Project 2: Trees and Heaps

Due: 02/23/2024 at 11:59PM

General Guidelines:

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not need to be public should generally be kept private (instance variables, helper methods, etc.). Additional file creation isn't allowed as Vocareum will only compile the files given in the startercode.

In general, you are not allowed to import any additional classes or libraries in your code without explicit permission from the instructor! Adding any additional imports will result in a 0 for the entire project.

Note on Academic Dishonesty:

Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. **You MUST do your own work**! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online!

Note on implementation details:

Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests:

The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview:

In this project, you will work on three problems. First, we will implement a generalized tree data structure, and then we will use the tree data structure to implement a pseudo filesystem that mimics the operations of a real filesystem. The system will provide the ability to create, read, update, and delete files and directories in a hierarchical structure. Finally, you will implement a heap that will be used to find the largest file currently in the filesystem.

Part 1: Tree (30 Points):

Main idea:

In part 1, you will implement a generalized tree data structure. Trees are a widely used data structure that emulates a hierarchical structure with nodes. Each node typically points to its children, allowing for various practical applications, from representing hierarchical data to aiding in efficient search algorithms. Trees are particularly important in areas like computer graphics, databases, and advanced algorithms.

Traditionally, trees use pointers or references to denote relationships between parent and child nodes. However, another effective approach to store a tree is by using an array where each node is accessed via its index. This method is beneficial for scenarios where frequent memory allocations and deallocations occur, as it can leverage continuous memory blocks and reduce fragmentation.

Breakdown of part 1:

Array-Based Representation: Instead of using pointers to represent the child-parent relationship, nodes are stored in an array. Each node's position (index) in the array implicitly represents its identity. Children and parents of a particular node can be accessed by storing their indices directly in the node.

Generic Class Implementation: By implementing the tree as a generic class in C++, we can store any data type within the tree nodes, making the tree more versatile. This is achieved through the use of C++ templates.

Pooling for Performance: Pooling is a technique where we replace “removal” with “recycling” and storing the available space into a “pool” so we can retrieve it the next time we need to. In our project, removing a node doesn’t delete the data and free the space from the memory; instead we mark it as “recycled” and put it into the pool. When a new node is required, we reuse a recycled node if one is available, otherwise we add it to the end of the node array. This boosts performance, especially in scenarios where nodes are frequently created and destroyed, by reducing overhead from dynamic memory allocation and preventing memory fragmentation.

Default Root Node: In this tree implementation every node must have a parent, with the exception of the root node. The node index 0 is reserved for the root and cannot be removed or reparented. When the tree is constructed, the root node should be created before any other operation.

In conclusion, by combining the contiguous memory layout of an array, the flexibility of C++ generics, and the efficiency of pooling, we can craft a highly performant tree data structure suitable for various high-performance applications. This approach can significantly enhance operations and traversal speeds while ensuring memory usage remains optimal.

Class: tree:

In tree.hpp, you will implement a template class named *tree_node* that represents the node of the tree and a template class named *tree* that holds a std::vector of *tree_node* and a std::queue of recycled node handles (aka indices). Since these are template classes, there's no .cpp involved. The implementation of each function/method of this class should be written inside the header file.

Public methods/function to implement:

1. You will find the functions and class members listed within the starter codes provided in this project for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started.

2. Friend class: In C++, the keyword "friend" is used to specify a unique relationship between classes or functions. When one class is declared as a friend of another, it means that the friend class is granted access to the private and protected members of the class in which it's declared as a friend. This breaks the usual encapsulation rules of C++, allowing for more flexibility in certain design scenarios. In this project, the *tree* class is a friend of the *tree_node* class, meaning methods of *tree* can freely manipulate the private members of *tree_node*.

3. Please make sure you use the std::vector functions *push_back()* or *emplace_back()* when trying to allocate a new node, and use std::queue functions *pop()* and *push()* when trying to retrieve or recycle nodes to the pool represented by the queue.

4. In tree.hpp there are several custom exception classes that should be thrown in various situations. For example, when a method argument specifies a node handle that is outside of the range of existing nodes, you should throw an *invalid_handle* exception:

   ```
   throw invalid_handle();
   ```

   The types and order of exceptions to throw are specified in each function below. These will be checked in the test cases.

5. Description of *tree_node* functions:

   1. ***tree_node_data& tree_node<tree_node_data>::ref_data():***
      Return a reference to the content of the node, if it is not recycled. Throw a *recycled_node* exception otherwise.

2. ***bool tree_node<tree_node_data>::is_recycled() const:***
   Return whether the node is recycled or not.

3. ***handle tree_node<tree_node_data>::get_handle() const:***
   Return the node's handle (its index into the node vector). Do not throw an exception if the node is recycled.

4. ***handle tree_node<tree_node_data>::get_parent_handle() const:***
   Return the handle of the node's parent, if the node isn't recycled. Throw a *recycled_node* exception otherwise.

5. ***const std::vector<handle>& tree_node<tree_node_data>::peek_children_handles():***
   Return a const reference to the array of handles to children of the node, if the node isn't recycled. Throw a *recycled_node* exception otherwise.

6. Description of *tree* functions:
   1. ***tree<tree_node_data>::tree():***
      Constructor of the *tree* class. You should allocate the root node here, with handle 0.

   2. ***handle tree<tree_node_data>::allocate(handle parentHandle):***
      Allocate a new node as the child of *parentHandle*, from the recycled pool (if non-empty), otherwise by extending the node vector using *push_back()*. Update the necessary variables for both cases. Throw any exceptions in the following order:
      i. If the *parentHandle* is not a valid index in the vector, throw *invalid_handle*.
      ii. If the *parentHandle* is a recycled node, throw a *recycled_node* exception.

   3. ***void tree<tree_node_data>::remove(const handle handle):***
      Remove the node indexed by the *handle* and add it to the pool. If the node has children, recursively remove and recycle the children first, followed by the node. Add the node's children to the pool queue before you add the node itself. Throw any exceptions in the following order:
      i. If the *handle* is not valid, throw *invalid_handle*.
      ii. If the *handle* refers to a recycled node, throw *recycled_node*.

   4. ***void tree<tree_node_data>::set_parent(const handle targetHandle, const handle parentHandle):***
      Set the parent of the node represented by *targetHandle* to the node represented by the *parentHandle*. You will also need to remove *targetHandle* from its original parent's child array. Throw any exceptions in the following order:
      i. If either handle is not valid, throw *invalid_handle*.
      ii. If either handle refers to a recycled node, throw *recycled_node*.

5. ***const std::vector<tree_node<tree_node_data>>& tree<tree_node_data>::peek_nodes() const:***
   Return a const reference to the vector of *tree_nodes*.

6. ***tree_node<tree_node_data>& tree<tree_node_data>::ref_node(handle handle):***
   Return a reference to the node represented by the *handle*. If the handle is not valid, throw an *invalid_handle* exception. Do not throw an exception if the node is recycled.

Part 2: Pseudo Filesystem (50 Points):

A pseudo filesystem is a representation of a filesystem structure and its operations without relying on actual underlying disk-based implementations. By simulating the operations of a real filesystem in software, it allows for the study, testing, or application of file management concepts in environments where actual file operations may not be possible or desired.

Breakdown of part 2:

Tree-Based Representation: At its core, every filesystem is hierarchically structured, akin to a tree. Every entity, be it a file, folder, or symbolic link, can be represented as a node within this tree. Our pseudo filesystem utilizes the tree structure from part 1 to simulate this hierarchy. By employing a template class for the tree's implementation, it ensures adaptability and robustness, allowing nodes to encapsulate different data types corresponding to files, folders, and symbolic links.

Core Operations:

1. *Files:* Users can create, delete, and rename files. This manipulation directly translates to operations on the nodes of our tree. Files also have a size, in bytes. The total size of all files in the filesystem may not exceed the filesystem capacity.
   Note: these are not real files, so you should not actually allocate the size in memory or on disk.
2. *Directories (Folders):* Like files, directories can also be created, deleted, and renamed. Directories also contain files, directories, and symlinks, mimicking the branching in our tree structure. A directory should only be deleted if it is empty.
3. *Symbolic Links:* Symbolic links, or symlinks (or just links), are special files that point to another file, or directory, or even another symbolic link. In our tree structure, they can be visualized as references or shortcuts to other nodes. In some cases, a symlink may point to an object that has been removed from the filesystem.

Path *Calculation*:

1. *Absolute Paths*: An absolute path gives the complete path to a particular file or directory from the root of the filesystem. Traversing our tree from the root node to the desired node generates the absolute path. An absolute path must start with "/", referring to the root directory.

In essence, by harnessing the power of tree data structures and the versatility of C++ generics, our pseudo filesystem effectively mimics the core operations and path calculations of a real-world filesystem.

Class: filesystem:

In filesystem.hpp, you will implement a class named *filesystem* and complete the function definitions included in filesystem.cpp. Similar to the tree in part 1, each object in the filesystem is referred to by its handle, i.e. the index into the underlying tree's node vector. For this part, we do not make a distinction between recycled handles and indices past the end of the array. If a handle does not refer to an allocated node, it is considered invalid (recycled or otherwise).

Public methods/function to implement:

1. ***filesystem::filesystem(size_t sizeLimit):***
   Constructor to initialize the file system with the given size limit. The *sizeLimit* defines the maximum size of the entire file system. Since the filesystem uses the tree from part 1 internally, the tree constructor will automatically create a root node at index 0. This root node should act as the root directory, which all new files, directories, and links will be descendents of.

2. ***bool exist(handle targetHandle):***
   Check and return if the given target handle exists (allocated and not deleted/recycled). This method should not throw any exceptions.

3. ***handle filesystem::create_file(size_t fileSize, const std::string& fileName):***
   handle filesystem::create_file(size_t fileSize, const std::string& fileName, handle parentHandle):
   Create a new file as a child of the directory pointed to by *parentHandle*, or the root directory if *parentHandle* isn't given. Instantiate the file with the given size and name. Throw any exceptions in the following order:
   - ***a.*** If the *parentHandle* doesn't exist or isn't a directory, throw *invalid_handle*.
   - ***b.*** If the *fileName* contains a '/' character, throw *invalid_name*.
   - ***c.*** If adding the file would exceed the file system capacity, throw *exceeds_size*.
   - ***d.*** If an object with the same name already exists in the directory, throw *file_exists*.

4. ***handle filesystem::create_directory(const std::string& directoryName):***
   handle filesystem::create_directory(const std::string& directoryName, handle parentHandle):
   Create a new directory as a child of the directory pointed to by *parentHandle*, or the root directory if *parentHandle* isn't given. Instantiate the directory with the given name. Throw any exceptions in the following order:
   - ***a.*** If the *parentHandle* doesn't exist or isn't a directory, throw *invalid_handle*.
   - ***b.*** If the *directoryName* contains a '/' character, throw *invalid_name*.
   - ***c.*** If an object with the same name already exists in the directory, throw *directory_exists*.

5. ***handle filesystem::create_link(handle targetHandle, const std::string& linkName):***
   handle filesystem::create_link(handle targetHandle, const std::string& linkName, handle parentHandle):
   Create a new link as a child of the directory pointed to by *parentHandle*, or the root directory if *parentHandle* isn't given. Instantiate the link with the given name and have it point to *targetHandle*. Throw any exceptions in the following order:
   - ***a.*** If either the *parentHandle* or the *targetHandle* do not exist, throw *invalid_handle*.
   - ***b.*** If the *parentHandle* isn't a directory, throw *invalid_handle*.
   - ***c.*** If the *linkName* contains a '/' character, throw *invalid_name*.
   - ***d.*** If an object with the same name already exists in the directory, throw *link_exists*.

6. ***bool filesystem::remove(handle targetHandle):***
   Delete the file, directory, or link indicated by *targetHandle*. If the target is a directory, remove only if it's empty. If the target is a link, only remove the link, not the object it points to. Return true if the target is removed, and false otherwise. If the *targetHandle* doesn't exist, throw *invalid_handle*.

7. ***void filesystem::rename(handle targetHandle, const std::string& newName):***
   Rename the file/directory/symlink referred to by *targetHandle* to *newName*. Throw any exceptions in the following order:
   - ***a.*** If the *targetHandle* doesn't exist, throw *invalid_handle*.
   - ***b.*** If the *newName* contains a '/' character, throw *invalid_name*.
   - ***c.*** If a file/directory/symlink with *newName* already exists, throw *name_exists*.

8. ***std::string filesystem::get_absolute_path(const handle targetHandle):***
   Return the absolute path from the root to the *targetHandle*. The absolute path should start with '/' and be followed by the name of each directory along the path from the root to the target directory/file/symlink, separated by '/' characters, and ending with the target's name. The path should not end in '/'. If the *targetHandle* doesn't exist, throw *invalid_handle*.

9. ***std::string filesystem::get_name(handle targetHandle):***
   Return the name of the file/directory/link referred to by *targetHandle*. If the *targetHandle* doesn't exist, throw *invalid_handle*.

10. ***handle filesystem::follow(handle targetHandle):***
    Return the handle of the file or directory pointed to by the symlink referred to by *targetHandle*. Symlinks to symlinks should be followed recursively to their final destination. If the *targetHandle* doesn't exist, throw *invalid_handle*.

11. ***size_t filesystem::get_file_size(handle targetHandle):***
    size_t filesystem::get_file_size(const std::string& absolutePath):
    Return the size of the file that is referred or linked to by *targetHandle* or *absolutePath*. See *get_handle()* below for details on resolving *absolutePath*. Throw any exceptions in the following order:
    - ***a.*** If the *absolutePath* doesn't point to an existing filesystem object, throw *invalid_path*.
    - ***b.*** If the *absolutePath* points to an existing filesystem object that's not a file or a link to a file (followed recursively), throw *invalid_handle*.
    - ***c.*** If the *targetHandle* doesn't exist, throw *invalid_handle*.
    - ***d.*** If the *targetHandle* doesn't point to a file or a link to a file (followed recursively), throw *invalid_handle*.

12. ***handle filesystem::get_handle(const std::string& absolutePath):***
    Return the handle of the directory/file/symlink represented by *absolutePath*. Follow symlinks in the middle of the path but not at the end. For example, if the last name in *absolutePath* refers to a symlink, return the handle of that symlink, not its destination. However, if a name in the middle of *absolutePath* refers to a symlink pointing to a directory, follow the link and continue the traversal. Symlinks to symlinks should be followed recursively to their final destination. If *absolutePath* does not point to an existing filesystem object, throw *invalid_path*.

13. ***size_t filesystem::get_available_size() const:***
    Return the size remaining in the filesystem.

Part 3: Maxheap for file size statistics (20 Points):

A max heap is a specialized tree-based data structure that satisfies the heap property: for any given node I, the value of I is greater than or equal to the values of its children. In simpler terms, the largest element is found at the root of a max heap. To manage files within a filesystem, especially when there's a need to consistently and quickly identify the largest file, a max heap can be an indispensable tool. By representing each file with a node in the max heap, where the value is the file's size, the heap ensures that the largest file always resides at the root. This allows for immediate finding of the largest file without scanning the entire filesystem.

Consider a scenario where a system admin needs to consistently monitor and perhaps offload the largest files to ensure optimal disk space usage. Instead of performing an exhaustive search each time, the max heap provides an immediate answer, making the management task far more efficient.

Public methods/function to implement:

1. You will find the functions and class members listed within the starter code provided in the file file_size_max_heap.hpp for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started.

2. Description of *file_size_max_heap* functions:
    1. ***void file_size_max_heap::push(const size_t fileSize, const handle handle)*:**
       Push an element with the corresponding *fileSize* and *handle* onto the max heap.

    2. ***void file_size_max_heap::remove(const handle handle):***
       Find and remove the element with the corresponding *handle* from the max heap. If the handle is not in the heap, throw *invalid_handle*.

    3. ***handle file_size_max_heap::top() const:***
       Return the handle of the element with the greatest file size in the heap in O(1) time. If the heap is empty, throw *heap_empty*.

3. In addition to implementing the max heap, you will also need to incorporate the *file_size_max_heap* class into your *filesystem* class. Each time you add or remove a file, you will need to update the heap. Finally, to access the largest file, you will implement the following method:
    1. ***handle filesystem::get_largest_file_handle() const:***
       Return the handle to the largest file in O(1) time. File sizes are guaranteed to be unique, so you do not have to worry about breaking ties. If the filesystem does not contain any files, throw *heap_empty*.

Building and Testing Your Code:

The following instructions are for Linux or Mac systems. For Windows, we recommend installing WSL as detailed in the Windows section below.

CMake:

Included with the starter code is a CMake configuration file. CMake is a cross-platform build system that generates project files based on your operating system and compiler. The general procedure for building your program is to run CMake, and then build using the generated project files or makefile. On Linux that process looks like this:

1. Navigate to your project directory, where CMakeLists.txt is located
2. Create a build directory for compiled executables:
   `$ mkdir build`
   Note: the $ symbol indicates the shell prompt - do not actually type $
3. Navigate to the new build directory
   `$ cd build`
4. Run CMake, which generates a Makefile
   `$ cmake ..`
   The .. indicates the parent directory where your CMake configuration is
5. Compile the code using the Makefile
   `$ make`

If you want to compile with debug symbols, replace step 4 with

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

If successful, the program will compile two executables: tree-app and filesystem-app. These programs accept input from stdin and print to stdout. You can use them to test your implementations. For details on how they work, consult the source code.

Testing:

Provided with the starter code are several sample inputs and expected outputs for tree-app and filesystem-app. To see if your implementation is correct, run the programs with the inputs, and compare the output with the expected outputs. For example:

```
$ ./tree-app < ../sample_tests/input/tree_test_00.txt > tree_test_00_output.txt
$ diff -qsZB tree_test_00_output.txt ../sample_tests/expected/tree_test_00.txt
```

If diff reports the files as identical, then your implementation matches the expected output.

Windows Setup:

For Windows users, we recommend using Windows Subsystem for Linux (WSL) and Visual Studio Code as your development environment. Using WSL, you can use all the same commands above to build and run on your local Windows computer. WSL can be installed through the Microsoft Store or by following the instructions here. The link also contains some customization/setup directions. We also recommend downloading Ubuntu from the Microsoft Store and using it to navigate your WSL and install packages.

You will need to install compilers and CMake in order to run and test your project. The required compilers as well as the "make" command can be installed using Ubuntu's build-essential package. This and CMake can be installed with the commands:

```
$ sudo apt update

$ sudo apt install build-essential cmake
```

You will have to enter your WSL password before installation begins to allow privileged access. You can check that the needed items were installed by running the corresponding command with the --version argument. If a version is printed, then that command was installed.

After downloading the work folder from Vocareum, you can move it into your desired directory in WSL using the WSL command line. You can access your Windows home directory from WSL using the path "/mnt/c/Users/<user>/"

To work on your code in WSL, you could use a text editor like Vim or Nano, or you could install the WSL extension in Visual Studio Code. After installing the extension, you can select the blue box at the bottom left of the Visual Studio window and select "Connect to WSL" from the dropdown window. This will allow you to open any WSL files and directories in your Visual Studio environment, and if you open a Terminal you can navigate your directories using the WSL Command Line. (Note: you may need to reinstall other extensions for WSL that you have already installed on your Windows version of VSCode.) From the Terminal, you can execute the `cmake` and `make` commands as specified above to build locally.

You should also test your code on Vocareum. After uploading your files, press the **Run** button on the top right. This will execute a script that will compile your code and run it against all of the sample test cases, and print a summary of which tests failed. You should run your code before submitting it to make sure it compiles and executes properly, since you will have a limited number of submissions.

Watch out for infinite loops! If your program enters into an infinite loop during a test, you will receive a 0 for the entire assignment. Make sure you have safeguards in your code to prevent this from happening.

Note that the sample test cases are not exhaustive and purposefully do not cover all possible cases. When you submit your code it will be tested against many more tests with more thorough cases, which will not be made available to inspect. It is your responsibility to ensure your program is correct, by devising new tests and manually inspecting their outputs.