ECE 2400 Computer Systems Programming, Fall 2017
Prelim 2 Prep

revision: 2017-11-04-22-42

These problems are not meant to be exactly like the problems that will be on the prelim. These problems are instead meant to represent the kind of understanding you should be able to demonstrate on the prelim.

In the following problems, we will explore two different data structures to track the directory hierarchy in an operating system. The data hierarchy could be used to store metadata about each directory in the system (e.g., number of files in each directory, access control for the directory), and the operating system would provide an *application level interface* (API) to enable system-level software to read and write the directory hierarchy. For example, the following commands will make four directories and then display the directory hierarchy.

```
% mkdir foo
% mkdir foo/bar0
% mkdir foo/bar1
% mkdir foo/bar0/baz
% tree
```

The directory hierarchy is made up of directories where each *parent directory* can have one or more *child directories*. So in the above example, `foo` is the parent of `bar0` and `bar1`. Or equivalently, `bar0` and `bar1` are the children of `foo`.

We will explore two data structures to store the directory hierarchy. The data structures should support making new directories and printing the full directory hierarchy. For this problem, you can assume that the maximum number of directories in the system is 16 and that there are no more than four child directories in any given parent directory. There are two common errors that we might want our data structure to check:

- **Directory Already Exists:** The data structure should cause an error if the user attempts to make a directory that already exists. *For this problem you are required to correctly detect this error and throw an* `std::invalid_argument` *exception.*
- **Parent of Directory Does Not Exist:** Ideally, the data structure should cause an error if the user attempts to create a new directory where one or more of the parents of that directory do not exist. *To simplify the problem, you can assume the user never attempts to create a new directory unless all of the parents of that directory already exist.*

Problem 1. Array-Based Directory Hierarchy Data Structure

Our first data structure simply uses an array to store all of the directories in the system. The class declaration is as follows:

```
class DirList
{
 public:

  DirList();
  void mkdir( const std::string& path );
  void print() const;

 private:
  size_t      m_paths_sz;
  std::string m_paths[16];
};
```

The `m_paths_sz` variable is used to store how many directories are presently in the system.

Part 1.A Implementing `DirList`

Implement the constructor, `mkdir`, and `print` member functions in the `DirList` class. Your implementation of `mkdir` should thrown an `std::invalid_argument` exception if the user attempts to make a directory that already exists. You can assume you have the following private helper member function

```
DirList::DirList()
  : m_paths_sz(0)
{ }

void DirList::mkdir( const std::string& path )
{
  for ( size_t i = 0; i < m_paths_sz; i++ ) {
    if ( m_paths[i] == path )
      throw std::invalid_argument("dir already exists!");
  }

  m_paths[m_paths_sz] = path;
  m_paths_sz++;
}
```

```
void DirList::print() const
{
  for ( size_t i = 0; i < m_paths_sz; i++ )
    std::cout << m_paths[i] << std::endl;
}
```

Part 1.B Algorithm Analysis for `DirList`

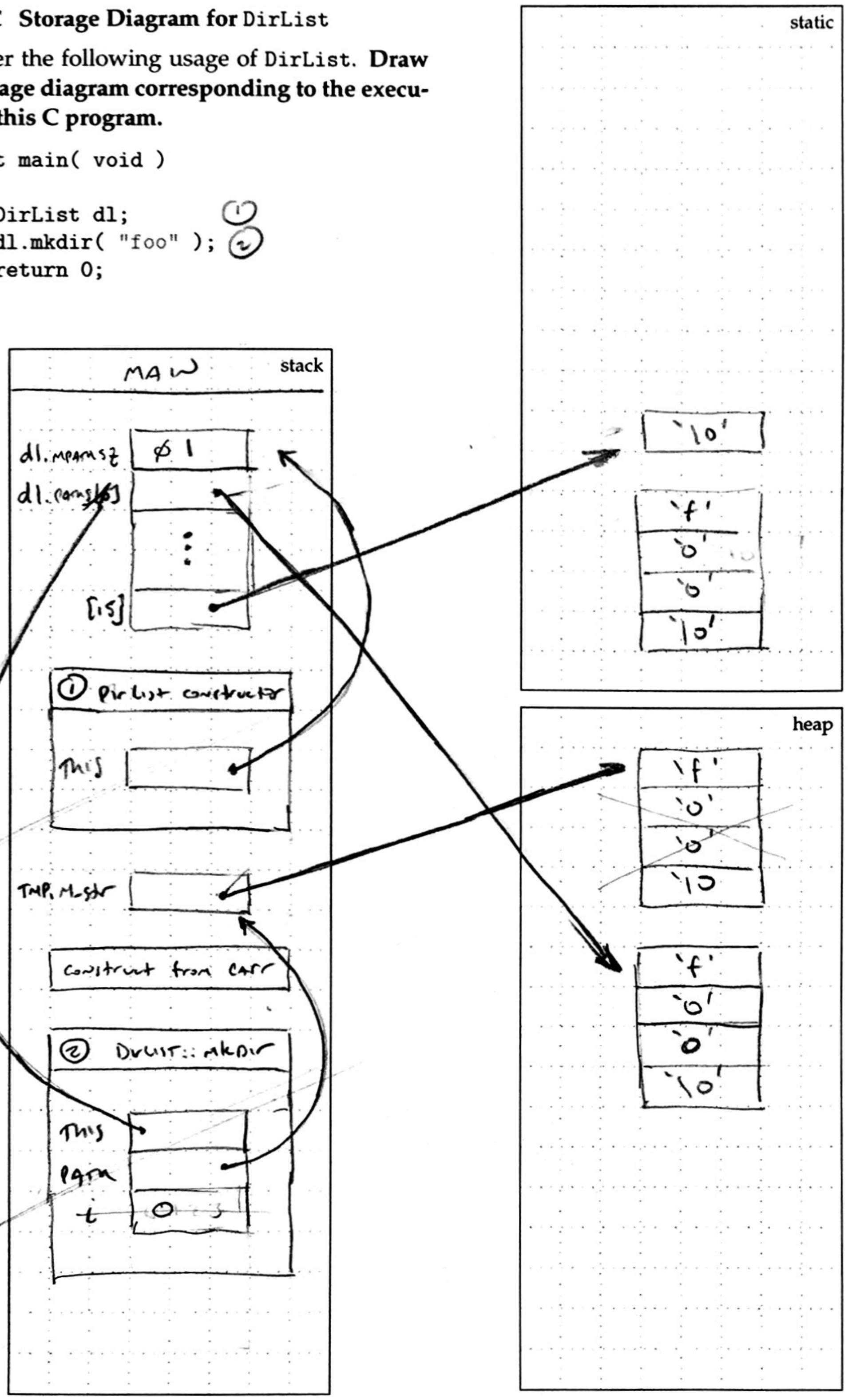
For this analysis assume we wish to store N paths in the directory hierarchy. **What is the best- and worst-case time complexity for `mkdir` as a function of N? What is the space complexity of this data structure as a function of N?**

Each call to `mkdir` will need to do a linear search through all of the paths in the data structure to verify that the path being added does not exist. This search will be $O(N)$. The data structure will have N entries in the `m_paths` array, so the space complexity is $O(N)$. Note that this analysis ignores the actual size in characters of each directory name which may or may not be a reasonable assumption.

Part 1.C Storage Diagram for `DirList`

Consider the following usage of `DirList`. **Draw the storage diagram corresponding to the execution of this C program.**

```
int main( void )
{
  DirList dl;             ①
  dl.mkdir( "foo" );      ②
  return 0;
}
```



Problem 2. Tree-Based Director Hierarchy Data Storage

Our second data structure uses a *tree*. A tree is like a linked list, except every node can have more than one "next node". The class declaration is as follows:

```
class DirTree
{
 public:

  DirTree() : m_root(Node()) { }

  void mkdir( std::string path );

  void print()
  {
    for ( size_t i = 0; i < m_root.children_size; i++ )
      print_h( "", m_root.children[i] );
  }

 private:

  class Node {
   public:

    Node( const std::string dir_ = "" )
      : dir(dir_), children_size(0)
    { }

    std::string dir;
    size_t      children_size;
    Node*       children[4];
  };

  size_t split( std::string dirs[], std::string path )
  {
    size_t i = 0;
    auto pos = path.find("/");
    while ( pos != std::string::npos ) {
      dirs[i] = path.substr(0,pos);
      i++;
      path.erase( 0, pos+1 );
      pos = path.find("/");
    }
    dirs[i] = path.substr(0,pos);
    i++;
    return i;
  }

  void print_h( const std::string& prefix, Node* node_p );

  Node m_root;

};
```

Part 2.A Implementing `DirTree::mkdir`

Implement the `mkdir` member function. This function should iterate across the directories in the path. For each directory, the function should check to see if that directory already exists and if not, the function should dynamically allocate a new `Node` to represent that directory. The end result should be a tree which represents the directory hierarchy. We have provided a private helper function to split a path into an array of strings, with one string per directory in the path.

```
void DirTree::mkdir( std::string path )
{
  std::string dirs[8];
  size_t dirs_size = split( dirs, path );
```

```
  // Search through parents

  Node* node_p = &m_root;
  for ( size_t i = 0; i < dirs_size-1; i++ ) {
    for ( size_t j = 0; j < node_p->children_size; j++ ) {
      if ( node_p->children[j]->dir == dirs[i] )
        node_p = node_p->children[j];
    }
  }

  // Make sure directory does not exist

  for ( size_t j = 0; j < node_p->children_size; j++ ) {
    if ( node_p->children[j]->dir == dirs[dirs_size-1] )
      throw std::invalid_argument("dir already exists!");
  }

  // Create new node

  Node* new_node_p = new Node(dirs[dirs_size-1]);
  node_p->children[node_p->children_size] = new_node_p;
  node_p->children_size += 1;
}
```

Part 2.B Implementing `DirTree::print_h`

Implement the `print_h` private helper member function. This should be a recursive function which prints out the current path corresponding to the given `Node` pointer and then recursively processes the current node's children.

```
void DirTree::print_h( const std::string& prefix, Node* node_p )
{
  std::string new_prefix = prefix + "/" + node_p->name;
```

```
  std::cout << new_prefix << std::endl;
  for ( size_t i = 0; i < node_p->children_size; i++ ) {
    print_h( new_prefix, node_p->children[i] );
  }
}
```

Part 2.C Algorithm Analysis of `DirTree`

For this analysis assume we wish to store N paths in the directory hierarchy. **What is the best-case and worst-case time complexity for `mkdir` as a function of N? What is the space complexity of this data structure as a function of N?**

Each call to `mkdir` will need to search all the way through the tree from the root to one of the leaves. We will need to do one comparison per node; we check to make sure the parents exist as we go through the tree, and when we get to the end of the tree we check to make sure that the new directory does not exist. In the worst case, every directory has exactly one child, which means our directory "tree" is really a chain of nodes, and thus this search requires $O(N)$ comparisons. In the best case, every directory has exactly four children, and the search requires only $O(log_4 N)$ comparisons.

Every new path adds one node to the tree, so our tree has $O(N)$ nodes and thus the space complexity is also $O(N)$. Note that this analysis ignores the actual size in characters of each directory name which may or may not be a reasonable assumption. The tree-based approach does exploit some redundancy in the way it stores paths, so this might warrant deeper analysis.