Data Structures

Table of Contents

Overview

The data structures lab is intended as a review of concepts that will be necessary in later labs. In this lab we will examine and implement three data structures: Queue-Lists, Hash-Maps, and Trees. When implementing this lab **do not** look at **any** code you have written before. ***Do not*** *copy and paste from another project.* Write the code from scratch and avoid the temptation to seek help from others.

Code Conventions

The grader of the lab will enforce some or all of these conventions by penalizing any violations.

- Do not change the `public`, `protected`, or `friend` interfaces of an existing class.
- If you absolutely have to add helper methods or variables, you must place them in `private` scope (not `protected` scope), and you must preserve `const` correctness.
- If you plan to declare helper function prototypes outside a class body, choose one of these options instead:
  - You may *define* such functions in the source (.cpp) file where they are needed.
  - You may declare them as `private static` methods.
- Do not `const_cast`. This implies a deficiency in the project API. File a bug report instead.
- Do not C-style cast in place of a `const_cast`.

Queue-List

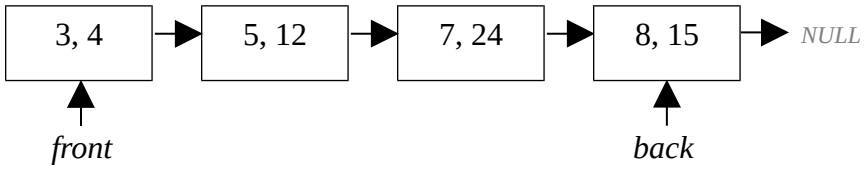
You should be able to complete this part of the lab within a short period of time (30 minutes or less). If you need more time, make sure you review the data structures concepts to prepare for future lab sessions.

Data Structure

A singly linked list is a type of linked list which is linked in only one direction: from front to back. It usually terminates in a pointer to `NULL` (0). Depending on how one adds to or remove items from either end, the linked list can behave either as a stack or as a queue. In this lab, you are to implement the add/remove behavior as if the list were a queue.

A queue-list is made up of ***nodes***. Each node in the list contains some data (in this case, a location represented by a pair of coordinates) and a pointer to the next node in the list. The first node in the list is called the *front*, and the last node is called the *back*. We should always keep separate pointers to the front and back of the list.

A queue-list with location coordinates as data



Implementation

Your `QueueList` class will span two header files. One holds the class definition, including the nested `Node` and `Iterator` classes, the `front` and `back` member variables, and the member function declarations. The other header file will hold the member function definitions, which you are to implement.

Todo List

Write the `QueueList::Iterator` operators and `QueueList` methods stated below. You must provide them from scratch. Look inside the primary header file for how the other methods are implemented in case you've forgotten.

QueueList<T>::Iterator

`T operator*() const`
Return the element at the iterator's current position in the queue.

`Iterator& operator++()`
Advance the operator one position in the queue. Return this iterator. ***NOTE***: if the iterator has reached the end of the queue (past the last element), point the **`node`** member to **`NULL`**.

QueueList<T>

`bool isEmpty() const`
Return `true` if there are no elements, `false` otherwise.

`void enqueue(T element)`
Insert the specified element to the list. Handle the special case when the list is empty.

`T getFront() const`

Return the first element in the list.

`void dequeue()`
Remove the first element from the list. Handle the special case when it is also the last element.

`void removeAll()`
Remove all elements from the list.

`bool contains(T element) const`
Return `true` if you find a node whose data equals the specified element, `false` otherwise.

`void remove(T element)`
Remove the first node you find whose data equals the specified element. Be sure to update the pointers appropriately. Test your code for the following scenarios:

- If you remove the first node from the list
- If you remove a node from the middle of the list
- If you remove the last node from the list
- If you remove the only node from the list

A remove method may look something like this:

```
void remove(data to remove)
{
    Setup up an iterator, starting at the beginning of the list;

    while (iterator has nodes to traverse)
    {
        if (the node matches the data)
        {
            remove the node, adjust the list pointers, and exit;
        }
    }
}
```

Hash-Map

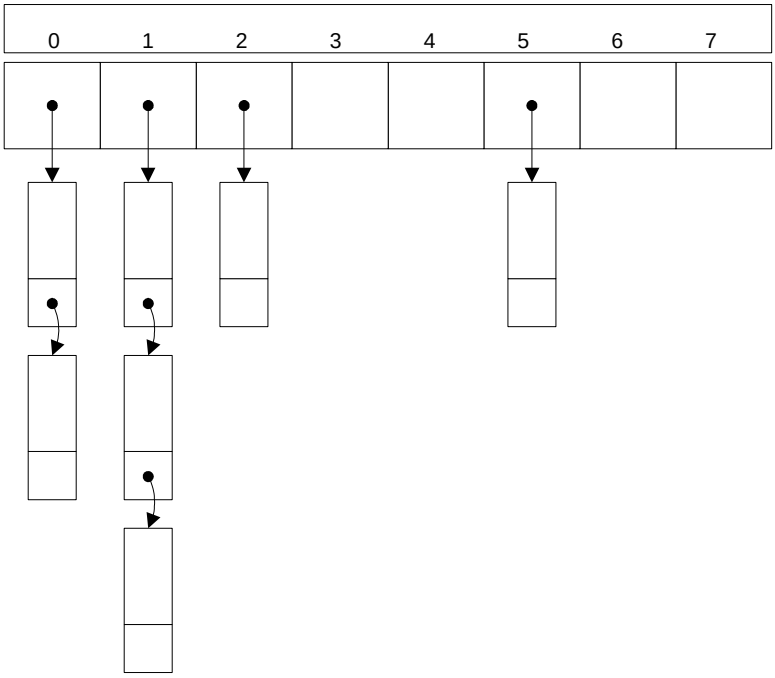
Sometimes it is useful to store information by associating some key with some value without knowing how many elements will be needed. For example, if we wanted to write a computer dictionary, we would want to associate words with definitions but might allow new words to be added later. A ***hash map*** can be used for such purposes.

Data Structure

A hash map, a type of associative array, is a data structure that associates a value with a key. (Our dictionary could be placed in a hash map, where the ***keys*** are the words and the ***values*** are their definitions.) The map's ***hash function*** generates a ***hash code*** from the key and reduces it (usually by modulo) to an index, and the value-key pair is placed in the table at the index (called the ***bucket***).

Suppose our dictionary's table size is 32. If the hash code for a key (word) in our dictionary is the ASCII value of the first character and we wanted to add the word “aardvark”, our hash code would be 97, so our hash function would return an index of 1. The word “aardvark” and its definition would be stored in bucket 1 in the table.

One problem with hash maps is collision between keys. If we also wanted to store the string “amnesia”, the hash function would generate the same hash code 97, so we’d be using bucket 1 again. If there are too many collisions, we might consider changing our hash function or the size of our table. To deal with collisions, we usually make each hash map bucket a linked list that can hold multiple pairs. When we want to store or retrieve a value, we first compute the bucket of the key and then step through the bucket's list until we find the correct value.



A common implementation of a hash map using linked lists as buckets.

Implementation

Your `HashMap` class will span two header files. One holds the class definition, including the `buckets` and `bucketCount` member variables, as well as the member function declarations. The other header file will hold the member function definitions, which you are to implement.

Any time you need to use a hash code to access the correct bucket, you should perform a modulo operation against the array size [`hashCode % bucketCount`] to ensure that any index generated will be within the range of the array. You will encounter heap-corruption errors at runtime if you do not perform this step.

Make sure your constructors initialize *everything* properly.

Todo List

Write the `HashMap::Pair` constructor, the `HashMap::Iterator` constructor and operators, and the `HashMap` constructor, destructor, and methods.

HashMap<K,V>::Pair

`Pair(K const& k)`
Initialize the key to the parameter passed in. Default-construct the value.

HashMap<K,V>::Iterator

`Pair const& operator*() const`
Return the key/value pair at the iterator's current position in the map.

`Iterator& operator++()`
Advance the operator one position in the map. Return this iterator. ***NOTE***: if the iterator has reached the end of the map (past the last pair), point the **`node`** member to **`NULL`**.

HashMap<K,V>

`HashMap(unsigned int count)`
Initialize the array to hold the specified number of buckets.

`~HashMap()`
Remove all elements from the hash map and clean up the memory that the array itself is stored in.

`void removeAll()`
Remove all elements from the hash map.

`V& operator[](K const& key)`
Use the key's hash code to determine the correct bucket.

- If the bucket is empty, add a node for the specified key, then return the node's value.
- If the bucket contains the specified key, return the associated value.
- Otherwise, add a node for the specified key to the end of the bucket, then return the node's value.

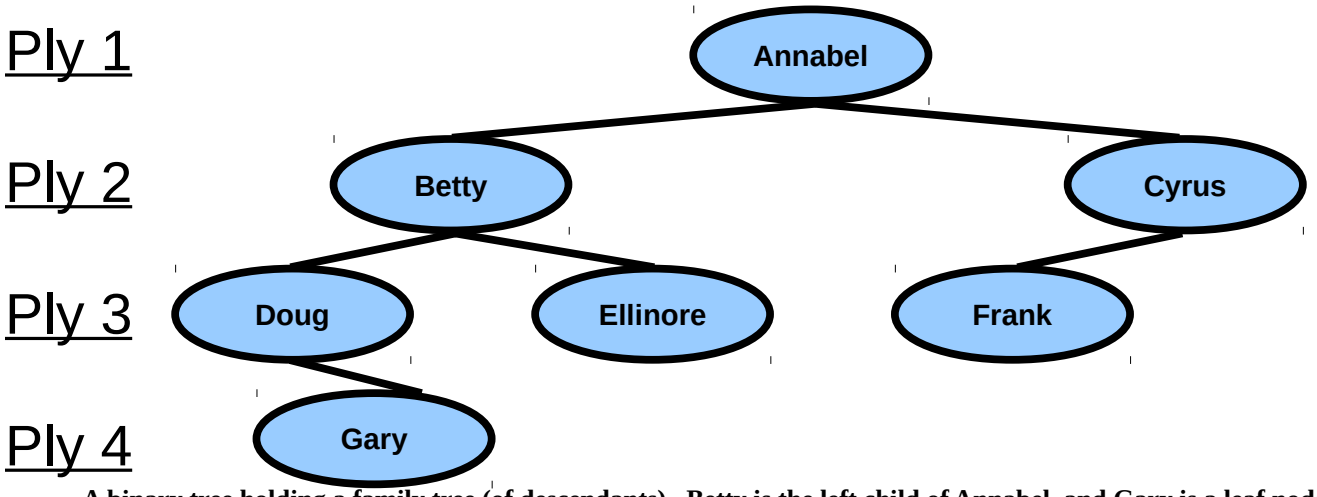
`void remove(K const& key)`
Use the key's hash code to determine the correct bucket, then find and remove the first node that contains the specified key.

Tree

Information is often non-linear: it cannot be easily placed “in line” because of more complex relationships that exist between pieces of data. This kind of data is usually difficult to store in arrays, lists, or tables. For example, a descendant family tree would be difficult to implement in a list fashion, because each parent may have several children. This information is more easily stored in a tree.

Data Structure

A ***tree*** is made up of a ***root*** node and its descendants. Each node in the tree may have zero or more ***child*** nodes. A node with no children is called a ***leaf*** node. The ply level of node is its depth.



A binary tree holding a family tree (of descendants). Betty is the left child of Annabel, and Gary is a leaf node.

Traversal

There are several ways to traverse a tree, each with benefits and drawbacks. Today we will examine three: ***depth-first preorder***, ***depth-first postorder***, and ***breadth first***.

Preorder Traversal

In a (depth-first) preorder traversal a node's data is processed before its children's data:

```
void traverse(Node node, F function)
{
      function(node);
      traverse(each child of node);
}
```

A preorder traversal of the example tree would yield:
Annabel, Betty, Doug, Gary, Ellinore, Cyrus, Frank

Postorder Traversal

In a (depth-first) postorder traversal a node's children's data is processed before its own:

```
void traverse(Node node, F function)
{
      traverse(each child of node);
      function(node);
}
```

A postorder traversal of the example tree would yield:
Gary, Doug, Ellinore, Betty, Frank, Cyrus, Annabel

Breadth-First Traversal

A breadth-first traversal is one that starts from the root and visits all the nodes of each ply before visiting the nodes in the next ply. The techniques involved in a breadth first-traversal are very different from those of depth-first traversals. A queue (or list) and a loop are used to store nodes rather than relying on recursive calls:

```
traverse(Node rootNode, F function)
{
      queue.enqueue(rootNode);
      while (queue.notEmpty())
      {
            Node node = queue.popfront();
            function(node);

            queue.enqueue(each child of node);
      }
}
```

The output of this traversal would look something like this:
Annabel, Betty, Cyrus, Doug, Elinore, Frank, Gary

Implementation

In this part of the lab you will be responsible for implementing a generic tree and performing three different traversals on that tree. Trees have many uses in AI and other computer gaming fields, so it is important that you understand them fully and are comfortable using them by the end of this lab.

Todo List

Write the following `TreeNode` constructors, destructor, and methods. You may assume that the type `T` has a default constructor.

`TreeNode(T element, TreeNode* p = 0)`
Initialize the appropriate member variables to the corresponding parameters. Default-construct the other member variables. If a parent is specified, add this node to the parent's list of children.

`TreeNode(TreeNode* p = 0)`
Initialize the appropriate member variable to the corresponding parameter. Default-construct the other member variables. If a parent is specified, add this node to the parent's list of children.

`~TreeNode()`
Recursively delete all children.

`TreeNode const* getParent() const`

Access the parent node.

T const& getData() const
T& getData()
Access the data.

size_t getChildCount() const
Access the number of children this node has.

TreeNode const getChild(size_t i) const*
TreeNode getChild(size_t i)*
Return the child at index *i* in the list.

void deleteChildren()
Recursively delete all children, but remember that this is not the destructor, so what else do you need to do to avoid crashing the program?

Write the following `Tree` methods from scratch. You may need to define helper methods in the `Tree` class: if you do so, remember to follow the code conventions.

*void breadthFirstTraverse(void (*dataFunction)(T const&)) const*
Traverse the root and all its sub-nodes in breadth-first fashion, passing each node's data to the specified function.

*void preOrderTraverse(void (*dataFunction)(T const&)) const*
Traverse the root and all its sub-nodes in pre-order fashion, passing each node's data to the specified function.

*void postOrderTraverse(void (*dataFunction)(T const&)) const*
Traverse the root and all its sub-nodes in post-order fashion, passing each node's data to the specified function.

Grading

Worth: 40 points.
There will be 20 case tests worth 2 points each: 8 for Queue-List, 16 for Tree, and 16 for HashMap.

Submissions

All lab projects are to be submitted via the Course Management System (CMS.) Edit the following files and submit them at the end of this lab:

- QueueList.h
- HashMap.h
- TreeNode.h
- Tree.h
-

These files reside in your project's Structures directory. Do not submit any other files. Place them in the ROOT directory of your zip file, not in a subdirectory.