Computer Science Project Report

# Data Structure Design:

Each node in the doubly linked list consists of ‘value’ (of type ItemType), ‘count’ (of type integer) and pointers to the next and previous node. The ‘count’ data member removes the necessity to create a separate node for a duplicate entry. The head and tail pointers point to the start and end of the linked list respectively. Certain operations, such as erase and eraseAll, require a different set of code for dealing with extreme nodes. The head and tail pointers facilitate easy access to these extreme nodes. Below is a diagrammatic representation of a typical Multiset.

1640

2180

Count = 2

Value = “Red”

Next = 1640

Prev = 2180

Count = 4

Next = 1040

Value = “Blue”

Head = 2180

Prev = nullptr

Next = 3790

Count = 3

Prev = 1640

Value = “Green”

Value = “Yellow”

Count = 1

Tail = 3790

Next = nullptr

Prev = 1040

1040

3790

An empty Multiset will have a head and tail that are null pointers. A diagrammatic representation of an empty Multiset is shown below.

Tail = nullptr

Head = nullptr

# Pseudocodes:

## Multiset::insert Function

*check if Mutiset is empty*

*create a new node*

*add value and count to the node*

*link the node to the head and tail pointers*

*return true*

*repeatedly*

*check if the value already exists in the node*

*if found, increment the count for that value by 1*

*return true*

*move to next node*

*create a new node*

*add value and count to the node*

*link the node to the former last node in the linked list*

*link the tail pointer to this node*

*return true;*

## Multiset::erase Function

*check if a value in the first node needs to be deleted*

*check if count is greater than 1*

*reduce count by 1*

*return 1*

*otherwise*

*link head pointer to the next node in the list*

*delete the current node*

*return 1*

*check if a value in the last node needs to be deleted*

*check if count is greater than 1*

*reduce count by 1*

*return 1*

*otherwise*

*link tail pointer to the previous node in the list*

*delete the current node*

*return 1*

*repeatedly*

*find node that consists the value to be erased*

*return 0 if no node was found*

*if the node was found, check if count is greater than 1*

*reduce count by 1*

*return 1*

*otherwise*

*link the previous and next nodes with each other*

*delete the current node*

*return 1*

## Multiset::eraseAll Function

(This function is similar to the above function except that this function does not deal with the count of the value, as it deletes the entire node corresponding to that value)

*check if the first node needs to be deleted*

*link head pointer to the next node in the list*

*delete the current node*

*return 1*

*check if the last node needs to be deleted*

*link tail pointer to the previous node in the list*

*delete the current node*

*return 1*

*repeatedly*

*find node that needs to be erased*

*break loop if the node was found*

*return 0 if no node was found*

*if the node was found,*

*link the previous and next nodes with each other*

*delete the current node*

*return 1*

## combine Function

*repeatedly*

*get value and count from the first Multiset*

*repeatedly (count times),\*[[1]](#footnote-1)*

*insert the value into a temporary Multiset*

*repeatedly*

*get value and count from the second Multiset*

*repeatedly (count times),\**

*insert the value into the same temporary Multiset*

*pass the value of the temporary Multiset to the ‘result’ Multiset*

## subtract Function

*repeatedly*

*get value1 and count1 from the first Multiset*

*repeatedly,*

*get value2 and count2 from the second Multiset*

*if both value1 and value2 are the same*

*find the difference in count (count1 – count2)*

*if the difference is not positive set the difference to 0*

*repeatedly (difference times),\**

*insert the value into a temporary Multiset*

*check if a match was NOT found in the second Multiset*

*repeatedly (count times),\**

*insert value1 into a temporary Multiset*

*pass the value of the temporary Multiset to the ‘result’ Multiset*

# Test Cases:

The following test cases apply when ItemType is defined as a std::string data type.

#include "Multiset.h"

#include <iostream>

#include <cassert>

using namespace std;

void test()

{

Multiset A; // default constructor

///////////////////////////////

//TESTS ON EMPTY MULTISETS

//////////////////////////////

assert(A.empty());

assert(A.size() == 0);

assert(A.erase("nothing") == 0);

assert(A.eraseAll("nothing") == 0);

ItemType value = "hello";

assert(A.get(3, value) == 0); // get function should leave value unchanged

assert(value == "hello"); // check whether value is unchanged by get function

// testing non-member functions with empty Multisets

Multiset B = A;

combine(A, B, A);

assert(A.empty() && B.empty());

subtract(B, A, B);

assert(A.empty() && B.empty());

//////////////////////////////////

//TESTS ON MULTISETS WITH DATA

/////////////////////////////////

//test insert function

assert(A.insert("red"));

assert(A.insert("blue"));

assert(A.insert("green"));

assert(A.insert("green"));

assert(A.insert("yellow"));

assert(A.insert("white"));

assert(A.insert("green"));

assert(A.insert("white"));

assert(A.insert("blue"));

assert(A.insert("blue"));

assert(A.insert("red"));

assert(A.insert("blue"));

assert(!A.empty()); // check if A is not empty

//test size function

assert(A.size() == 12);

//test uniqueSize function

assert(A.uniqueSize() == 5);

//test contains function

assert(A.contains("red"));

assert(A.contains("blue"));

assert(A.contains("green"));

assert(A.contains("yellow"));

assert(A.contains("white"));

assert(!A.contains("purple"));

//test count function

assert(A.count("red") == 2);

assert(A.count("blue") == 4);

assert(A.count("green") == 3);

assert(A.count("yellow") == 1);

assert(A.count("white") == 2);

assert(A.count("purple") == 0);

//test get function

ItemType value1 = "hello";

assert(A.get(100, value1) == 0); // get function should leave value1 unchanged in

// this case

assert(value1 == "hello");

assert(A.get(0, value1) == 2);

assert(value1 == "red");

assert(A.get(1, value1) == 4);

assert(value1 == "blue");

assert(A.get(2, value1) == 3);

assert(value1 == "green");

assert(A.get(3, value1) == 1);

assert(value1 == "yellow");

//test erase function

assert(A.erase("orange") == 0);

assert(A.erase("white") == 1);

assert(A.erase("green") == 1);

assert(A.erase("yellow") == 1);

assert(A.erase("white") == 1);

assert(A.erase("purple") == 0);

assert(A.count("green") == 2);

assert(A.count("yellow") == 0);

assert(A.count("white") == 0);

//test eraseAll function

assert(A.eraseAll("purple") == 0);

assert(A.eraseAll("red") == 2);

assert(A.eraseAll("blue") == 4);

assert(A.eraseAll("green") == 2);

assert(A.eraseAll("orange") == 0);

assert(A.empty());

//test swap function

assert(A.insert("red"));

assert(A.insert("blue")); assert(A.insert("blue"));

assert(A.insert("green")); assert(A.insert("green"));

assert(A.insert("yellow"));

assert(B.insert("purple")); assert(B.insert("purple"));

assert(B.insert("orange")); assert(B.insert("orange"));

assert(B.insert("cyan"));

assert(B.insert("brown"));

A.swap(B);

assert(B.contains("red"));

assert(B.count("blue") == 2);

assert(B.count("green") == 2);

assert(B.count("yellow") == 1);

assert(A.contains("brown"));

assert(A.count("purple") == 2);

assert(A.count("orange") == 2);

assert(A.count("cyan") == 1);

//test copy constructor

Multiset C(A);

Multiset D = B;

assert(C.contains("brown"));

assert(C.count("purple") == 2);

assert(C.count("orange") == 2);

assert(C.count("cyan") == 1);

assert(D.contains("red"));

assert(D.count("blue") == 2);

assert(D.count("green") == 2);

assert(D.count("yellow") == 1);

//test assignment operator

D = C;

D = D;

assert(D.contains("brown"));

assert(D.count("purple") == 2);

assert(D.count("orange") == 2);

assert(D.count("cyan") == 1);

A = C = B;

assert(A.contains("red"));

assert(A.count("blue") == 2);

assert(C.count("green") == 2);

assert(C.count("yellow") == 1);

Multiset E;

assert(E.empty());

C = E;

assert(C.empty());

////////////////////////////////////////////////

//TESTS ON THE COMBINE AND SUBTRACT FUNCTIONS

////////////////////////////////////////////////

Multiset X, Y, Z;

assert(X.insert("red"));

assert(X.insert("blue")); assert(X.insert("blue"));

assert(X.insert("green")); assert(X.insert("green"));

assert(Y.insert("red")); assert(Y.insert("red"));

assert(Y.insert("blue")); assert(Y.insert("blue")); assert(Y.insert("blue"));

assert(Y.insert("yellow"));

assert(Y.insert("green"));

//test combine function

combine(X, Y, Z);

assert(Z.count("red") == 3);

assert(Z.count("blue") == 5);

assert(Z.count("green") == 3);

assert(Z.count("yellow") == 1);

Z = D;

combine(Y, X, Z);

assert(Z.count("red") == 3);

assert(Z.count("blue") == 5);

assert(Z.count("green") == 3);

assert(Z.count("yellow") == 1);

//test subtract function

X = D;

subtract(Z, Y, X);

assert(X.count("red") == 1);

assert(X.count("blue") == 2);

assert(X.count("green") == 2);

assert(X.count("yellow") == 0);

subtract(X, Y, Z);

assert(Z.count("red") == 0);

assert(Z.count("blue") == 0);

assert(Z.count("green") == 1);

assert(Z.count("yellow") == 0);

Multiset Empty;

subtract(Empty, X, Z); // subtracting X from an empty Multiset should result

// an empty Multiset

assert(Z.empty());

//test combining and subtracting a Multiset with itself (aliasing)

combine(Y, Y, Y);

assert(Y.count("red") == 4);

assert(Y.count("blue") == 6);

assert(Y.count("green") == 2);

assert(Y.count("yellow") == 2);

subtract(Y, Y, Y);

assert(Y.empty());

}

int main()

{

test();

cout << "All tests have been passed" << endl;

}

1. *\*By ‘count times’ or ‘difference times’ it is meant that the loop is executed ‘count’ or ‘difference’ no. of times, depending on their integer values.* [↑](#footnote-ref-1)