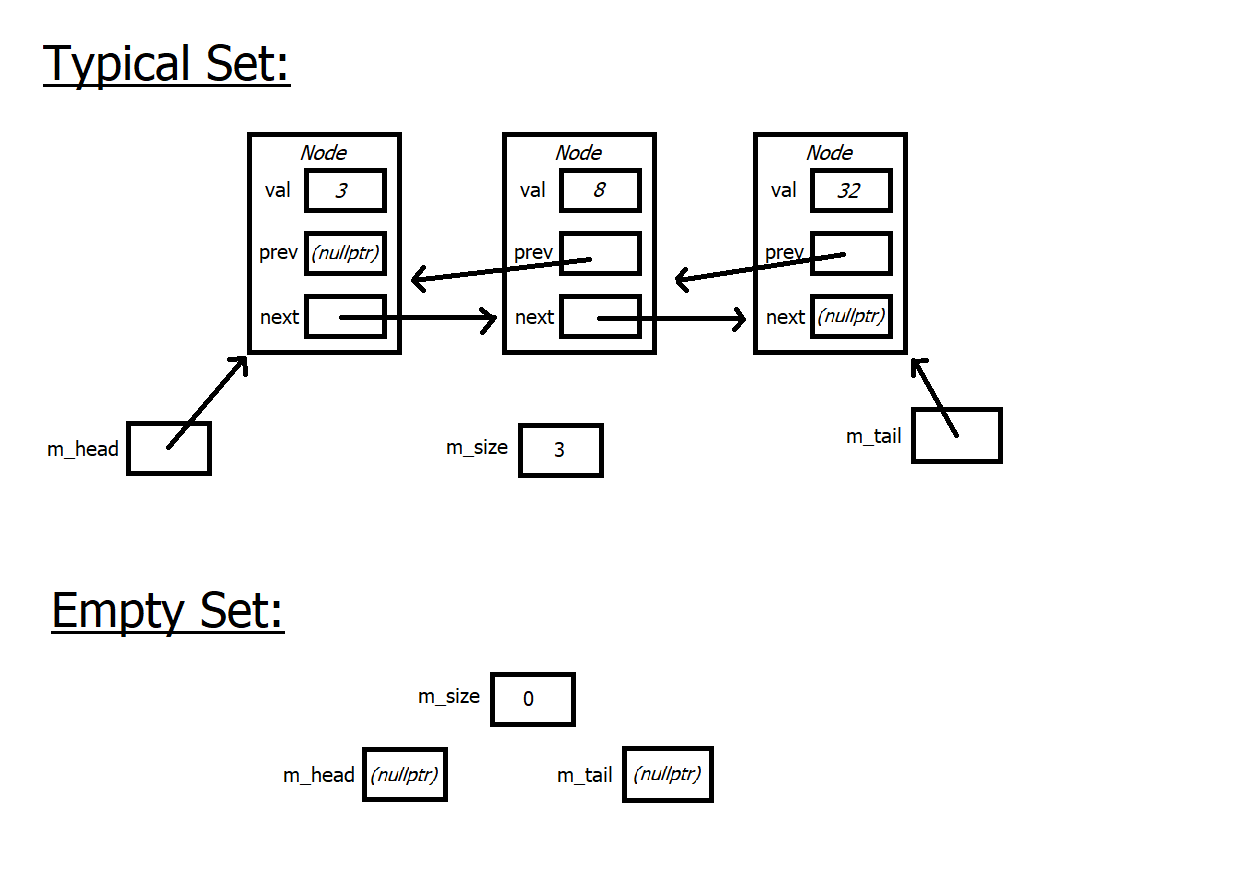
Harrison Cassar

ID: 505114980

Discussion 1C

TA: Hackett, T.R.

Report for CS32 Project 2: Double Trouble

1. Brief description of the design of my doubly-linked list implementation:
   * My doubly-linked list consists of a series of connected "Node" objects that each consist of a "previous" and "next" Node pointer that essentially link themselves to the Node both in front of and behind them. The first Node, as it has no Node before it to connect to, has its "previous" pointer set to the null pointer, and, similarly, the last Node, since it has no next Node to connect to, has its "next" pointer set to the null pointer. Since this is a regular doubly-linked list (not a circular one), the Set class contains both a "head" and "tail" pointer that indicates where in memory the beginning and end of the linked list exists. The Node struct itself (which is located in the private section of the Set class) additionally contains a "value" variable that holds the actual value of that object in the set. The implementation of the linked list ensures that all the elements in the set is sorted. This ensures that the “get()” function of Set has a much easier time determining the correct target position while not sacrificing much difference in performance.
   * Here is a helpful diagram of a typical and empty set:  
     
2. Psuedocode for non-trivial algorithms:
   * subtract():  
     Result Set gets the elements in Set1  
     repeating for Set2’s size:  
      receive value of element of Set2  
      call erase on result with that value
   * unite():  
     Result Set gets the elements in Set1  
     repeating for Set2’s size:  
      receive value of element of Set2  
      call insert on result with that value
   * erase():  
     check for empty list  
     if erasing first item in list:  
      delete Node and update all pointers and size  
      if list was only one-node long, tail should be nullptr  
     traverse linked list:  
      if found Node with the desired value:  
      break early  
     if we did find a matching value:  
      delete Node and update next/prev pointers and size  
      if Node was last in the list, tail is updated
   * insert():  
     check for list already containing the value (no duplicates allowed)  
     traverse linked list:  
      if found the sorted position for the new value :  
      break early  
     if sorted position is at beginning:  
      check if adding to an empty list  
      create new Node and update all pointers and size properly  
     if sorted position is at end:  
      create new Node and update all pointers and size properly  
     otherwise  
      create new Node and update all pointers and size properly
3. List of test cases that would thoroughly test the functions:

//default constructor

Set s1;

//for an empty set

assert(s1.size() == 0); //test size

assert(s1.empty()); //test empty

assert(!s1.erase("roti")); //nothing to remove

//functions of one set

assert(s1.insert("roti")); //test insert

assert(s1.insert("pita")); //test insert again

assert(!s1.insert("roti")); //test insert with already present element

assert(s1.size() == 2); //test size

assert(!s1.empty()); //test empty

assert(s1.contains("pita")); //test contains on a present value

assert(!s1.contains("spaghetti")); //test contains on a non-present value

assert(s1.erase("roti")); //test erase on present value

assert(!s1.erase("spaghetti")); //test erase on present value

assert(s1.insert("shells")); //insert another element for later tests

ItemType x = "laobing"; //ItemType variable for get functionality

assert(s1.get(0, x) && x == "pita"); //testing get on element at the front

assert(s1.get(1, x) && x == "shells"); //testing get on element beyond the front

assert(!s1.get(-1, x)); //testing get on invalid inputted size

assert(!s1.get(3, x)); //testing get on invalid inputted size

//copy constructor

Set s2(s1);

//assignment operator

Set s3;

s3 = s2;

//functions of two sets

assert(s3.insert("apple")); //making s3 different from s2 to test swap

function

assert(s3.get(0, x) && x == "apple"); //testing implementation: seeing

that the first element is apple as expected

int sizeFirst = s3.size(); //record before-swap size of s3

int sizeSecond = s2.size(); //record before-swap size of s2

s3.swap(s2); //swap s3 and s2's contents

assert(!s3.contains("apple")); //test for lack of presence of s3's unique

item before swap

assert(s2.contains("apple")); //test for presence of s3's unique item

before swap (now s2's unique item)

assert(s2.get(0, x) && x == "apple"); //test to see if apple is still in the

first spot as expected (i.e. order is kept the same)

assert(s3.size() == sizeSecond); //test successful size swapping

assert(s2.size() == sizeFirst); //test successful size swapping

Set result; //construct result set for later testing

unite(s2, s3, result); //perform unite call on two sets for later testing

for (int i = 0; i < 2; i++) //making sure values in either are accurate

assert(result.get(i, x) && (x == "pita" || x == "shells" || x == "apple"));

assert(!result.get(3, x)); //making sure only 3 elements were in either of the

two sets (testing unite)

subtract(s2, s3, result); //call subtract function on two sets for later testing

assert(result.get(0, x) && x == "apple"); //testing for inclusion of unique

element to s2 in result after calling the subtract function

assert(!result.get(1, x)); //making sure only 1 element is unique to s2

subtract(s3, s2, result); //call subtract function in the opposite way (s3

includes no unique elements)

assert(result.empty()); //testing for no elements in result, as expected