CS146Section8Lab5 Name: Michael Huang Due date: March 9th, 2016

Ex-2.7a

sum = 0;

for( i = 0; i < n; i++ )

sum++;

This is O(n) because the number of loops are dependent on n, which results in n loops.

sum = 0;

for( i = 0; i < n; i++ )

for( j = 0; j < n; j++ )

sum++;

This is O(n2) because there is a nested loop with n iterations inside another loop with n iterations, which is a total of n \* n loops, which is O(n2).

sum = 0;

for( i = 0; i < n; i++ )

for( j = 0; j < n \* n; j++ )

sum++;

This is O(n3) because there is a nested loop with n \* n iterations inside another loop with n iterations, which is a total of n \* (n \* n) loops, which is O(n3).

sum = 0;

for( i = 0; i < n; i++ )

for( j = 0; j < i; j++ )

sum++;

This is O(n2) because there is a nested loop with j iterations, but j is dependent on i, which is dependent on n. The outer loop will have n iterations, and the inner will have j iterations, which roughly i iterations which is equal to n iterations, for a total of n \* n iterations.

sum = 0;

for( i = 0; i < n; i++ )

for( j = 0; j < i \* i; j++ )

for( k = 0; k < j; k++ )

sum++;

This O(n5), because the outer loop has n iterations, the first inner loop has i \* i iterations, which is roughly the same as n \* n iterations. The innermost loop has k iterations, which is roughly the same as j iterations, where j is roughly n \* n for a total of n \* (n \* n \* (n \* n)) iterations, which is O(n5).

sum = 0;

for( i = 1; i < n; i++ )

for( j = 1; j < i \* i; j++ )

if( j % i == 0 )

for( k = 0; k < j; k++ )

sum++;

This is O(n4). The outer two loops combine for a total of O(n3). However, the if statement of j % i == 0 is only true if j is a multiple of i (j = i, 2i…i2), and is only true i times. Thus, since the inner n \* n loop occurs only i times, the amount of operations balances and averages out to an effective n amount of iterations per if statement. Thus, the algorithm is

n \* (n \* n \* (n)) = O(n4).

Ex-2.12:

An algorithm takes 0.5ms for input size 100. How large a problem can be solved in 1 min if the running time is (1 m = 60000ms):

1. linear

c \* g(n) = t. c \* 100 = 0.5ms. c = 0.5/100.

c \* n = 60000ms. n = 60000 \* 100 / 0.5.

n = 12,000,000 inputs.

1. O(N logN)

c \* g(n) = t. c \* 100\*log(100) = 0.5ms. c = 0.5/200.

c \* nlog(n) = 60000ms. nlog(n) = 60000 \* 200 / 0.5.

n = 3,656,807 (used a graphing calculator, too difficult to solve otherwise).

1. quadratic

c \* g(n) = t. c \* 1002 = 0.5ms. c = 0.5/1002.

c \* n2 = 60000ms. n2 = 60000 \* 1002 / 0.5. n = (60000 \* 1002 / 0.5)1/2.

n = 34641.0161514, or ~ 34641 inputs.

1. cubic

c \* g(n) = t. c \* 1003 = 0.5ms. c = 0.5/1003.

c \* n3 = 60000ms. n3 = 60000 \* 1003 / 0.5. n = (60000 \* 1003 / 0.5)1/3.

n = 4932.42414866 or ~4932 inputs.

Ex-2.20:

1. Write a program to determine if a positive integer N is prime.

Code:

|  |
| --- |
| public class Prime {  public static boolean isPrime(int n) {  if (n == 2) {  return true;  } else if (n % 2 == 0) {  return false;  } else {  for (int i = 2; i <= Math.sqrt(n); i++) {  if (n % i == 0) {  return false;  }  }  }  return true;  }    public static void main(String[] args) {  System.out.println(isPrime(71));  }  } |

1. What is the run time of your program?

My run time should be O(√n), because I used the Sieve of Eratosthenes, which checks the number if the number is divisible by all numbers until the square root of N.

1. Let B equal the number of bits in the binary representation of N. What is the value of B?

N = 2B.

log2B = log2N.

log(N) = B.

B = O(log(n)).

1. In terms of B, what is the worst-case running time of your program?

O(sqrt(N)) = O(sqrt(2B)) = O(2B/2)

1. Compare the running times to determine if a 20-bit number and a 40-bit number are prime, assume it takes T sec to determine if a 20-bit number is a prime, in terms of T how long does it take for a 40-bit number.

20 bits: 220/2 = 210 = T.

40 bits: 240/2 = 220 = T2.

Ex-2.31: Suppose that line 15 in the binary search routine had the statement low = mid instead of low = mid + 1. Would the routine still work?

It would not work for all cases, for example if the array only had 1 and 2, and you are searching for 3. Low is 0, high is 1, and mid would be 0 since an int of ½ is 0. a[0] is 1, which is less than 3, so low is set to mid, which results in an infinite loop because low is always 0.

Ex-3.8:

public static void removeFirstHalf(List<?> lst) {

int theSize = lst.size( ) / 2;

for( int i = 0; i < theSize; i++ )

lst.remove( 0 );

}

1. Why is theSize saved prior to entering the for loop?

Once elements are removed, the list’s size changes, and this would affect the loop.

1. What is the running time of removeFirstHalf if lst is an ArrayList?

It is O(n2). ArrayList’s remove method is O(n), since it must shift every element forward after removing in the beginning. We are removing n/2 elements, so the complexity is

n/2 \* n which is n2/2, which is O(n2).

1. What is the running time of removeFirstHalf if lst is a LinkedList?

It’s O(n). LinkedList’s remove method is O(1), since the list is not traversed and only updates the next references, the amount of work per remove is constant. Therefore, the complexity is n/2 \* 1 = n/2, which is O(n).

1. Does using an iterator make removeHalf faster for either type of List?

No, because after removing in an ArrayList, all of the elements still need to be shifted, and would still be O(n) per remove. Since the LinkedList’s remove is O(1), there’s no point in using an iterator to speed it up.

Ex-3.10: Provide an implementation of removeAll to MyLinkedList.

Code (it’s pretty long):

|  |
| --- |
| public class MyLinkedList<T> implements Iterable<T> {  public MyLinkedList() {  doClear();  }  private void clear() {  doClear();  }  public void doClear() {  beginMarker = new Node<>(null, null, null);  endMarker = new Node<>(null, beginMarker, null);  beginMarker.next = endMarker;  theSize = 0;  modCount++;  }  public int size() {  return theSize;  }  public boolean isEmpty() {  return size() == 0;  }  public boolean add(T x) {  add(size(), x);  return true;  }  public void add(int idx, T x) {  addBefore(getNode(idx, 0, size()), x);  }  private void addBefore(Node<T> p, T x) {  Node<T> newNode = new Node<>(x, p.prev, p);  newNode.prev.next = newNode;  p.prev = newNode;  theSize++;  modCount++;  }  public T get(int idx) {  return getNode(idx).data;  }  public T set(int idx, T newVal) {  Node<T> p = getNode(idx);  T oldVal = p.data;  p.data = newVal;  return oldVal;  }  private Node<T> getNode(int idx) {  return getNode(idx, 0, size() - 1);  }  private Node<T> getNode(int idx, int lower, int upper) {  Node<T> p;  if (idx < lower || idx > upper)  throw new IndexOutOfBoundsException("getNode index: " + idx + "; size: " + size());  if (idx < size() / 2) {  p = beginMarker.next;  for (int i = 0; i < idx; i++)  p = p.next;  } else {  p = endMarker;  for (int i = size(); i > idx; i--)  p = p.prev;  }  return p;  }  public T remove(int idx) {  return remove(getNode(idx));  }  private T remove(Node<T> p) {  p.next.prev = p.prev;  p.prev.next = p.next;  theSize--;  modCount++;  return p.data;  }  public String toString() {  StringBuilder sb = new StringBuilder("[ ");  for (T x : this)  sb.append(x + " ");  sb.append("]");  return new String(sb);  }  public java.util.Iterator<T> iterator() {  return new LinkedListIterator();  }  private class LinkedListIterator implements java.util.Iterator<T> {  private Node<T> current = beginMarker.next;  private int expectedModCount = modCount;  private boolean okToRemove = false;  public boolean hasNext() {  return current != endMarker;  }  public T next() {  if (modCount != expectedModCount)  throw new java.util.ConcurrentModificationException();  if (!hasNext())  throw new java.util.NoSuchElementException();  T nextItem = current.data;  current = current.next;  okToRemove = true;  return nextItem;  }  public void remove() {  if (modCount != expectedModCount)  throw new java.util.ConcurrentModificationException();  if (!okToRemove)  throw new IllegalStateException();  MyLinkedList.this.remove(current.prev);  expectedModCount++;  okToRemove = false;  }  }  private static class Node<T> {  public Node(T d, Node<T> p, Node<T> n) {  data = d;  prev = p;  next = n;  }  public T data;  public Node<T> prev;  public Node<T> next;  }  private int theSize;  private int modCount = 0;  private Node<T> beginMarker;  private Node<T> endMarker;  public void removeAll(Iterable<? extends T> items) {  for (T a: items) {  java.util.Iterator<T> iter = new LinkedListIterator();  T current = iter.next();  while(!current.equals(a)) {  current = iter.next();  }  if (current.equals(a)) {  iter.remove();  }  }  }  public static void main(String[] args) {  MyLinkedList<Integer> lst = new MyLinkedList<>();  MyLinkedList<Integer> items = new MyLinkedList<>();  for (int i = 0; i < 10; i++)  lst.add(i);  System.out.println("List: " + lst);  for (int i = 0; i < 5; i++)  items.add(i);  System.out.println("Items: " + items);  lst.removeAll(items);  System.out.println("After removing: " + lst);  }  } |

Console output:



The removeAll method should be O(n2). For every item to be removed, the iterator must traverse the list until it matches the removed item (O(n)), and the iterator removes it (O(1)). This goes on for n amount of items, so the complexity is O(n2).

Ex-3.29: Write an algorithm for printing a singly linked list in reverse, using only constant extra space. This instruction implies that you cannot use recursion, but you may assume that your algorithm is a list member function.

Reverse all the nodes, print them, then reverse it again to restore it to the correct order.

Code:

|  |
| --- |
| import java.util.Stack;  public class SinglyLinkedList<T> {  private Node<T> root;    @SuppressWarnings("hiding")  class Node<T> {  T data;  Node<T> next;    public Node (T object) {  data = object;  }  }    public SinglyLinkedList() {  root = null;  }    public void add(T object) {  Node<T> adding = new Node<T>(object);  if (root == null) {  root = adding;  } else {  Node<T> current = root;  while (current.next != null) {  current = current.next;  }  current.next = adding;  }  }    public String toString() {  String toString = "[";  Node<T> current = root;  while(current != null) {  toString += current.data + " ";  current = current.next;  }  return toString.substring(0, toString.length() - 1) + "]";  }    public void printReverse() {  reverse();  System.out.println(toString());  reverse();  }    public void reverse() {  Node<T> start = root;  Node<T> next = start.next;  while (start != null) {  if (next == null) {  root.next = next;  root = start;  break;  } else {  Node<T> nextRef = next.next;  next.next = start;  start = next;  next = nextRef;  }  }  }    public static void main(String[] args) {  SinglyLinkedList<Integer> a = new SinglyLinkedList<Integer>();  for(int i = 0; i < 10; i++) {  a.add(i);  }  System.out.println(a);  a.printReverse();  }  } |

Console output:



Ex-4.2: For node B in Figure 4.70:

1. Name the parent node: A.
2. List the children: D, E.
3. List the siblings: C.
4. Compute the depth (from root to B, where root is 0): 1.
5. Compute the height (longest path from a leaf to B): 3.

Ex-4.26: Write the methods to perform the double rotation without the inefficiency of doing two single rotations.

Add these two methods to the source code.

|  |
| --- |
| private AvlNode<AnyType> DoubleRotateLeft(AvlNode<AnyType> node) {  AvlNode<AnyType> one = node.left;  AvlNode<AnyType> two = one.left;;  one.right = two.left;  node.left = two.right;  two.left = one;  two.right = node;  one.height = Math.max(height(one.left), height(one.right)) + 1;  node.height = Math.max(height(node.left), height(node.right)) + 1;  two.height = Math.max(one.height, node.height) + 1;  return two;  }    private AvlNode<AnyType> DoubleRotateRight(AvlNode<AnyType> node) {  AvlNode<AnyType> one = node.right;  AvlNode<AnyType> two = one.left;  node.right = two.left;  one.left = two.right;  two.left = node;  two.right = one;  node.height = Math.max(height(node.left), height(node.right)) + 1;  one.height = Math.max(height(one.left), height(one.right)) + 1;  two.height = Math.max(node.height, one.height) + 1;  return two;  } |