1 Submission Instructions

- In the submission instructions, we mention your ASURITE ID several times. Your ASURITE ID is the user name you use to log in to ASU computer systems such as MyASU and Canvas, e.g., the instructor's ASURITE ID is kburger2. It is not the same as your ASU ID number printed on your SunCard (that identifier is a 10-digit integer).
- Note that for each homework assignment, only some of the exercises will be graded. A list of the exercises that will be graded can be found in the Module 7: HW4 Due submission page in Canvas.
- There are two primary reasons we grade only some of the exercises: first, this course has a large enrollment so there are many assignments to be graded. Second, in the accelerated time frame for online courses, we want to return graded homework exercises as quickly as we can.
- Please understand that it is time-consuming to manually grade assignments, particularly programming exercises, so it may take as long as a week to return scores. We will always try to return them sooner.
- Not grading all of the exercises does not mean that the ungraded exercises are unimportant or that you will not be tested on the concepts in the exercise. They are equally important as the graded exercises and the concepts you learn may be on the exams. Consequently, we **strongly recommend** that you complete *all* of the exercises.
- Some of your solutions must be submitted in a PDF document. To start the assignment, create a word processing document named *asuriteid*-h04.*ext*, where *asurite* is your ASURITE ID and *ext* is DOCX if you are using Word or ODT if you are using Libre Writer or Open Office Writer. For example, the instructor's file would be named kburger2-h03.odt because he uses Libre Writer.
- Please type your name, ASURITE ID, and the homework number (HW4) near the top of the document.
- For the **short-answer and description** exercises (e.g., see Exs. 3.2, 3.5, 5.2, etc), please neatly type your solution in the document. Clearly number each exercise so there is no confusion for the grader.
- When you are done with the document, please convert it **Adobe PDF format** and name the file *asuriteid*-h04.pdf, e.g., the instructors file would be named kburger2-h04.pdf.
- Next, create an empty folder named *asuriteid*-h04 and copy *asuriteid*-h04.pdf to that folder.
- For the linked list exercises 4.1 and 4.3, copy your modified *DList.java* source code file to the *asurite*-h04 folder. For the binary tree tree exercise 5.3 copy your modified *BinaryTree.java* source code folder to the *asuriteid*-h04 folder. (Note: Java source code files are the files with a *.java* file name extension; do not copy the *.class* files or any data files as we do not need those.)
- Then compress the *asuriteid*-h04 folder creating a **zip archive** file named *asuriteid*-h04.zip. Upload *asuriteid*-h04.zip to Canvas using the submission link on the Module 7: HW4 Due submission page before the assignment deadline.
- Please see the Course Summary section on the Syllabus page in Canvas for the deadline. The deadline can also be found on the CSE205 course calendar, look for the event named Module 7: HW4 Due.
- Consult the Syllabus for the late and academic integrity policies.

2 Learning Objectives

1. To use the merge sort and quick sort sorting algorithms to sort a list of elements.
2. To analyze the time complexity of the merge sort and quick sort sorting algorithms.
3. To implement linked list, stack, queue, and tree data structures.
4. To analyze the time complexity of linked list, stack, queue, and tree operations.
5. To implement a binary search tree (BST) data structure.
6. To analyze the time complexity of BST operations.

3 Sorting

3.1 Learning Objectives for Exs. 3.1-3.4: To demonstrate that the student understands how the merge sort algorithm sorts a list. In the video lecture *Sorting Algorithms: Section 7 : Merge Sort Example* we traced how merge sort would recursively sort *list* = { 4, 2, 7, 3, 5, 13, 11, 8, 6, 2 }. For this exercise, I would like you to draw a similar diagram showing how merge sort would sort *list* = { 5, 3, 1, 6, 2, 4 }. Scan this diagram and insert it into your final PDF.

3.2 In *Sorting Algorithms : Section 9 : Merge Sort Pseudocode* we discussed the high-level pseudocode for the merge sort algorithm. I wrote three methods: *recursiveMergeSort*(), *merge*(), and *copyRest*(). Continuing the previous exercise, how many times will *recursiveMergeSort*() be called when sorting the list of Exercise 3.1. Include the original non-recursive call to *recursiveMergeSort*() and all of the subsequent recursive calls in the answer.

3.3 Continuing Exs. 3.1-3.2, how many times will *merge*() be called?

3.4 Continuing Exs. 3.1-3.2, during the final call to *merge*()—when we are merging $list_L$ and $list_R$ to form the final sorted *list*—*copyRest*() will be called. **(a)** When *copyRest*() executes, which list will be *srcList* ($list_L$ or $list_R$)? **(b)** What will be the value of *srcIndex*? **(c)** Which list will be *dstList*? **(d)** What will be the value of *dstIndex*?

3.5 Learning Objectives for Exs. 3.5-3.7: To demonstrate that the student understands how the quick sort algorithm sorts a list. Consider *list* = { 5, 4, 2, 9, 1, 7, 3, 8, 6 } and the quick sort algorithm. Assume we always choose the first list item in the list being partitioned as the pivot, as you did in Project 3. Trace the partition method showing how *list* is partitioned into $list_L$ and $list_R$. To get you started, here is the format of what I am looking for in your solution:

```
list = { 5, 4, 2, 9, 1, 7, 3, 8, 6 }, pivot = 5, leftIndex = -1, rightIndex = 9
While loop pass 1:
    leftIndex ends up at 0, rightIndex ends up at 6
    leftIndex < rightIndex so swap list[0] and list[6]: list = { 3, 4, 2, 9, 1, 7, 5, 8, 6 }
While loop pass 2:
    ...
While loop pass 3:
    ...
While loop terminates because leftIndex = ??? is >= rightIndex = ???
partition() returns ??? so list_L = { ??? }, list_R = { ??? },
```

3.6 Choosing the first list element as the pivot does not always lead to a good partitioning (ideally, the sizes of $list_L$ and $list_R$ will be approximately equal). Suppose *list* = { 1, 2, 3, 4, 5, 6, 7, 8 } and we again select the first element as the pivot. What would $list_L$ and $list_R$ be? For this exercise, you do not have to write a detailed trace of the partition method as you did for the previous exercise; simply list what index would be returned by *partition*() and the contents of $list_L$, and $list_R$.

3.7 Starting with $list_R$ from the previous exercise, repeat Exercise 3.6 on $list_R$. Explain what pattern is going to hold if we continue to partition each successive $list_R$.

4 Linked Lists

Important Note: There are two version of *DList.java* in the *Week 6 Source Code* and *Week 7 Source Code* zip archives. The two versions are mostly identical with this difference: the *DList* class in the *Week 6 Source Code* zip archive is designed to store elements only of the *java.lang.Integer* data type, e.g., we could not use this class to store *Circle*s or *Banana*s. The *DList* class in the *Week 7 Source Code* zip archive is a **generic class**[1] which means in English that we can use that *DList* class to store elements of any data type (of course, all of the elements still have to be of the same type or belong to the same inheritance hierarchy). For example, we may wish to create a *DList* of *Shape*s by writing `DList<Shape> list = new DList<>();` Then, we could add any *Shape* subclass object to *list*, such as a *Rectangle* or *Circle* object. For Homework Assignment 4, use the *DList* class from the *Week 6* archive that stores *Integers*. (The generic *DList* class from the *Week 7* archive is provided because you will use it in Programming Project 4.)

1 https://docs.oracle.com/javase/tutorial/java/generics/types.html. As an example, *ArrayList* is a generic class, e.g., we can create an ArrayList of *Integer*s by writing ArrayList<Integer> list = new ArrayList<>(); and we can create an ArrayList of *Banana*s by writing ArrayList<Banana> list = new ArrayList<>();

4.1 Learning Objectives for Exs. 4.1-4.6: The student shall be able to write a linked list class storing elements of the *Integer* data type. The student shall be able to explain the time complexities of the linked list class methods. **(Include your modified *DList.java* source code file in your homework solution zip archive)** Using whatever Java IDE you prefer, create a project and add *DList.java* and *DListTest.java* to it (these files are provided in the *Week 6 Source* zip archive). Modify *DList* to implement a method that removes all occurrences of a specific integer from the list. Here is the pseudocode:

```
Method removeAll(pData : Integer) Returns Nothing
    Declare index variable i and initialize i to 0
    While i < the size of this Dlist Do
        If get(i) equals pData Then
            remove(i)
        Else
            Increment i
        End If
    End While
End Method removeAll
```

Next, modify *DListTest*() to add a new test case 21 which tests that *removeAll*() works correctly. For example, in *testCase21*() declare and create a new *DList* object named *list*. Let x be the integer we are going to remove all of from *list*, e.g., $x = 1$. Call *list.append*() several times to append random integers to *list*; make sure to append several x's to *list*. After initializing *list*, call *list.removeAll*(x) to remove all occurrences of x. Then call *passOrFail*() as in the other test cases to verify that there are no x's in *list*.

4.2 Let n be the size of a *DList*, i.e., the number of elements. The *remove*(*index*) method is $O(n)$. The *get*(i) method is $O(n)$ because in the worst case, we have to traverse almost the entire list to locate the element at index i. Why? *get*(i) calls *getNodeAt*(i) to obtain a reference to the node at index i so the time complexity of *get*(i) is proportional to the time complexity of *getNodeAt*(i).

Now what is the time complexity of *getNodeAt*(i)? The key operations in *getNodeAt*() are the assignments of *getHead().getNext*() before the loop starts and the assignment of *node.getNext*() to *node* during each iteration of the for loop. For *getNodeAt*(0) and *getNodeAt*(n - 1) the key operations will never be performed so the best case time complexity of *getNodeAt*() is $O(1)$. In the worst case, i would be n - 2 and the key operations would be performed 1 + n - 2 = n - 1 times so the worst case time complexity is $O(n)$.

The key operations of *removeAll*() are the key operations of *getNodeAt*(). For this exercise, define a function $f(n)$ which specifies the maximum number of times the key operations will occur as a function of the list size n. Then specify what the worst case time complexity of *removeAll*() is in big O notation (you do not have to provide a formal Big Oproof; just provide a decent explanation).

4.3 **(Remember to include *DList.java* in your solution zip archive)** Here is the Java implementation of three useful methods (which are not currently in *Dlist*).

```
/**
 * Removes the head node from this DList. It would be inadvisable to call this method on an
 * empty list because we do not check for that condition. Returns the data stored in the head
 * node. */
protected Integer removeHead() {
    Integer data = getHead().getData();  // Save the data at the head node.
    if (getSize() == 1) {                // Are we removing the head from a list of size 1?
        setHead(null);                   // Yes, so set the head and tail references to be null.
        setTail(null);                   // This creates an empty list.
```

```
        } else {
            // Change the prev reference of the node following head to null as that node will become
            // the new head.
            getHead().getNext().setPrev(null);
            // Change the head reference to the node following the current head.
            setHead(getHead().getNext());
        }
        setSize(getSize() - 1);  // Decrement the size of the list.
        return data;             // Return the data that was at the head.
    }

    /**
     * Removes an interior node pNode from this DList. It would be inadvisable to call this method
     * when pNode is null because we do not check for that condition. Returns the data stored in
     * pNode.
     */
    protected Integer removeInterior(Node pNode) {
        Integer data = pNode.getData();           // Save the data at pNode.
        pNode.getPrev().setNext(pNode.getNext()); // These two statements unlink pNode from
        pNode.getNext().setPrev(pNode.getPrev()); // the list.
        setSize(getSize() - 1);                   // Decrement the size of the list.
        return data;                              // Return the data that was at pNode.
    }

    /**
     * Removes the tail node from this DList. It would be inadvisable to call this method on an
     * empty list because we do not check for that condition. Returns the data stored in the tail
     * node.
     */
    protected Integer removeTail() {
        Integer data = getTail().getData();  // Save the data at the tail node.
        if (getSize() == 1) {                // Are we removing the tail from a list of size 1?
            setHead(null);                   // Yes, so set the head and tail references to be null.
            setTail(null);                   // This creates an empty list.
        } else {
            // Change the next reference of the node preceding tail to null as that node will become
            // the new tail.
            getTail().getPrev().setNext(null);
            // Change the tail reference to the node preceding the current tail.
            setTail(getTail().getPrev());
        }
        setSize(getSize() - 1);  // Decrement the size of the list.
        return data;             // Return the data that was at the tail.
    }
```

Add these three methods to *DList*. Next, using these three methods, rewrite the provided *remove*(*index*) method to make the code in that method simpler and more readable (my new and improved *remove*() method is a half dozen lines of code). Be sure to run the test cases in *DListTest* to ensure that your new *remove*() method works correctly—those are test cases 13–18. Also, make sure *remove*() throws an *IndexOutOfBoundsException* if *pIndex* is less than 0 or greater than or equal to *getSize*(). Hint: if *pIndex* is 0 then call *removeHead*() to remove the head node, which is at index 0. If *pIndex* is the index of the tail node, then call *removeTail*(). Otherwise, we are removing an interior node. Get the node at index *pIndex* by calling *getNodeAt*() and then pass that node to *removeInterior*().

4.4 Here is the Java implementation of three useful methods which are not currently in *Dlist*.

```
/**
 * Adds a new node storing pData to be the new head of this DList.
 */
protected void addHead(Integer pData) {
    Node newNode = new Node(pData, null, getHead());
    if (getHead() == null) {
        setTail(newNode);
    } else {
        getHead().setPrev(newNode);
    }

    setHead(newNode);
    setSize(getSize() + 1);
}

/**
 * Adds a new node storing pData to be the predecessor to pNode pNode (pNode may be head or tail).
 */
protected void addInterior(Integer pData, Node pNode) {
    if (pNode == getHead()) {
        addHead(pData);
    } else {
        Node newNode = new Node(pData, pNode.getPrev(), pNode);
        pNode.getPrev().setNext(newNode);
        pNode.setPrev(newNode);
        setSize(getSize() + 1);
    }
}

/**
 * Adds a new node storing pData to be the new tail of this DList.
 */
protected void addTail(Integer pData) {
    Node newNode = new Node(pData, getTail(), null);
    if (getTail() == null) {
        setHead(newNode);
    } else {
        getTail().setNext(newNode);
    }
    setTail(newNode);
    setSize(getSize() + 1);
}
```

Using these three methods, rewrite *add*(*index, data*) to make the code in that method simpler and more readable (my new and improved *add*() method is a half dozen lines of code). Be sure to run the test cases in *DListTest* to ensure that your new *add*() method works correctly. Also, make sure *add*() still throws an *IndexOutOfBoundsException* if *pIndex* is less than 0 or greater than *getSize*().

4.5 If you determined the correct answer to Exercise 4.2, you may wonder if the pseudocode of Exercise 4.1 is really the best way to remove all nodes containing a specific value from the list. For this exercise, implement a new remove all method named *DList.removeAllBetter*() which provide a more efficient implementation of the remove all function. Copy *DListTest.testCase21*()—which tests the *removeAll*() method—to *DListTestCase.testCase22*(). Then modify *testCase22*() so it calls *removeAllBetter*() to verify that your new implementation works correctly. Here is the procedure for the faster remove all method:

```
Method removeAllBetter(pData : Integer) Returns Nothing
    -- Declaring a Node pointer and using the while loop below is the standard way to traverse
    -- all of the elements of a link list.
    Declare a Node named node and initialize it to refer to the head node
    While node is not null Do
        Declare a Node named nextNode and initialize it to refer to the next Node following node
        -- Check to see if the data at node needs to be deleted
        If the data at node equals pData Then  -- We need to delete node
            -- Check to see if we are removing the head node
            If node is the head node Then
                remove the head node
                nextNode ← the new head node
            -- Check to see if we are removing the tail node
            Else If node is the tail node Then
                remove the tail node
                nextNode ← null
            -- Otherwise, we are removing an interior node
            Else
                Call removeInterior() to remove node
            End If
        End If
        node ← nextNode  -- Update node to refer to the next node in the list and continue looping
    End While
End Method removeAll
```

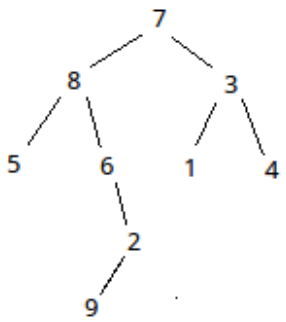
4.6 Give an informal proof of the worst case time complexity of your new *removeAllBetter*() method of Exercise 4.5. Basically, what I am looking for is the description of the key operation, a function $f(n)$ which counts the maximum number of times the key operation is performed as a function of the list size n, and then state that $f(n)$ is $O(g(n))$ for some function $g(n)$.

5 Binary Trees and BSTs

Important Note: The *BinaryTree*<*T*> and *BinaryTreeVisitor*<*T*> classes in the *Week 7 Source Code* zip archive are **generic classes** which means in English that we can use the *BinaryTree*<*T*> class to store elements of any data type (of course, all of the elements still have to be of the same type or in the same inheritance hierarchy). For example, we may wish to create a *BinaryTree*<*T*> of *Student*s by writing `BinaryTree<Student> tree = new BinaryTree<>();`

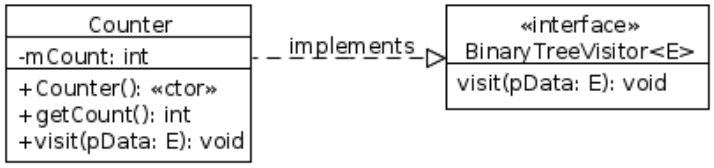
Learning Objectives for Exs. 5.1-5.5: The student shall be able to write a generic binary tree class and shall be able to explain the characteristics of binary trees.

5.1 Consider this binary tree. **(a)** List the descendants of node 8. **(b)** List the ancestors of 1. **(c)** List the leaf nodes. **(d)** List the internal nodes. **(e)** What are the levels of nodes 3, 1, and 9? **(f)** What is the height of the tree? **(g)** What is the height of the subtree rooted at 6? **(h)** Is this a full binary tree? Explain. **(i)** Explain how we could transform this tree to be a complete binary tree, i.e., state which nodes we would move and where we would move them to.



5.2 **(a)** List the nodes in the order they would be visited during a level order traversal. **(b)** List the nodes in the order they would be visited during an inorder traversal. **(c)** List the nodes in the order they would be visited during a preorder traversal. **(d)** List the nodes in the order they would be visited during a postorder traversal.

5.3 (Include your modified *BinaryTree.java* source code file in your homework solution zip archive) Add a method `int getSize()` to *BinaryTree* that returns the size of the binary tree, where the size of the tree is defined to be the number of nodes. Here is how I want you to implement this method. Write a **local class** (see *Module 3 : Objects and Classes II : Section 2*) in method *public int getSize*() named *Counter* which implements the *BinaryTreeVisitor* <*E*> interface:



The *Counter* constructor initializes the counter variable *mCount* to 0, and the *Counter.visit*() method will be called each a time a node is visited either from *BinaryTree.traverse*(int, Node<E>, BinaryTreeVisitor<E>) or *BinaryTree.traverseLevelOrder*(Node<E>, BinaryTreeVisitor<E>) depending on whether you performed a level order traversal or one of the three other types of traversals as described below. Within *Counter.visit*(), we simply increment *mCount* to count the node as visited.

Once the local class is completed, we can count the nodes in the tree by performing a traversal (it does not matter which type of traversal we perform because each node will be visited during the traversal; the order in which we visit them does not matter for this application). To perform the traversal write:

```
public int getSize() {
    // Implement local class named Counter here
    ???

    // Create a Counter object to count the nodes
    Counter counter = new Counter();

    // I have chosen to perform a level order traversal, but try the other ones to verify they work
    // the same way as well, e.g., traverse(PRE_ORDER, counter).
    traverse(LEVEL_ORDER, counter);

    // After we finish the traverse, Counter.mCount is equal to the number of nodes visited, which
    // is the size of the tree.
    return counter.getCount();
}
```

5.4 If *n* is the size of the tree, what is the worst case time complexity of *getSize*() in Big O notation?

5.5 The *BinaryTree.Iterator*<*E*> class uses a stack (the *mStack* instance variable) to store references to parent nodes as the iterator moves left and right downward in the tree. The stack of parent nodes is necessary because the *moveUp*() method needs to change the iterator's current node reference to be the parent node of the current node when *moveUp*() is called.

However, storing the parent nodes on a stack is not the only way to implement *moveUp*(). Furthermore, in certain tree methods (that are not currently implemented), it would be very helpful to have a reference to the parent node. For this exercise we will modify the *BinaryTree*<*E*>, *BinaryTree.Node*<*E*>, and *BinaryTree.Iterator*<*E*> classes so each node will store a reference to its parent (for the root node, the parent reference will be null).

First, modify the *BinaryTree.Node*<*E*> class to add a new instance variable `Node<E> mParent` which will always contain a reference to the parent node of a node (for the root node, *mParent* will be null). Add accessor and mutator methods for *mParent* to the *Node* class. Modify the *Node* constructors thusly:

```
public Node() {
    this(null, null);
}
```

```
public Node(E pData, Node<E> pParent) {
    this(pData, null, null, pParent);
}
public Node(E pData, Node<E> pLeft, Node<E> pRight, Node<E> pParent) {
    setData(pData);
    setLeft(pLeft);
    setRight(pRight);
    setParent(pParent);
}
```

Second, modify the *BinaryTree.Iterator<E>* class to eliminate the *mStack* instance variable and the *getStack*() and *setStack*() accessor and mutator methods. Then modify these *Iterator* methods:

```
public Iterator(BinaryTree<E> pTree) {
    setTree(pTree);
    setCurrent(getTree().getRoot());
    setStack(new Stack<Node<E>>());  // Delete this line
}
public void addLeft(E pData) throws EmptyTreeException {
    if (getTree().isEmpty()) throw new EmptyTreeException();
    pruneLeft();
    getCurrent().setLeft(new Node<E>(pData, getCurrent()));
}
public void addRight(E pData) throws EmptyTreeException {
    if (getTree().isEmpty()) throw new EmptyTreeException();
    pruneRight();
    getCurrent().setRight(new Node<E>(pData, getCurrent()));
}
public void moveLeft() {
    if (getCurrent().hasLeft()) {
        getStack().push(getCurrent());
        setCurrent(getCurrent().getLeft());
    }
}
public void moveRight() {
    if (getCurrent().hasRight()) {
        getStack().push(getCurrent());
        setCurrent(getCurrent().getRight());
    }
}
public void moveToRoot() {
    getStack().clear();
    setCurrent(getTree().getRoot());
}
public void moveUp() {
    setCurrent(getStack().pop());
    if (getCurrent().getParent() != null) {
        setCurrent(getCurrent().getParent());
    }
}
```

Finally, modify this *BinaryTree* constructor:

```
public BinaryTree(E pData, BinaryTree<E> pLeft, BinaryTree<E> pRight) {
    Node<E> leftChild = pLeft == null ? null : pLeft.getRoot();
    Node<E> rightChild = pRight == null ? null : pRight.getRoot();
    setRoot(new Node<E>(pData, leftChild, rightChild, null));
    if (leftChild != null) leftChild.setParent(getRoot());
    if (rightChild != null) rightChild.setParent(getRoot());
}
```

This driver routine will test things out (put this in *Main.java*):

```
BinaryTree<Integer> treeLeft = new BinaryTree(3);                       //   3
BinaryTree.Iterator<Integer> itLeft = treeLeft.iterator();              //  / \
itLeft.addLeft(10); itLeft.addRight(20);                                // 10  20

BinaryTree<Integer> treeRight = new BinaryTree(5);                      //    5
BinaryTree.Iterator<Integer> itRight = treeRight.iterator();            //   / \
itRight.addLeft(100); itRight.addRight(200); itRight.moveLeft();        // 100 200

BinaryTree<Integer> treeTest = new BinaryTree(9, treeLeft, treeRight);  //       9
// Prints: 9 3 5 10 20 100 200                                          //     /   \
treeTest.traverse(BinaryTree.LEVEL_ORDER, this);                        //   3       5
System.out.println();                                                   //  / \     / \
BinaryTree.Iterator<Integer> itTest = treeTest.iterator();              // 10  20 100 200
itTest.moveLeft(); itTest.moveLeft();
System.out.println(itTest.get());                                       // Prints: 10
itTest.moveUp();
System.out.println(itTest.get());                                       // Prints: 3
itTest.moveUp();
System.out.println(itTest.get());                                       // Prints: 9
itTest.moveUp();
System.out.println(itTest.get());                                       // Prints: 9
```

5.6 Learning Objectives for Exercises 5.6-5.9: The student shall be able to explain what a BST is and to explain the characteristics of BST's. A BST is created (it is initially empty) where the key associated with the data in each node is an integer. Elements are added to the BST with these keys in this order: 5, 4, 8, 7, 6, 9, 3, 2, 1. **(a)** Draw the resulting BST. **(b)** What is the height of the tree?

5.7 For the tree of Exercise 5.6 complete this table which lists how many key comparisons will be made to locate each of the keys in the tree.

Key	Number of Key Comparisons to Locate Node Containing Key
1	
2	
3	
4	
5	
6	
7	
8	
9	

What is the average number of comparisons?

5.8 Continuing, assume the keys of Exercise 5.6 are integers which are appended to a linked list of integers, i.e., the elements of the list will be 5, 4, 8, ..., 2, 1. Assume we always start from the head node when searching for an element. Complete this table which lists the number of comparisons that are made to locate each element in the list.

Element	Number of Comparisons to Locate the Element
1	
2	
3	
4	
5	
6	
7	
8	
9	

What is the average number of comparisons?

5.9 How much faster, expressed as a percentage, are searches in this particular BST than searches in this particular linked list?