1 Submission Instructions

Create a document using your favorite word processor and type your exercise solutions. At the top of the document be sure to include your name and the homework assignment number, e.g. HW4. Convert this document into **Adobe PDF format** and name the PDF file *<asuriteid>.pdf* where *<asuriteid>* is your ASURITE user id (for example, my ASURITE user id is *kburger2* so my file would be named *kburger2.pdf*). To convert your document into PDF format, Microsoft Office versions 2008 and newer will export the document into PDF format by selecting the proper menu item from the File menu. The same is true of Open Office and Libre Office. Otherwise, you may use a freeware PDF converter program, e.g., *CutePDF* is one such program.

Next, create a folder named *<asuriteid>* and copy *<asuriteid>.pdf* to that folder. Copy any requested Java source code files to this folder (note: Java source code files are the files with a *.java* file name extension; **do not** copy the *.class* files as we do not need those).

Next, compress the *<asuriteid>* folder creating a **zip archive** file named *<asuriteid>.zip*. Upload *<asuriteid>.zip* to the Homework Assignment 4 link by the assignment deadline. Consult the online syllabus for the late and academic integrity policies.

Note: not all of these exercises will be graded, i.e., random ones will be selected for grading.

Exercises 4.7–4.9, 5.4, and 5.8–5.9 are optional and are worth bonus credit. Any points earned on these exercises will be added to your homework assignment point total before computing your homework assignment percentage. Your homework assignment percentage is limited to a maximum of 100%, e.g., if you ended up earning 95 homework points your homework percentage would be calculated as *min*(95 / 87.5, 1.0).

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 compexity of BST operations.

3 Sorting

3.1 In the video lecture for *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. The objective of this exercise is to essentially see if you understand how the merge sort procedure works.

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 nonrecursive call to *recursiveMergeSort*() and all of the recursive calls in the sum.

3.3 Continuing, how many times will *merge*() be called?

3.4 Continuing, 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 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. 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 (next page):

```
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 = ??? >= 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

4.1 **(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(In: Integer pData) Returns Nothing
    Define 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 test case 21 which tests that *removeAll*() works correctly.

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 don't have to provide a formal proof; just do a little hand waving).

4.3 **(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();
    if (getSize() == 1) {
        setHead(null);
        setTail(null);
    } else {
        getHead().getNext().setPrev(null);
        setHead(getHead().getNext());
    }
    setSize(getSize() - 1);
    return data;
}

/**
 * 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();
    pNode.getPrev().setNext(pNode.getNext());
    pNode.getNext().setPrev(pNode.getPrev());
    setSize(getSize() - 1);
    return data;
}

/**
 * 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();
    if (getSize() == 1) {
        setHead(null);
        setTail(null);
    } else {
        getTail().getPrev().setNext(null);
        setTail(getTail().getPrev());
    }
    setSize(getSize() - 1);
    return data;
}
```

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 half-a-dozen lines of code). Be sure to run the test cases in *DListTest* to ensure that your new *remove*() method still works correctly. Also, make sure *remove*() throws an *IndexOutOfBoundsException* if *pIndex* is less than 0 or greater than or equal to *getSize*().

4.4 **(Include *DList.java* in your solution zip archive)** 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 half-a-dozen lines of code). Be sure to run the test cases in *DListTest* to ensure that your new *remove*() method still works correctly. Also, make sure *add*() still throws an *IndexOutOf BoundsException* if *pIndex* is less than 0 or greater than *getSize*().

4.5 **(Include *DList.java* in your solution zip archive)** 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, comment out the statements in *removeAll*() that that you implemented in Exercise 4.1, and provide a more efficient implementation of this method. Reuse your same test case from Exercise 4.1 to verify that your new implementation works correctly.

Rather than giving you pseudocode, I will give you a hint by describing the general procedure. Create a *Node* object named *node* which initially refers to the head of the list. Compare the data in the head node to *pData*. If they match, call *removeHead*() to remove the head node (note that the next node you will visit will be the new head node). If they do not match, make *node* refer to the node succeeeding head. Compare the data in the element in the node to *pData*. if they match, and if this is not the tail node, call *removeInterior*() to remove the node. Continue moving *node* to each successive node, checking for a match and removing when necessary. Like the head node, removing the tail node has to be handled specially by calling *removeTail*(). Do not even think about calling *getNodeAt*(index) or *remove*(index); the only methods my solution uses are *getHead*(), *removeHead*(), *remove Interior*(), *removeTail*(), and *getNext*() on the node when we need to move to the next node.

4.6 Give an informal proof of the worst case time complexity of your new *removeAll*() method of Exercise 4.5. Basically, what I am looking for is an identification 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 $g(n)$.

4.7 **(1 bonus pt for each) (Include *DList.java* in your solution zip archive)** Using the new add methods, rewrite *append*(*data*) and *prepend*(*data*).

4.8 **(4 bonus pts) (Include *DList.java* in your solution zip archive)** Write a method `void orderedAdd (Integer pData)` that will insert *Integer*s into a *DList* such that ascending sort order is maintained. For example,

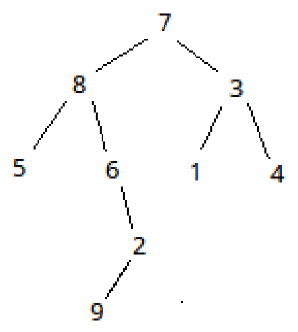
```
DList list = new DList(); // list = { }
list.orderedAdd(5);       // list = { 5 }
list.orderedAdd(3);       // list = { 3 5 }
list.orderedAdd(1);       // list = { 1 3 5 }
list.orderedAdd(7);       // list = { 1 3 5 7 }
list.orderedAdd(9);       // list = { 1 3 5 7 9 }
list.orderedAdd(-5);      // list = { -5 1 3 5 7 9 }
```

4.9 **(4 bonus pts)** Write a method `void split(int pIndex, DList pLeft, DList pRight)` that will "split" this *DList* (the one on which the method is invoked) into a left sublist *pLeft* and a right sublist *pRight*. The elements of *pLeft* will consist of list elements at indices 0, 1, 2, ..., *pIndex* - 1. The elements of *pRight* will consist of list elements at indices *pIndex*, *pIndex* + 1, ..., *getSize*() - 1. Note that *pLeft* and *pRight* must be created (and are assumed to be empty) before calling *split*(). For example:

```
DList list = new DList(); // list = { }
list.append(3); list.append(5); list.append(7); list.append(11);
list.append(13); list.append(17); list.append(19); list.append(29);
// list = { 3, 5, 7, 11, 13, 17, 19, 29 }
DList left = new DList, right = new DList();
list.split(5, left, right);
// left = { 3, 5, 7, 11, 13 }, right = { 17, 19, 29 }
```

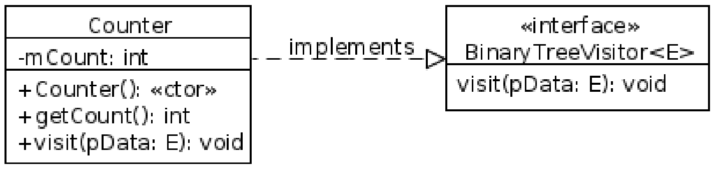
5 Binary Trees and BSTs

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 *Week 3 : Objects and Classes II : Section 2*) in *getSize*() named *Counter* which implements the *BinaryTree Visitor*<*E*> interface:



The *Counter* constructor initializes *mCount* to 0. *visit*() simply increments *mCount* when it is called. 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 performe 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
    ???
    Counter counter = new Counter();
    traverse(LEVEL_ORDER, counter);
    return counter.getCount();
}
```

5.4 **(1 bonus pt)** If *n* is the size of the tree, what is the worst case time complexity of *getSize*() in big O notation?

5.5 **(Include *BinaryTree.java* in your solution zip archive)** 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*, *BinaryTree.Node*, and *BinaryTree.Iterator* 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* class to eliminate the *mStack* instance variable and the *getStack*() and *setStack*() accessor and mutator methods. Modify these *Iterator* methods:

```
public Iterator(BinaryTree<E> pTree) {
    setTree(pTree);
    setCurrent(getTree().getRoot());
    ~~setStack(new Stack<Node<E>>());~~
}
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 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 **(1 bonus pt)** Continuing, assume the keys of Exercise 5.6 are integers which are appened 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 **(1 bonus pt)** How much faster, expressed as a percentage, are searches in this particular BST than searches in this particular linked list?