Winston Shih
WXS190012
CS 3345.002
9/20/2024

CS 3345 Assignment 1 README report

Array Stack:

```
import java.util.EmptyStackException;
public class ArrayStack implements BKStack{
	private int top;
	private static final int INITIAL_CAPACITY=10;
	private double[] stack;
```

ArrayStack is a class that represents an array implementation of the BK Stack class and its methods. It imports the EmptyStackException from java.util that pop and peek operations trigger when there are no elements in Array Stack. It has three instances: top, INITIAL_CAPACITY, and stack. Top is an instance with an integer data type that represents the pointer of the array called stack. Stack has a data type of double and is the array stack of this program. The INITIAL_CAPACITY is a constant that has an integer data type and represents the initial fixed capacity of the array stack before the resize method is invoked. The time and space complexity of this class is O(n) because the memory space required for the class increases as the number of elements in the array stack increases. As the size of the array stack increases, the running time increases.

```
public ArrayStack()
{
	top=-1;
	stack=new double[INITIAL_CAPACITY];
}
```

The ArrayStack class has a no arguments constructor method. The constructor sets the top pointer to index of -1 and instantiates a double array named stack with an initial capacity of 10. This constructor has a time and space complexity of O(1) because the method only performed two operations to instantiate the stack array and set the index value of the top each time it was called. Those two operations used a fixed amount of memory and its running time is not affected by the size of the array stack.

The ArrayStack class implements BKStack interface's isEmpty(), count(), pop(), push(double d), and peek().

```
public boolean isEmpty()
{
	return (top==-1);
}
```

The ArrayStack class's isEmpty method returns whether the arrayStack is empty or not by testing if the top is equal to -1. It returns a boolean value of true or false. The isEmpty() method has a time complexity of O(1) since the method only has one comparison operation that does not depend on stack size. The comparison operation only takes a constant amount of time to execute and a constant amount of space to store.

```
public int count()
{
	return (top+1);
}
```

The count method returns the number of elements in the array stack (integer value). It increments top by 1 and returns that result because the stack pointer top started at index -1. The time complexity and space complexity are O(1). The reason is that there is only one operation executed in the method, which only takes a constant amount of time to execute and does not need additional space to store.

```
public double pop()
{
	if(isEmpty())
	{
		throw new EmptyStackException();
	}
	return stack[top--];
}
```

The pop operation checks to see if the stack is empty. If the stack pointer is equal to -1, then the program throws an EmptyStackException. Otherwise, the method pops the top stack value and returns its double value. The time and space complexity is O(1). The pop method’s access and remove top value operations take a constant amount of time to execute and use a constant portion of memory.

```
public void push(double d)
{
	if(top==stack.length-1)
	{
		resize();
	}
	stack[++top]=d;
}
```

The push method takes in a double type element as input. It checks to see if the stack pointer named top’s index is equal to the current size of array stack minus one. If the stack pointer is equal to the size of the array stack or higher, then the resize method is called. Top gets preincrement so that the value of the newest element of stack is equal to element that should be pushed into array stack. Push has an O(1) complexity if stack capacity is not exceeded because it takes constant time to push an element onto stack and adding a single element to stack does not require a program to create more data structures. If the resize method is invoked, the complexity of push will be O(n) since the resized array will need to copy all of the original array elements to its first n spaces of stack and the new array will need n more space.

```
public double peek()
{
	if(isEmpty())
	{
		throw new EmptyStackException();
	}
	return stack[top];
}
```

Peek method checks to see if the stack is empty. If Array Stack is empty, the method throws an EmptyStackException. If the array is not empty, the peek operation returns the element on top of the Array Stack. Peek operation has a time and space complexity of O(1) because the only operation it executes is accessing the top element of the array stack. This operation can be done with a constant amount of time and space regardless of input size of the array stack.

```
private void resize()
{
	double[] stack2=new double[stack.length*2];
	for(int i=0;i<stack.length;i++)
	{
		stack2[i]=stack[i];
	}
	stack=stack2;
}
```

Resize method creates a new double array called stack2, but sets the size of the array to be two times the capacity of the array stack. It then uses a for loop to copy all elements of array stack to array stack 2 from index o to stack.length-1. After copying the original array stack's elements, the method assigns stack2 array to stack array so that stack array has the original array's elements plus extra memory space. This method implementation has a time complexity of O(n) because the for loop has to iterate through n elements when it copies each element in stack array into stack2 array. Theis method also has a space complexity of O(n) since stack2 is a copy of stack and has twice the amount of memory space than the original array stack.

Linked Stack:

```
import java.util.EmptyStackException;
public class ListStack implements BKStack, Iterable<Double>{
	private int size;
	private int modCount=0;
	private ListStackNode begin;
	private ListStackNode end;
	private ListStackNode top;
```

The ListStack class is a linked list stack implementation of the BKStack class interface and Iterable class interface. The data type of the class is Double because the Iterable interface only accepts Double wrapper class for double data type numbers. This class imports the EmptyStackException from java.util package and uses it when one of the pop and peek operations finds out that the linked list stack is empty. This class has three ListStackNode instances called begin, end, and top. The top is the stack pointer for linked list stack. Begin and end are the beginning and ending nodes for the linked list stack. The class also has two integer instances called size and modCount. modCount is the modification counter of the class and has an initial value of 0. The size instance represents the current size of the linked list stack of ListStack class. The time complexity and space complexity of ListStack class is O(n) because the linked list stack stores each element in its own separate node on the linked list. The memory space and running time increases as the linked list stack adds more nodes.

```
private static class ListStackNode
{
	private Double data;
	private ListStackNode previous;
	private ListStackNode next;
```

The ListStackNode is an inner Node class of the ListStack class that is used as a linked-list node for this class's linked list stack. It has a Double instance called data representing a node's data. The other two instances are previous and next. They have a ListStackNode data type and they represent the nodes preceding and succeeding the current node. The time complexity is O(1) because all of the node class's operations can be run in constant time. The space complexity is O(n) since each node in the linked list stack needs more space to store the node data, previous node, and next node.

```
public ListStackNode(Double d, ListStackNode p, ListStackNode n)
{
	data=d;
	previous=p;
	next=n;
}
```

This is a parameterized constructor of ListStackNode class if the parameters for node data, previous node, and next node are defined. The constructor assigns the data, previous node, and next node instances the values from the data, previous node, and next node parameters. The time and space complexity is O(1) since it requires constant space and running time to assign the instances its respective parameter values.

```
public ListStackNode(Double d)
{
	data=d;
	previous=null;
	next=null;
}
```

This is another parameterized constructor of the ListStackNode class for when the node data parameter is defined. The constructor assigns the data instance the data parameter's value. The previous and next instances are given the value of null because there are no previous and next node parameters. The time and space complexity is O(1) since it requires only a constant amount of memory space and running time to assign the data instance the value of its parameter value and set value of previous and next nodes to null.

```
public ListStack()
{
	clear();
}
```

This is a no arguments constructor for ListStack class. It calls the clear method to create a new linked list stack. The time and space complexity are O(1) because it only calls the clear method and that method call has constant time complexity and does not require any more memory space.

```
public void clear()
{
	begin=new ListStackNode(null,null,null);
	end=new ListStackNode(null,begin,null);
	begin.next=end;
	size=0;
	modCount++;
}
```

This clear method instantiates begin and end nodes, links the nodes together, sets linked list stack size to 0, and increments modification counter. The clear method has a time and space complexity since the method reuses variables and nodes when the linked list stack needs to be reset. The method performs a constant number of operations and reassigns the variables and nodes that already exist in the program so it does not need more memory space for new nodes or variables.

```
public int size()
{
	return size;
}
```

This method returns the size of the linked list stack, which has a data type of integer. The time and space complexity is O(1) because the size method returning size of a list stack does not need additional running time or memory.

```
public boolean isEmpty()
{
	return (top==null);
}
```

isEmpty returns whether the linked list stack is empty by checking if the top pointer is equal to null and returning boolean values. If the top is equal to null, the stack is empty and it returns true. Otherwise, the stack is full and isEmpty returns false. The isEmpty time and space complexity is O(1) since the program only executes one conditional operation and does not require additional memory space.

```
public boolean add(Double x)
{
	add(size(), x);
	return true;
}
```

add(Double x) takes in a parameter named x that takes in Double values as input. The method calls add void method to add the value of x to the node located at index of current size of linked list array. It returns true after the new node is added. The time and space complexity is O(1) since adding a new node operation only requires a constant running time and constant amount of extra memory space.

```
public void add(int idx, Double x)
{
	addBefore(getNode(idx, 0, size()),x);
}
```

This void add method only calls addBefore method and getNode methods to add a node before a specified index. For the p parameter of addBefore, it uses the method call of getNode method. For the x parameter of add, it inputs its value of x variable for addBefore's x parameter. In the method call for getNode, the method inputs its value of idx for getNode's idx parameter. It also inputs 0 and method call of size for low and high parameters. The time and space complexity for this void method are O(n) and O(1). In the worst case scenario, the program will be forced to traverse through all n elements in the linked list stack to look for idx to insert a node before an existing node. This operation only needs a constant amount of extra memory space.

```
private void addBefore(ListStackNode p, Double x)
{
	ListStackNode nextNode=new ListStackNode(x, p.previous, p);
```

```
        nextNode.previous.next=nextNode;
        p.previous=nextNode;
        size++;
        modCount++;
}
```

The addBefore void method adds a node with data x before a current node p. The addBefore method has two parameters called p and x, which have data types of ListStackNode and x. P is the node that will be added before the current node and x is the data of the node that will be added. The method creates and instantiates a new ListStackNode object called nextNode by passing its x and p values to ListStackNode constructor's node data and next node parameters. For the previous node parameter, it passes the node preceding p node. After creating nextNode, its value gets passed to the node after the node preceding nextNode to link previous node to current node. The node preceding p node gets its value set as value of nextNode so p is linked to the new node. The method increments the size of the linked list stack and its modification counter. The time and space complexity of the method is O(1) because the method only uses a constant amount of running time to change values of pointers and a constant amount of new memory space is used to create new nodes.

```
private ListStackNode getNode(int idx)
{
        return getNode(idx, 0, size()-1);
}
```

This getNode method only has one parameter called idx for index of node and has a data type of ListStackNode. This method uses a recursive call of a different getNode method call that has a parameter for node index, low index, and high index. The recursive call inputs this method's idx for the other method's idx parameter, inputs 0 for low index, and size()-1 for high index to get a node at an index between 0 and size of array minus one. The time and space complexity is O(log n) and O(1) because this method uses binary search to look for the node at the correct index and only uses a constant amount of extra space in stack memory.

```
private ListStackNode getNode(int idx, int low, int up)
{
        ListStackNode p;
        if(idx>up||idx<low)
        {
                throw new IndexOutOfBoundsException();
        }
        if(idx<(size()/2))
        {
                p=begin.next;
                for(int i=0;i<idx;i++)
                {
                        p=p.next;
                }
        }
        else
        {
                p=end;
                for(int i=size();i>idx;i--)
                {
                        p=p.previous;
                }
        }
        return p;
}
```

/**
	* Method gets Node at idx that must be between index low and index high and checks to see if index is before or after midpoint of list stack to find the node
	* more efficiently.
	* @param idx index to search.
	* @param low lowest index.
	* @param up highest index.
	* @return p internal node related to idx
	* @throws IndexOutOfBoundsException if index is higher than last index or lower than first index (inclusive).
	*/

This getNode method has a data type of ListStackNode and integer parameter of idx (index) like the other method, but it has two more parameters. The integer parameters low and up represent the minimum and maximum indexes in the linked list stack. The method checks to see if the index imputed for idx does not go below low or above up by throwing an IndexOutOfBoundsException if any condition is violated. After that it checks to see if the index is before the middle index or after the middle index. If the index is lower midpoint, the pointer starts from the first element and traverses from index 0 to the index from parameter. If index is greater than or equal to middle index, the stack pointer starts at index stack length-1 and traverses from the ending index to parameter index or middle index. After retrieving node data from the index, the method returns the node data. The time complexity and space complexity are O(n) and O(1) since the number of iterations to traverse the list starting from start index or end index is proportional to list stack size. The space complexity is O(1) because the amount of extra memory space needed will not change even if size of linked list stack does.

```
public Double remove(int idx)
{
        return remove(getNode(idx));
}
```

This method returns Double value of linked list node by using two recursive calls. It first recursively calls remove method with a parameter of p that has a data type of ListStackNode and represents the previous node. The input for that parameter is a recursive call to getNode with a parameter for idx. The method inputs its idx value for that parameter. The time complexity is O(n) and space complexity is O(1) because find operation requires traversing through a linked list stack. The find operation’s O(n) time complexity dominates the remove operation’s O(1) time complexity. However, the method uses a constant amount of memory space since the list stack does not increase in input size.

```
public Double set( int idx, Double newVal )
{
        ListStackNode p = getNode(idx);
        Double oldVal = p.data;
        p.data = newVal;
        return oldVal;
}
```
set() changes thenode located at idx index to value specified in newVal parameter. The method first creates a new ListStackNode object at idx index, creates a Double object called oldVal with value of p's node data, reassigns node p's data to value of newVal, and returns original value of node at index idx. The time and space complexity is O(n) and O(1). The reason time complexity is O(n) is that in the worst case scenario, the method will have to traverse the entire linked list stack to find a specific node at a given index. The space complexity is O(1) because each variable only needs a constant amount of extra memory space.
```
public Double remove(ListStackNode p)
{
        p.next.previous=p.previous;
        p.previous.next=p.next;
        size--;
        modCount++;
        return p.data;
}
```
Method removes node based on what node is imputed in its p parameter. It assigns node before next node after p the value of previous node before p and assigns the node after previous node preceding p the value of next node after p. The method decrements the size after removing p and increments modification counter by 1. The method ends by returning the node data of p. The time and space complexity is O(1) because it only takes a constant amount of time to remove a node from the linked list stack. Also, the input size decreases so there is no need for additional memory space in the linked list stack.
```
public java.util.Iterator<Double> iterator()
{
        return new ListStackIterator();
}
```
This iterator class creates and returns an instance of ListStackIterator class that is an iterator for Double value in linked list stack. The time and space complexity is O(1) because the only operations are creating and returning new objects for the ListStackIterator class. The operations do not depend on input size so they have constant time and space complexity.
```
private class ListStackIterator implements java.util.Iterator<Double>
{
        private ListStackNode current=begin.next;
        private int expectedModCount=modCount;
        private boolean okToRemove=false;
```
The ListStackIterator is an iterator implementation of java.util.Iterator<Double> class. It has three instances. The first instance is current, which has a data type of ListStackNode and is assigned the value of the next node after begin. expectedModCount has an integer data type and is assigned the modCount's value so the program knows how many changes should be made to the linked list stack. The boolean instance is okToRemove and is set to false so any node can not be removed without permission. The time

and space complexity is O(1) since it uses a constant amount of space for each of its instances. The methods hasNext() and next() only use a constant amount of time to access the next node in the list.

```
public boolean hasNext()
{
        return (current!=end);
}
```

hasNext returns if there is another node after the current node by checking if the current node does not equal the end node . The time and space complexity is O(1) since the method only performs one operation that requires a constant amount of running time and additional space.

```
public Double next()
{
        if(modCount!=expectedModCount)
        {
                throw new java.util.ConcurrentModificationException();
        }
        if(!(hasNext()))
        {
                throw new java.util.NoSuchElementException();
        }
        Double nextItem=current.data;
        current=current.next;
        okToRemove=true;
        return nextItem;
}
```

next() checks if the modification counter is correct and throws a ConcurrentModificationException if the modification counter is incorrect. next() then checks if there is another node after the current node and throws NoSuchElementException if there is no next node. After those checks, the method sets nextItem to data of current node, sets current node's value to next node, allows the previous current node to be removed, and returns the Double value in the current node. The time and space complexity is O(1) since each return or check operation requires constant running time and does not need to create additional nodes. It only needs a constant amount of memory for its return or error catching operations.

```
public void remove()
{
        if(modCount!=expectedModCount)
        {
                throw new java.util.ConcurrentModificationException();
        }
        if(!okToRemove)
        {
                throw new IllegalStateException( );
        }
        ListStack.this.remove(current.previous);
        expectedModCount++;
        okToRemove = false;
}
```
The remove method implementation checks to see if the modification counter is equal to the expected modification counter. It throws a ConcurrentModificationException if the modification counter is incorrect. It also checks to see if the remove operation is invoked more than once. If the remove method was called more than once, then the program throws IllegalStateException. After checking if the modification counter is correct, the method removes the current node, increments expectedModCount by 1, and sets ability to remove node to false. The time and space complexity is O(1) because it only performs only 3 operations that only utilizes a constant space of memory.
```
public int count()
{
	int nodeCount=0;
	ListStackNode current=top;
	for(Double data:this)
	{
		if(data!=null)
		{
			nodeCount++;
			current=current.next;
		}
	}
	return nodeCount;
}
```
Count method uses an enhanced for loop to search for Double values in the program's array that is represented by this keyword. It uses a nodeCount that is initialized to 0 and the ListStackNode pointer current is assigned the value of the top pointer. During the duration of the loop, the method checks to see if the current node has a non null value. If node is not empty, the nodeCount increases and the current pointer points to the next node in the list. After the enhanced for loop reaches the end of the list, the method returns how many nodes are in the linked list stack, which have an integer data type. The space complexity is O(1) since the method only utilizes a constant amount of memory space in the linked list stack. The time complexity is O(n) because the method has to iterate through an n amount of elements in an n amount of time to count how many elements are in the linked list space.
```
public void push(double d)
{
	ListStackNode node=new ListStackNode(d);
	if(!(isEmpty()))
	{
		node.next=top;
	}
	top=node;
}
```
Push creates a new node using node data from the parameter. It checks to see if the linked list is not empty before making the next pointer point to the current top value. After that, the top is updated with the new node so the new node is the top element of the linked list stack. The time and space complexity is O(1)

since the creation of a node and reassignment of stack pointers do not depend on the input size of the linked list stack and only require a constant amount of memory.

```
public double pop()
{
	if(isEmpty())
	{
		throw new EmptyStackException();
	}
	Double data=top.data;
	top=top.next;
	return data;
}
```

Pop checks to see if the stack is empty. If the linked list is empty, it throws an EmptyStack Exception. If the linked list stack is not empty, it removes the top element of stack by reassigning top's value to top.next and returning the value of the previous top value. The data type of data is double. The time and space complexity is O(1) because remove and return operations do not require the program to use more memory space or run time.

```
public double peek()
{
	if(isEmpty())
	{
		throw new EmptyStackException();
	}
	return top.data;
}
```

Peek method checks to see if the linked list stack is empty. If the linked list is empty, it throws an EmptyStackException. Otherwise, it returns the top element of the linked list stack, which is a double value. The time and space complexity is O(1) because it performs one return operation that does not need more running time and space to execute.