Neil Daterao
CSC-151
Project 5 ADT Writeup

For my PageList ADT, I decided to go with a Linked List Implementation.

## Advantages

**Sequential Insertion and Retrieval:** Linked lists maintain the sequence in which elements are inserted, which aligns with the requirement of saving pages in the order they are encountered.

**Dynamic Size:** Linked lists can dynamically adjust their size as elements are added or removed, which allows for flexibility in handling different numbers of pages for each word. Additionally, obtaining the length of the linkedlist is an O(1) operation, so we don't have to iterate through the entire container to get the total number of pages if we are creating a program that is dependent on size. Our project wouldn't be too affected by this though since we would only have a max of 4 pages.

**Memory Efficiency:** Linked lists can be more memory-efficient compared to other data structures like arrays, especially when dealing with large pageLists where memory allocation might become a concern.

## Disadvantages

**Not Random Access:** Since LinkedLists are not random access, the average case for search is O(n). If there is a need to access pages at arbitrary positions frequently, a linked list might not be the best choice. However, since our project deals with a max of 4 pages, we don't lose efficiency as O(4) == O(1).

**Limited Search Efficiency:** Searching for a specific page number within a linked list requires traversing the list sequentially, resulting in a linear time complexity. This can be inefficient for large pageLists, especially when searching for specific page numbers multiple times.

**Memory Overhead:** Each node in a linked list requires additional memory overhead for storing pointers to the next node. This overhead can become significant for large pageLists, potentially impacting memory usage and performance.

Overall, for the needs of this project, I feel the LinkedList was the best choice. If we were dealing with a much larger page list, perhaps a HashSet or HashTable for constant look up times would be suitable, however these don't maintain sequence. Despite the ADT, there is going to be some tradeoffs.

```java
package proj5;

/**
 * Driver for the index maker project
 *
 * @author Neil Daterao
 * @version 3/12/2024
 * @note I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 */
public class Client
{
    public static void main(String[] args)
    {
        makeIndex("src/resources/uscons.txt"); // replace with correct path
    }

    /**
     * Makes an index out of fileName. Gradescope needs this function.
     *
     * @param fileName path to text file that you want to index
     */
    public static void makeIndex(String fileName) {

        Index index = new Index(fileName);
        index.createIndex();
    }
}

```

```java
package proj5;

/**
 * Index class.
 * Indexes will only include words such that there are less than MAXPAGESINPAGELIST occurances of the word
 * in a file on different pages. Additionally, indexes will only contain words longer than MINWORDLENGTH
 * Pages breaks should be denoted using "#" within file.
 * Words not included in index will be stored in a Dictionary.
 *
 * @author Neil Daterao
 * @version 3/09/2024
 */
public class Index {

    private BinarySearchTree<IndexEntry> contents;
    private FileReader reader;
    private int currentPage;
    private Dictionary dictionary;

    public final int MINWORDLENGTH = 2;
    public final int MAXPAGESINPAGELIST = 4;


    /**
     * Constructor to initialize an index given a filePath
     * @param filePath
     */
    public Index(String filePath) {
        dictionary = new Dictionary();
        contents = new BinarySearchTree<>();
        currentPage = 1;
        reader = new FileReader(filePath);

    }
```

```java

    /**
     * Driver code to create index. Will print the to the standard output after being created.
     */
    public void createIndex() {
        String token;
        while ((token = reader.nextToken()) != null) {
            if (token.equals("#")) { nextPage(); }
            else {
                if (token.length() > MINWORDLENGTH && !getDictionary().contains(token)) {
                    if (this.contains(token)) { updateIndex(token); }
                    else { this.add(token);  }
                }
            }

        }
        System.out.println(this.toString());
        System.out.println(dictionary);
    }

    /**
     * Getter for Dictionary
     * @return
     */
    public Dictionary getDictionary() {
        return dictionary;
    }

    /**
     * Private helper method to move to next page
     */
    private void nextPage() {
        currentPage++;
    }

    /**
     * Private getter to get the current page
```

```
     * @return
     */
    private int getCurrentPage() {
        return currentPage;
    }

    /**
     *
     * @param word
     * @return true if index contains word, false if not.
     */
    public Boolean contains(String word) {
        IndexEntry temp = new IndexEntry(word);
        return contents.search(temp);
    }

    /**
     *
     * @param word
     * @return Index Entry from index given a word
     */
    public IndexEntry getIndexEntry(String word) {
        IndexEntry temp = new IndexEntry(word);
        return (IndexEntry)contents.get(temp).data;
    }

    /**
     * Delete an index entry from an index given a word.
     * Will also print "Deleting {IndexEntry} from index."
     * @param word
     */
    public void deleteEntryFromIndex(String word) {
        if (contains(word)) {
            System.out.println("Deleting '" + getIndexEntry(word).toString() + "' from index.");
            IndexEntry temp = new IndexEntry(word);
```

```
            contents.delete(temp);
        }
    }

    /**
     * Adds a new word to the index with the page number.
     * Handles merge cases so index is formatted properly
     * with no duplicate words and upper and lowercase versions
     * of the same words.
     * @param word
     */
    public void add(String word){
        if (!this.contains(word)) {

            if (isUpperCase(word)) {
                String lowerCaseWord = word.toLowerCase();
                if (this.contains(lowerCaseWord)){
                handleUpperCaseWordWithLowerCaseInIndex(lowerCaseWord);
                }
                else {  addNewEntry(word); }
            }
            else {
                String upperCaseWord = capitalizeWord(word);
                if (this.contains(upperCaseWord)) {
                handleLowerCaseWordWithUpperCaseInIndex(upperCaseWord, word);
                }
                else { addNewEntry(word); }
            }
        }
    }

```

```java
    /**
     * String version of Index. Each IndexEntry printed in ASCII Alphabetical order.
     * Each IndexEntry on new lines.
     * @return String verison of Index.
     */
    public String toString() {
        return contents.toStringReverseInOrderTraversal();
    }

    /**
     * Private helper method to check if a word is uppercase.
     * An uppercase word is uppercase if the first letter is capitalized.
     * @param word
     * @return True if uppercase, false if not
     */
    private Boolean isUpperCase(String word) {
        return Character.isUpperCase(word.charAt(0));
    }

    /**
     * Private helper method to handle the case when word is lowercase and
     * the uppercase word exists in the index already, copy the pages from
     * the upper case word to the lower case pagelist, then add the current page.
     * @param upperCaseWord
     * @param word
     */
    private void handleLowerCaseWordWithUpperCaseInIndex(String upperCaseWord, String word){
        IndexEntry newEntry = new IndexEntry(word);
        PageList pageListForUppercaseWord = this.getIndexEntry(upperCaseWord).getPageList();
        PageList pageListForLowerCaseWord =
```

```
newEntry.getPageList();
        pageListForLowerCaseWord.addPageList(pageListForUppercaseWord);
        contents.insert(newEntry);
        deleteEntryFromIndex(upperCaseWord);
        getDictionary().remove(upperCaseWord); //we don't want the uppercase word to be placed in dictionary
        updateIndex(word);
    }

    /**
     * Private helper method to handle the case wher if
     * word is uppercase and the lower case word exists in the index already.
     * Add the current page to the lowercase pagelist.
     * @param lowerCaseWord
     */
    private void handleUpperCaseWordWithLowerCaseInIndex(String lowerCaseWord) {
        updateIndex(lowerCaseWord);
    }


    /**
     * Private helper method to capitalize the first letter of a word
     * @param token
     * @return
     */
    private String capitalizeWord(String token) {
        return Character.toUpperCase(token.charAt(0)) + token.substring(1);
    }

    /**
     * Private helper method to add a new entry to the index given a word
```

```
     * @param word
     */
    private void addNewEntry(String word) {
        IndexEntry newEntry = new IndexEntry(word, getCurrentPage());
        contents.insert(newEntry);
    }

    /**
     * Private helper method to update the index.
     * @param token
     */
    private void updateIndex(String token) {
        PageList pageListForWord = this.getIndexEntry(token).getPageList();
        if (!pageListForWord.containsPage(getCurrentPage())) {
            if (pageListForWord.size() < MAXPAGESINPAGELIST) {
                pageListForWord.addPage(getCurrentPage());
            } else {
                deleteEntryFromIndex(token);
                getDictionary().add(token);
            }
        }
    }

}

```

```java
package proj5;

/**
 * Class for Index Entries. An index entry has a word and a corresponding page list.
 * @author Neil Daterao
 * @version 03/09/2024
 */
public class IndexEntry implements Comparable<IndexEntry> {

    private String data;
    private PageList pageList;

    /**
     * Constructor for a new Index Entry
     * @param data
     */
    public IndexEntry(String data) {
        this.data = data;
        pageList = new PageList();
    }

    /**
     * Constructor to initliaze an index entry with the data and the first page seen.
     * @param data
     * @param firstPage
     */
    public IndexEntry(String data, int firstPage) {
        this.data = data;
        pageList = new PageList();
        pageList.addPage(firstPage);
    }


    /**
     *
     * @return The word keeping track of
     */
    public String getWord() {
```

```
        return data;
    }

    /**
     *
     * @return The page list of the word
     */
    public PageList getPageList() {
        return pageList;
    }

    /**
     * CompareTo method
     * @param otherEntry other index entry
     * @return 1 if this is greater than other
     * -1 if other is greater than this
     *  0 if they are equal.
     */
    public int compareTo(IndexEntry otherEntry) {
        return this.getWord().compareTo(otherEntry.getWord());
    }

    /**
     * @return String version of Index Entry
     * example:
     *  "java {3, 8}"
     */
    public String toString() {
        return getWord().toString() + " " + pageList.toString();

    }
}

```

```java
package proj5;

/**
 * Generic Binary Search Tree Abstract Data Type (ADT). Can hold any element that extends the Comparable Interface.
 *
 * @author Neil Daterao
 * @version 3/08/2024
 */
public class BinarySearchTree<Element extends Comparable<Element>>
{
    private BSTNode<Element> root;

    /**
     * Default constructor, initialize empty BST
     */
    public BinarySearchTree() {
        root=null;
    }

    /**
    * Private helper method to insert a BSTNode recursively
    *
    * @param subroot inserts into subtree rooted at subroot
    * @param newNode node to insert
    * @return the BST rooted at subroot that has newNode inserted
    */
    private BSTNode<Element> recursiveInsert(BSTNode<Element> subroot, BSTNode<Element> newNode) {
      if (isEmpty(subroot)) {
        return newNode;
      }
      else if (newNode.data.compareTo(subroot.data) > 0) {
        subroot.rlink = recursiveInsert(subroot.rlink,newNode);
```

```
        return subroot;
      }
      else {
        subroot.llink = recursiveInsert(subroot.llink,newNode);
        return subroot;
      }
    }

    /**
    * Inserts a new element into BST
    *
    * @param newElement to insert
    */
    public void insert(Element newElement){
        if (!search(newElement)) {
          BSTNode<Element> newNode = new BSTNode<>(newElement);
          root = recursiveInsert(root, newNode);
        }
    }


    /**
    * Private recursive function to build the string version of the BST.
    *
    * @param subroot subroot of tree to print
    */
    private String print(BSTNode<Element> subroot)
    {
      StringBuilder builder = new StringBuilder();

      if (!isEmpty(subroot)) {
          builder.append("(");
          builder.append(print(subroot.llink));
          builder.append(" " + subroot + " ");
          builder.append(print(subroot.rlink));
          builder.append(")");
      }

```

```
      return builder.toString();
    }


    /**
     *  Prints a parenthesized version of this tree that shows
     *  the subtree structure.  Every non-empty subtree is
     *  encased in parentheses. Example: (( A ) B ( C )) means
     *  B is the parent of A (left kid) and C (right kid).
     * @return String version of Binary Search Tree.
     */
    public String toString()
    {

      return print(root);

    }

    /**
     * String version of BST in ascending order on seperate lines.
     * @return
     */
    public String toStringReverseInOrderTraversal() {
        return reverseInOrder(root);
    }

    /**
     * Helper method for reverse in-order traversal and building the string representation. Builds in ascending order (least->greatest)
     */
    private String reverseInOrder(BSTNode<Element> subRoot) {
      StringBuilder builder = new StringBuilder();
```

```java

        if (!isEmpty(subRoot)) {
            builder.append(reverseInOrder(subRoot.llink));
            builder.append(subRoot.data).append("\n");
            builder.append(reverseInOrder(subRoot.rlink));
        }

        return builder.toString();
    }

    /**
     *
     * @return number of data items (nodes) in the tree
     */
    public int size() {
        return size(root);
    }

    /**
     * Private recursive method to return size of BST
     * @param subRoot
     * @return
     */
    private int size(BSTNode<Element> subRoot) {
        if (isEmpty(subRoot)) { return 0; }
        else { return 1 + size(subRoot.llink) + size(subRoot.rlink);  }

    }

    /**
     *
     * @param target value to search for
     * @return true if target is in BST and false if not
     */
```

```java
    public boolean search(Element target) {
        return search(target, root);
    }

    /**
     * Returns the BSTNode of the target
     * @param target
     * @return BSTNode of target, or null if target is not in BST.
     */
    public BSTNode<Element> get(Element target) {
        return searchAndReturnNode(target, root);
    }


    /**
     * Private recursive method to search for target given a subroot in a BST
     * @param target
     * @param subRoot
     * @return true if target is in BST and false if not
     */
    private boolean search(Element target, BSTNode<Element> subRoot) {
        if (isEmpty(subRoot)) { return false; }
        else {
            int comparison = subRoot.data.compareTo(target);

            if (comparison == 0) { return true; }
            else if (comparison > 0) { return search(target, subRoot.llink); }
            else { return search(target, subRoot.rlink); }
        }
    }

    /**
     * Private recursive method to search and return the node for target given a subroot in a BST
```

```
     * @param target
     * @param subRoot
     * @return true if target is in BST and false if not
     */
    private BSTNode<Element> searchAndReturnNode(Element target, BSTNode<Element> subRoot) {
        if (isEmpty(subRoot)) { return null; }
        else {
            int comparison = subRoot.data.compareTo(target);

            if (comparison == 0) { return subRoot; }
            else if (comparison > 0) { return searchAndReturnNode(target, subRoot.llink); }
            else { return searchAndReturnNode(target, subRoot.rlink); }
        }
    }

    /**
     * Deletes value from a tree, if value is not there, do nothing
     * @param value value to be deleted
     */
    public void delete(Element value) {
       root = delete(root, value);
    }

    /**
       * deletes value from tree rooted at subroot
       *  @param subroot  root of tree to be deleted from
       * @param value  element to delete
       * @return pointer to tree rooted at subroot that has value removed
       *
     */
      private BSTNode<Element> delete(BSTNode<Element> subroot, Element value) {
```

```
        if (isEmpty(subroot)) { return null; }
        else if (subroot.data.compareTo(value) > 0 ) {
          subroot.llink = delete(subroot.llink, value);
          return subroot;
        }
        else if (subroot.data.compareTo(value) < 0 ) {
          subroot.rlink = delete(subroot.rlink, value);
          return subroot;
        }

        else {
          // Case 1: Node is a leaf, return null to delete it
          if (isEmpty(subroot.llink) && isEmpty(subroot.rlink)) { return null; }

          // Case 2: Node has only a right subtree, return right subtree
          else if (isEmpty(subroot.llink)) { return subroot.rlink; }

          // Case 3: Node has only a left subtree, return left subtree
          else if (isEmpty(subroot.rlink)) { return subroot.llink; }

          // Case 4: Node has 2 subtrees, locate successor.
          else {
              BSTNode<Element> minNode = findMin(subroot.rlink);
              subroot.data = minNode.data;
              subroot.rlink = delete(subroot.rlink, subroot.data);
          }

      }
```

```java

        return subroot;


    }

    /**
     * Helper method to find the minimum value node in a subtree
     * @param node
     * @return
     */
    private BSTNode<Element> findMin(BSTNode<Element> subroot) {
        while (!isEmpty(subroot.llink)) {
            subroot = subroot.llink;
        }
        return subroot;
    }

    /**
     * Return true if a given subroot is empty and false if not
     * @param subRoot
     * @return
     */
    private Boolean isEmpty(BSTNode<Element> subRoot) {
        return subRoot == null;
    }



}

```

```java
package proj5;

/**
 * Generic BinarySearchTree Node class.
 *
 * @author Neil Daterao
 * @version 3/08/2024
 */
public class BSTNode<Element extends Comparable<Element>>{

    public Element data;
    public BSTNode<Element> llink;
    public BSTNode<Element> rlink;

    /**
     * non-default constructor
     * @param newKey Element that node will hold
     */
    public BSTNode(Element newKey)
    {
        data = newKey;
        llink = null;
        rlink = null;
    }

    /**
     * returns key as printable string
     */
    public String toString()
    {
        return data.toString();
    }
}

```

```java
package proj5;

/**
 * Dictionary Class for Strings, doesn't allow for duplicates
 * @author Neil Daterao
 * @version 3/09/2024
 */
public class Dictionary {

    private BinarySearchTree<String> contents;

    /**
     * Constructs an empty dictionary.
     */
    public Dictionary() {
        contents = new BinarySearchTree<>();
    }

    /**
     * Adds a word to the dictionary. If dictionary already has word, nothing will happen.
     * @param word the word to be added to the dictionary
     */
    public void add(String word) {
        contents.insert(word);
    }

    /**
     * Checks if a word is present in the dictionary.
     * @param String to be added
     * @return true if the String is present in the dictionary, false otherwise
     */
    public boolean contains(String word) {
       return contents.search(word);
    }

    /**
```

```
     * Removes a word from the dictionary. If String is not there, do nothing.
     * @param word the String to be removed from the dictionary
     */
    public void remove(String word) {
        contents.delete(word);
    }

    /**
     *
     * @return Size of dictionary
     */
    public int size() {
        return contents.size();
    }

    /**
     *
     * @return True if dictionary is empty, false if not
     */
    public Boolean isEmpty() {
        return contents.size() == 0;
    }

    /**
     * @return String version of dictionary in ascending order with a word per line.
     */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Dictionary:").append("\n");
        sb.append(contents.toStringReverseInOrderTraversal());
        return sb.toString();
    }


}

```

```java
package proj5;

/**
 * PageList class. Container of pages that can contain duplicates
 *
 * @author Neil Daterao
 * @version 3/09/2024
 */
public class PageList {

    private LinkedList<Integer> contents;


    /**
     * Default constructor, creates empty pageList
     */
    public PageList() {
        contents = new LinkedList<>();
    }

    /**
     * @param pageNumber to add to pagelist
     */
    public void addPage(int pageNumber) {
        contents.insertAtTail(pageNumber);
    }

    /**
     *
     * @param pageNumber to remove from pagelist
     * If pageNumber not in pagelist, will print "Page not in PageList"
     */
    public void removePage(int pageNumber) {
        try { contents.removeAtIndex(contents.indexOf(pageNumber)); }
        catch(IndexOutOfBoundsException e) {
            System.out.println("Page not in PageList");
        }
```

```
    }

    /**
     *
     * @param pageNumber
     * @return True if pageNumber in in list and false if not
     */
    public Boolean containsPage(int pageNumber) {
        return contents.contains(pageNumber);
    }

    /**
     *
     * @return Size of page list
     */
    public int size() {
        return contents.size();
    }

    /**
     *
     * @return true if pagelist is empty and false if not
     */
    public boolean isEmpty() {
        return contents.isEmpty();
    }

    /**
     * Adds the contents of another pagelist to this pagelist. Does not change what is in other page list.
     * @param otherPageList
     */
    public void addPageList(PageList otherPageList) {
        LinkedList<Integer> otherList = otherPageList.contents.clone();
        while (!otherList.isEmpty()) {
            addPage(otherList.removeHead());
```

```
        }
    }


    /**
     * @return String version of page list. Enclosed in Squiggly Brackets, seperated by commas. If consecutive pages, start and end are seperated by dashes.
     * Ex: {2,4,7,9}
     * Ex: {2-4,7} == {2,3,4,7}
     *
     */
    public String toString(){
        StringBuilder sb = new StringBuilder();
        sb.append("{");

        if (contents.size() > 0) {
            int start = contents.getIthItem(0);
            int end = start;

            for (int i = 1; i < contents.size(); i++) {
                if (contents.getIthItem(i) == end + 1) {
                    end++;
                } else {
                    if (start == end) {
                        sb.append(start);
                    } else {
                        sb.append(start).append("-").append(end);
                    }

                    sb.append(", ");
                    start = contents.getIthItem(i);
                    end = start;
                }
            }

            if (start == end) {
```

```
                sb.append(start);
            } else {
                sb.append(start).append("-").append(end);
            }
        }

        sb.append("}");
        return sb.toString();
    }
}

```

```java
package proj5;

/**
 * Linked List is a collection of data nodes.  All methods here relate to
 * how one can manipulate those nodes. Generic linked list class that can hold any element that extends the Comparable Interface.
 *
 * @author Neil Daterao
 * @version 3/09/2024
 */
public class LinkedList<Element extends Comparable<Element>>
{
    private int length;
    private ListNode<Element> firstNode;

    /**
     * default constructor
     */
    public LinkedList()
    {
        length=0;
        firstNode=null;
    }

    /**
     * return number of items in list
     * @return number of items (nodes) in the list
     */
    public int getLength()
    {
        return length;
    }

    /**
     * tells if this linked list is empty or not
     * @return true if this list has no nodes; else false
     */
```

```java
    public boolean isEmpty() {
        return getLength()==0;
    }

    /** insert new Element at LL head
     *
     * @param newData
     */
    public void insertAtHead(Element newData)
    {
        ListNode<Element> newnode = new ListNode<>(newData);
        if (isEmpty()) { firstNode=newnode; }
        else
        {
            newnode.next=firstNode;
            firstNode=newnode;
        }
        length++;
    }

    /**
     * returns printable version of this list
     * @return string version of the list
     */
    public String toString(){
        String toReturn = "(";
        ListNode<Element> runner = firstNode;
        while(runner != null){
            toReturn = toReturn + runner;
            runner = runner.next;
            if(runner != null){
                toReturn = toReturn + ", ";
            }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }


    /** remove and return data at the head of the
```

```
list
     *
     *  @return the Element the deleted node contains.  Returns null if list empty.
     */
    public Element removeHead()
    {
        if (!isEmpty())
        {
            ListNode<Element> secondNode = firstNode.next;
            Element deletedNodeContents = firstNode.data;
            firstNode = secondNode;
            length--;
            return deletedNodeContents;
        }
        else { return null; }

    }


    /** insert data at end of list
     *
     * @param newData new Element to be inserted
     */
    public void insertAtTail(Element newData)
    {
        if (isEmpty()) { insertAtHead(newData); }

        else
        {
            ListNode<Element> newNode = new ListNode<>(newData);
            ListNode<Element> runner = firstNode;

            while (runner.next != null)
            {
                runner = runner.next;

            }
```

```
            runner.next = newNode;
            length++;
        }

    }


    /**
     * search for first occurrence of value and return index where found
     *
     * @param value Element to search for
     * @return index where Element occurs (first node is index 0).  Return -1 if value not found.
     */
    public int indexOf(Element value)
    {
        if (isEmpty()) { return -1; }

        ListNode<Element> runner = firstNode;
        int index = 0;

        while (runner != null) {
            if (runner.data.equals(value)) { return index; }

            else
            {
                runner = runner.next;
                index++;
            }

        }
        return -1;
    }


    /**
     * returns the ith item in the list (where first item is index 0)
     * @param index
```

```
     * @return the data inside the node at given index. Return null if not there.
     */
    public Element getIthItem(int index){
        ListNode<Element> runner = firstNode;
        while (runner!=null){
            if (index==0){
                return runner.data;
            }
            else {
                runner = runner.next;
                index--;
            }
        }
        return null;
    }

    /**
     * Alias for get length, maintain convention in Java Collections.
     * @return size (amount of items in Linked List)
     */
    public int size() { return getLength(); }



    /**
     * Method to insert data at a specified index.
     * @param index Index to be inserted at
     * @param newData
     */
    public void insertAtIndex(int index, Element newData) {
        if (index < 0 || index > length) {
            throw new IndexOutOfBoundsException("Invalid index: " + index);
        }
        ListNode<Element> runner = firstNode;
        ListNode<Element> newNode = new ListNode<>(newData);
```

```java

        if (index == 0) { insertAtHead(newData); }

        else {

            for (int i = 0; i < index - 1; i++) {
                runner = runner.next;
            }

            newNode.next = runner.next;
            runner.next = newNode;
            length++;
        }


    }

    /**
     * Return a deepcopy of a linkedlist.
     */
    public LinkedList<Element> clone() {
        LinkedList<Element> newList = new LinkedList<>();
        ListNode<Element> runner = firstNode;

        while (runner != null) {
            newList.insertAtTail(runner.data);
            runner = runner.next;
        }

        return newList;
    }

    /**
     * Returns true if two objects are equal, false if not.
     */
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
```

```java
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        LinkedList<Element> otherList = (LinkedList<Element>) obj;

        if (this.size() != otherList.size()) {
            return false;
        }

        ListNode<Element> thisRunner = this.firstNode;
        ListNode<Element> otherRunner = otherList.firstNode;

        while (thisRunner != null) {
            if (!thisRunner.data.equals(otherRunner.data)) {
                return false;
            }
            thisRunner = thisRunner.next;
            otherRunner = otherRunner.next;
        }

        return true;
    }

    /**
     *
     * @param value
     * @return True if linked list contains value, false if not
     */
    public boolean contains(Element value) {
        ListNode<Element> runner = firstNode;
        while (runner != null) {
            if (runner.data.equals(value)) {
                return true;
            }
            runner = runner.next;
```

```
        }
        return false;
    }


    /**
     * Method to remove data at a specified index.
     * @param index Index to be removed
     */
    public void removeAtIndex(int index) {
        if (index < 0 || index >= length) {
            throw new IndexOutOfBoundsException("Invalid index: " + index);
        }
        if (index == 0) { removeHead(); }
        else {

            ListNode<Element> previousNode = getNodeAtIndex(index - 1);
            ListNode<Element> currentNode = previousNode.next;
            previousNode.next = currentNode.next; //skip over currentNode.
            length--;
        }


    }

    /**
     * Private helper/getter to get the node at a specified index
     * @param index given index
     * @return ListNode at given index.
     */
    private ListNode<Element> getNodeAtIndex(int index) {
        ListNode<Element> current = firstNode;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
```

```
        return current;
    }


    /**
     * Empty the LinkedList.
     */
    public void clear() {
        firstNode = null;
        length = 0;
    }



}

```

```java
package proj5;

/**
 * Generic ListNode class.
 * The ListNode class is more data-specific than the LinkedList class.  It
 * details what a single node looks like.  This node has one data field.
 * @author Neil Daterao
 * @version 3/09/2024
 */
public class ListNode<Element extends Comparable<Element>>
{

    public Element data;
    public ListNode<Element> next;

    /**
     * Constructor for a ListNode given an Element newData
     * @param newData
     */
    public ListNode(Element newData) {
        data = newData;
        next = null;
    }

    /**
     * Return a string version of the ListNode.
     */
    public String toString(){
        return data.toString();
    }


}


```

```java
package proj5;

/**
 * A FileReader object will read tokens from an input file.  The name of
 * the input file is given when the constructor is run.  The lone method,
 * nextToken(), will return the next word in the file as a String.
 *
 * @author Chris Fernandes
 * @version 2/27/17
 */
import java.io.*;

public class FileReader
{

    private StreamTokenizer st;     //file descriptor

    /**
     * @param fileName path to input file. Can either be a full
     *  path like "C:/project7/proj7_input.txt" or a relative one like
     *  "src/proj7_input.txt" where file searching begins inside
     *  the project folder.
     *  Be sure to use forward slashes to specify folders.  And
     *  don't forget the quotes (it's a String!)
     */
    public FileReader(String fileName)
    {
        try {
        st = new StreamTokenizer(
                new BufferedReader(
                    new InputStreamReader(
                        new FileInputStream(fileName))));
```

```
        } catch (IOException e) {}
        st.resetSyntax();                    // remove default rules
        st.ordinaryChars(0, Character.MAX_VALUE); // turn on all chars
        st.wordChars(65,90);
        st.wordChars(97,122);       //make the letters into tokens
        st.whitespaceChars(0, 34);
        st.whitespaceChars(36,64);  //make everything else except "#" into
        st.whitespaceChars(91,96);  //whitespace
        st.whitespaceChars(123, Character.MAX_VALUE);
    }

    /**
     * returns the next token as a String. Possible tokens are
     * words, the pound symbol "#" signifying the end of a page,
     * and null which is returned when the end of the file is reached.
     *
     * @return a word, "#", or null
     */
    public String nextToken()
    {
        try
        {
        while (st.nextToken() != st.TT_EOF) {
            if (st.ttype < 0)
            {
                return st.sval;
            }
            else
            {
                return String.valueOf((char)st.ttype);
            }
        }
```

```
        return null;
        } catch (IOException e) {}
        return "error";
    }
}

```

```java
package test;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import proj5.BSTNode;
import proj5.BinarySearchTree;

/**
 * Testing class for Binary Search Tree Abstract Data Type (ADT)
 *
 * @author Neil Daterao
 * @version 2/29/2024
 */
public class BST_Tester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private BinarySearchTree<String> bst;
    private BinarySearchTree<String> largeBst;
    private BinarySearchTree<Integer> intBst;

    @Before
    public void setUp() throws Exception {
        bst = new BinarySearchTree<>();
        intBst = new BinarySearchTree<>();
    }

    @After
    public void tearDown() throws Exception {
        bst = null;
        intBst = null;
        largeBst = null;
    }

```

```java
    private void createLargeBST() {
        largeBst = new BinarySearchTree<>();
        largeBst.insert("D");
        largeBst.insert("H");
        largeBst.insert("C");
        largeBst.insert("J");
        largeBst.insert("M");
        largeBst.insert("E");
        largeBst.insert("A");
        largeBst.insert("F");
        largeBst.insert("G");
        largeBst.insert("K");
        largeBst.insert("I");
        largeBst.insert("L");

    }


    @Test
    public void testToString() {
        bst.insert("B");
        bst.insert("A");
        bst.insert("C");
        assertEquals("should be (( A ) B ( C ))", "(( A ) B ( C ))", bst.toString());
    }

    @Test
    public void testToStringEmptyTree() {
        assertEquals("", bst.toString());
    }

    @Test
    public void testToStringOneNode() {
        bst.insert("B");
        assertEquals("( B )", bst.toString());
    }

    @Test
    public void testToStringMultipleNodes() {
        bst.insert("B");
```

```
        bst.insert("A");
        bst.insert("C");
        bst.insert("E");
        bst.insert("D");
        assertEquals("(( A ) B ( C (( D ) E )))", bst.toString());
    }

    @Test
    public void testSizeEmptyTree() {
        assertEquals(0, bst.size());
    }

    @Test
    public void testSizeOneNode() {
        bst.insert("B");
        assertEquals(1, bst.size());
    }

    @Test
    public void testSizeMultipleNodes() {
        bst.insert("B");
        bst.insert("A");
        bst.insert("C");
        assertEquals(3, bst.size());

        bst.insert("E");
        bst.insert("D");
        assertEquals(5, bst.size());
    }

    @Test
    public void testInsert_M() {
        bst.insert("M");
        assertEquals("( M )", bst.toString());
    }

    @Test
    public void testInsert_MV() {
        bst.insert("M");
        bst.insert("V");
```

```
        assertEquals("( M ( V ))", bst.toString());
    }

    @Test
    public void testInsert_MVP() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        assertEquals("( M (( P ) V ))", bst.toString());
    }

    @Test
    public void testInsert_MVPDN() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        bst.insert("D");
        bst.insert("N");
        assertEquals("(( D ) M ((( N ) P ) V ))", bst.toString());
    }

    @Test
    public void testInsert_MVPNDF() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        bst.insert("D");
        bst.insert("N");
        bst.insert("F");
        assertEquals("(( D ( F )) M ((( N ) P ) V ))", bst.toString());
    }

    @Test
    public void testInsert_MVPDNFB() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        bst.insert("D");
```

```
        bst.insert("N");
        bst.insert("F");
        bst.insert("B");
        assertEquals("((( B ) D ( F )) M ((( N ) P ) V ))", bst.toString());
    }

    @Test
    public void testInsert_MVPDNFBC() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        bst.insert("D");
        bst.insert("N");
        bst.insert("F");
        bst.insert("B");
        bst.insert("C");
        assertEquals("((( B ( C )) D ( F )) M ((( N ) P ) V ))", bst.toString());
    }

    @Test
    public void testInsert_MVPDNFBCQR() {
        bst.insert("M");
        bst.insert("V");
        bst.insert("P");
        bst.insert("D");
        bst.insert("N");
        bst.insert("F");
        bst.insert("B");
        bst.insert("C");
        bst.insert("Q");
        bst.insert("R");
        assertEquals("((( B ( C )) D ( F )) M ((( N ) P ( Q ( R ))) V ))", bst.toString());
    }

    @Test
    public void testSearchExistingValue() {
        createLargeBST();
        assertTrue("Value 'D' should exist in the
```

```
tree", largeBst.search("D"));
    }

    @Test
    public void testSearchNonExistingValue() {
        createLargeBST();
        assertFalse("Value 'X' should not exist in the tree", largeBst.search("X"));
    }

    @Test
    public void testSearchRootValue() {
        createLargeBST();
        assertTrue("Root value 'H' should exist in the tree", largeBst.search("H"));
    }

    @Test
    public void testSearchLeafs() {
        createLargeBST();
        assertTrue("Last node value 'A' should exist in the tree", largeBst.search("A"));
        assertTrue("Last node value 'C' should exist in the tree", largeBst.search("C"));
        assertTrue("Last node value 'M' should exist in the tree", largeBst.search("M"));
    }

    @Test
    public void testSearchMiddleNodeValue() {
        createLargeBST();
        assertTrue("Middle node value 'L' should exist in the tree", largeBst.search("L"));
    }

    @Test
    public void testSearchEmptyTree() {
        assertFalse("Empty Tree", bst.search("A"));

    }

```

```java
    @Test
    public void testDeleteLeafNode() {
        intBst.insert(5);
        intBst.insert(3);
        intBst.insert(7);

        intBst.delete(3);
        assertFalse(intBst.search(3));
        assertEquals("( 5 ( 7 ))", intBst.toString());
    }

    @Test
    public void testDeleteNodeWithOneChild() {
        intBst.insert(5);
        intBst.insert(3);
        intBst.insert(7);
        intBst.insert(6);

        intBst.delete(7);
        assertFalse("7 should no longer be in the BST",intBst.search(7));
        assertTrue("Should not delete child", intBst.search(6));
    }

    @Test
    public void testDeleteNodeWithTwoChildren() {
        intBst.insert(5);
        intBst.insert(3);
        intBst.insert(7);
        intBst.insert(6);
        intBst.insert(8);

        intBst.delete(7);
        assertFalse("7 should not be in the BST", intBst.search(7));
        assertTrue("Should not delete child", intBst.search(8));
        assertTrue("Should not delete child", intBst.search(6));
```

```
    }

    @Test
    public void testDeleteRootNode() {
        intBst.insert(5);
        intBst.insert(3);
        intBst.insert(7);
        intBst.insert(6);
        intBst.insert(8);

        intBst.delete(5);
        assertFalse("5 should not be in BST", intBst.search(5));
        assertTrue("Should not delete child", intBst.search(3));
        assertTrue("Should not delete child", intBst.search(7));
    }

    @Test
    public void testDeleteForItemNotThere() {
        intBst.insert(5);
        intBst.insert(3);
        intBst.insert(7);
        intBst.insert(6);
        intBst.insert(8);

        intBst.delete(9);
        assertEquals("(( 3 ) 5 (( 6 ) 7 ( 8 )))", intBst.toString());

    }

    @Test
    public void testGetExistingElement() {
        intBst.insert(10);
        intBst.insert(8);
        intBst.insert(4);

        Integer target = 8;
        BSTNode<Integer> resultNode = intBst.get(
```

```
target);
        assertNotNull(resultNode); // Check that the result is not null
        assertEquals(target, resultNode.data); // Check that the correct node is returned
    }

    @Test
    public void testGetNonExistingElement() {
        intBst.insert(10);
        intBst.insert(8);
        intBst.insert(4);

        Integer target = 2;
        BSTNode<Integer> resultNode = intBst.get(target);
        assertNull(resultNode); // Check that the result is not null

    }
}
```

```java
package test;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import proj5.Dictionary;


/**
 * Tests for Dictionary class.
 * @author Neil Daterao
 * @version 03/12/2024
 */
public class Dictionary_Tester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Dictionary dictionary;


    @Before
    public void setUp() throws Exception {
        dictionary = new Dictionary();
    }

    @After
    public void tearDown() throws Exception {
        dictionary = null;
    }

    @Test
    public void testAdd() {
        dictionary.add("apple");
        assertTrue(dictionary.contains("apple"));
    }

```

```java
    @Test
    public void testAddDuplicate() {
        dictionary.add("apple");
        dictionary.add("apple");
        assertTrue(dictionary.contains("apple"));
        assertEquals(1, dictionary.size());
    }

    @Test
    public void testAddMultiple() {
        dictionary.add("apple");
        dictionary.add("banana");
        assertTrue(dictionary.contains("apple"));
        assertTrue(dictionary.contains("banana"));
        assertEquals(2, dictionary.size());
    }

    @Test
    public void testEmptyDictionary() {
        assertTrue(dictionary.isEmpty());
    }

    @Test
    public void testNonEmptyDictionary() {
        dictionary.add("apple");
        assertFalse(dictionary.isEmpty());
    }

    @Test
    public void testToStringEmpty() {
        assertEquals("Dictionary:\n", dictionary.toString());
    }

    @Test
    public void testToStringSingleWord() {
        dictionary.add("apple");
        assertEquals("Dictionary:\napple\n", dictionary.toString());
    }

```

```java
    @Test
    public void testToStringMultipleWords() {
        dictionary.add("apple");
        dictionary.add("banana");
        dictionary.add("cherry");
        assertEquals("Dictionary:\napple\nbanana\n cherry\n", dictionary.toString());
    }

    @Test
    public void testToStringWithDuplicates() {
        dictionary.add("apple");
        dictionary.add("apple");
        dictionary.add("banana");
        assertEquals("Dictionary:\napple\nbanana\n", dictionary.toString());
    }



}

```

```java
package test;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import proj5.*;

/**
 * Testing class for Index. Contains various edge cases and tests for basic functionality
 * @author Neil Daterao
 * @version 03/09/2024
 */
public class Index_Tester {

    @Rule
    public Timeout timeout = Timeout.millis(100);


    private Index index;
    private final String testTextFile = "src/resources/test.txt";


    @Before
    public void setUp() throws Exception {
       index = new Index(testTextFile);
       index.createIndex();
    }

    @After
    public void tearDown() throws Exception {
        index = null;
    }

    @Test
    public void testCreateIndexCourseNotInIndex(){
```

```
        assertFalse("Index does not contain course", index.contains("course"));
        assertFalse("Index does not contain Course", index.contains("Course"));

        assertTrue("Dictionary contains course", index.getDictionary().contains("course"));

    }

    @Test
    public void testCreateIndexCoursesInIndex() {
        assertTrue("Index contains courses",index.contains("courses"));
        assertTrue("Courses shown on page 6", index.getIndexEntry("courses").getPageList().containsPage(6));
        assertTrue("Courses shown on page 12", index.getIndexEntry("courses").getPageList().containsPage(6));
    }


    @Test
    public void testCreateIndexChrisNotInIndex(){
        assertTrue("Index contains 'Chris'", index.contains("Chris"));
        assertTrue("Courses shown on page 3", index.getIndexEntry("Chris").getPageList().containsPage(3));
        assertFalse("Dictionary does not contain 'Chris'", index.getDictionary().contains("Chris"));
    }

    @Test
    public void testCreateIndexComputerInIndex(){
        assertTrue("Index contains 'computer'", index.contains("computer"));
        assertTrue("computer shown on page 2", index.getIndexEntry("computer").getPageList().containsPage(2));
```

```
        assertTrue("computer shown on page 6", index.getIndexEntry("computer").getPageList().containsPage(6));
        assertTrue("computer shown on page 10", index.getIndexEntry("computer").getPageList().containsPage(10));
        assertFalse("Dictionary does not contain 'computer'", index.getDictionary().contains("computer"));
    }

    @Test
    public void testCreateIndexTakingInIndex(){
        assertTrue("Index contains 'taking'", index.contains("taking"));
        assertFalse("Dictionary does not contains 'taking'", index.getDictionary().contains("taking"));
    }

    @Test
    public void testMergeOfUpperCaseWordWithLowerCaseInIndex() {
        assertTrue("Index contains 'computer'", index.contains("computer"));
        assertTrue("computer shown on page 6, this was Computer originally", index.getIndexEntry("computer").getPageList().containsPage(6));
        assertFalse("Index does not contain 'Computer'", index.contains("Computer"));
        assertFalse("Dictionary does not contain 'Computer'", index.getDictionary().contains("Computer"));
    }

    @Test public void testMergeOfLowerCaseWordWithUpperCaseInIndex(){
        assertTrue("Index contains 'the'", index.contains("the"));
        assertTrue("the shown on page 4, this was The originally", index.getIndexEntry("the").
```

```
getPageList().containsPage(4));
        assertFalse("Index does not contain 'The'", index.contains("The"));
        assertFalse("Dictionary does not contain 'The'", index.getDictionary().contains("The"));
    }


    @Test
    public void testCreateIndexTheInIndexWithFullPageList(){
        assertTrue("Index contains 'the'", index.contains("the"));
        assertEquals(4, index.getIndexEntry("the").getPageList().size());
    }


    @Test
    public void testWordsLessThan2CharsAreNotAddedToIndex() {
        String indexContent = index.toString();

        String[] lines = indexContent.split("\\r?\\n");
        for (String line : lines) {
            String word = line.split("\\s+")[0];
            assertTrue(word.length() > 2);
        }

    }

    @Test
    public void testMultipleWordsOnSamePageOnlyCountsOnce(){
        assertTrue("Index contains 'boggle'", index.contains("boggle"));
        assertEquals(1, index.getIndexEntry("boggle").getPageList().size());
        assertTrue("Boggle shown on page 13 4 times, 13 should only be added once", index.
```

```
getIndexEntry("the").getPageList().containsPage(13));
    }


    @Test
    public void testIndexAlphabeticalOrder() {
        String previousWord = null;
        String indexContent = index.toString();

        String[] lines = indexContent.split("\\r?\\n");

        for (String line : lines) {
            String word = line.split("\\s+")[0];
            if (previousWord != null) {
                assertTrue("'" + word + "' should come after '" + previousWord + "'",
                           word.compareTo(previousWord) >= 0);
            }
            previousWord = word;
        }
    }


}

```

```java
package test;

import org.junit.*;
import org.junit.rules.Timeout;
import static org.junit.Assert.*;

import proj5.LinkedList;

/**
 * LinkedList testing class for all methods in class. Tests generic linked list.
 * @author Neil Daterao
 * @version 2/29/2024
 */

public class LL_Tester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private LinkedList<String> stringLL;
    private LinkedList<Integer> intLL;

    @Before
    public void setUp() throws Exception {
        stringLL = new LinkedList<>();
        intLL = new LinkedList<>();
    }

    @After
    public void tearDown() throws Exception {
      stringLL = null;
      intLL = null;
    }



    @Test
    public void testRemoveHead() {

        stringLL.insertAtHead("A");
```

```
        stringLL.insertAtHead("B");
        stringLL.insertAtHead("C");
        stringLL.insertAtHead("D");
        stringLL.insertAtHead("E");
        stringLL.insertAtHead("F");
        stringLL.insertAtHead("G");

        stringLL.removeHead();

        assertEquals("(F, E, D, C, B, A)", stringLL.toString());
        assertEquals(6, stringLL.getLength());

    }

    @Test
    public void testRemoveHeadFromListWithOneElement() {

        stringLL.insertAtHead("A");

        assertEquals("A", stringLL.removeHead());
        assertTrue(stringLL.isEmpty());
    }

    @Test
    public void testRemoveHeadFromEmptyList() {
        // Create an empty linked list


        // Test removing from an empty list
        assertNull(stringLL.removeHead());
    }

    @Test
    public void testInsertAtTailInEmptyList() {
        // Create an empty linked list


        // Insert at the tail of an empty list
        stringLL.insertAtTail("A");
```

```java

        // Verify the list after insertion
        assertEquals("(A)", stringLL.toString());
        assertEquals(1, stringLL.getLength());
    }

    @Test
    public void testInsertAtTailInListWithOneElement() {
        stringLL.insertAtHead("A");

        // Insert at the tail of a list with one element
        stringLL.insertAtTail("B");

        // Verify the list after insertion
        assertEquals("(A, B)", stringLL.toString());
        assertEquals(2, stringLL.getLength());
    }

    @Test
    public void testInsertAtTailInListWithMultipleElements() {
        stringLL.insertAtHead("A");
        stringLL.insertAtTail("B");
        stringLL.insertAtTail("C");
        stringLL.insertAtTail("D");
        stringLL.insertAtTail("E");
        stringLL.insertAtTail("F");
        stringLL.insertAtTail("G");

        // Verify the list after insertion
        assertEquals("(A, B, C, D, E, F, G)", stringLL.toString());
        assertEquals(7,stringLL.getLength());
    }


    @Test
    public void testIndexOfInEmptyList() {
```

```java

        assertEquals(-1, intLL.indexOf(1));
        assertEquals(-1, intLL.indexOf(2));
        assertEquals(-1, intLL.indexOf(3));
    }

    @Test
    public void testIndexOfInListWithOneElement() {

        intLL.insertAtHead(1);

        assertEquals(0, intLL.indexOf(1));
        assertEquals(-1, intLL.indexOf(2));
        assertEquals(-1, intLL.indexOf(3));
    }

    @Test
    public void testIndexOfInListWithMultipleElements() {

        intLL.insertAtHead(1);
        intLL.insertAtTail(2);
        intLL.insertAtTail(3);
        intLL.insertAtTail(4);
        intLL.insertAtTail(5);
        intLL.insertAtTail(6);
        intLL.insertAtTail(7);

        assertEquals(0, intLL.indexOf(1));
        assertEquals(1, intLL.indexOf(2));
        assertEquals(2, intLL.indexOf(3));
        assertEquals(3, intLL.indexOf(4));
        assertEquals(4, intLL.indexOf(5));
        assertEquals(6, intLL.indexOf(7));
        assertEquals(-1, intLL.indexOf(8));


    }

    @Test
    public void testIndexOfWithDuplicateValues() {
```

```java

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(1);

        // Test indexOf with duplicate values
        assertEquals(0, intLL.indexOf(1));
        assertEquals(1, intLL.indexOf(2));
        assertEquals(-1, intLL.indexOf(3));
    }

    @Test
    public void testInsertAtIndex() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);

        // Test inserting at index 0
        intLL.insertAtIndex(0, 4);
        assertEquals("(4, 3, 2, 1)", intLL.toString());

        // Test inserting in the middle
        intLL.insertAtIndex(2, 5);
        assertEquals("(4, 3, 5, 2, 1)", intLL.toString());

        // Test inserting at the end
        intLL.insertAtIndex(5, 6);
        assertEquals("(4, 3, 5, 2, 1, 6)", intLL.toString());

        // Test inserting at index equal to length (append)
        intLL.insertAtIndex(6, 7);
        assertEquals("(4, 3, 5, 2, 1, 6, 7)", intLL.toString());
    }

    @Test
```

```
    public void testInsertAtIndexEmptyList() {

        intLL.insertAtIndex(0, 1);
        assertEquals("(1)", intLL.toString());
    }

    @Test
    public void testInsertAtIndexInvalidIndex() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);

        // Test inserting at index out of bounds
        assertThrows(IndexOutOfBoundsException.class, () -> {
            intLL.insertAtIndex(3, 3);
        });

        assertThrows(IndexOutOfBoundsException.class, () -> {
            intLL.insertAtIndex(-1, 3);
        });


    }

    @Test
    public void testGetIthItem() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);

        // Test getting the first item (index 0)
        assertEquals((Integer)3, intLL.getIthItem(0));

        // Test getting an item in the middle
        assertEquals((Integer)2, intLL.getIthItem(1));

```

```
        // Test getting the last item
        assertEquals((Integer)1, intLL.getIthItem(2));
    }

    @Test
    public void testGetIthItemIndexOutOfBounds() {

        intLL.insertAtHead(1);

        // Test getting item at index out of bounds
        assertNull(intLL.getIthItem(1));
    }

    @Test
    public void testGetIthItemEmptyList() {

        assertNull(intLL.getIthItem(0));
    }

    @Test
    public void testClone() {
        LinkedList<String> originalList = new LinkedList<>();
        originalList.insertAtHead("A");
        originalList.insertAtHead("B");
        originalList.insertAtHead("C");

        // Clone the original list
        LinkedList<String> clonedList = originalList.clone();

        // Test that the cloned list is not the same object reference as the original list
        assertNotSame(originalList, clonedList);

        // Test that the data of each node in the cloned list matches the data of the corresponding node in the original list
        assertEquals(originalList.toString(), clonedList.toString());
```

```java

        LinkedList<Integer> emptyList = new LinkedList<>();
        LinkedList<Integer> clonedEmptyList = emptyList.clone();

        assertEquals(emptyList.toString(), clonedEmptyList.toString());
    }

    @Test
    public void testEqualsSameObject() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);

        assertTrue(intLL.equals(intLL)); // Same object should be equal
    }

    @Test
    public void testEqualsNullObject() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);

        assertFalse(intLL.equals(null)); // Null object should not be equal
    }

    @Test
    public void testEqualsDifferentClass() {

        stringLL.insertAtHead("A");
        stringLL.insertAtHead("B");
        stringLL.insertAtHead("C");

        assertFalse(stringLL.equals("hello")); // Different class should not be equal
```

```
    }

    @Test
    public void testEqualsDifferentLists() {
        LinkedList<String> list1 = new LinkedList<>();
        list1.insertAtHead("A");
        list1.insertAtHead("B");
        list1.insertAtHead("C");

        LinkedList<String> list2 = new LinkedList<>();
        list2.insertAtHead("X");
        list2.insertAtHead("Y");
        list2.insertAtHead("Z");

        assertFalse(list1.equals(list2)); // Different lists should not be equal
    }

    @Test
    public void testEqualsEqualLists() {
        LinkedList<Integer> list1 = new LinkedList<>();
        list1.insertAtHead(1);
        list1.insertAtHead(2);
        list1.insertAtHead(3);

        LinkedList<Integer> list2 = new LinkedList<>();
        list2.insertAtHead(1);
        list2.insertAtHead(2);
        list2.insertAtHead(3);

        assertTrue(list1.equals(list2)); // Equal lists should be equal
    }


    @Test
    public void testEqualsDifferentLLTypes(){
```

```
        LinkedList<Integer> list1 = new LinkedList<>();
        list1.insertAtHead(1);
        list1.insertAtHead(2);
        list1.insertAtHead(3);

        LinkedList<String> list2 = new LinkedList<>();
        list2.insertAtHead("X");
        list2.insertAtHead("Y");
        list2.insertAtHead("Z");

        assertFalse(list1.equals(list2));

    }



    @Test
    public void testEqualsUnequalLengthLists() {
        LinkedList<String> list1 = new LinkedList<>();
        list1.insertAtHead("A");
        list1.insertAtHead("B");
        list1.insertAtHead("C");

        LinkedList<String> list2 = new LinkedList<>();
        list2.insertAtHead("A");
        list2.insertAtHead("B");

        assertFalse(list1.equals(list2)); // Lists of unequal length should not be equal
    }

    @Test
    public void testEqualsEmptyLists() {
        LinkedList<Integer> list1 = new LinkedList<>();
        LinkedList<Integer> list2 = new LinkedList<>();
```

```java

        assertTrue(list1.equals(list2)); // Empty lists should be equal
    }

    @Test
    public void testContainsEmptyList() {

        assertFalse(intLL.contains(1)); // Empty list should not contain any value
    }

    @Test
    public void testContainsSingleElement() {

        stringLL.insertAtHead("A");
        assertTrue(stringLL.contains("A")); // stringLL with a single element should contain that value
        assertFalse(stringLL.contains("B")); // stringLL with a single element should not contain other values
    }

    @Test
    public void testContainsMultipleElements() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);

        assertTrue(intLL.contains(1));
        assertTrue(intLL.contains(2));
        assertTrue(intLL.contains(3));
        assertFalse(intLL.contains(4));
    }

    @Test
    public void testContainsDuplicateElements() {

        stringLL.insertAtHead("A");
```

```
        stringLL.insertAtHead("A");

        assertTrue(stringLL.contains("A")); // List with duplicate elements should contain "A"
    }


    @Test
    public void testRemoveAtIndexFromEmptyList() {

        assertThrows(IndexOutOfBoundsException.class, () -> {
            intLL.removeAtIndex(0);
        });
    }

    @Test
    public void testRemoveAtIndexOutOfBounds() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);
        assertThrows(IndexOutOfBoundsException.class, () -> {
            intLL.removeAtIndex(3); // Index out of bounds
        });
    }

    @Test
    public void testRemoveAtIndexFirstNode() {

        stringLL.insertAtHead("A");
        stringLL.insertAtHead("B");
        stringLL.insertAtHead("C");
        stringLL.removeAtIndex(0);
        assertEquals("(B, A)", stringLL.toString());
    }

    @Test
```

```
    public void testRemoveAtIndexLastNode() {

        stringLL.insertAtHead("A");
        stringLL.insertAtHead("B");
        stringLL.insertAtHead("C");
        stringLL.removeAtIndex(2);
        assertEquals("(C, B)", stringLL.toString
    ());
    }

    @Test
    public void testRemoveAtIndexMiddleNode() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.insertAtHead(3);
        intLL.removeAtIndex(1);
        assertEquals("(3, 1)", intLL.toString());
    }


    @Test
    public void testClearEmptyList() {

        intLL.clear();
        assertTrue(intLL.isEmpty());
        assertEquals(0, intLL.getLength());
    }

    @Test
    public void testClearNonEmptyList() {

        stringLL.insertAtHead("A");
        stringLL.insertAtHead("B");
        stringLL.insertAtHead("C");
        stringLL.clear();
        assertTrue(stringLL.isEmpty());
        assertEquals(0, stringLL.getLength());
    }

    @Test
```

```
    public void testClearMultipleTimes() {

        intLL.insertAtHead(1);
        intLL.insertAtHead(2);
        intLL.clear();
        intLL.clear();
        assertTrue(intLL.isEmpty());
        assertEquals(0, intLL.getLength());
    }



}
```

```java
package test;

import static org.junit.Assert.*;

import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

import proj5.PageList;

public class PageList_Tester {

    @Rule
    public Timeout timeout = Timeout.millis(100);

    private PageList pageList;
    private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    private final PrintStream originalOut = System.out;


    @Before
    public void setUp() throws Exception {
        pageList= new PageList();
        System.setOut(new PrintStream(outContent));
    }

    @After
    public void tearDown() throws Exception {
        pageList = null;
        System.setOut(originalOut);
    }


    @Test
```

```java
    public void testAddPage() {
        pageList.addPage(1);
        assertTrue(pageList.containsPage(1));
        assertEquals(1, pageList.size());
        pageList.addPage(2);
        assertTrue(pageList.containsPage(2));
        assertEquals(2, pageList.size());
        assertEquals("{1-2}", pageList.toString());
    }

    @Test
    public void testRemovePage() {
        pageList.addPage(1);
        pageList.removePage(1);
        assertFalse(pageList.containsPage(1));
        assertEquals(0, pageList.size());

        pageList.addPage(1);
        pageList.addPage(2);
        pageList.addPage(3);
        pageList.removePage(1);
        assertFalse(pageList.containsPage(1));
        assertEquals(2, pageList.size());
        assertEquals("{2-3}", pageList.toString());
    }

    @Test
    public void testRemovePageNonExisting() {
        pageList.removePage(1);
        assertEquals("Page not in PageList\n", outContent.toString());
    }


    @Test
    public void testContainsPage() {
        assertFalse(pageList.containsPage(1));
        pageList.addPage(1);
        assertTrue(pageList.containsPage(1));
    }

```

```
    @Test
    public void testContainsPageMultiple() {
        pageList.addPage(1);
        pageList.addPage(2);
        pageList.addPage(3);
        assertTrue(pageList.containsPage(2));
    }

    @Test
    public void testContainsPageMultipleNotThere() {
        pageList.addPage(1);
        pageList.addPage(2);
        pageList.addPage(3);
        assertFalse(pageList.containsPage(4));
    }

    @Test
    public void testSize() {
        assertEquals(0, pageList.size());
        pageList.addPage(1);
        assertEquals(1, pageList.size());
    }

    @Test
    public void testIsEmpty() {
        assertTrue(pageList.isEmpty());
        pageList.addPage(1);
        assertFalse(pageList.isEmpty());
    }

    @Test
    public void testAddPageList() {
        PageList otherPageList = new PageList();
        otherPageList.addPage(1);
        otherPageList.addPage(2);
        otherPageList.addPage(3);

        pageList.addPageList(otherPageList);

        assertTrue(pageList.containsPage(1));
```

```
        assertTrue(pageList.containsPage(2));
        assertTrue(pageList.containsPage(3));
        assertEquals(3, pageList.size());

        PageList thirdPageList = new PageList();
        thirdPageList.addPageList(otherPageList);
        thirdPageList.addPageList(otherPageList);
        assertEquals(6, thirdPageList.size());
        assertEquals("{1-3, 1-3}", thirdPageList.toString());
    }

    @Test
    public void testAddPageListPageListNotChanged(){
        PageList otherPageList = new PageList();
        otherPageList.addPage(1);
        otherPageList.addPage(2);
        otherPageList.addPage(3);

        pageList.addPageList(otherPageList);

        assertTrue(pageList.containsPage(1));
        assertTrue(pageList.containsPage(2));
        assertTrue(pageList.containsPage(3));
        assertEquals(3, pageList.size());


        assertTrue(otherPageList.containsPage(1));
        assertTrue(otherPageList.containsPage(2));
        assertTrue(otherPageList.containsPage(3));
        assertEquals(3, otherPageList.size());

    }

    @Test
    public void testToString() {
        pageList.addPage(1);
        pageList.addPage(2);
        pageList.addPage(4);
        pageList.addPage(5);
```

```
        pageList.addPage(6);
        pageList.removePage(4);

        assertEquals("{1-2, 5-6}", pageList.toString());
    }

    @Test
    public void testToStringNoDashes() {
        pageList.addPage(2);
        pageList.addPage(4);
        pageList.addPage(6);
        pageList.addPage(8);
        pageList.addPage(10);

        assertEquals("{2, 4, 6, 8, 10}", pageList.toString());
    }

}

```