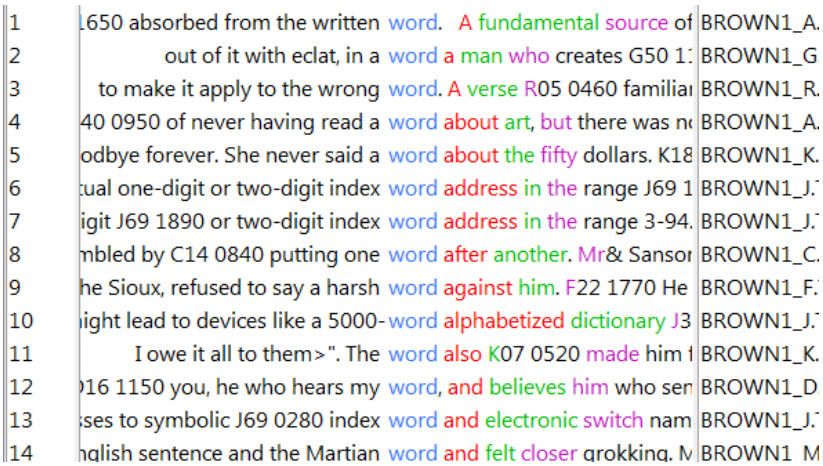
Erika Parayno
November 18, 2019
CSS 501
Program 4 Design

Program 4: Concordance

Problem statement: Similar to an index, a concordance is an alphabetical list of distinct words in a body of text, with every instance of each word shared with its immediate context. While indices only contain words that correspond to important concepts, concordance shows every instance of the words in a text, possibly excluding trivial words such as “a”, “an”, “the”, etc.



1	L650 absorbed from the written	word. A fundamental source of	BROWN1_A.
2	out of it with eclat, in a	word a man who creates G50 1	BROWN1_G
3	to make it apply to the wrong	word. A verse R05 0460 familia	BROWN1_R.
4	40 0950 of never having read a	word about art, but there was n	BROWN1_A.
5	odbye forever. She never said a	word about the fifty dollars. K18	BROWN1_K.
6	tual one-digit or two-digit index	word address in the range J69 1	BROWN1_J.
7	igit J69 1890 or two-digit index	word address in the range 3-94.	BROWN1_J.
8	mbled by C14 0840 putting one	word after another. Mr& Sanso	BROWN1_C.
9	he Sioux, refused to say a harsh	word against him. F22 1770 He	BROWN1_F.
10	ight lead to devices like a 5000-	word alphabetized dictionary J3	BROWN1_J.
11	I owe it all to them>". The	word also K07 0520 made him	BROWN1_K.
12	16 1150 you, he who hears my	word, and believes him who se	BROWN1_D
13	ses to symbolic J69 0280 index	word and electronic switch nam	BROWN1_J.
14	nglish sentence and the Martian	word and felt closer grokking. M	BROWN1_M

Figure 1. Screenshot from AntConc program

Purpose: The purpose of this program is to read a corpus text file and output a concordance in KWIC format, as demonstrated in the above illustration. The program will exclude a specified set of stop words, stored in a text file and located in the same directory as the program.

The program flow is:

1. read corpus text file from command line, using an array as fixed window reading frame
2. create a binary tree containing nodes with key and adding occurrence as first node on linked list
3. any occurrence of word will be stored as additional LinkedList node of the linkedlist in that BSTNode
4. then format and print each keyword in alphabetical order and content of linkedlist in the order it was added.

Relevant structures and algorithms:

- *Reading from command line*: To pass command line arguments, we typically define main() with number of command line arguments (argc) and list of

command-line arguments. So if we pass a value to a program, value of argc would be 2 as there is one for argument and one for file name.

- *Binary tree*: Binary trees are most use in storing hierarchical structures, file structures, and organizational structures. Binary trees hold nodes. The root node is located on the top of the tree. In a binary tree, a parent has no more than two branches extending to each child. The branches are arranged so that the left child is less than the parent root while the right child is greater. This structure will be used to store unique words from the text file.
    - *BST Traversal*: There are 3 common traversals. In this assignment, we will be using preorder (neutral, left, and right). Each node will be visited before either of the children, then the left subtree and right subtree are traversed in that order.
        - *Recursion*: Recursion will be used to traverse through the tree. For example, we will be using this on the clear and insert method of the BST. Its recursive because it is passing current node's two children into itself, and its children's children into themselves until they reach an end node. They will have to wait until the most inner call (aka the leaf) node has returned a Boolean value before the most outer call is completed.

- *Linked list*: Recall, linked list is a structure that represents a sequence of nodes. The order of the nodes is not given by physical placement, but its position relative to its neighbor. In this program, we will be using a singly linked list, as the data will be inputted in the direction, and read in the same direction. This structure will be used to store each occurrence of a specific word and is in a node of the binary search tree structure. Each linked list node will contain pre-context, post-context, and information about the next node.

- *Array*: Arrays will be used as fixed window reading frames to store pre- and post-context adjacent to the key word in the text file. It will hold five string values before and after the key word, totaling in an array containing 11 strings (index 0-10). If pre- or post- context contain less than 5 string values, some of the array will contain null values. A series of array illustrations shown below depicts reading frame behavior. If the string located at index 5 is a non-stop word, we will create a new node with key equal to string in index 5 and store its pre and post context in a new node added to the linked list node for that key.

					Fourscore	and	seven	years	ago	our

				Fourscore	and	seven	years	ago	our	fathers

			Fourscore	and	seven	years	ago	our	fathers	brought

- *Ostream formatting:* We'll make ostream modifications so it produces output similar to printf in JAVA. Tentative plan is to calculate the maximum character length for all pre-context and post-context and set that as ostream width. In order

to format pre-context so that it is aligned to the right, we will use std::right ostream fill with " " until desired width is reached. Meanwhile, the key and post-context column will use std::left.

Input data		
Name & Description	Variable Type	Assumptions
file.txt: This will be the corpus text file prompted from command line as an argument after the executable.	String (text file)	- May or may not contain punctuation - Likely to contain "stop words" - Would be of "reasonable" size.
stopwords.txt: It contains identified stop words such as "a", "as", "the", etc. In the program, these will be excluded in the BST structure, but included in the Linked List Nodes.	String (text file)	- Each stop word will be separated by an end-line. - Everything can be simplified to all lowercase

An example input from command line may look like:

```
./ex file.txt
```

An example file.txt file may look like (truncated):

```
Fourscore and seven years ago our fathers brought forth, on this
continent, a new nation, conceived in liberty, and dedicated to
the proposition that all men are created equal.
```

An example stopwrods.txt file may look like:

```
A
An
The
On
Are
```

Output data		
Name & Description	Variable Type	Constraints

Concordance: The output showcases each distinct word on cout, with each occurrence. Adjacent to each occurrence of the word, is the pre- and post-context (5 words before are left and 5 words after are right to the key word).	String	- Should be formatted with word and context aligned properly (as demonstrated on illustration in problem statement & below).

An example output to cout may look like (truncated):

```
                        Fourscore  and         seven years ago our fathers
new nation conceived in liberty  and         dedicated to the proposition that
                                 fourscore   and seven years ago our
                   Fourscore and  seven       years ago our fathers brought
                Fourscore and seven  years    ago our fathers brought forth
```

Error Handling:

- The program must handle errors and display appropriate messages for each error.
- The program must verify that the file.txt to be read exists and is not empty.
- The final BST must contain that each non-stop word in a BST, containing key and all of the key's occurrences.
- The binary tree must preserve order characteristic of a binary tree throughout the whole program.
- The concordance display must be shown in KWIC format without manipulating data and data source.
- Valgrind results must return no leaks possible and no suppressed errors.

Implementation Plan:

Class: BST (Binary Search Tree)		
Name & description	Tentative implementation (pseudo-code)	Test plan
BST: Default constructor.	`BST::BST(){}`	- If no value was set to a node, it should contain null.

~BST: Default destructor clears	`BST::~BST(){`
	`  // call clear with root`
	`}`
		- Valgrind results should indicate

and deletes entire tree by deleting the tree node.		program is free of memory leaks.

Add: Adds a string value into the binary tree, placing it in an appropriate location. If tree does yet exist, the method will set it as root.	`bool BST::add(string value){`
	`  if (root is null) {`
	`  // create root`
	`  // set root key to value`
	`  // return true`
	`  } else {`
	`  // calls insert starting at root`
	`  }`
	`}`
		- Attempt to add to a null root should create a root with that value
		- If tree root is not null, method should return the same value as insert call.

Insert: Recursive function that will continue to call itself until it has found the correct parent node. The parent node must have less than 2 children for the insertion to occur.	`bool BST::insert(Node* &subTree, string value) {`
	`  if (reached an empty child node) {`
	`  // set child node key to value`
	`  // return true;`
	`  }`
	`  if (value already exists in BST) {`
	`  // return false;`
	`  }`
	
	`  if (subTree is greater than value){`
	`  // call insert to left of subTree`
	`  } else {`
	`  // call insert to right of subTree`
	`  }`
	`}`
		- Binary search tree order should be preserved after each insertion.
		- Method should return false if value was not added on to the tree.

Clear: A recursive function to clear values in node and delete its individual subtrees. Uses post order delete traversal.	`Void BST::clear(Node* &subtree) {`
	`  If (has no children) {`
	`  // return;`
	`  }`
	`  If (left child has value) {`
	`  // call clear on left subtree`
	`  }`
	`  If (right child has value) {`
	`  // call clear on right subtree`
	`  }`
	`  // delete subtree`
	`}`
		- Entire subtree should be deleted, without affecting other subtrees in binary tree.
		- If subtree passed is root, entire tree should be cleared and deleted.

Class: BSTNode (Binary Search Tree Node)		
Name & Description	Tentative implementation (pseudo-code)	Test plan

getKey: returns string variable key.	`string BSTNode::getKey() const {`
	`  // return key`
	`}`
		- Should return key as stored in node.

setKey: sets string variable key as	`void BSTNode::setKey(const string & value) {` `  If (contains punctuation) {`	- No key should contain punctuations.

parameter value	`// modify value using substring`
	`}`
	`// set key to value`
	`}`	

setRightChild: sets rightChild as parameter node pointer.	`void BSTNode::setRightChild(Node *n) {`
	`rightChild = n;`
	`}`
		- Can ensure this works by calling getRightChild and see if it matches node pointer parameter

getRightChild: returns node pointer variable rightChild	`Node* BSTNode::getRightChild() const {`
	`// return rightChild`
	`}`
		- Can confirm that the return value is rightChild by looking at BST structure

setLeftChild: sets leftChild as parameter node pointer.	`void BSTNode::setLeftChild(Node *n) {`
	`leftChild = n;`
	`}`
		- Can ensure this works by calling getLeftChild and see if it matches node pointer parameter

getLeftChild: returns node pointer variable left child	`Node* BSTNode::getLeftChild() const {`
	`// return leftChild`
	`}`
		- Can confirm that the return value is leftChild by looking at BST structure

getOccurences: returns pointer to occurences LinkedList	`LinkedList* BSTNode::getOccurences() const{`
	`// return occurrences`
	`}`
		- Must return pointer pointing at beginning of linked List, thus should able to iterate through structure with Linked List implementation.

Class: LinkedList		
Name & Description	Tentative implementation (pseudo-code)	Test plan

LinkedList: Default constructor sets headptr as nullptr and itemCount as 0.	`LinkedList::LinkedList() {`
	`// set headPtr as nullptr;`
	`// set itemCount equal to 0;`
	`}`
		- Default linked list should contain nullptr as headptr and 0 item count.

LinkedList: Constructor that accepts a string	`LinkedList::LinkedList(string preContent, string postContent) {`
	`// create new node`
	`// set new node pre and post content`
		- Since at the creation of each node, we already have the

content as parameter. This will the constructor used.	`// set node as headptr`
	`// set itemCount equal to 1`
	`}`
		information to create the first occurence. All linkedlist in the tree must contain headPtrs that are not equal to null.

getItemCount: Returns member variable itemCount.	`Int LinkedList::getItemCount(){`
	`// returns itemCount;`
	`}`
		- itemCount should equal to number of nodes in the linkedList at that given time

add: Iterates through linkedList until it reaches the last node and inserts new node at the end of the linked list.	`void LinkedList::add(string preContent, string postContent) {`
	`// create new node ptr`
	`While ( node->next is not null) {`
	`// iterate through linked list`
	`}`
	`// create new node`
	`// set new node pre and post content`
	`// set nodeptr next as new node`
	`}`
		- This method should add new node to the end of the list
		- The occurrences in the linked list should be in the order it was read in the text file.

Class: LinkedListNode		
Name & Description	Tentative implementation (pseudo-code)	Test plan

setPreContent: sets member variable preContent to variable passed as parameter.	`Void LinkedListNode::setPreContent(string content) {`
	`// set precontent to content`
	`}`
		- preContext should show identical string to content

getPreContent: returns string member variable preContent.	`String LinkedListNode::getPreContent(){`
	`// return preContent`
	`}`
		-preContent should accurately depict content in text file
		-cannot contain more than 5 strings

setPostContent: sets member variable postContent to variable passed as parameter.	`Void LinkedListNode::setPostContent(string content) {`
	`// set postContent to content`
	`}`
		- postContent should show identical string to content

getPostContent:	`String LinkedListNode::getPostContent(){`	-preContent should

returns string member variable postContent.	`// return postContent` `}`	accurately depict content in text file -cannot contain more than 5 strings

UML Diagrams: