Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Курский государственный университет»

Кафедра программного обеспечения и администрирования информационных систем

Направление подготовки математическое обеспечение и администрирование информационных систем

Форма обучения очная

**Отчет**

**по лабораторной работе №4**

«Линейные списки»

Выполнил:

студент группы 213 Файтельсон А.А.

Проверил:

ассистент кафедры ПОиАИС Овсянников А.В.

Курск, 2024

**Цель работы:** Научиться составлять процедуры (функции), реализующие операции для работы с линейными списками.

**Постановка задачи:**

1. Изучить описание двусвязного списка.
2. Программно реализовать двусвязный список и методы работы с ним.
3. Программно пересечение двусвязного
4. Составить программу для лексического анализа текста, которая должна для данного текстового файла T найти все различные слова и указать количество их повторений, а также номер первой строки, в которой они встречаются.списка.

Таблица 1 – Индивидуальный вариант

|  |  |
| --- | --- |
| Вариант 5 | 4. Объединение двух двусвязных списков. |
| 5. Программа для лексического анализа текста |

**Алгоритмы решения задачи в текстуальном виде.**

Двусвязный список позволяет двигаться по цепочке элементов в обоих направлениях, поскольку доступны следующий и предыдущий элементы.

Пересечение двусвязных списков:

1. Создать пустое множество (или хэш-таблицу) для хранения уникальных узлов списка A.
2. Пройти по списку A и добавить каждый узел в множество.
3. Затем пройти по списку B и для каждого узла проверять, содержится ли он в множестве.
4. Если узел содержится в множестве, добавить его в результирующий список пересечения.

Программа для лексического анализа текста:

1. Открытие файла: Открыть текстовый файл для чтения.
2. Инициализация структуры данных:
   1. Создать словарь для хранения слов и их статистики (количество повторений и номер первой строки).
3. Чтение файла построчно:
   1. Разделить строку на слова, используя функцию, которая игнорирует символы, не являющиеся буквами.
   2. Привести каждое слово к нижнему регистру.
4. Обработка слов:
   1. Для каждого слова проверить, есть ли оно в словаре, если нет то, добавить, иначе увеличить счетчик

**Обоснование правильности выбора алгоритма**

Эффективную работу программ обеспечивает:

1. Линейная сложность: Алгоритм проходит по тексту один раз, что обеспечивает линейную сложность O(n), где n — количество символов в тексте.
2. Использование стандартных библиотек: Использование стандартных библиотек Go (например, bufio и strings) упрощает работу с вводом/выводом и обработкой строк.
3. Удобство представления данных: Хранение информации о словах в виде словаря и связного списка позволяет быстро получать доступ к данным и эффективно управлять ими.

**Листинг программы**

**package main**

**import (**

**"fmt"**

**"slices"**

**"strings"**

**"github.com/emirpasic/gods/v2/utils"**

**)**

**// Assert List implementation**

**// List holds the elements, where each element points to the next and previous element**

**// type List[T comparable] struct {**

**// first \*element[T]**

**// last \*element[T]**

**// size int**

**// }**

**//**

**// type element[T comparable] struct {**

**// value T**

**// prev \*element[T]**

**// next \*element[T]**

**// }**

**//**

**// // New instantiates a new list and adds the passed values, if any, to the list**

**// func New[T comparable](values ...T) \*List[T] {**

**// list := &List[T]{}**

**// if len(values) > 0 {**

**// list.Add(values...)**

**// }**

**// return list**

**// }**

**// Add appends a value (one or more) at the end of the list (same as Append())**

**func (list \*List[T]) Add(values ...T) {**

**for \_, value := range values {**

**newElement := &element[T]{value: value, prev: list.last}**

**if list.size == 0 {**

**list.first = newElement**

**list.last = newElement**

**} else {**

**list.last.next = newElement**

**list.last = newElement**

**}**

**list.size++**

**}**

**}**

**func (list \*List[T]) Merge(values ...T) {**

**for \_, value := range values {**

**newElement := &element[T]{value: value, prev: list.last}**

**if list.size == 0 {**

**list.first = newElement**

**list.last = newElement**

**} else {**

**if !list.Contains(value) {**

**list.last.next = newElement**

**list.last = newElement**

**}**

**}**

**list.size++**

**}**

**}**

**// Append appends a value (one or more) at the end of the list (same as Add())**

**func (list \*List[T]) Append(values ...T) {**

**list.Add(values...)**

**}**

**// Prepend prepends a values (or more)**

**func (list \*List[T]) Prepend(values ...T) {**

**// in reverse to keep passed order i.e. ["c","d"] -> Prepend(["a","b"]) -> ["a","b","c",d"]**

**for v := len(values) - 1; v >= 0; v-- {**

**newElement := &element[T]{value: values[v], next: list.first}**

**if list.size == 0 {**

**list.first = newElement**

**list.last = newElement**

**} else {**

**list.first.prev = newElement**

**list.first = newElement**

**}**

**list.size++**

**}**

**}**

**// Get returns the element at index.**

**// Second return parameter is true if index is within bounds of the array and array is not empty, otherwise false.**

**func (list \*List[T]) Get(index int) (T, bool) {**

**if !list.withinRange(index) {**

**var t T**

**return t, false**

**}**

**// determine traveral direction, last to first or first to last**

**if list.size-index < index {**

**element := list.last**

**for e := list.size - 1; e != index; e, element = e-1, element.prev {**

**}**

**return element.value, true**

**}**

**element := list.first**

**for e := 0; e != index; e, element = e+1, element.next {**

**}**

**return element.value, true**

**}**

**// Remove removes the element at the given index from the list.**

**func (list \*List[T]) Remove(index int) {**

**if !list.withinRange(index) {**

**return**

**}**

**if list.size == 1 {**

**list.Clear()**

**return**

**}**

**var element \*element[T]**

**// determine traversal direction, last to first or first to last**

**if list.size-index < index {**

**element = list.last**

**for e := list.size - 1; e != index; e, element = e-1, element.prev {**

**}**

**} else {**

**element = list.first**

**for e := 0; e != index; e, element = e+1, element.next {**

**}**

**}**

**if element == list.first {**

**list.first = element.next**

**}**

**if element == list.last {**

**list.last = element.prev**

**}**

**if element.prev != nil {**

**element.prev.next = element.next**

**}**

**if element.next != nil {**

**element.next.prev = element.prev**

**}**

**element = nil**

**list.size--**

**}**

**// Contains check if values (one or more) are present in the set.**

**// All values have to be present in the set for the method to return true.**

**// Performance time complexity of n^2.**

**// Returns true if no arguments are passed at all, i.e. set is always super-set of empty set.**

**func (list \*List[T]) Contains(values ...T) bool {**

**if len(values) == 0 {**

**return true**

**}**

**if list.size == 0 {**

**return false**

**}**

**for \_, value := range values {**

**found := false**

**for element := list.first; element != nil; element = element.next {**

**if element.value == value {**

**found = true**

**break**

**}**

**}**

**if !found {**

**return false**

**}**

**}**

**return true**

**}**

**// Values returns all elements in the list.**

**func (list \*List[T]) Values() []T {**

**values := make([]T, list.size, list.size)**

**for e, element := 0, list.first; element != nil; e, element = e+1, element.next {**

**values[e] = element.value**

**}**

**return values**

**}**

**// IndexOf returns index of provided element**

**func (list \*List[T]) IndexOf(value T) int {**

**if list.size == 0 {**

**return -1**

**}**

**for index, element := range list.Values() {**

**if element == value {**

**return index**

**}**

**}**

**return -1**

**}**

**// Empty returns true if list does not contain any elements.**

**func (list \*List[T]) Empty() bool {**

**return list.size == 0**

**}**

**// Size returns number of elements within the list.**

**func (list \*List[T]) Size() int {**

**return list.size**

**}**

**// Clear removes all elements from the list.**

**func (list \*List[T]) Clear() {**

**list.size = 0**

**list.first = nil**

**list.last = nil**

**}**

**// Sort sorts values (in-place) using.**

**func (list \*List[T]) Sort(comparator utils.Comparator[T]) {**

**if list.size < 2 {**

**return**

**}**

**values := list.Values()**

**slices.SortFunc(values, comparator)**

**list.Clear()**

**list.Add(values...)**

**}**

**// Swap swaps values of two elements at the given indices.**

**func (list \*List[T]) Swap(i, j int) {**

**if list.withinRange(i) && list.withinRange(j) && i != j {**

**var element1, element2 \*element[T]**

**for e, currentElement := 0, list.first; element1 == nil || element2 == nil; e, currentElement = e+1, currentElement.next {**

**switch e {**

**case i:**

**element1 = currentElement**

**case j:**

**element2 = currentElement**

**}**

**}**

**element1.value, element2.value = element2.value, element1.value**

**}**

**}**

**// Insert inserts values at specified index position shifting the value at that position (if any) and any subsequent elements to the right.**

**// Does not do anything if position is negative or bigger than list's size**

**// Note: position equal to list's size is valid, i.e. append.**

**func (list \*List[T]) Insert(index int, values ...T) {**

**if !list.withinRange(index) {**

**// Append**

**if index == list.size {**

**list.Add(values...)**

**}**

**return**

**}**

**var beforeElement \*element[T]**

**var foundElement \*element[T]**

**// determine traversal direction, last to first or first to last**

**if list.size-index < index {**

**foundElement = list.last**

**beforeElement = list.last.prev**

**for e := list.size - 1; e != index; e, foundElement = e-1, foundElement.prev {**

**beforeElement = beforeElement.prev**

**}**

**} else {**

**foundElement = list.first**

**for e := 0; e != index; e, foundElement = e+1, foundElement.next {**

**beforeElement = foundElement**

**}**

**}**

**if foundElement == list.first {**

**oldNextElement := list.first**

**for i, value := range values {**

**newElement := &element[T]{value: value}**

**if i == 0 {**

**list.first = newElement**

**} else {**

**newElement.prev = beforeElement**

**beforeElement.next = newElement**

**}**

**beforeElement = newElement**

**}**

**oldNextElement.prev = beforeElement**

**beforeElement.next = oldNextElement**

**} else {**

**oldNextElement := beforeElement.next**

**for \_, value := range values {**

**newElement := &element[T]{value: value}**

**newElement.prev = beforeElement**

**beforeElement.next = newElement**

**beforeElement = newElement**

**}**

**oldNextElement.prev = beforeElement**

**beforeElement.next = oldNextElement**

**}**

**list.size += len(values)**

**}**

**// Set value at specified index position**

**// Does not do anything if position is negative or bigger than list's size**

**// Note: position equal to list's size is valid, i.e. append.**

**func (list \*List[T]) Set(index int, value T) {**

**if !list.withinRange(index) {**

**// Append**

**if index == list.size {**

**list.Add(value)**

**}**

**return**

**}**

**var foundElement \*element[T]**

**// determine traversal direction, last to first or first to last**

**if list.size-index < index {**

**foundElement = list.last**

**for e := list.size - 1; e != index; {**

**fmt.Println("Set last", index, value, foundElement, foundElement.prev)**

**e, foundElement = e-1, foundElement.prev**

**}**

**} else {**

**foundElement = list.first**

**for e := 0; e != index; {**

**e, foundElement = e+1, foundElement.next**

**}**

**}**

**foundElement.value = value**

**}**

**// String returns a string representation of container**

**func (list \*List[T]) String() string {**

**str := "DoublyLinkedList\n"**

**values := []string{}**

**for element := list.first; element != nil; element = element.next {**

**values = append(values, fmt.Sprintf("%v", element.value))**

**}**

**str += strings.Join(values, ", ")**

**return str**

**}**

**// Check that the index is within bounds of the list**

**func (list \*List[T]) withinRange(index int) bool {**

**return index >= 0 && index < list.size**

**}**

**package main**

**import (**

**\_ "embed"**

**"fmt"**

**"strings"**

**)**

**type List[T comparable] struct {**

**first \*element[T]**

**last \*element[T]**

**size int**

**}**

**type element[T comparable] struct {**

**value T**

**prev \*element[T]**

**next \*element[T]**

**}**

**func New[T comparable](values ...T) \*List[T] {**

**list := &List[T]{}**

**if len(values) > 0 {**

**list.Add(values...)**

**}**

**return list**

**}**

**// func (list \*List[T]) Add(values ...T) {**

**// for \_, value := range values {**

**// newElement := &element[T]{value: value, prev: list.last}**

**// if list.size == 0 {**

**// list.first = newElement**

**// list.last = newElement**

**// } else {**

**// list.last.next = newElement**

**// list.last = newElement**

**// }**

**// list.size++**

**// }**

**// }**

**//**

**// func (list \*List[T]) Append(values ...T) {**

**// list.Add(values...)**

**// }**

**//**

**// func (list \*List[T]) Prepend(values ...T) {**

**// for v := len(values) - 1; v >= 0; v-- {**

**// newElement := &element[T]{value: values[v], next: list.first}**

**// if list.size == 0 {**

**// list.first = newElement**

**// list.last = newElement**

**// } else {**

**// list.first.prev = newElement**

**// list.first = newElement**

**// }**

**// list.size++**

**// }**

**// }**

**//**

**// func (list \*List[T]) Get(index int) (T, bool) {**

**// if !list.withinRange(index) {**

**// var t T**

**// return t, false**

**// }**

**//**

**// if list.size-index < index {**

**// element := list.last**

**// for e := list.size - 1; e != index; e, element = e-1, element.prev {**

**// }**

**// return element.value, true**

**// }**

**// element := list.first**

**// for e := 0; e != index; e, element = e+1, element.next {**

**// }**

**// return element.value, true**

**// }**

**//**

**// func (list \*List[T]) withinRange(index int) bool {**

**// return index >= 0 && index < list.size**

**// }**

**func (list \*List[T]) MergeList(anotherList \*List[T]) {**

**for \_, val := range anotherList.Values() {**

**list.Merge(val)**

**}**

**}**

**//go:embed test.txt**

**var file string**

**type word struct {**

**word string**

**count int**

**pointedWord \*string**

**}**

**func main() {**

**Test1 := New(1, 2, 4, 5)**

**Test2 := New(5, 4, 8, 9)**

**Test1.MergeList(Test2)**

**fmt.Println(Test1.String())**

**// file, err := os.Open("test.txt")**

**// if err!=nil{**

**// fmt.Print("error: opening file")**

**// return**

**// }**

**wordMap := make(map[string]struct {**

**count int**

**line int**

**pointedWord \*string**

**})**

**Ans := New(word{})**

**splitedString := strings.Split(file, " ")**

**for i, value := range splitedString {**

**word := strings.ToLower(value) // Приводим слово к нижнему регистру**

**if entry, exists := wordMap[word]; exists {**

**entry.count++**

**wordMap[word] = entry**

**} else {**

**wordMap[word] = struct {**

**count int**

**line int**

**pointedWord \*string**

**}{count: 1, line: i, pointedWord: &value}**

**}**

**}**

**for key, value := range wordMap {**

**Ans.Add(word{key, value.count, value.pointedWord})**

**}**

**fmt.Println(Ans)**

**}**