Sean Ashton (14866636)

Data Structures and Algorithms

Semester 2 2018 Assignment Documentation

Contents

Overview

The program reads in data from a hard coded folder, ElectionData.
Ideally the user would be prompted if the data could not be found in that folder.

The files are read via FileIO class to a queue of string arrays which are then passed to the ElectionData object, which parses the input and creates/ updates Candidate, Party, State and Division objects. The lists that these objects are stored in is sorted by the end of the data import phase, so allow for fast radix style sorting of user queries.

The phase of reading in vote data is done by spawning one thread per file and running the threads in the background while the user navigates the menu, the program will wait for threads to complete when the user first asks for a report that contains vote data, usually this wait will not be present due to the user setting filters and otherwise passing time for the threads to complete.

All objects are stored in Linked Lists that implement ListIterator to allow for forward/backward stepping to insert values into a specific position which allows for sorting on insert instead of after the fact.

Initially Candidates are also stored in a hash table to facilitate quick and easy access to specific candidates when reading in the votes as each candidate has multiple, out of order lines representing their votes as the input file is ordered by polling place.

When the user specifies filters, those requests are sent to a Filter object to process. Once processed the Filter object has options for output via a linked list of Candidates or a queue of String arrays, ready for printing or saving to file.

Ordering is represented by a 4 element integer array as the data is trivial to store and represent. This is later passed to Filter to determine sort order.

Sorting is done in reverse order to the user specified priorities as all the sorts used are stable, so if the data is sorted by the least important factor first subsequent sorts that have duplicate keys will not rearrange elements relative to each other.

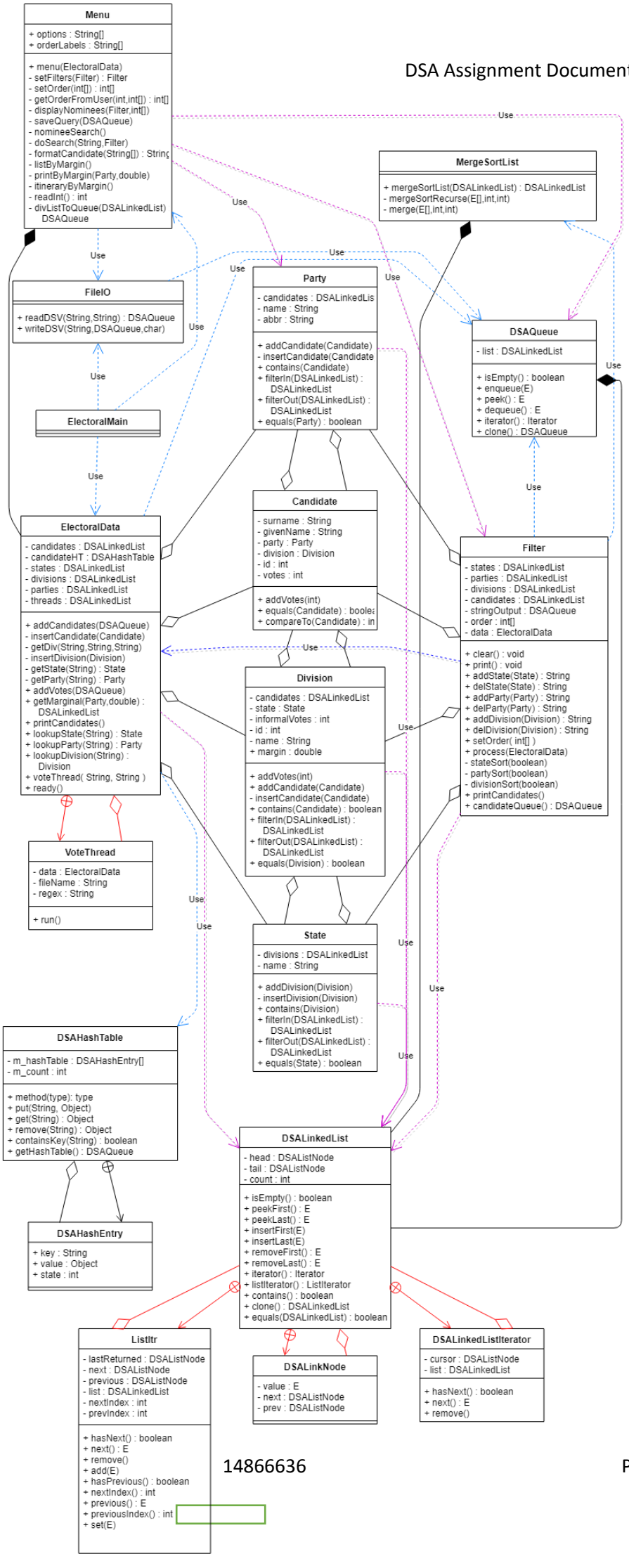
Marginal seats are calculated as part of ElectionData, each Division object has a margin field that is used as scratch space for the margin calculation and ordering, it is overwritten on each call to getMarginal(), this is similar behaviour to the visited flag on graph nodes.

Filtering is done by iteration through all candidates and filter criteria and copying each match to an output list that is used by the next stage of filtering, and so on. As the filtering iterates through all filter criteria in order, it also sorts the data while filtering (sort by surname is done via Merge Sort, as it is not a filter criteria).

Desired changes

Ideally the insert candidate and related methods would use a persistent List Iterator to reduce the traversal of the linked list that stores them all.

UML


Menu
+ options : String[]
+ orderLabels : String[]
+ menu(ElectoralData)
- setFilters(Filter) : Filter
- setOrder(int[]) : int[]
- getOrderFromUser(int,int[]) : int[]
- displayNominees(Filter,int[])
- saveQuery(DSAQueue)
- nomineeSearch()
- doSearch(String,Filter)
- formatCandidate(String[]) : String
- listByMargin()
- printByMargin(Party,double)
- itineraryByMargin()
- readInt() : int
- divListToQueue(DSALinkedList) DSAQueue
MergeSortList
+ mergeSortList(DSALinkedList) : DSALinkedList
- mergeSortRecurse(E[],int,int)
- merge(E[],int,int)
FileIO
+ readDSV(String,String) : DSAQueue
+ writeDSV(String,DSAQueue,char)
Party
- candidates : DSALinkedLis
- name : String
- abbr : String
+ addCandidate(Candidate)
- insertCandidate(Candidate
+ contains(Candidate)
+ filterIn(DSALinkedList) : DSALinkedList
+ filterOut(DSALinkedList) : DSALinkedList
+ equals(Party) : boolean
DSAQueue
- list : DSALinkedList
+ isEmpty() : boolean
+ enqueue(E)
+ peek() : E
+ dequeue() : E
+ iterator() : Iterator
+ clone() : DSAQueue
ElectoralMain
Candidate
- surname : String
- givenName : String
- party : Party
- division : Division
- id : int
- votes : int
+ addVotes(int)
+ equals(Candidate) : boolea
+ compareTo(Candidate) : in
ElectoralData
- candidates : DSALinkedList
- candidateHT : DSAHashTable
- states : DSALinkedList
- divisions : DSALinkedList
- parties : DSALinkedList
- threads : DSALinkedList
+ addCandidates(DSAQueue)
- insertCandidate(Candidate)
- getDiv(String,String,String)
- insertDivision(Division)
- getState(String) : State
- getParty(String) : Party
+ addVotes(DSAQueue)
+ getMarginal(Party,double) : DSALinkedList
+ printCandidates()
+ lookupState(String) : State
+ lookupParty(String) : Party
+ lookupDivision(String) : Division
+ voteThread(String, String)
+ ready()
Filter
- states : DSALinkedList
- parties : DSALinkedList
- divisions : DSALinkedList
- candidates : DSALinkedList
- stringOutput : DSAQueue
- order : int[]
- data : ElectoralData
+ clear() : void
+ print() : void
+ addState(State) : String
+ delState(State) : String
+ addParty(Party) : String
+ delParty(Party) : String
+ addDivision(Division) : String
+ delDivision(Division) : String
+ setOrder(int[])
+ process(ElectoralData)
- stateSort(boolean)
- partySort(boolean)
- divisionSort(boolean)
+ printCandidates()
+ candidateQueue() : DSAQueue
Division
- candidates : DSALinkedList
- state : State
- informalVotes : int
- id : int
- name : String
+ margin : double
+ addVotes(int)
+ addCandidate(Candidate)
- insertCandidate(Candidate)
+ contains(Candidate) : boolean
+ filterIn(DSALinkedList) : DSALinkedList
+ filterOut(DSALinkedList) : DSALinkedList
+ equals(Division) : boolean
VoteThread
- data : ElectoralData
- fileName : String
- regex : String
+ run()
State
- divisions : DSALinkedList
- name : String
+ addDivision(Division)
- insertDivision(Division)
+ contains(Division)
+ filterIn(DSALinkedList) : DSALinkedList
+ filterOut(DSALinkedList) : DSALinkedList
+ equals(State) : boolean
DSAHashTable
- m_hashTable : DSAHashEntry[]
- m_count : int
+ method(type): type
+ put(String, Object)
+ get(String) : Object
+ remove(String) : Object
+ containsKey(String) : boolean
+ getHashTable() : DSAQueue
DSALinkedList
- head : DSAListNode
- tail : DSAListNode
- count : int
+ isEmpty() : boolean
+ peekFirst() : E
+ peekLast() : E
+ insertFirst(E)
+ insertLast(E)
+ removeFirst() : E
+ removeLast() : E
+ iterator() : Iterator
+ listIterator() : ListIterator
+ contains() : boolean
+ clone() : DSALinkedList
+ equals(DSALinkedList) : boolean
DSAHashEntry
+ key : String
+ value : Object
+ state : int
ListItr
- lastReturned : DSAListNode
- next : DSAListNode
- previous : DSAListNode
- list : DSALinkedList
- nextIndex : int
- prevIndex : int
+ hasNext() : boolean
+ next() : E
+ remove()
+ add(E)
+ hasPrevious() : boolean
+ nextIndex() : int
+ previous() : E
+ previousIndex() : int
+ set(E)
DSALinkNode
- value : E
- next : DSAListNode
- prev : DSAListNode
DSALinkedListIterator
- cursor : DSAListNode
- list : DSALinkedList
+ hasNext() : boolean
+ next() : E
+ remove()
Use

Classes

ElectoralMain

The purpose of this class is to be the entry point for the program, it checks for the presence of the required files and reads them in to an ElectoralData class before handing over control flow to Menu. This class serves to prepare the application for general use. There is an opportunity to either create a class to handle finding and prompting for input files, I chose not to due to time constraints. This functionality could have been part of Menu, though again due to time constraints it is not.

FileIO

The purpose of this class is to read in a delimiter (in this case comma) separated file. This was written during the lab sessions (specifically lab 6) as it is generally useful to separate the reading and parsing of files. The ADT used for returning the parsed data is a Queue built on a LinkedList, this was chosen as the file is in a sequential format such that the first line read will be the first data parsed. A LinkedList could have been used as it provided similar functionality, Linked Lists were used as opposed to arrays as the size of the structure is not known ahead of time and would therefore require either resizing arrays or reading the file twice.

Menu

The purpose of this class is to handle all interactions with the user of the program. I chose to create this class as there is a clear job that this class does, which abstracts any interaction with the user away from all other classes. There are some hard coded values in arrays for sorting order, since the data has many different fields, there isn't a programmatic way to select which to sort by without including all of them, while this could be a possible feature addition the method to sort by arbitrary fields has not been covered and provides little value given the input data format is known by the program at compile time.

The menu has a method for saving any report the user generates. This report is in a queue to be used by the FileIO writeDSV method without having to convert it. The reasoning behind using queues for file IO is described above.

ElectoralData

The purpose of this class is to read in and hold all the data from files, it stores various container classes representing Parties, States, Candidates and Divisions. When reading in data there are functions to get the appropriate object given the data about it, if it doesn't exist (new Party for example) then it creates it. When being queried externally the data has already been read in, so null is returned if the object can't be found instead of creating it. While there are two functions with very similar purposes they are separate because of the different applications in use.

All objects being read in are stored in an ordered fashion in their linked list. This is achieved by implementing a List Iterator in the Linked List class and moving through the list to the appropriate location to insert or by using a merge sort. Ideally a persistent iterator would be used to reduce the overheads of creating one every time a new element is inserted and minimising traverse of the list.

Since all the data is readily present, the marginal seat calculation is implemented here, which returns a linked list in order of margin.

Linked lists were chosen as the ADT for all data as in every use after reading in votes, all values must be iterated over, hence the limitation of linked list (Accessing data not at the ends) is not a problem. Arrays could have been used for slightly fast iteration, however it would slow down an already slow read in process, while the reports run very fast using linked lists.

Party

The purpose of this class is to hold all party information. This includes the name, abbreviation and candidates within the party. This class also has a utility function to filter from an input list candidates that are either in or not in the party. The ADT linked list was chosen for holding candidates for the same reasons described in ElectoralData. The filter methods of this class are not used, the plan was to use each class to filter elements related to it, however it was far more straight forward and efficient to iterate through once in the Filter class which results in sorting and filtering in one pass.

Candidate

The purpose of this class is to hold all candidate information. This includes the name, party, division, id number and votes within the party. There is a function to add votes to a candidate within this class.

Division

The purpose of this class is to hold all division information. This includes the name, id number, informal votes, state and candidates within the division. This class also has a utility function to filter from an input list candidates that are either in or not in the division. There is a public margin variable used when calculating margins that is overwritten every time a new party's margin is calculated, in a similar fashion to a graph vertex's visited field. Adding candidates to this object uses a List iterator to insert it in the list at the appropriate location. The ADT linked list was chosen for holding candidates for the same reasons described in ElectoralData. The filter methods of this class are not used, the plan was to use each class to filter elements related to it, however it was far more straight forward and efficient to iterate through once in the Filter class which results in sorting and filtering in one pass.

State

The purpose of this class is to hold all state information. This includes the name and candidates within the state. This class also has a utility function to filter from an input list candidates or divisions that are either in or not in the state, which ultimately was not used, see Party and Division. The ADT linked list was chosen for holding candidates for the same reasons described in ElectoralData.

Filter

The purpose of this class is to filter and sort a list of Candidates using a set of filters and sort preferences set by the user.

Filtering and sorting is done by sorting the list by the least important sort criteria first and the most important last, this results in a preferentially sorted list as all sorts used are stable. For sorting the party, state and division a radix style sort is used as the sort key is highly duplicated. This works by iterating through each sort criteria (State, Party, Division) and adding matching candidates to a linked list, this is effectively filling one bucket at a time, Ideally assigning each candidate to a list (held by array/hashtable) and only iterating through once would be more efficient, however the method used is easier to implement and runs very fast for the given input data (~1000 candidates). For the surname sort a List based merge sort is used by having Candidate implement Comparable and returning the surname comparison. This merge sort was adapted from the Prac 6 Graphs Lab.

The order is passed to the filter before running processing, this step is optional as there is a default order.

MergeSortList

The purpose of this class is to sort a DSALinkedList that contains values that implement Comparable, this is not a part of the DSALinkedList as non-Comparable values are needed to be stored in a linked list. A special SortedList class could have been used, inheriting from DSALinkedList and implementing various functions relating to sorting such as a ListIterator and a MergeSort however I chose to use a separate class as the Java rules regarding generics were proving problematic while attempting to implement the sorting in the labs. This was created in Lab 6 then modified for the assignment.

DSAHashTable

The purpose of this class is to implement the hash table ADT for use in adding votes to candidates as it is far quicker than iterating through a linked list for every candidate and every polling location. Hash table was chosen due to its ability to grow with the data as it is read in and index by an arbitrarily large value without requiring as much space as an array (Candidate IDs go almost to 30,000 where there are only approximately 1000 candidates in the data). No other ADT implemented through the practicals so far offers these features. A 2-3-4, Red-Black or B tree could be used to get a similar result, however DSAHashTable has already been tested and works well.

DSAQueue

The purpose of this implementation of a queue ADT is for use in cases where data needs to be handed between methods and classes in a sequential manner. Particularly it is used for the lines read in from files and printed to the screen or out to files. The role of this ADT could be fulfilled by the LinkedList that it is built on, though the abstract nature means that a more efficient or otherwise different method of storing the queue could be used without changing code in the main program. (Queue on top of a hash table or tree perhaps?)

DSALinkedList

The purpose of this class is to serve as the basis of all object storage in this project. Linked Lists work well for the majority of tasks required in this assignment as most operations need to iterate through every element in the list and move dynamically sized lists around. In some cases, a tree or array would be more efficient to do the operation with however the versatility of the Linked List when random accesses are not required trumped these considerations when selecting an ADT for this purpose.

DSALinkedList.DSALinkedListIterator

This inner class of DSALinkedList provides a useful way to iterate through the list which is required in most operations that this program provides. It falls short in the case of inserting elements as it can't insert elements at all, leaving just the methods insertFirst and insertLast of the parent list class. This led to the creation of the next class;

DSALinkedList.ListItr

This inner class of DSALinkedList provides an implementation of a List Iterator that rectifies the limitations of the List and Iterator combination, primarily it allows for moving backwards through a list in addition to forwards which means that new elements can be inserted (with arbitrary position rules) in any position in the list. The result of this is reduced need for external sorting methods which may not work due to the required sorting to happen based on different criteria.

Additional decisions

Itinerary

Initially the itinerary will be sorted by state, as this is a very computationally inexpensive optimisation that will result in a better than the by margin route.

For a closer to optimal route all locations are read in to a Graph representing the travel times between connected vertices, Each vertex has a flag mustVisit that shows if a vertex is a margin seat. For each node starting from either Canberra or a user defined start point Dijkstra's algorithm is run to find the closest marginal seat and traverse to it writing out the route to a DSAQueue before marking the vertex as NOT mustVisit. The result is a route that minimises travel time between vertices, though not optimally (See travelling salesman problem for information on why this is the case)

For each marginal seat an extra "edge" is added to the route with a duration of 3 hours to represent mingling time.

Once the route has been calculated an itinerary is produced by adding prep time (within certain hours 2200 – 0800)

Graph's are the obvious choice for calculating the itinerary as no other ADT described in this unit stores the distances between arbitrary nodes and allows for traversal or the addition of traversal algorithms.