Scott Eno
Jamie Kahle
HSDA Restaurant Design

Most Frequent Operation & Optimization:

Most frequent operation: Searching a collection for a specific key.

This operation has been keeping collections sorted by the most commonly searched key type (whether it be a seat count, table name, or party name). Maintaining this sorting has allowed a binary search to be utilized in order to reduce the number of comparisons required each time a search is performed.

Implemented ADTs:

We have utilized AscendinglyOrderedList, AscendinglyOrderedListD, and DEQ to support our implementation.

Our usage of the AscendinglyOrderedList has allowed us to optimize the item searching that occurs within a majority of the driver options. Furthermore, each instance is sorted by the attribute that is searched most frequently, such as the seats of an available Table when seating or the party name of a Seated Party when removing.

The DEQ is used to maintain the behavior and benefits of a Queue while decreasing the number of operations required to both search for a party to seat and return the queue back to its original FIFO order. With a normal queue, this action would always require 2n - 1 queue operations. However, a DEQ paired with our optimized reordering logic performs less operations when a party is found before the midpoint of the queue before reaching a maximum of 2n - 1 operations performed, all without extra storage.

ADT Instances:

available (AscendinglyOrderedListD of type Table): contains information on the empty tables in a restaurant section. New Tables are inserted into this AOL. Sorted by table capacity to optimize seating.

serving (AscendinglyOrderedList of type SeatedParty): Contains information about all parties and the table each is seated at. Upon seating a party a new instance is created and the table is added. Sorted by party name to optimize search for a party that wishes to leave.

tableNames (AscendinglyOrderedList of type String): Contains the names of all the tables in the section, available or not. Used to optimize the search for a table’s existence and avoid a linear unordered search through available and serving.

parties(AscendinglyOrderedList of type String): Contains the names of all customers in alphabetical order. Used to optimize the search for a party's existence in the restaurant and avoid slow search through the waiting DEQ.

DEQ:
waiting (DEQ of type Party): contains information on the parties currently waiting to be seated. New parties will be added to this queue in FIFO order upon entering the restaurant. When being seated, if a party is unable to, it will temporarily be added to the back of the queue until a party is able to be seated, where if more than half the parties were unable to be seated: the queue will continue to cycle. If less than half were unable to be seated, then the queue will put the parties from the start of the queue, back to the front in their original order.

Driver

Handles Input/Output. Maintains a List of sections in the restaurant and a List of waiting parties

Functionality:

void initialize(List<Sections> sections) - prompts user to specify table names and capacity for each section upon program startup.

void welcomeParty(List<Sections> sections) - adds a new unique user-specified Party to waitingQueue

void seatNext(List<Sections> sections, Queue<Party> waiting) - attempts to seat next Party in line

void leaveParty(List<Sections> sections, Queue<Party> waiting) - attempts to remove user specified party from the restaurant
void addTable(List<Sections> sections) - adds a unique user-specified table to a section

void removeTable(List<Sections> sections) - attempts to remove user specified table from a section

void displayAvailableTables(List<Sections> sections) - displays information on the currently available tables

void displayWaiting(Queue<Party> waiting) - displays information on the currently waiting parties

void displayServing(List<Sections> sections) - displays information on the currently occupied tables

Section

Represents a distinct section within a restaurant. This class provides an interface for adding/removing tables, seating customers, and freeing tables once customers have left.

Functionality:

boolean hasTableName(String tableName) - returns true if this section contains a table by the specified tableName, false otherwise.

void addTable(Table table) - adds a new unoccupied Table to this section.

Table removeTable(String tableName) throws RemoveOccupiedTableException - removes table of specified name from this section. Returns null if the table does not exist. Throws RemoveOccupiedTableException if the specified table is still occupied when attempting removal.

boolean seatParty(Party party) - Returns true if the party was successfully seated, false otherwise. The table that leaves the least number of excess seats will be prioritized; when seating a party of 3, they will be given a table for 4 over a table of 6 if both are available.

OccupiedTable removeParty(String partyName) - Removes and returns the party specified by partyName from this section and frees the table they were seated at if it exists, otherwise returns null if the specified party could not be found.

String getAvailableTableInfo() - Returns a formatted string detailing the section name, the number of available tables, and the details of each available table.

String getServingInfo() - Returns a formatted string detailing the section name and the details of each occupied table.

Party extends KeyedItem<String>

Represents a group of one or more customers identified by a name, number of people, and a preference of section. Keyed by the the party name

Functionality:

String getKey() - returns the name associated with this party

int getSize() - returns the number of people in this party

String getSection() - returns true if this party has pets with them, otherwise returns false.

String toString() - returns a formatted string detailing the name, head count, and this party's preferred section.

Scott Eno
Jamie Kahle
HSDA Restaurant Design

Table extends KeyedItem<Integer>

Represents a table within a section identified by a name and number of seats. Keyed by its seat count.

Functionality:

String getName() - returns the name of this table.

int getKey() - returns the number of seats this table has.

int compareTo(Table otherTable) - Compares this table with the specified Table based on the number of seats. Return 0 if seats are equal, < 0 if this table has less seats, > 0 if this table has more seats.

String toString() - returns a formatted string detailing the name and number of seats of this table.

SeatedParty extends Party

Represents a Party occupying a Table, which has a name, a preferred section, a size, and the table it occupies.

Functionality:

Table getTable() - returns the table that this party is occupying

String toString() - returns a formatted string detailing this party and the table it occupies.

Dataflow:

Option	Dataflow
Startup	- For each section - Prompt user for number of tables - For [number of tables] loops - Prompt user for tableName until unique name is given - If section.hasTable(tableName) == true → tableName is not unique, prompt again - Repeat until unique name is given - Prompt for number of seats - Create newTable - section.addTable(newTable) - newTable is added to **availableTables** - Print menu and begin

Scott Eno
Jamie Kahle
HSDA Restaurant Design

Option 1: Party Enters	- Prompt for partyName until unique name is given - if **waitingQueue** contains party with partyName OR section.hasParty(partyName) == true → partyName is not unique, prompt again - Prompt for group size - Prompt for hasPets - Create newParty - Enqueue newParty onto **waitingQueue**
Option 2: Serve Party	- Dequeue nextParty from the front of **waitingQueue** - Store nextParty's name to keep track of original order of the queue - Loop until nextParty is seated OR no party could be seated (nextParty == original front of queue) - Get the section that nextParty prefers - preferredSection.add(nextParty) - Search through **availableTables** to find a table with the least number of excess seats. - If a Table is found, create a new SeatedParty from nextParty and Table and add it to **serving** - .add(nextParty) returns true - if .add(nextParty) returns false → enqueue nextParty, dequeue new nextParty, repeat - Display failure - If a party was successfully seated → rotate through queue items until original FIFO order is restored - dequeue and enqueue from **waitingQueue** until peek returns original front of queue - Display success - If no parties could be seated - Display failure
Option 3: Party Leaves	- Prompt for partyName of leaving party - For each section until a match is found - section.remove(partyName) - Search **serving** for matching - if a match is found: - remove and secure matching SeatedParty from **serving** - Add table stored in the SeatedParty back to **availableTables** - Table is now freed - Return matching SeatedParty

	- Display successful remove - If a match is not found within any of the sections → search through **parties** to verify that specified party is not still waiting - Display failure - Else → match not found, display failure
Option 4: Add table	- Prompt user for section to add to - Prompt user for tableName until unique name is given - If section.hasTable(tableName) == true → tableName is not unique, prompt again - Repeat until unique name is given - Prompt for number of seats - Create newTable - section.addTable(newTable) - newTable is added to **availableTables**
Option 5: Remove table	- Prompt user for section to remove from - Prompt user for tableName - section.removeTable() - If matching tableName is found in **availableTables** - Remove, secure, and return found Table - Display successful remove - If matching tableName is found in **occupiedTables** - Throw RemoveOccupiedTableException - Driver catches exception, informs user that table is occupied - If matching tableName could not be found - Throw NoSuchElementException - Driver catches exception, informs user that table could not be found
Option 6: Display available tables	- For each section - section.getAvailableTableInfo() - Display name of section along with size and contents of **availableTables** (.size() & .toString())
Option 7: Display waiting parties	- Display contents of **waitingList** (.toString())

Option 8: Display parties being served	- For each section - section.getServingInfo() - Display name of section along with contents of **occupiedTables** (.size() & .toString())

Division of Work:

Scott: Responsible for
- Implementing Section, Table, SeatedParty, Driver Frame & Options 4,5,6

Jamie: Responsible for
- Designing Section's internal methods (sorting, searching) Party, Custom Exceptions, Driver Options 1,2,3, 7, 8, Extra Credit