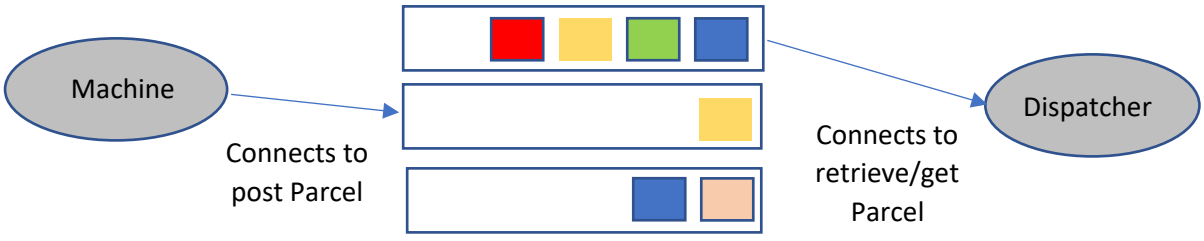
Data Structures and Algorithms: Assignment 2

Due: 27 October 2020, 11:59pm – 100 marks possible (20% of final grade)

When submitting, zip up your entire Netbeans project into a folder with the following structure: *lastname_firstname_studentID.zip* (using your own name and ID) and submit to the Assignment 2 folder on Blackboard BEFORE the due date and time. Late submissions will incur a penalty. Anti-plagiarism software will be used on all assignment submissions, so make sure you submit YOUR OWN work and DO NOT SHARE your answer with others. Any breaches will be reported and will result in an automatic fail and possible removal from the course.

Question 1) – Factory Simulator (30 marks); Recommended Readings – 1.3, 1.4, 3.2, 4.4

The purpose of this question is to create a factory simulator – a multi-threaded program which uses the producer-consumer design pattern. A *Machine* can run as a thread and connects to a *ConveyorBelt*, once connected it creates *Parcel* obects to post to the belt. The *ConveyorBelt* encapsulates a priority queue which keeps parcels in order of their *Comparable*. A *Dispatcher* also running as a thread connects to an available *ConveyorBelt*, once connected it can retrieve *Parcel* objects. Both the *Machine* and *Dispatcher* can monitor several conveyor belts, but should only connect to one at a time. Each *ConveyorBelt* can only hold a limited amount of *Parcel* objects.



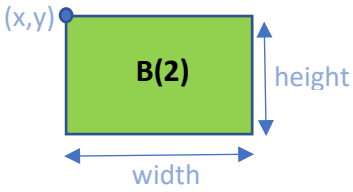
3 ConveyorBelt objects

Part A: Using the UML below create a class that represents a *Parcel* used in this factory simulator.

Parcel<E> implements Comparable<Parcel<?>>
- E : element - colour : Color - consumeTime : int - priority : int - timestamp : long
+Parcel(element, colour, consumeTime, priority) +consume():void +toString():String +drawBox(g:Graphics,x:int, y:int, width:int, height:int):void +compareTo(p : Parcel<?>) : int

Each parcel is instantiated with a generic element, a priority, colour and a *consumeTime* – the time it takes to process this parcel. During creation of the Parcel, a timestamp is set (in nano second time) to indicate its creation time. A *Parcel* is *Comparable* so can be compared with another *Parcel* even though they might hold a different generic. The ordering is determined firstly by priority and secondly via creation time (timestamp). The class also has a *toString* which prints out the element and priority

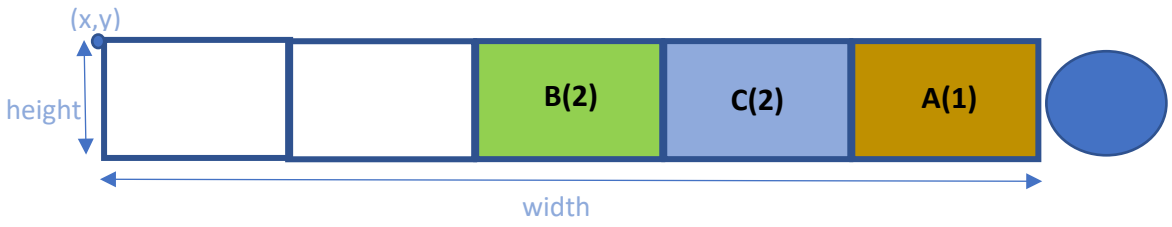
in parenthesis, and a *consume* function, which *sleeps* the calling code for the duration specified by the parcels *consumptionTime*. A *Parcel* can also be drawn with a *Graphics* object and method parameters as shown below. The specified *x,y* pixel coordinate represents the top left corner of the drawn box and *width* and *height* parameters determines its dimension.



Part B: Using the following UML, create a class called *ConveyorBelt* which encapsulates a *PriorityQueue* specifically designed to hold *Parcel* objects.

ConveyorBelt
-maxCapacity : int -connectedMachine : Machine -connectedDispatcher : Dispatcher -PriorityQueue<Parcel<?>> : queue
+ConveyorBelt(maxCapacity:int) +ConveyorBelt() +connectMachine(machine:Machine):boolean +connectDispatcher(machine:Dispatcher):boolean +disconnectMachine(machine:Machine):boolean +disconnectDispatcher(machine:Dispatcher):boolean +size():int +isEmpty():boolean +isFull():boolean +postParcel(p:Parcel<?>, machine:Machine):boolean +getFirstParcel(dispatcher:Dispatcher):Parcel<?> +retrieveParcel(dispatcher:Dispatcher):Parcel<?> +drawBelt(g:Graphics,x:int, y:int, width:int, height:int):void

The conveyor belt holds a *maximum capacity* of parcels in which it holds. It cannot exceed this capacity which is given to the constructor. A default capacity of 10 is used if no max capacity is specified during construction. The *ConveryorBelt* only allows one *Machine* and one *Dispatcher* to connect to it at a time (using a null if there is none currently connected). Only the currently connected *Machine* can post parcels to the conveyor belt (if not full) and only the currently connected *Dispatcher* can get and retrieve parcels from the conveyor belt (if any). A connected machine or dispatcher should pass themselves in as reference when posting/retrieving parcels so that the *ConveyorBelt* can check if they are currently connected or not. A *ConveyorBelt* can also be drawn with a *Graphics* object and the given method parameters. The specified *x,y* pixel coordinate represents the top left corner of the belt and width and height parameters determines its full dimension. The entire belt is drawn with or without parcels. If the belt has a connected dispatcher or machine, it should draw them as circles on each side of the belt. The example belt below has a capacity of 5, but only has 3 parcels in it with a connected dispatcher but no connected machine.



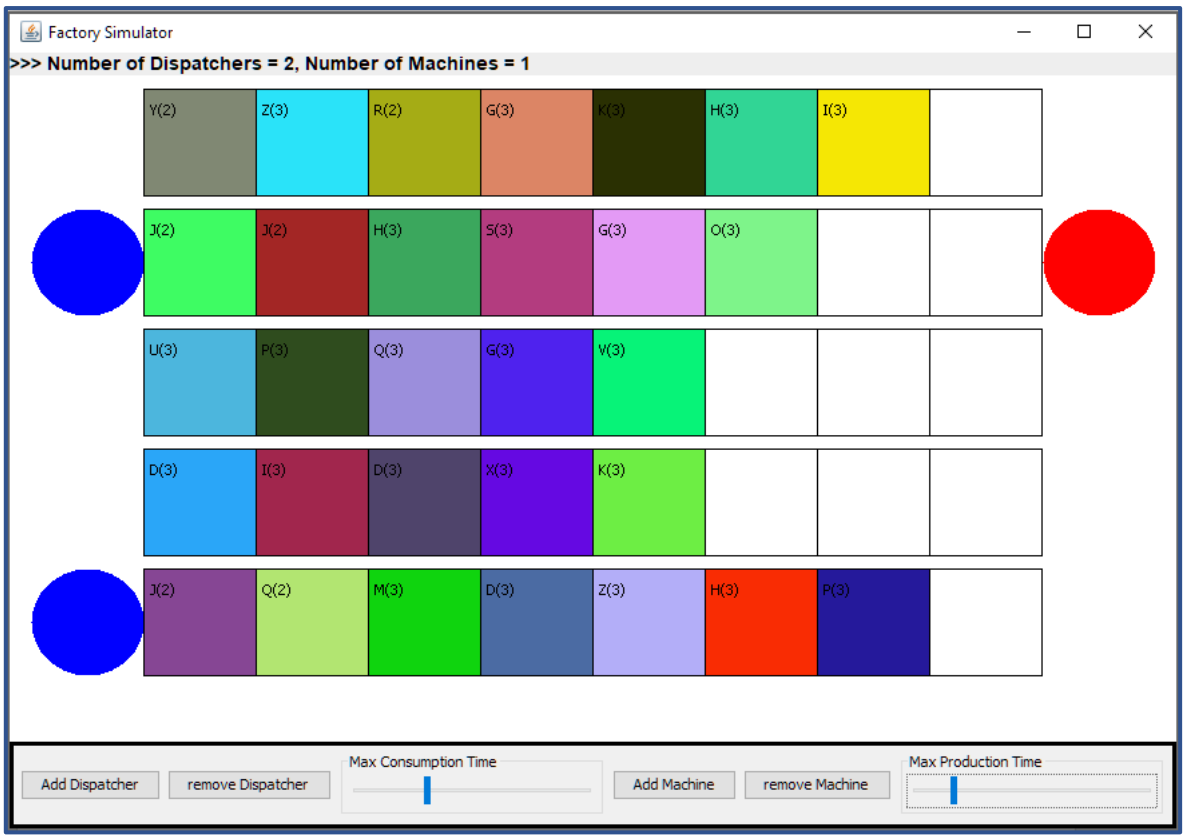
Note: As a *ConveyorBelt* is shared between many threads, it should be made "thread safe" so no *race conditions* arise in the simulator.

Part C: Create a class called *Machine* that runs continuously as a thread until requested to stop. When a machine is instantiated an array of *ConveyorBelt* objects should be passed into its constructor. It also holds the following public static variables: *MIN_CONSUMPTION_TIME, MAX_CONSUMPTION_TIME, MIN_PRODUCTION_TIME and MAX_PRODUCTION_TIME*. Set these initially to something suitable.

While a *Machine* is running, it should constantly circulate the *ConveyorBelt* objects that it is monitoring. If it comes across an available belt that is not full, it should try connecting to it. If the belt is full or it is unable to connect (as perhaps another machine is already connected), it simply moves on to the next *ConveyorBelt* in the array and repeats. If the machine successfully connects, it then starts posting *Parcel* objects to it. Before creating each parcel, the *Machine* must *sleep* for a *random* duration between min and max production time variables. A *Machine* should create a *Parcel* with a random colour, a random priority (between 1-3), a random letter (between A-Z) for its element and a random *consumeTime* (between min and max consumption time variables) – all passed into the *Parcel* constructor. Once the *Machine* detects that the *ConveyorBelt* becomes full, it should disconnect from the belt and then repeat the entire process by trying to connect to the next *ConveyorBelt* in the array.

Create another class called *Dispatcher* which also runs continuously as a thread until requested to stop. It has a similar behaviour to *Machine*, monitoring an array of *ConveyorBelt* objects passed into the constructor. However, its behaviour is slightly different. Once again, a *Dispatcher* will try to successfully connect to a *ConveyorBelt* if it is not empty and does not already have an existing *Dispatcher* connected to it. If it connects successfully it *gets* a *Parcel* from the *ConveyorBelt* and calls the *consume* method on it – effectively sleeping the thread for the parcels specified *consumeTime*. Once consumed it retrieves/removes the parcel from the belt and disconnects – repeating the process by moving to the next available *ConveyorBelt* in the array. Note, the *Dispatcher* only removes a single *Parcel* from the belt per connection, whereas the *Machine* posts multiple parcels until the belt is full per connection.

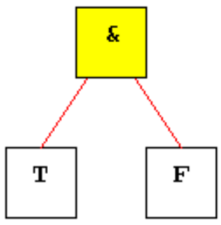
Part D: Finally, make a GUI called *FactorySimulatorGUI* which demonstrates an animated visual of the system. It should create **FIVE** *ConveyorBelt* objects of capacity eight. It should also maintain a *List* of *Machine* objects and a separate *List* of *Dispatcher* objects. It should have *JButtons* to add and remove *Dispatcher* or *Machine* threads. It should also have a *JLabel* which specifies the amount of current machine and dispatcher threads actively running. It also should have two *JSlider's* which can adjust the *MAX_CONSUMPTION_TIME* or *MAX_PRODUCTION_TIME static* values effectively increasing the range of random duration that either the *Machine* takes to produce a *Parcel* or a *Dispatcher* takes to consume the *Parcel*. An example screenshot of a working *FactorySimulatorGUI* is shown below with two connected *Dispatcher* objects (blue) and one connected *Machine* object (red) monitoring the same five *ConveyorBelt* objects, each containing randomly coloured parcels with assigned priority numbers.



Question 2) – Boolean Expression Tree (20 marks); Recommended Readings – 3.1, 4.1, 4.2

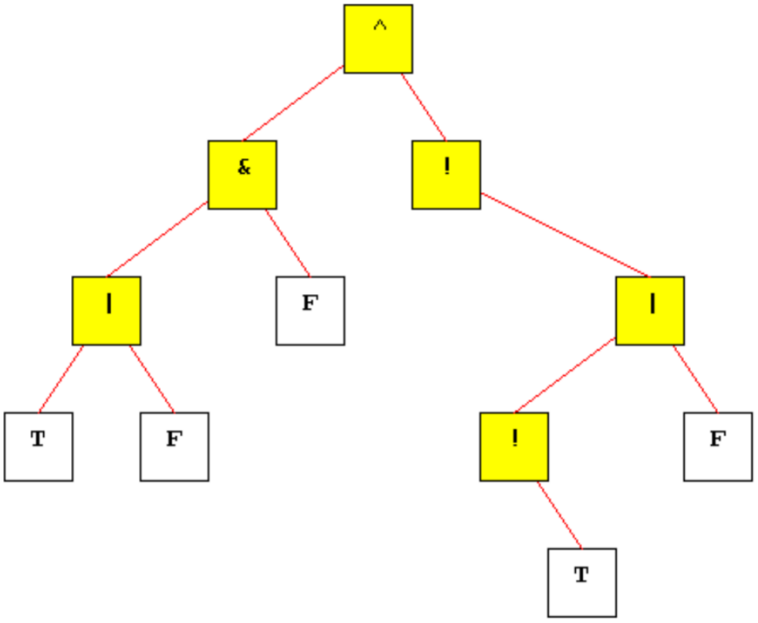
An *expression tree* is a binary tree representing mathematical expressions built from *postfix notation* and converting to *infix notation*. Each internal node in the tree can be an *operator* (mathematical operations) whereas leaf nodes are *operands* (numbers or values). An operator can calculate an expression from the output of its left child and right child if it is a binary operator, or simply taking the expression from its right child for a unary operator. A *boolean expression tree* is an expression tree for *boolean logic.* It is comprised of operands {true (T) or false (F)} as well as operators {OR (|), AND (&), NOT(!), XOR(^)}.

Eg: The postfix notation input **TF&** would produce the expression tree:



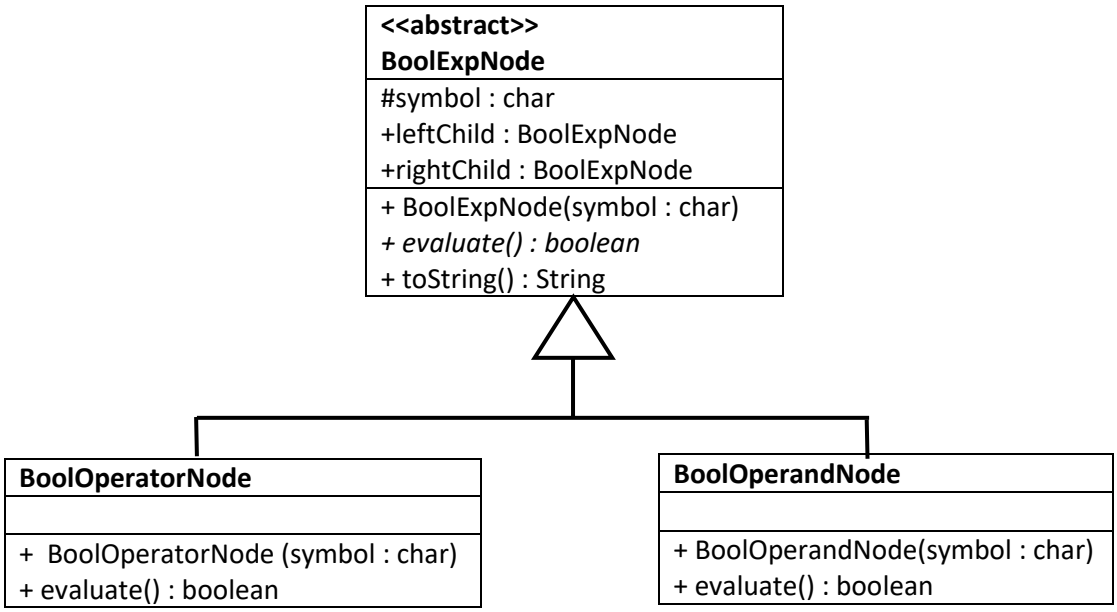
This would evaluate to the infix notation **(T & F) = false**.

Eg: The postfix notation of **TF|F&T!F|!^** as input would produce the expression tree:



This would evaluate to the infix notation: **(((T|F)&F)^!(!T|F)) = true**.

Using the UML below create the abstract class *BoolExpNode* and concrete subclasses *BoolOperatorNode* and *BoolOperandNode*. The super class keeps an expression *symbol* as a char as well as links to its left and right subtrees. The constructors accept a *symbol* used to represent the operand (T,F) or operator (!&^|) throwing an appropriate exception if the symbol is invalid. The *toString* simply prints out the symbol as a *String*. The subclasses must override the abstract method *evaluate,* which for an operand, will simply return **true** or **false** depending on the symbol. However, for an operator node it must evaluate the correct boolean expression by evaluating the left subtree and right subtree depending on the symbol (care should be taken for the unary operator NOT).



Create another class called *BooleanExpressionTreeBuilder* using the UML below. Note all methods should be declared static.

BooleanExpressionTreeBuilder
+ buildExpressionTree (expression : char[]) : BoolExpNode + toInfixString(node : BoolExpNode) : String + countNodes(node : BoolExpNode) : int

The method *buildExpressionTree* which builds an expression tree comprised of operator and operand nodes by reading symbols from the postfix input char array. It does this using a stack (similar to the postfix calculator example) looping through the character array to determine whether the symbol represents either an operator or operand. If the symbol represents an operand (T or F) it simply pushes a new *BoolOperandNode* onto the stack, if the symbol represents an operator it creates a new *BoolOperatorNode*, pops two nodes back off the stack and attaches them as left and right children to the operator node. It then pushes the operator node back on the stack. If done correctly the stack should only have 1 node (the root) remaining in which case it should return it. Feel free to add any private helper methods which could be used to determine whether a symbol is an operator or operand. The following pseudo code may be useful.

Create Stack S;

for(symbol -> expression)

if symbol is an operand

create P as new operand
push(P) onto S

if symbol is operator

create P as new Operator
pop two Nodes (or only one if unary operator) and set as P.left and P.right
push(P) onto S

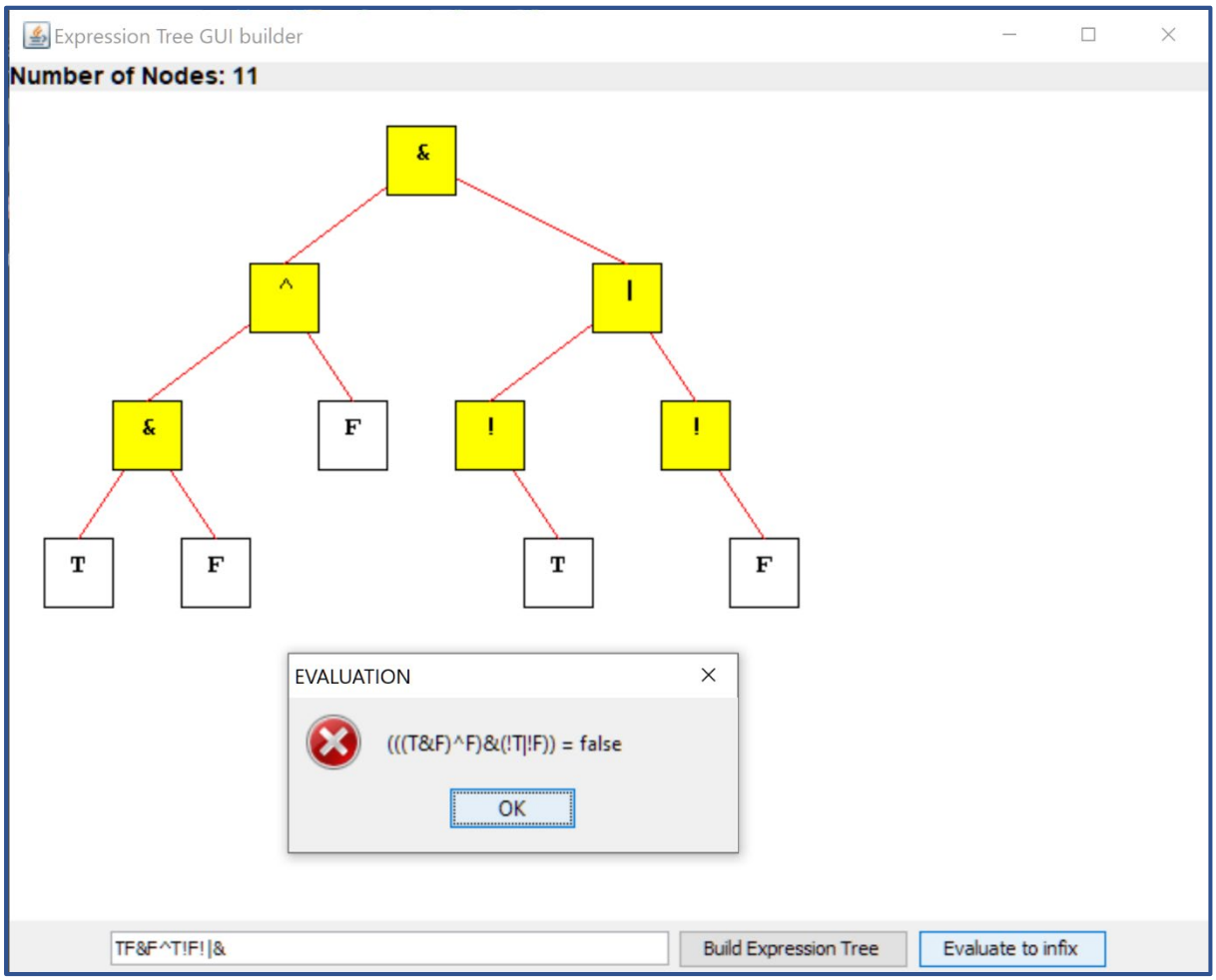
return pop P from S.

The recursive method *toInfixString* can be used to recursively convert a binary expression tree into infix notation using brackets starting from the given parameter node – **hint** use an appropriate modified tree traversal algorithm.

The method *countNodes* should recursively count the number of nodes down the left and right subtrees from the parameter node.

Obtain the **partially complete** class *BooleanExpressionTreeGUI* from Blackboard. The class can display an expression tree, drawing the tree from the root node (if not null) down. It has an input text field to enter a post fix expression and associated buttons. One button is used to build the tree, the other to evaluate the tree, outputting the infix notation string, and the resulting evaluated output. It also needs to update the *JLabel* to count the number of nodes in the tree. Note, all drawing and GUI components are already done for you, you simply need to just wire it in to work with *BooleanExpressionTreeBuilder*

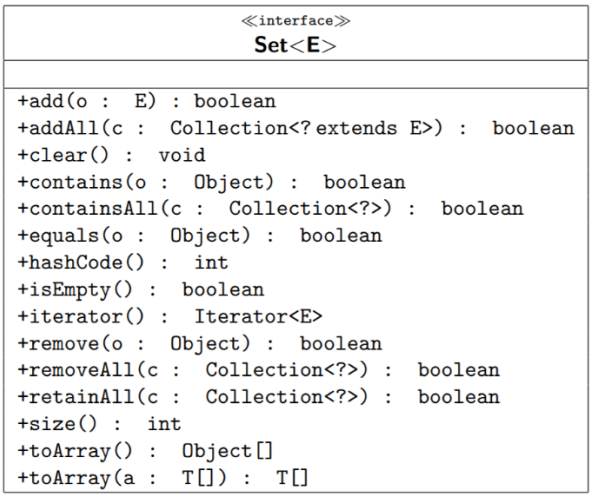
and *BoolExpNode* plus subclasses. Once working your GUI should look like something like this once tree is built and the evaluate to infix button pressed:



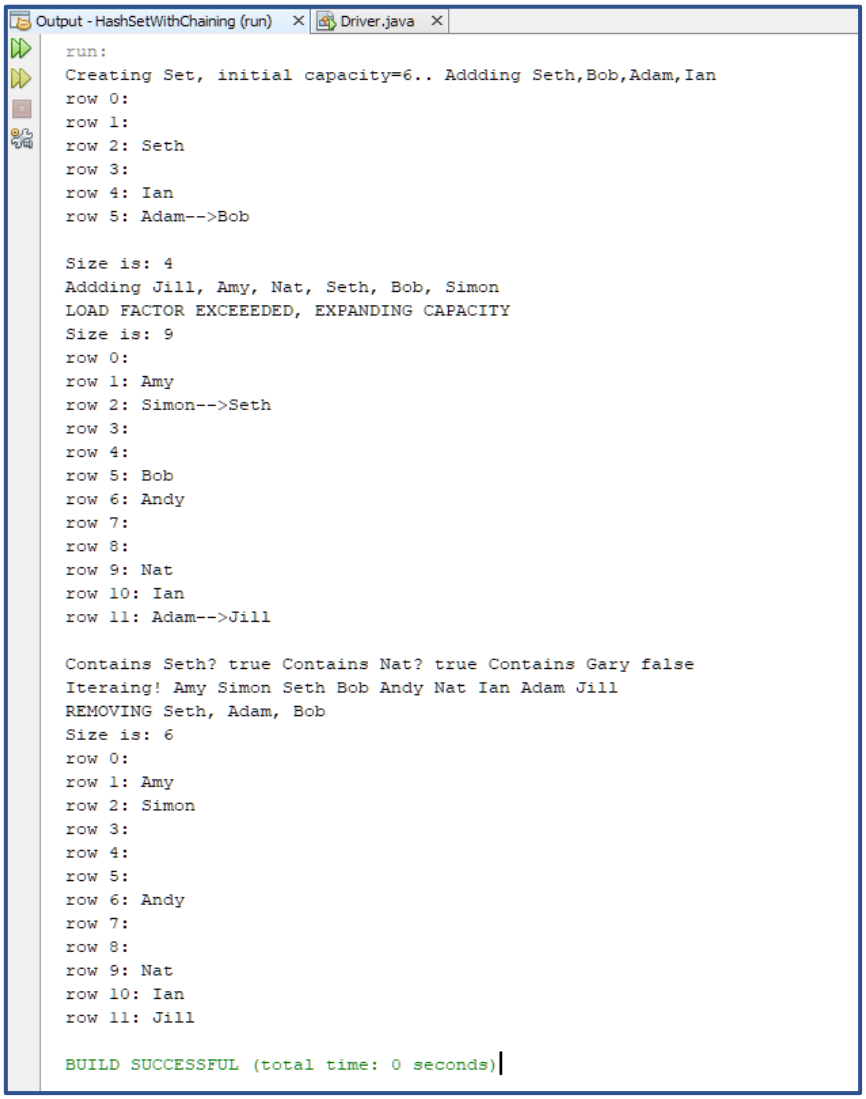
Make any necessary modifications to both the GUI and *BooleanExpressionTreeBuilder* which can handle any unexpected user input such as invalid symbols and incorrect postfix formula.

Question 3) – HashSetWithChaining (20 marks); Recommended Readings – 2.1, 2.2, 5.1

A *set* is a collection of elements with no repetitions and does not necessarily care about ordering of its internal elements. A set can be described with the *Set<E>* interface available in the *Java Collections API* and the following UML. Create a class called *HashSetWithChaining* which implements an efficient set using an underlying *hash table*, resolving any possible *collisions* by *chaining*. Feel free to look at the *Set* Java documentation to get a full understanding of how these methods should operate. Make sure you implement this class from scratch, creating your own hash table and *Node* class. Do not encapsulate an existing Java data structure.



Your set should maintain a default *load factor* of 75%, otherwise it will have to expand capacity. A custom load factor and *initial capacity* of the hash table can also be given as optional parameters during construction. Create a test class with a suitable *main* method which "effectively" tests most of the operations of your *HashSetWithChaining* class. For testing modify the *toString* method so that it prints out the entire contents of the hash table showing chained elements, where necessary. See example screenshot below of a simple test.



Question 4) Bus Journey Planner (30 marks); Recommended Readings – 5.2, 6.1, 6.2

Obtain the completed class called *BusTrip,* available from Blackboard, which represents a single trip for a bus travelling between two locations with associated departure and arrival times. A *BusTrip* has a cost (given by static stage constants) and a unique bus route identifier. A bus leaves from a *departLocation* on the *departTime* and arriving at the *arrivalLocation* on the *arrivalTime*. You can assume a *BusTrip* is express, and only end to end with no location stops in between. All bus attribute information is passed into its constructor. The class has *getter* methods for each attribute and a *toString* representation for the object. A *BusTrip* is considered *equal* to another if the departure time, departure location, and route identifier are the same.

Part A: Using the following UML diagram below create a class called *BusJourney* that represents a journey comprised of one or more *BusTrip* objects between multiple locations. The class encapsulates a *List* data structure of *BusTrip* objects, keeping the dated order of the necessary trips which comprise a complete journey. There are two constructors, firstly a default constructor creating a journey with no initial trips, secondly a constructor that adds an existing list of *BusTrip* objects to the underlying data structure.

BusJourney
- busList : List<BusTrip>
+ BusJourney() + BusJourney(list : List<BusTrip>) + addBus(bus : BusTrip) : boolean + removeLastTrip() : boolean + containsLocation(location : String) : boolean + getOrigin() : String + getDestination() : String + getOriginTime() : LocalTime + getDestinationTime() : LocalTime + cloneJourney() : BusJourney + getNumberOfBusTrips() : int + getTotalCost() : double + toString() : String

The *addBus* method should add a *BusTrip* to the journey, returning true and only adding the trip to the current journey if it meets the following criteria:

- The journey's end location is equal to the newly added *BusTrip* departure location (so the new bus trip departs from the same location as journey's current destination).
- The journey's current destination time is earlier in the day (or equal) to the newly added *BusTrip* parameter's departure time (so the new bus trip cannot be added if its departures earlier in the day to the journey's current destination time, i.e. missed the bus).
- The journey so far does not already contain the parameters arrival location somewhere in its busList (meaning the same location is never visited twice – no closed paths).

The *containsLocation* method should return true if the given location is in the journey so far. The getter methods return the *origin* and *destination* location and dates (or null) of the *Journey* (so origin location and time are the first *BusTrip's* departure values, whereas the destination location and time are the last *BusTrip's* arrival values). The *removeLastTrip* method removes the lastly added *BusTrip* from the current journey (if any). The *getTotalCost* returns the total cost of all the bus trips in this

journey. The *getNumberOfTrips* returns the number of bus trips which comprise this journey. The *toString* method should print out a string representation of all trips in this journey and the total cost in a nicely formatted way. Finally, the *cloneJourney* method returns a new *BusJourney* object, passing its *busList* to the new instance by calling the second constructor.

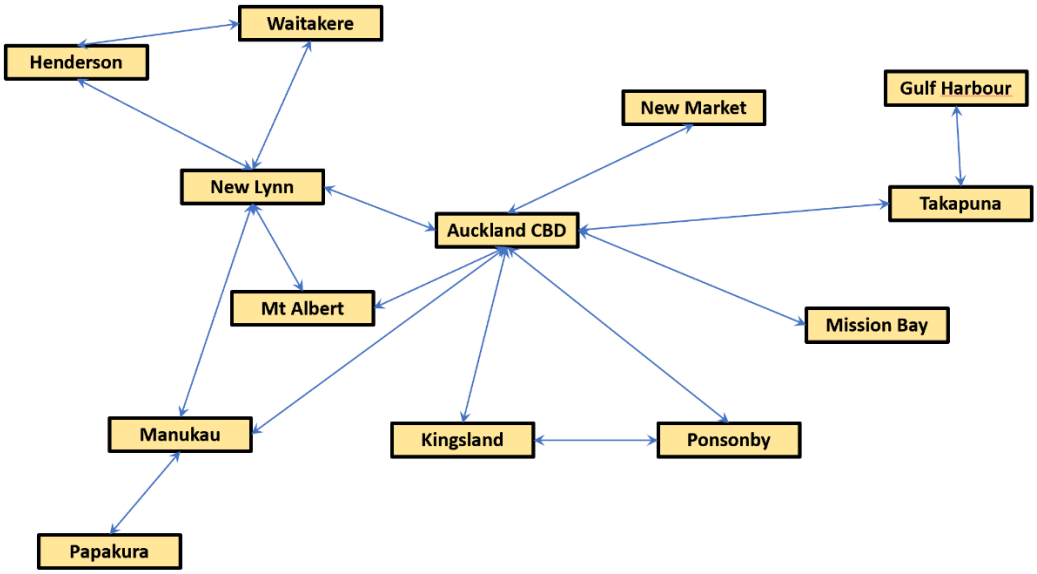
Part B: Using the following UML, create a class called *JourneyPlanner*. The class holds an efficient *Map* data structure of unique locations as keys and a **set** of *BusTrip* instances which depart that location as values. The *add* method is used to add a unique *BusTrip* to this map (care taken when the departure location already exists in the *locationMap*).

The *getPossibleJourneys* method is used to return list of all the uniquely possible routes from the start location and time, arriving at the end location before the end time. It does this by calling the recursive method *findPaths* which uses a *depth first search* technique to build up all the possible journeys that can be undertaken between the start and end location between the specified time ranges (some journeys might not exist). It does this by building up a *currentJourney* parameter and when the target location is found it will "*clone*" the current journey and add it to the *List* of *BusJourney* values found for this search. **Hint**: Use *BusJourney* like a stack while recursively finding paths.

JourneyPlanner
- locationMap : Map<String, Set<BusTrip>>
+ JourneyPlannner() + add(bus: BusTrip) : boolean + getPossibleJourneys(startLocation:String,startTime:LocalTime, endLocation:String, endTime:LocalTime): List<BusJourney> -findPaths(currentLocation:String, currentTime:LocalTime, endLocation:String, endTime:LocalTime currentJourney:CruiseJourney, journeyList:List<BusJourney>):void

Once developed, a suitable test class called *JourneyPlannerTest* (also available on Blackboard) can be used to find all *BusJourney* instances between any two locations within the specified time range. The test class adds numerous *BusTrip* objects to the *JourneyPlanner,* essentially building a graph. The graph below provides a visual representation showing the different bus trips travelling between locations. The buses all have differing schedules starting from 6am to 11pm at night. Some routes have a more frequent schedule (departing every 15 mins) whereas others less frequent (departing every two hours). Some even have different bus route identifiers between the same two locations. Use this test class to test different scenarios of travel. If you are having problems and for simple debugging, try using only a **small subset** of these trips and build your own smaller graph before attempting this one. However, make your own static *buildGraph* method, do not modify the existing one.

Once completed modify *JourneyPlannerTest*, so that once all the possible bus journeys have been found, use the *Collections* sort with *customized ordering* so that the retrieved results are printed in the order of *destination time*, so that journeys that arrive at the destination location earliest take precedence.



Example output from graph: Leaving New Lynn at 8.30am and arrive at Ponsonby before 10am:

There are 7 possible journeys for your search from New Lynn (8:30) to Ponsonby (10:00)

Journey Option 1: TOTAL COST: $11.0!!!
> BUS: 11N LEAVING New Lynn (08:30) and ARRIVING Auckland CBD (09:00) $7.5
> BUS: 00P LEAVING Auckland CBD (09:00) and ARRIVING Ponsonby (09:15) $3.5

Journey Option 2: TOTAL COST: $13.0!!!
> BUS: 11N LEAVING New Lynn (08:30) and ARRIVING Auckland CBD (09:00) $7.5
> BUS: 02P LEAVING Auckland CBD (09:15) and ARRIVING Ponsonby (09:30) $5.5

Journey Option 3: TOTAL COST: $11.0!!!
> BUS: 11N LEAVING New Lynn (08:30) and ARRIVING Auckland CBD (09:00) $7.
> BUS: 00P LEAVING Auckland CBD (09:30) and ARRIVING Ponsonby (09:45) $3.5

Journey Option 4: TOTAL COST: $11.0!!!
> BUS: 10N LEAVING New Lynn (09:00) and ARRIVING Auckland CBD (09:30) $7.5
> BUS: 00P LEAVING Auckland CBD (09:30) and ARRIVING Ponsonby (09:45) $3.5

Journey Option 5: TOTAL COST: $13.0!!!
> BUS: 11N LEAVING New Lynn (08:30) and ARRIVING Auckland CBD (09:00) $7.5
> BUS: 02P LEAVING Auckland CBD (09:45) and ARRIVING Ponsonby (10:00) $5.5

Journey Option 6: TOTAL COST: $13.0!!!
> BUS: 10N LEAVING New Lynn (09:00) and ARRIVING Auckland CBD (09:30) $7.5
> BUS: 02P LEAVING Auckland CBD (09:45) and ARRIVING Ponsonby (10:00) $5.5

Journey Option 7: TOTAL COST: $13.0!!!
> BUS: 11N LEAVING New Lynn (09:15) and ARRIVING Auckland CBD (09:45) $7.5
> BUS: 02P LEAVING Auckland CBD (09:45) and ARRIVING Ponsonby (10:00) $5.5

Example: Leaving Papakura at 10.20am and arriving at Gulf Harbour before 3pm:

There are 0 possible journeys for your search from Papakura (10.20) to Gulf Harbour (15:00)

Example: Leaving Papakura at 10.00am (20 mins earlier) and arriving at Gulf Harbour before 3pm:

There is 1 possible journey for your search from Papakura (10.00) to Gulf Harbour (15:00)

Journey Option 1: TOTAL COST: $37.5!!!
> BUS: 88P LEAVING Papakura (10:15) and ARRIVING Manukau (10:45) $7.5
> BUS: 92M LEAVING Manukau (11:10) and ARRIVING Auckland CBD (11:55) $10.5
> BUS: 44B LEAVING Auckland CBD (12:00) and ARRIVING Takapuna (12:30) $7.5
> BUS: 99G LEAVING Takapuna (13:00) and ARRIVING Gulf Harbour (14:30) $12.0

Example: Leaving Auckland CBD at 1pm and arriving at Manukau before 3pm:

There are 5 possible journeys for your search from Auckland CBD (13.00) to Manukau (15:00)

Journey Option 1: TOTAL COST: $10.5!!!
> BUS: 92M LEAVING Auckland CBD (13:10) and ARRIVING Manukau (13:55) $10.5

Journey Option 2: TOTAL COST: $21.0!!!
> BUS: 11N LEAVING Auckland CBD (13:30) and ARRIVING New Lynn (14:00) $9.0
> BUS: 82M LEAVING New Lynn (14:00) and ARRIVING Manukau (14:50) $12.0

Journey Option 3: TOTAL COST: $21.0!!!
> BUS: 10N LEAVING Auckland CBD (13:15) and ARRIVING New Lynn (13:45) $9.0
> BUS: 82M LEAVING New Lynn (14:00) and ARRIVING Manukau (14:50) $12.0

Journey Option 4: TOTAL COST: $23.0!!!
> BUS: 22N LEAVING Auckland CBD (13:00) and ARRIVING Mt Albert (13:30) $5.5
> BUS: 23M LEAVING Mt Albert (13:30) and ARRIVING New Lynn (13:55) $5.5
> BUS: 82M LEAVING New Lynn (14:00) and ARRIVING Manukau (14:50) $12.0

Journey Option 5: TOTAL COST: $10.5!!!
> BUS: 92M LEAVING Auckland CBD (14:10) and ARRIVING Manukau (14:55) $10.5

Example: Leaving Auckland CBD at 8pm and arriving at Kingsland before 9pm:

There are 4 possible journeys for your search from Auckland CBD (20:00) to Kingsland (21:00)

Journey Option 1: TOTAL COST: $3.5!!!
> BUS: 21K LEAVING Auckland CBD (20:00) and ARRIVING Kingsland (20:20) $3.5

Journey Option 2: TOTAL COST: $7.0!!!
> BUS: 00P LEAVING Auckland CBD (20:00) and ARRIVING Ponsonby (20:15) $3.5
> BUS: 10P LEAVING Ponsonby (20:15) and ARRIVING Kingsland (20:30) $3.5

Journey Option 3: TOTAL COST: $3.5!!!
> BUS: 22K LEAVING Auckland CBD (20:15) and ARRIVING Kingsland (20:35) $3.5

Journey Option 4: TOTAL COST: $3.5!!!
> BUS: 21K LEAVING Auckland CBD (20:30) and ARRIVING Kingsland (20:50) $3.5

Example: Leaving Manukau 6am and arriving Ponsonby before 11pm (large time range):

There are 16804 possible journeys for your search from Manukau (6:00) to Ponsonby (23:00)!

Note: The possible journeys for all searches above are unique, although some may have the same bus route number, the departure and arrival times are different. Also note that each consecutive bus trip in a journey departs from the same location as the previous trips arrival location and cannot depart earlier in time than the previous trips arrival time (departures and arrivals with the same time are accepted). All times specified in 24-hour format.