Hungry Students: Cafeteria Simulation

Intro:

This project is somehow similar to the previous from a structural and behavioral view of programming point.

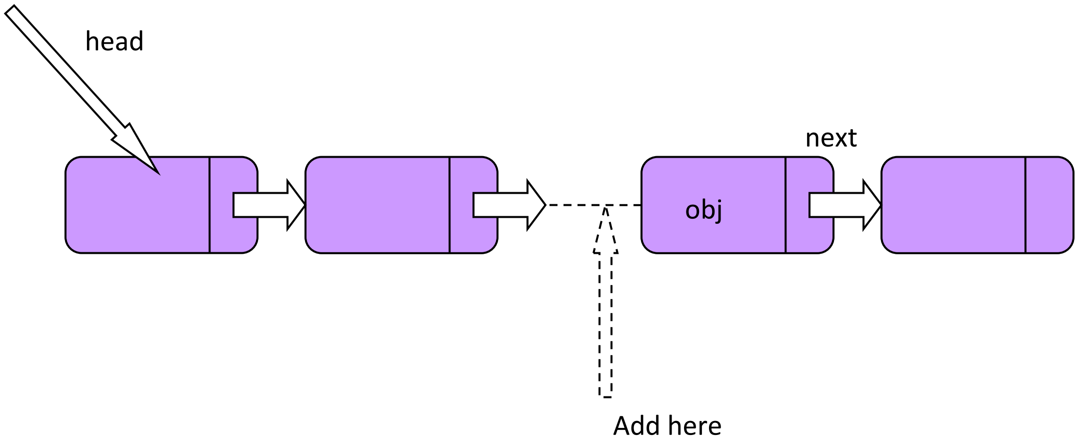
The basic structures mentioned previously MyStack, MyQueue are the same here, in the same package, but added to it additional classes to do the simulation properly.

The graphics basically works in the same way, we used the same technique of double buffering and partial image draw and update, and every object is responsible of drawing himself in his parents space.

We'll mention the added classes and the way they work.

Basic Data Structures:

Besides MyStack and MyQueue we got MySortedList



sorted linked list implementation, it is required that inserted elements implements Comparable interface.

So, the elements inserted into the list are inserted in their proper place- a sorted list- so we need not to sort the list again when we use it.

The sorting as said before depends that the inserted object implements **Comparable** interface which has one method :

Public int **compareTo**(Objetc target)

And returns 0 on equality, a negative when less and positive when greater than.

This inheritance is not forced at compile time, so the program will throw a **RunTimeException** :**ClassCastException**.

Also the list has a find method to find a relative object in the list and returns a pointer to it (an object reference), thus the inserted object should have equals () method properly overridden to tell the equality, or it will use Object equality which depends on the reference.

to iterate through the elements of this list you have to do this: call **beginIteration**() method to initiate the iteration then get the objects using **next**() method this was done to isolate the programmer from how the list is linked or even works.

Though the rules here seem a little bit tedious, yet they were meant for good, and to build a program with the least mistakes as possible, the mistakes that could be guarded by the original design of the programming elements used.

This notation is quite similar to the Java's linked list implementations, or the TreeSet class which uses almost the same rules here (Comparable, equals, iteration…)

Field Summary	
protected Node	head

Constructor Summary	
MySortedList()	

Method Summary	
void	beginIteration() starts the iteration from the head of this list
java.lang.Object	find(java.lang.Object obj) find an object in the list, the passed object must implement Comparable interface the returned object is the first the equals() this object in the list
java.lang.Object	insert(java.lang.Object obj) the inserted object must implement Comparable interface or a runtime exception ClassCastException will rise
boolean	isEmpty() check content availability
java.lang.Object	next() get next object in the current iteration
java.lang.Object	remove(java.lang.Object obj) not implemented yet
java.lang.String	toString() do a nice concatenation

Now we'll talk about the packages of this project:

Packages and their Classes:

Packages	
hungrystudents	The UI
hungrystudents.core	Core classes that make the simulation
myUtils	Basic data structures

The same talk as in the first assignment

Package myUtils

Class Summary	
MyQueue	Discussed before
MySortedList	Discussed before
MyStack	Discussed before
Node	Discussed before
SchoolSortedList	A school linked list
StudentQueue	A queue of students
StudentSoretedList	A student sorted list
TrayStack	A stack of trays

Class SchoolSortedList:

```
java.lang.Object
  └myUtils.MySortedList
      └myUtils.SchoolSortedList
```

Field Summary

java.awt.Color	color
Constructor Summary	
SchoolSortedList(int x, int y, int width, int height, int schoolWidth)	coordinates of the list and the width of each school
Method Summary	
void	draw(java.awt.Graphics2D g) draw the schools linked list and pass action to each school
School	insert(School s) insert a school and update coords
void	updateCoords(int x, int y, int width, int height) update the coords of each school upon every change in the list

a graphical representation of a school sorted linked list, this class extends the MySortedLinkedList and overrides some of it's methods to get a fine graphical view

the methods overridden are one:

insert(…), which is now take a specific parameter of type School and inserts it in the linked list, and updated the graphical coordinate of the whole list.

Class StudentSoretedList

```
java.lang.Object
  └ myUtils.MySortedList
      └ myUtils.StudentSoretedList
```

a graphical representation of a student sorted linked list, this class extends the MySortedLinkedList and overrides some of it's methods to get a fine graphical view, students sorted list is required for every school, the list is drawn vertically.

When defining the school space, this list gets it’s space too and each student inherits his own space from this class through UpdateCoords method.

Field Summary	
java.awt.Color	color color of the list
(package private) int	margin margin between school and astudent space

Constructor Summary	
StudentSoretedList()	passing the coords of the list, and the height of student space to draw against
StudentSoretedList(int x, int y, int width, int height, int studentHeight)	

Method Summary	
void	draw(java.awt.Graphics2D g) draw the students linked list and pass action to each student
Student	insert(Student s) insert a student and update coords
void	updateCoords() call update for original values [shortcut nothing more]
void	updateCoords(int x, int y, int width, int height, int studentHeight) students are drawn vertically against their schools

Class StudentQueue

```
java.lang.Object
  └ myUtils.MyQueue
```

└ **myUtils.StudentQueue**

Graphical Hungry Student Queue, it extends the MyQueue and overrides few functions for added functionality
over graphics coords drawing will be from right to left

Field Summary	
(package private) static int	margin

Constructor Summary	
StudentQueue(int x, int y, int width, int height)	passing coords of the queue we construct a graphical queue

Method Summary	
Student	dequeue() commit an update command to update each student coords
void	draw(java.awt.Graphics2D g) draw the queue shape then pass the drawing to each student in the queue
Student	queue(Student std) coordinate data are inherited from head to tail
void	updateCoords(int x, int y, int width, int height) method is called upon every change in the queue structure

As seen above the only method overridden are queue and dequeue because they deform the shape of the queue upon their call, so their implementation is simple:

Call parent implementation over the passed object

Call updateCoords(…) to validate the change.

Drawing of the queue is like drawing anything else in these two assignments: typically draw the object space then draw his children

Class TrayStack

```
java.lang.Object
  └myUtils.MyStack
      └myUtils.TrayStack
```

Graphical Tray data, this class has the stack functionality and few of his own drawing will be from right to left

Field Summary	
java.awt.Color	color color of stack
(package private) static int	margin between stack and trays

Constructor Summary	
TrayStack(int x, int y, int width, int height, int trayWidth)	

Method Summary	
void	draw(java.awt.Graphics2D g) draw stack space then pass drawing command to trays
Tray	pop() pop top from stack
Tray	push(Tray tray) coordinate data are inherited from the head each insertion propagates change through the whole stack
void	updateCoords(int x, int y, int width, int height, int trayWidth) upon each change in the stack, this method gets called

really nothing to talk about.

Now the second package in this project:

Package hungrystudents.core

Class Summary	
CafeteriaCoords	Holds main objects coordinates in the cafeteria
Controller	Assembles everything and does the logic
LogMsg	Discussed before
Prices	Holds list of static values(prices)
Raw	A raw is: a hungry student queue and two stacks of trays
School	a school has info and a list of students
Student	A student has a info and belongs to a school
Tray	A tray has a color

Class CafeteriaCoords

Field Summary	
static java.awt.Rectangle	dessert not drawn(unimportant)
static java.awt.Rectangle	entree not drawn(unimportant)
static java.awt.Rectangle	foodPlace has stacks of trays and food

static java.awt.Rectangle[]	hungryStds two queues
static java.awt.Rectangle	salad not drawn(unimportant)
static java.awt.Rectangle	schools 6 schools and their students
static java.awt.Rectangle[]	trays 4 stacks of trays inside the food place

All main visual objects in the scene have space to draw them selves inside, these spaces are defined here and they are all relative to each other and to the total width and height of the target graphics.

Class Prices

Field Summary	
static double	cheesecake
static double	chicken
static double	fishsticks
static double	lasagna
static double	pudding
static double	salad

All static values, change these to change the final bill.

Class Raw

Field Summary	
(package private) StudentQueue	stdQueue

(package private) TrayStack	trays1
(package private) TrayStack	trays2

simple class to represent a raw in the cafeteria, a raw consists of a hungry student queue and two stacks of trays, this class was added just to ease iteration.

Class School:

a school has a name and a sorted list of students, and two colors.
since schools are added to list them selves they should implement Comparable interface
a school knows how to draw it self and passes it's drawing command and surface to the students to draw them selves too.

Field Summary	
java.awt.Color[]	colors
java.lang.String	name
StudentSoretedList	studentList

Constructor Summary	
School()	
School(java.lang.String name)	construct school by name only, colors are set to default tell they are set from else where

Method Summary	
double	calcBill() bill of a school is the sum of it's students bills

int	compareTo(java.lang.Object obj) comparison of two schools depends on their names
void	draw(java.awt.Graphics2D g) draw school space and call student list to draw it self
boolean	equals(java.lang.Object obj) equality of two schools will depend only on school name
double	getBill() control access to bill private data you must call calcBill() first
java.lang.String	toString() concatenate the school name and it's student info too
void	updateCoords(int x, int y, int width, int height) control access to coords private data and update student list coords too

Class Student:

a student knows all about him self, his own info and his bill, and his school too
it also can draw it self.
this class implements Comparable interface because it will be added to a MySortedList object.

Field Summary	
(package private) java.awt.Color	color
java.lang.String	dessertName
java.lang.String	entreeName
java.lang.String	name
int	ouncesSalad
School	school

Constructor Summary	
Student()	

Method Summary	
double	calcBill() depending on his requested food the bill is calculated
int	compareTo(java.lang.Object obj) comparing two students using names (in the same school)
void	draw(java.awt.Graphics2D g) draw a student in his space
double	getBill() return the bill value [call it after calculation]
java.lang.String	toString() view it in a fine way
void	updateCoords(int x, int y, int width, int height) control access to coords private data

Class Tray:

a single tray, has a color and knows how to draw itself

Field Summary	
java.awt.Color	color

Constructor Summary	
Tray()	

Tray(java.awt.Color c)	every tray has a color
Method Summary	
void	draw(java.awt.Graphics2D g) draw the try
void	updateCoords(int x, int y, int width, int height) just to control access to it's private coord data

Class Controller:

this class assembles all other objects to work together and simulate the cafeteria
the controller has: hungry students queues, tray stacks, a schools list and much more
all graphical info is held her.
drawing is done using double buffering, and clip repaint only so: if a tray stack is changed it's alone is redrawn and nothing else, so motion is very smooth and consumes the less processing possible.

Nested Class Summary	
(package private) class	Controller.SimulationThread
Field Summary	
boolean	enableSwitching stops simulation if set to true
int	height height of target graphics
(package private) StudentQueue[]	hungryStds two
static java.awt.image.BufferedImage	image off screen image

java.awt.Graphics2D	img the image graphics
(package private) long	randNumSeed random number seed
(package private) SchoolSortedList	schools 6
long	step time step between two changes
java.awt.Graphics2D	targetGraphics on screen graphics(should be) to view final results
(package private) TrayStack[]	trays 4
javax.swing.JFrame	ui (bad reference for tests)
int	width width of target graphics

Constructor Summary	
Controller(java.lang.String dataPath, int width, int height, java.awt.Graphics2D g)	passing a file path to load data from, width and height of the target surface and it's graphics

Method Summary	
void	beginSimulation() fire a new thread to begin the simulation and view motion on the screen
void	billSchools(java.lang.String fileName) each school bills it self
Void	generateTrays(long seed) this method will accept a seed of a random number generator, which will be used to generate a number from 0 and less than 6 and choose one of 6 colors and 20 trays of each color to add to the tray stack, which in turn : there are 4 tray stacks in the cafeteria, each will have 30 random trays to hold all trays

void	loadSchoolsData(java.lang.String filename) we assume this: initially students are considered hungry, so they are all added to the hungry students queue later when a student is served he's marked as SERVED and removed from the hungry students queue and added back into his school this method loads the data and calls the tray generation method file is expected to hold information of all students and their meals in the following format: SeedNumber [first line only as a seed for the random numbers generation] StudentName SchoolName Number EntreeName DessertName .
void	viewSchoolsData() view school data on the standard output

This class has the magic that plays with all other classes and populates the simulated images onto screen.

For generation of trays this what happens:

The random number is generated by the Java.util.Random class that has a constructor that accepts a SEED and gets the next value using nextDouble() for instance,

So the target is as explained before:

accepting a seed of a random number generator, which will be used to generate a number from 0 and less than 6 and choose one of 6 colors and 20 trays of each color to add to the tray stack, which in turn : there are 4

so 120 trays, 20 trays of each color, 30 trays in each tray stack, and randomly distributed over the 4 stacks:

Generate a random number using the seed and normalize it to 6

Initialize color vector of 6 elements, another vector of 6 elements to count the 6

Colors, and a vector of 4 elements to count the trays stacks

```
Color[] colors = new Color[6];
int[] count = new int[6];
int[] stacks = new int[4];
for (i=0;i<4;i++){ //4 stacks of trays
        While(stacks[i]<30){
            if ((randNum >= 0) && (randNum < 1) && (count[0] <= 20)) {
                   trays[i].push(new Tray(colors[0]));
                    stacks[i]++;
            } else if ((randNum >= 1) && (randNum < 2) && (count[1] <= 20)) {
                    trays[i].push(new Tray(colors[1]));
                    stacks[i]++;
            } else if ((randNum >= 2) && (randNum < 3) && (count[2] <= 20)) {
                    trays[i].push(new Tray(colors[2]));
                    stacks[i]++;
           } else if ((randNum >= 3) && (randNum < 4) && (count[3] <= 20)) {
                    trays[i].push(new Tray(colors[3]));
                    stacks[i]++;
           } else if ((randNum >= 4) && (randNum < 5) && (count[4] <= 20)) {
                    trays[i].push(new Tray(colors[4]));
                    stacks[i]++;
           } else if ((randNum >= 5) && (randNum < 6) && (count[5] <= 20)) {
                    trays[i].push(new Tray(colors[5]));
                    stacks[i]++;
            }
```

```
                }

        }
```

The previous will do the work just fine and get the effect needed.

The begin simulation method, fires a thread of the next internal class:

Class Controller.SimulationThread

```
java.lang.Object
  └ java.lang.Thread
      └ hungrystudents.core.Controller.SimulationThread
```

when this thread runs it does the simulation of the cafeteria tell there are no hungry students left the thread can be stopped by setting the enableSwitching variable of the parent Controller object and view speed can be controlled by the step value of the same object

Field Summary	
(package private) java.awt.image.AffineTransformOp	op

Constructor Summary	
Controller.SimulationThread()	

Method Summary	
void	commitScene() all changes to the image draw is commited to the screen now
void	drawCaf (java.awt.Graphics2D g, int width, int height) initially we draw the cafeteria it self
void	**run**() do the trick
void	**step**() one step by sleeping for a while and updating the figure drawn

Methods of this class does the same as those in the SwitchingThread in the previous assignment, with drawCaf as drawYard there, and run here as run there with different heart.

It’s algorithm is something like this:

```
While(enableSwitchng){
    If both hungry student queues are empty then return
    If either raw has no more trays then miss happens(return)
    For each raw do a similar work:{
        Pick first tray stack (the longest)
        Set the second tray stack to the shortest
        While(true){
            If(first stack empty then)
                 reverse first and second handlers)
                if (this round 2) then
                    pick top of second stack and break the loop
            Else{
                Peek top tray
                If color matches one of those for the peeked student then
                    Pop tray away and pop student into his school
                Else
                    Pop tray from first stack onto the second stack
            }
        }
    }
}
```

Graphics:

Nothing to talk about, it’s exactly as in the first assignment, done in the same principle and uses the same methods.

Algorithm Complexity:

The most significant methods are listed bellow along with their Complexity in formal context.

Method name	Method algorithm	complexity

TrayStack.push(…)	Super.push(tray)
	updateCoords(…)
		C(updateCoords)+1=O(n)

TrayStack.pop(…)	Super.pop()
	updateCoords(…)
		C(updateCoords)+1=O(n)

TrayStack.updateCoords(…)	Temp=head;
	While(temp!=null){
	
	updateCoords(temp.obj)
	}
		O(n) where n is a simple update operation of 4 value assignments

TrayStack.draw(…)	Temp=head;
	While(temp!=null){
	Temp.trayobj.draw(g)
	}
		On(n) where n is a drawing operation of a single rect

All other methods of MyQueue and MySortedList matched the complexity of the MyStack including my StudentQueue and MyStudentSortedList, complexities of O(1) or O(n), the only one that differes from the basic structure classes is the SchoolSortedList because it’s a list and each element has a list		

SchoolSortedList.insert(…)	Super.insert(school)
	updateCoords(…)
		O(n)+C(updateCoords)

SchoolSortedList.draw(…)	Temp=head;
	Draw the list space
	While(temp!=null){
	Draw a school
	Temp=temp.next
	}
		=O(1)+O(n)*O(schoolDrawing)= O(n2) where n is a simple drawing operation

SchoolSortedList.updateCoords()	Temp=head
	While(temp!=null){
	Update(temp-school)
	Temp=temp.next;
	}
		O(n2) also of a simple operations

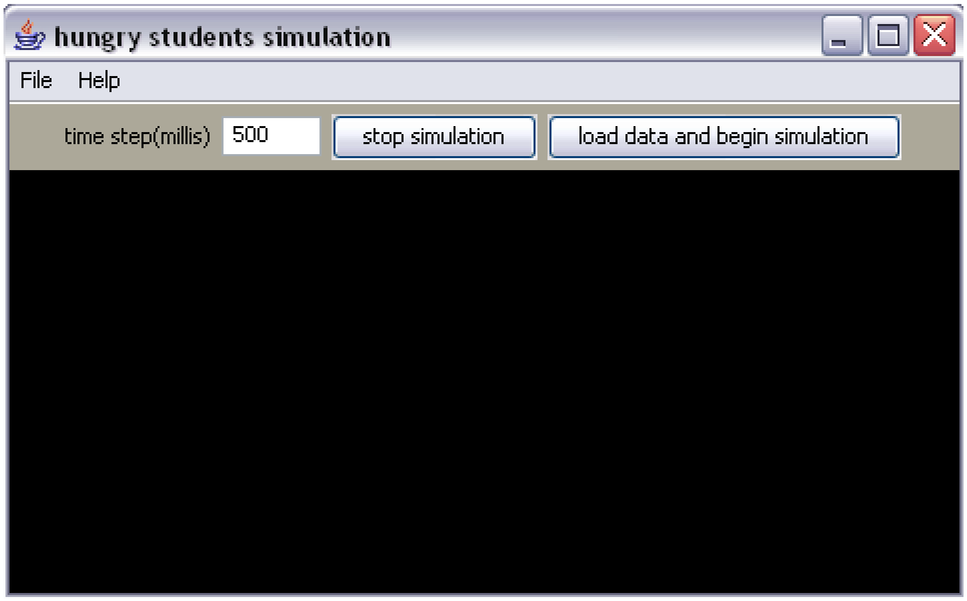
School.UpdateCoords(…)	Update current and children list	O(n)
School.draw(…)	Draw current and pass to children	O(n) of simple drawing

Controller. generateTrays(…)	for (i=0;i<4;i++){
	While(stacks[i]<30){
	if (something)
		4*30*O(n) if assumed 4 and 30 as constants then it’s
		O(n)

	trays[i].push()	If they were variables it could be O(n3) cubic
SimulationThread.run()	Algorithm above and can be assessed by	O(n)*(O(n)+O(n2))= O(n3) cubic
Darwing of schools space is actually done once and before the simulation begins, that's why it was omitted from the complexity calculation of the last function which other wise could be a quadric function. So what happens that through running the simulation: that only the schools student list is redrawn[not the school it self or the schools list itself] So drawing calculations are minimized almost to the least it can.		

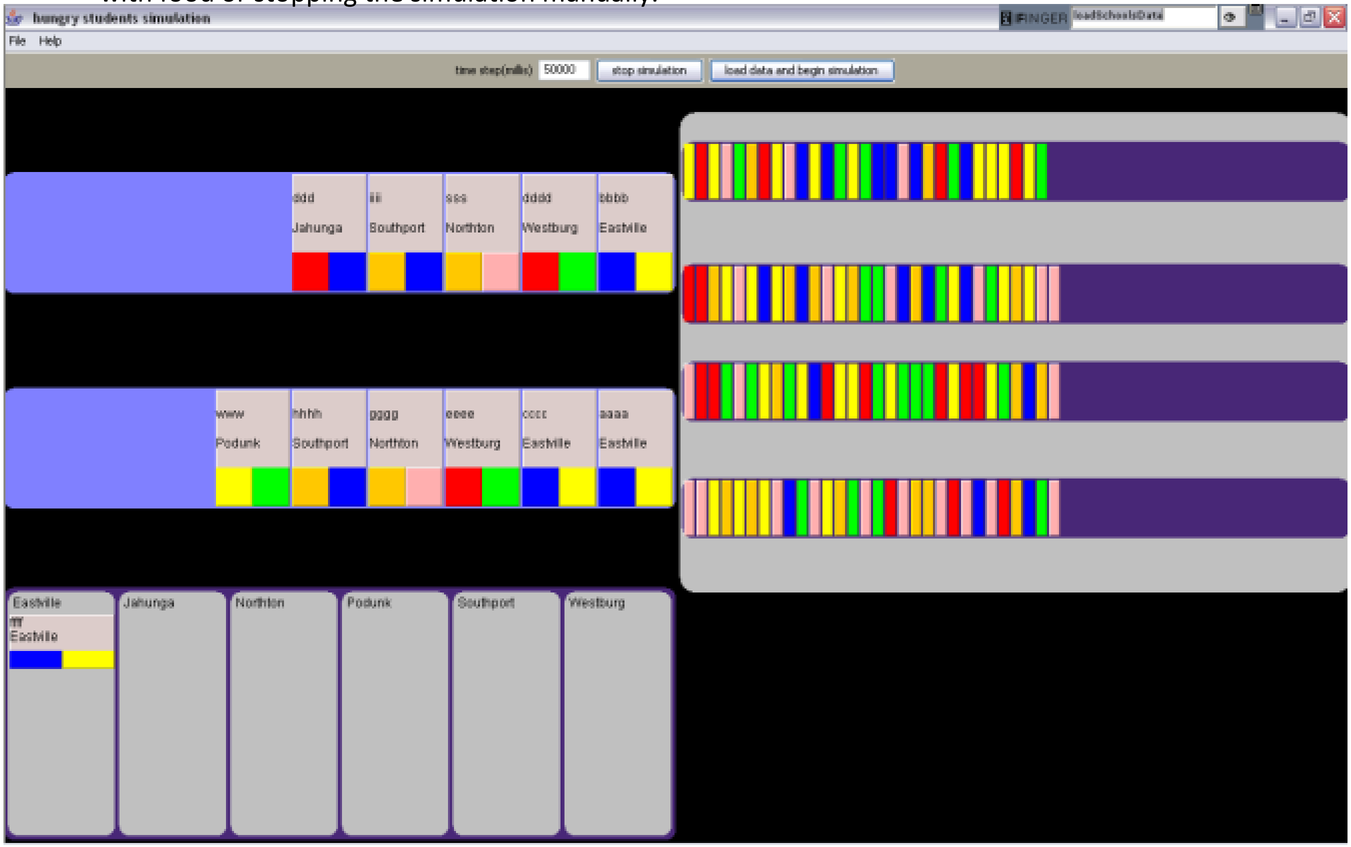
Now we'll see the UI and some drawing results:

Initially:

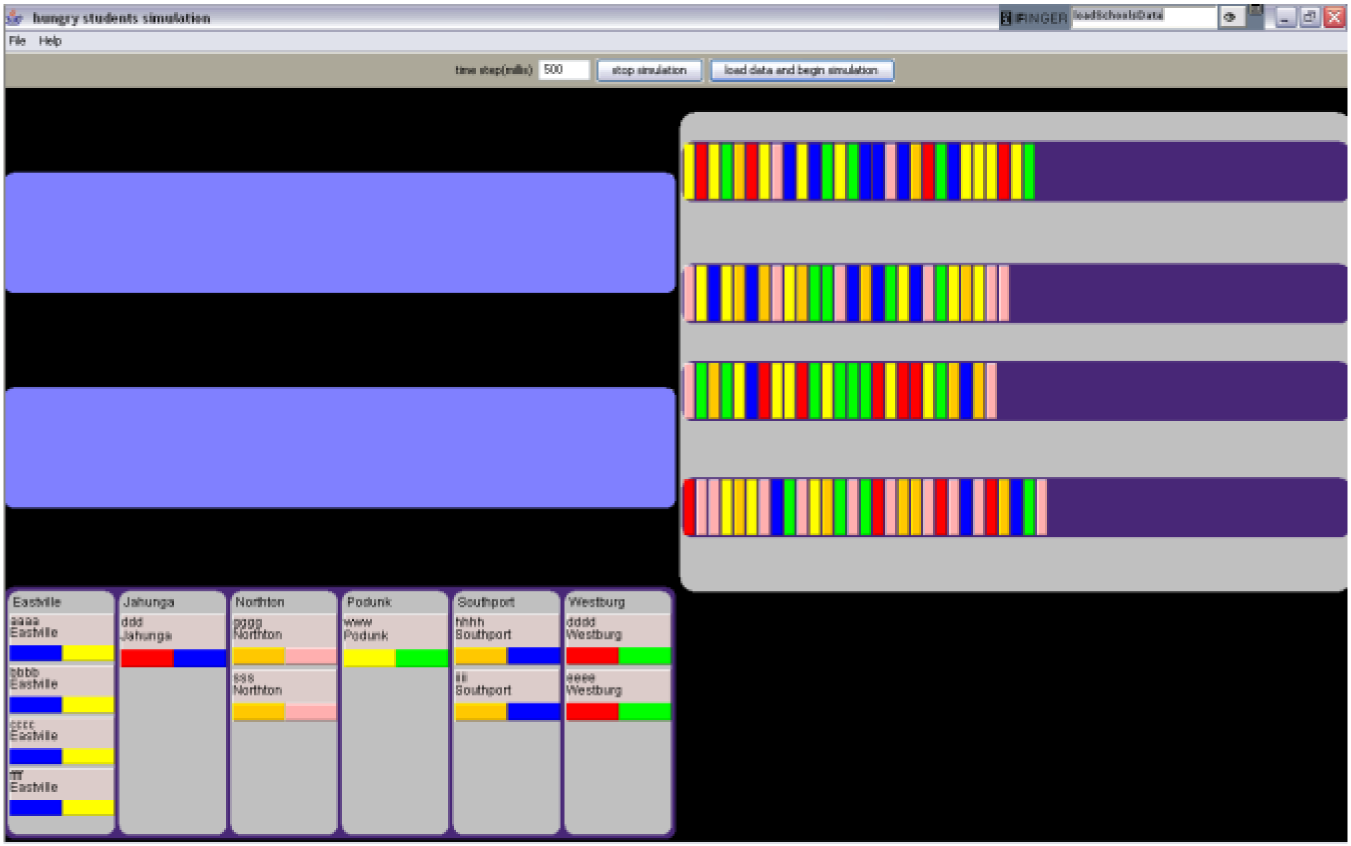


time step(millis) 500

After selecting the students file the simulation begins and goes on until all students are stuffed with food or stopping the simulation manually.



The simulation at the begining



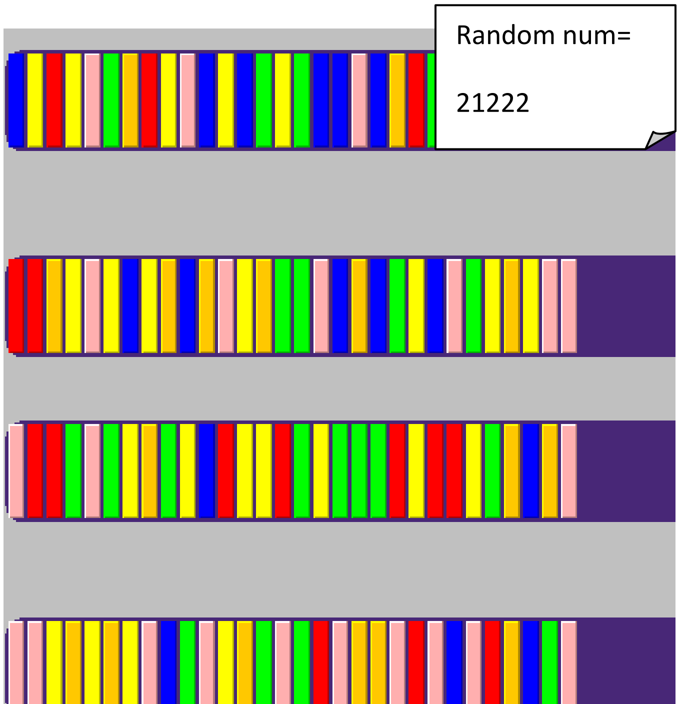
time step(millis) 500

The simulation was done

The standard output looks like this:

```
Schools listed:

            Eastville:

            Jahunga:

            Northton:

            Podunk:

            Southport:

            Westburg:


Students in hungry queue[0]

            ffff --->[chicken,cheesecake,1]

            bbbb --->[chicken,cheesecake,1]

            dddd --->[fishsticks          ,pudding,2]

            sss --->[lasagna,cheesecake,2]

            iiii --->[lasagna,cheesecake,2]

            ddd --->[lasagna,cheesecake,2]


Students in hungry queue[1]
            aaaa --->[chicken,cheesecake,1]

            cccc --->[chicken,cheesecake,1]

            eeee --->[fishsticks,pudding,2]
```

```
        gggg --->[lasagna,cheesecake,2]

        hhhh --->[lasagna,cheesecake,2]

        www --->[lasagna,cheesecake,2]


University of:Eastville
        aaaa:8.75
        bbbb:8.75
        cccc:8.75
        ffff:8.75
Total:35.0


University of:Jahunga
        ddd:9.25
Total:9.25


University of:Northton
        gggg:9.25
        sss:9.25
Total:18.5


University of:Podunk
        www:9.25
        Total:9.25


University of:Southport
        hhhh:9.25
        iiii:9.25
Total:18.5


University of:Westburg
        dddd:4.25
        eeee:4.25

Total:8.5
```



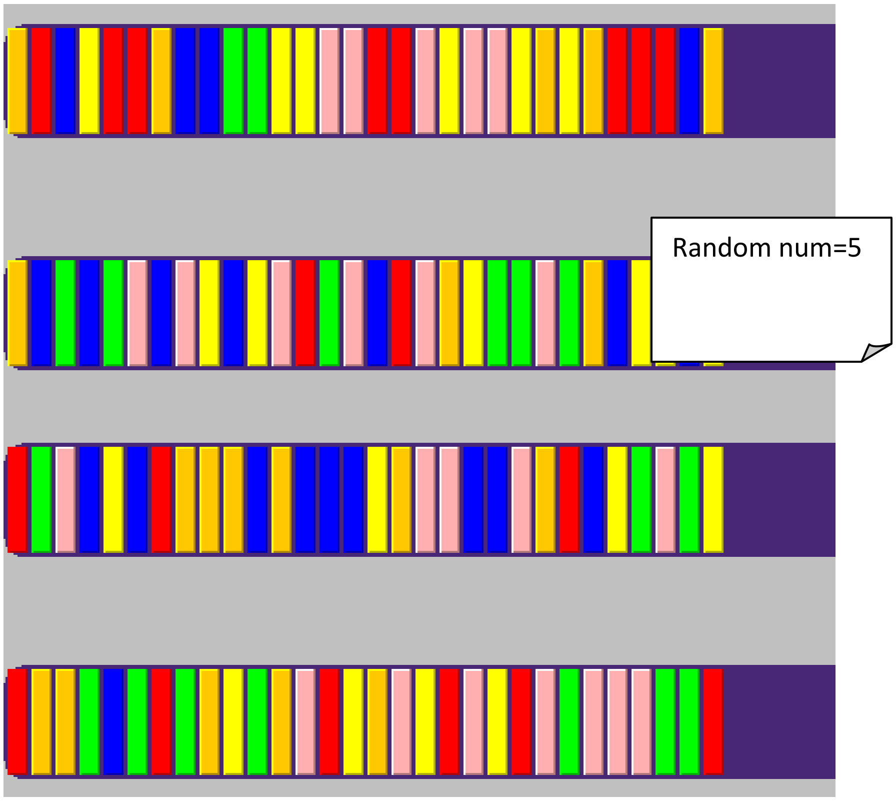
Random num=

21222

Random num=

33222

Samples of tray generation



The End.