Gillespie Documentation

Connor Kordes

July 17, 2015

Contents

1 Introduction

This paper intends to document the problems, solutions, and progress of creating a versatile Gillespie library in C++. The Gillespie Algorithm is one that creates a statistically correct trajectory for a stochastic equation. The problem in making a library or algorithm in C++ for this function is that the Gillespie algorithm is used in a plethora of cases, each of which contains its own unique conditions. Additionally finding an efficient way to store and search values with this algorithm is another problem in creating this tool. As to understand the rest of the introduction, a quick side note will be made on how the Gillespie Algorithm works at its core.

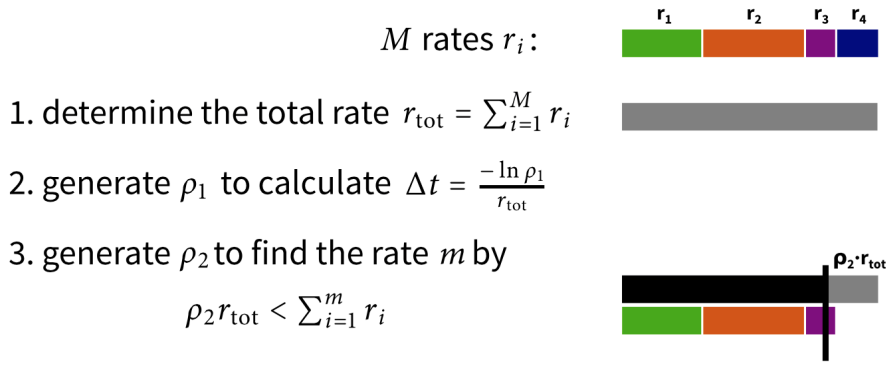
1.1 Gillespie Algorithm

Again, the Gillespie Algorithm is used to portray an accurate model. The model usually contains molecules, rates, and state changes. The molecules each contain rates which act as a probability for a certain state change.

The Algorithm

1. One must initialize the number of molecules, rates, and states.
2. Next one needs to generate random numbers that help decide the next time change increment, as well as which state change will occur through selecting a rate.
   - Use Mersienne Twister random number generator to create a random number γ_1
   - calcaulate the $\Delta Time$ with $\frac{\ln \gamma_1}{\sum R}$ with the $\sum R$ being the sum of the rates.
   - Lastly one needs to produce a place value which is created by creating a second random number in the range [0,1). Then they need to multiply this by $\sum R$ and choose a rate that falls in the domain of this.
3. One needs to update the the variables initialized in the beginning based on the selected rate and other factors.
4. Lastly one should start again at step 2 either until they have reached the desired time or there population has reached a desired state.

The second step is also well explained by this diagram



Photos/GillespieSteps.png

This sounds fairly general mostly because it is. This algorithm is extremely versatile which makes it difficult to implement in a generalized way. The most important part in making this algorithm however is finding an efficient way of storing, accessing, and modifying the rates because this is the one part that the user will not have controll over.

2 One Dimensional Walker

The first challenge was to get a working version that implemented some model. One of the most simple and well known models is the one-dimensional walker model, which models an organism that can only move in two directions along a line. This model thus only has two rates: one that changes its position positively and another that changes it negatively. THe first step taken was implementing a Gillespie class with the following private variables:

```
    double currentTime;
    double position;
    double sum;
    int seed;
    int limit;
    std::vector<double> rates;
    std::vector<double> cummlativeRates;
    std::vector< std::vector< double > > data;
```

With basic constructors to initialize all of the variables, them main function was run which is listed here:

```
void run(){
    double place = 0;
    double deltaT = 0;
    std::mt19937 mt_rand;
    mt_rand.seed(seed);
    auto die = std::bind(std::uniform_real_distribution<double>(0,1), mt_rand);
    while(currentTime<limit){
        deltaT = (-log(die()))/sum;
        currentTime+=deltaT;
        place = die()*sum;
        for (double i = 0; i<cummlativeRates.size();i++){
            if (place<cummlativeRates[i]){
                if (i==1)
                    position=position+1;
                else
                    position=position-1;
                data.push_back({currentTime,i,position});
                break;
            }
        }
    }
}
```

This function is where the main algorithm takes place. The data (current time and position) is stored into the data vector which is then outputted to a .txt file that is interpretted and graphed by a python file. Again the end goal of this program is to be versatile and fast - at this stage it is neither of which. The user can make slight adjustments to the rates and limit, but beyond this there is little that the user can do to implement this into their more complicated program. In terms of speed, the only thing that makes this program slightly faster than a standard one is that it stores the largest rates at the beggining of the cummulative vector so that the random place that is selected has a larger chance of finding these more probable rates first. The last problem with this implementation is that there is no way to know which rate was selected because there is no key value associated with the rates so if 2 rates are identical there will be no way to distinguish between them. The next program - Two Dimensional Walker - was meant to be more versatile but no more efficient.

3 Two Dimensional Walker

3.1 Two Dimensional Walker - Rate Structure

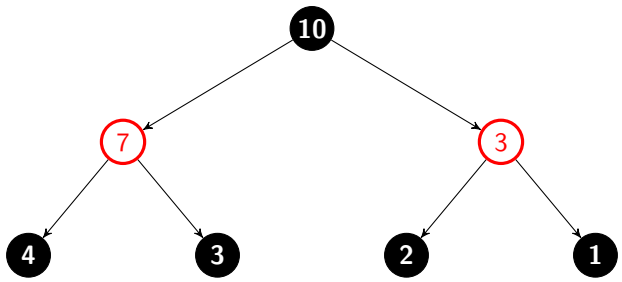
The first problem to tackle was supporting more rates and giving these rates an additional key that would allow them to be identified by the user further allowing them to change a state corresponding to the identified key. This sounds extremly similiar to a map, but a map coudn't be used for these specific purposes so a maplike structure called RateStruct was created in place. It served the same purpose as the rate vector in the previous one dimensional walker however a find function was implemented in this class instead of the Gillespie class and the Rates now had a key value that allowed them to be interpretted by the user. In this case I interpretted the keys like so:

```
//ratestructure is rs
 double vecPos = rs.find(place);
        switch ( (int)vecPos )
        {
            case 1:
                positionY++;
                break;
            case 2:
                positionY--;
                break;
            case 3:
                positionX++;
                break;
            case 4:
                positionX--;
                break;
            default:
                break;
        }
```

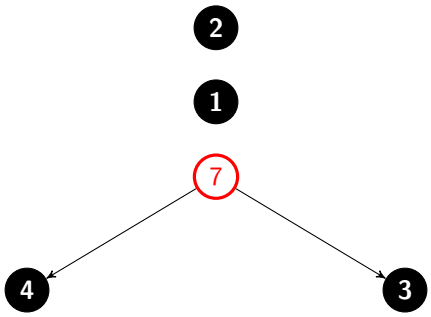
Only four rates were entered, each one corresponding to a change in X or Y position. The problems with this algorithm was that it still lacked versatility and speed. It was implementing the same find() method as the one dimensional walker and the user has no way of adjusting the switch statement unless they of course manually adjust the code. Additionally, many implementations need an ability to support mulitple organism however this can only support one which can move left, right, up, and down. With all of these problems, the most important to tackle was speed. Many understand basic switch statements but few understand how to create and efficient find function.

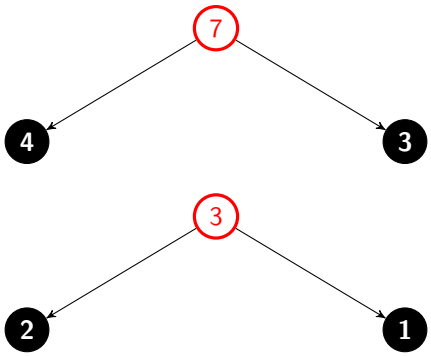
3.2 Two Dimensional Walker - BinaryTree

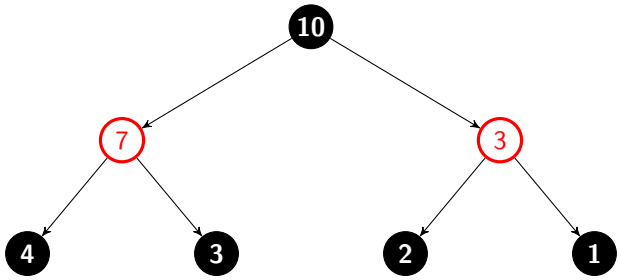
There is an efficient method to search rates given that they are stored in a binary tree. To store them you make each rate a leaf and the parent node a sum of the two children. For example if one had the rates 1,2,3,4 and wanted to store them in a binary tree it would look like this:



Programatically, the user inputs a vector of doubles that looks similiar to: [[1,10],[2,11],[3,4]] where the first number is the key and the second the rate. From here they are initialized as leaf nodes - their left and right pointers are initialized as nullptrs - and then are added to a queue. The queue is used to initialize as shown above. It pops the first two elements off the front and then adds a new Node to the back that has the sum of the two popped nodes as the parent node and the two popped nodes as the children. It continues this process untill there is only one element left in the queue which will be the entire tree. Graphically it looks something like this: lets say the queue is initialized with rate values 1,2,3,4.







Here it shows shows how it was dones in C++:

```
BinaryTree(std::vector< std::vector< double > > r)
{
    std::vector< Node* > q(r.size());
    for (int i = 0; i < q.size(); i++)
    {
        q[i] = new Node(r[i][1],r[i][0]);
    }

    Node* rate1;
    Node* rate2;

    while (q.size()>1)
    {
        rate1 = q.front(); q.erase(q.begin());
        rate2 = q.front(); q.erase(q.begin());

        Node* parent = new Node(rate1,rate2);
        q.push_back(parent);
    }

    head = conductor = q[0];
}
```

One next has to traverse the tree to find a rate given a place. To do this the following method was created.

1. if the double place is greater than the left node than go right and make place equal to what it was previously minus the left node
2. On the other hand go left
3. Repeat until you have reached a leaf
4. This is the rate that you have chosen

Lets say the place 8 was chosen. 8 is greater than 7 so we should subtract 7 and go right. Now place is 1. 1 is less than 2 so we go left. We have arrived at 2 which is the final rate.

This is what it looks like in C++:

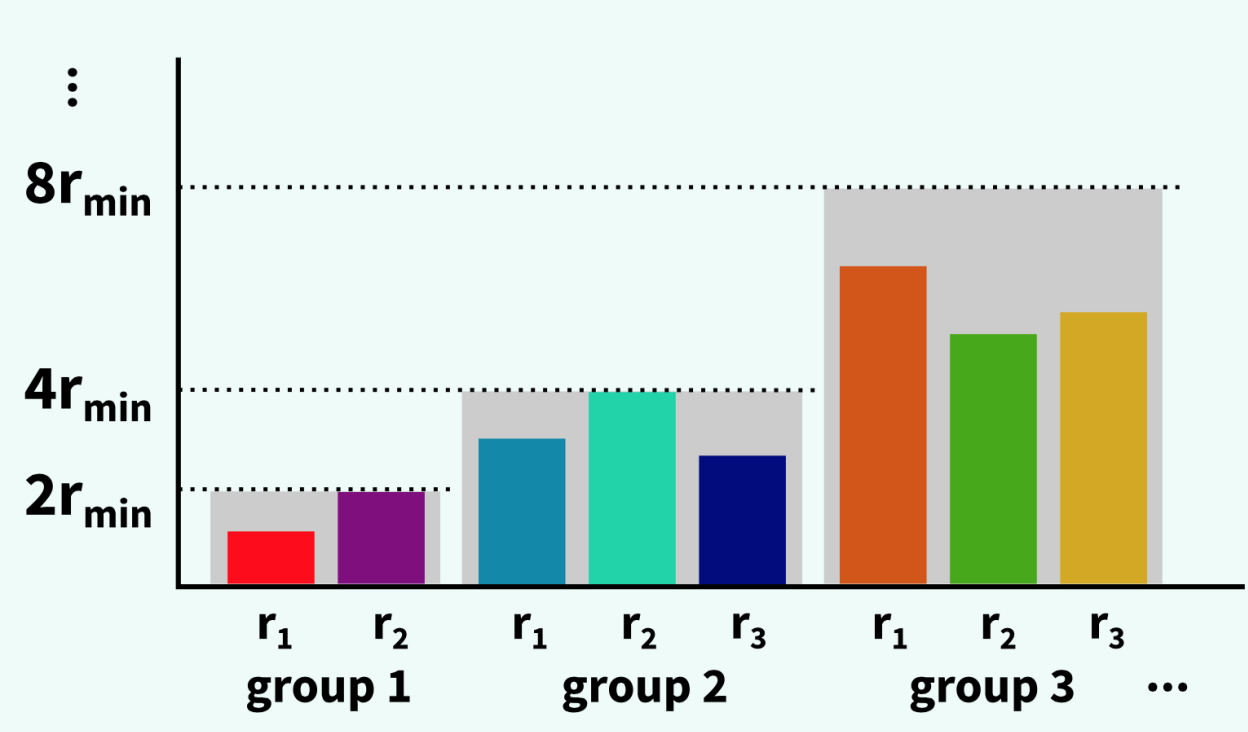
```
double find(double place)
{
    conductor = head;
    while (place<conductor->getRate() && conductor->getRight() != NULL)
    {
        if (place>conductor->getLeft()->getRate())
        {
            place -= conductor->getLeft()->getRate();
            conductor = conductor->getRight();

        }

        else
        {
            conductor = conductor->getLeft();
        }

    }

    return conductor->getKey();
}
```

This method is much faster with O(log n) time instead of O(n). However the program can still handle few cases and their is one other well known implementation that is faster than this given that the rates are very different from eachother.

3.3 Composition and Rejection

Composition and Rejection follows a completely different method than the original Gillespie Algorithm. The first step necessary is to sort the rates into group according to the minimum rate inputed. Each succesive group should have rates that are two times the previous group with the first group being two times the minimum. This graphic demonstrates what the results should look like given a set of rates.

Photos/CompGroups.png



As one can see, each group has elements that are succesively large than the last which will help make things more efficient when rejection is implemented.

The rejection algorithm is similiar to Gillespie except it follows a different pattern.

1. One must select a group similiar to how rates were selected in the original Gillespie. This time however, each group is treated as a rate that is the sum of its elements
2. Once the group is selected. One should next choose a rate. However this process is different than Gillespie. Let's say we are looking at group two.
3. A new random number from 0 to 1 is generated and multiplied by the group height which in this case is 4 times the minimum rate
4. Then one randomly selects a rate in the group by generating a random integer that is within the bounds of the rates vector that is holding them
5. If number created in step 3 is larger than the selected rate than one should return to step 3 until a rate is larger than the number created in step 3.

To create this in C++ two structures were created. A group structure that holds the sum of the elements, which level it is, and the elements themselves. The rest of the functions in this structure are made to support the find method which is represented by steps 3 and downward. The other structure created was the Composition structure which initializes all the rates in their corresponding level and perfroms step 1.

The group structure is here:

```
class Group{
private:
    int level{0};
    double gSum{0};
    std::vector< std::pair<double, std::pair<int, int>>> elements;
public:
    Group() = default;
    Group(int l){
        level = l;
    }
    void add(std::pair<double,std::pair<int, int>> p){
        gSum+=p.first;
        elements.push_back(p);
    }
    std::pair<double,std::pair<int, int>> find(double compMin){
        double levelHeight = pow(2,level)*compMin;
        auto seed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
        std::mt19937 mt_rand;
        mt_rand.seed(seed);
        auto die = std::bind(std::uniform_real_distribution<double>(0,1), mt_rand);
        auto vecPlace = std::bind(std::uniform_int_distribution<int>(0,elements.size()-1), mt_rand);
        std::pair<double,std::pair<int, int>> current = elements[vecPlace()];
        double place = die() * levelHeight;
        while(current.first<place){
            current = elements[vecPlace()];
            place = die() * levelHeight;
        }
        return current;
    }
    double getGSum(){return gSum;}
    void updateGSum(){
        gSum = 0;
        for (int i = 0; i<elements.size();i++)
            gSum+=elements[i].first;
    }
};
```

And the main initializing step to the Composition structure is here:

```
Composition(std::vector< std::pair<double,std::pair<int, int>>> r){
    min = *std::min_element(std::begin(r),std::end(r),[]( std::pair<double,std::pair<int, int>> r1,
        std::pair<double,std::pair<int, int>> r2){return r1.first<r2.first;});
    Group a(1);
    groups.push_back(a);
    for (int x = 0; x < r.size(); x++){
        for (int y = 0; y < groups.size();y++){
            if (r[x].first<pow(2,y+1)*min.first){
                groups[y].add(r[x]);
                break;
            }
            else if(y == groups.size()-1)
            {
                Group a = *new Group(y+2);
                groups.push_back(a);
            }
        }
    for (int i = 0; i<groups.size(); i++)
        groupSums += groups[i].getGSum();
}
```

For the two dimensional walker tests that were committed this structure was no were near as efficient due to the lack of difference between rates I entered. If one was given a diverse set of rates this algorithm would be much faster than the other (O(1)) however few rates were entered that were identical except for the key associated with them. At this point speed was established however there was no ability to add or delete rates or support more than one organism. The next task would be to support more diverse cases of implementation for the algorithm that supported both the binary and composition and rejection structure.

4 Versatility

4.1 Creatures and States

Now that speed has been established the versatility of the program would need to improve. To begin, this would mean more rates, creatures, and editing capabilities. To implement these things a new structure that works in tandem with the rate structures (BinaryTree and Composition) would have to store different creatures and their corresponding state variables that are changed as a result of the selected rate - in the past examples these state variables have been the X and Y position. Now these states should be able to be entered by the user and adjusted by a function that they enter. This would be adjust by the creation of a state structure that would hold the states of each creature created. each state would be a string corresponding to an integer value which would be adjusted by the user. The most simple implementation of this would be a map that holds the state strings and values and can be adjusted by the user. This structure was implemented in C++ as so:

```
class States{
private:
    std::map<std::string, double> myStates{};

public:
    States() = default;
    States(std::vector<std::string> states){
        for (int i = 0; i<states.size();i++){
            myStates[states[i]]=0;
        }
    }

    void initialize(std::vector<std::string> states){
        for (int i = 0; i<states.size();i++){
            myStates[states[i]]=0;
        }
    }
    void increment(int reference){
                switch(reference){
                case 1:
                    myStates["positionX"]+=20;
                    break;
                case 2:
                    myStates["positionX"]-=20;
                    break;
                case 4:
                    myStates["dead"]=1;
                    break;
                default:
                    break;
            }
    }
    int size(){return myStates.size();}
    double get(std::string a){
        return myStates.at(a);
    }
    void set(double pX){myStates["positionX"] = pX;}

};
```

While the increment function is at the moment not taking in a function from the user this can be easily changed later on. However one can see that this system can take many states instead of those that are initialized as private variables. The next steps are to make the current algorithm even more versatile because it currently can only support few rates and creatures. To do this the it was decided that implementing a creature that can die and birth would demonstrate these abilities the best. Furthermore, an implementation of the One Dimensional walker would be made that can support multiple rates and creatures with both the composition and rejection structure as well as the binary tree structure.

4.2 Things to Consider

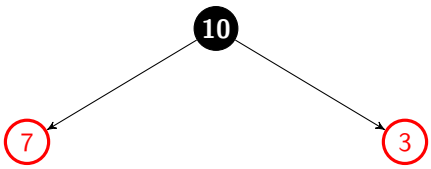
A new creature is represented as more rates. For example the original rates for a one dimensional walker were those that would negatively and positively change its position. If there were two creatures present than ther would be four rates would have four rates: Two for one creature and two for another. So the problem of adding more creatures is really just adding the capability to add more rates. However, one other problem arrises. It is necessary to link certain rates to certain creatures as to change the correct states. To address this problem, both structures (binary and compositon) are using a special entry type that comes in the form of [a,[b,c]]. A being the double that represents the rate value, b being an integer that provides which creature the a value is linked to, and lastly the c value which is the one taken in by the increment function above (it links the rate to some action).For both programs the seed should be passed in by the user, and the creatures stored in the Gillespie struct that was originally created.

4.3 Binary Tree

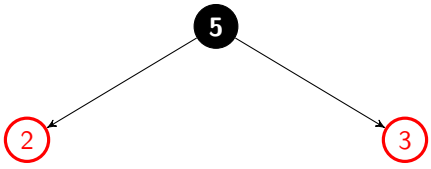
Creating an efficient insertion and deletion method for the binary tree is vital to its operations. The first step is figuring where to insert things and what criteria should effect this. Clues of how to do this lie in the search method which is the basis to how efficient this program will be. We want each side of the tree to be relatively balanced in terms of rate values, but in the case that there is an exceptionally large rate value one must be strategic in terms of placement. The steps to this method looks something like this.

1. First check if the node is empty. If it is return 0.
2. If the node is not empty the see if it's a leaf (has no children).
3. If it's a leaf add a new node thats children are the current leaf and the node that wants to be inserted. and then put it in the position of the current leaf and update all the values above.
4. If it's not a leaf check if the value being inserted is greater than both the left and right nodes of the current Node. If it is then make a new Node whose left branch is the current Node and right is the value that want's to be inserted then put this node in the place of the current node, update the parent values and end the process.
5. If neither of these conditions is met then check which node is less (left or right). Then set your current node to the node that is less and repeat at step three.

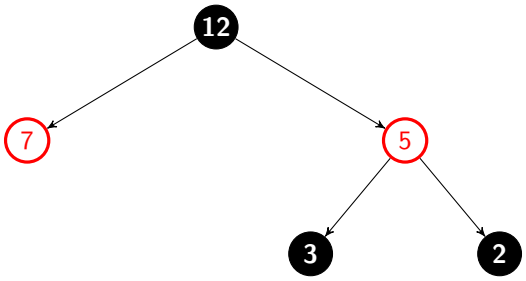
This method insures that if there is a large rate value one will find it early on in the program. It also ensures that the branches stay relatively balanced because it will choose which side to insert itself based on the least value. The combined effect is an extremely efficient binary tree that can insert rates in an intelligent way. Let's say this is our tree:



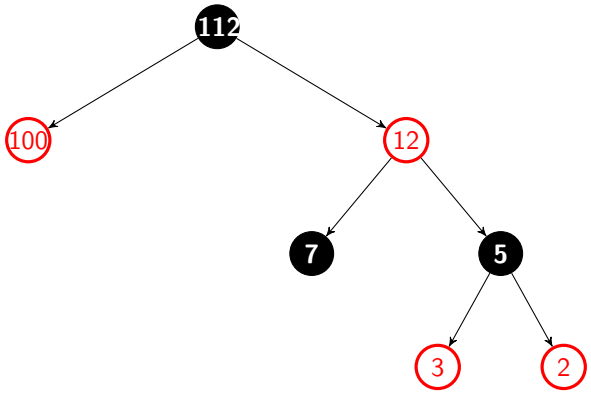
We want to insert the value two into the tree. We start at ten. The Node has children and is less than both 7 abd 3 so we go to step 5. Now we go right because 3 is less than 7 and start at step 3. It is a leaf so a new node is formed according to step 3.



This node is then added to three's position in the first tree with all of the above values updated like so:



Now lets say that we want to add 100 to this tree. Following the steps we get to step 4 and find that 100 should be inserted high up the tree like so:



Again the rates in a sense are probabilities and putting the larger ones closer to to the top makes sense because we are statistically more likely to pick these rates so now we do not have to go down the tree very far to find the larger values.
Deletion is less efficient for a binary tree. In performing deletion one has to check every possible node to see if it is the on that needs to be deleted. One can not use a search function because the structure of the tree does not allow this. Furthermore the search algorithm follows the following pattern:

1. Check if the Node is the one you are looking for
2. If not go left
3. If not go right
4. If it's the one you are looking for delete the then set the other child that has the same parent to the parent position and set it's children as nullptrs
5. Lastly updated all of the parent nodes

With this pattern one is able to go through all nodes recursively and to delete the necessary ones. Unfortunately, this takes O(n) time to perform.

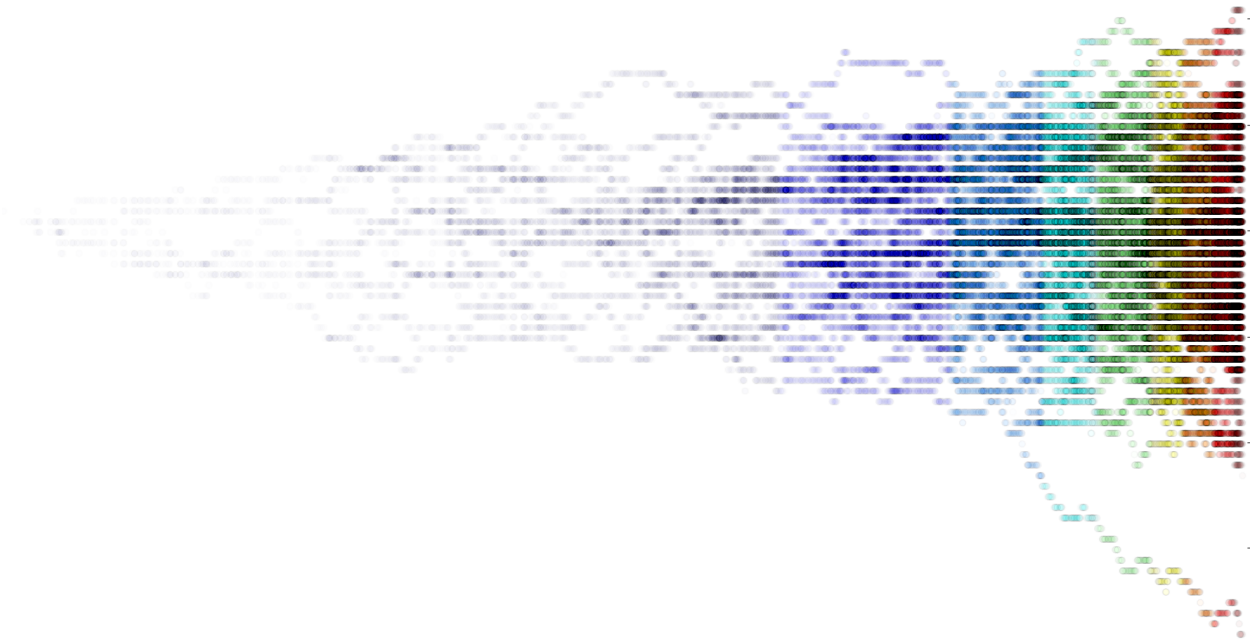
The insertion, deletion and update methods follow:

```
void insert(std::pair<double,std::pair<int,int>> a){
    conductor = head;
        while(!isLeaf()){
            if (a.first>conductor->getLeft()->getRate()&&a.first>conductor->getRight()->getRate()){
                if (conductor == head){
                    head = conductor = new Node(head, new Node(a.first,a.second));
                }
                else{
                    Node* nPP = conductor->getParent();
                    Node* nParent = new Node(conductor, new Node(a.first,a.second));
                    if (equals(conductor, nPP->getLeft()))
                        nPP->setLeft(nParent);
                    else
                        nPP->setRight(nParent);
                    nParent->setParent(nPP);
                    update(nPP,(nPP->getLeft()->getRate()+nPP->getRight()->getRate())-nPP->getRate());
                    conductor = head;
                }
                return;
            }
            if (conductor->getRight()->getRate() >= conductor->getLeft()->getRate())
                conductor = conductor->getLeft();
            else
                conductor = conductor->getRight();
        }
        conductor->setLeft(new Node(conductor->getRate(),conductor->getKey()));
        conductor->getLeft()->setParent(conductor);
        conductor->setRight(new Node(a.first,a.second));
        conductor->getRight()->setParent(conductor);
        double difference = conductor->getRate() - (conductor->getRight()->getRate()+conductor->getLeft()->
        conductor->setKey({0,0});
        update(conductor,-difference);
        conductor = head;
}

void remove(int creatureNum, Node* temp){
    int id = 0;
    if (temp == nullptr)
        return;
    if(temp->getLeft()==NULL && temp->getRight()==NULL && temp->getKey().first==creatureNum){
        if (temp->getParent()==nullptr) {
            head = conductor = nullptr;
            return;
        }
        if (equals(temp, temp->getParent()->getLeft())){
            Node* parentR = temp->getParent()->getRight();
            parentR->setParent(temp->getParent()->getParent());
            if (parentR->getParent()==nullptr){
                head = conductor = parentR;
                return;
            }
            if (equals(temp->getParent(), temp->getParent()->getParent()->getLeft()))
                parentR->getParent()->setLeft(parentR);
            else
                parentR->getParent()->setRight(parentR);

            update(parentR->getParent(),-temp->getRate());
        }
        else{
            Node* parentL = temp->getParent()->getLeft();
            parentL->setParent(temp->getParent()->getParent());
            if (parentL->getParent()==nullptr){
                head = conductor = parentL;
                return;
            }
```

```
            if (equals(temp->getParent(), temp->getParent()->getParent()->getLeft()))
                parentL->getParent()->setLeft(parentL);
            else
                parentL->getParent()->setRight(parentL);

            update(parentL->getParent(),-temp->getRate());
        }
    }
    remove(creatureNum,temp->getLeft());
    remove(creatureNum, temp->getRight());
}

//checks to see if two nodes rates and keys are equal
bool equals(Node* a, Node* b){
    return a->getRate() == b->getRate() && a->getKey() == b->getKey();
}
//will update all parent nodes by adding the increment to their rate val
void update(Node* updatePoint,double increment){
    if (updatePoint == nullptr) return;
    updatePoint->setRate(updatePoint->getRate()+increment);
    if (updatePoint->getParent()!=NULL)
        update(updatePoint->getParent(),increment);
}
```

The code can be confusing but it performs the basic steps in the list and diagram above. With this established the structure is ready for the implementation. Graphically the result is something like this with the x axis time and the y as position.

Photos/example.png



The different trails represent organisms branching off from one another.

4.4 Composition and Rejection

The task of insertion and deletion is much easier with composition and rejection. To insert something one follows the same method that was used for initialization for the two dimensional walker. Deletion is just as inefficient as for a binary tree. One has to go through each group to check if it is the correct rate to delete and if it is one can delete. However one could increase the efficiency of this method if the know the rate that they are deleting, because they can then figure out which group it lies in by performing the following function $\lceil \log_m x \rceil$. take the log of the rate with the base m being 2 times the minimum. This would take O(1) time instead of O(n).

4.5 Results

After these new implementations comparisons were run and it became clear that composition and rejection was much more efficient than the binary tree. However the cases were very specific and more will need to be run to the see the advantage of the structures in different situations.

5 Next Steps

The current structures are very efficient and relatively versatile; they can delete, add, store, search, and update rates. While their could be some cases that these structures wouldn't work for, they cover almost all cases. The only problem however is that these structures are far from being user friendly. The big decision is whether to just keep a documentation of the code and how certain obstacles were overcome and to let users implement it themselves or to further work on the Gillespie class and make it more user intuitive and versatile so that users can include it in their code. However, both of these choices are at the extremes. What one real needs is just the binary and composition structures to store the rates and run the simulation. These structures should be able to supply sufficient information about the users program without having errors.

5.1 Key

For both structures a key is utilized which has in every case been a pair of integers however this is very limiting and the user should be able to decide how they want to associate rates with states. Furthermore, the best next step would be able to make both structures take in a structure specified by the user to make the program less reliant on specific identifiers. The best way to allow a class to take in an unspecified member is to make it a template class that takes in one class called ID which will be the bases to the other classes.

5.2 Simulation Tests

All of the tests have been to see if the structures look as if they are working. However, no statistical tests have been implemented that would test the accuracy and differences in both the structures speed and results. The best way to implement these tests would run 1000's of simulations of just a 1DWalker that can't reproduce or die with everystructure. Then the average position at different exact times would be calculated.

5.3 Combined Structures

At this point the binary structure and composition structure have been tested with two different main.cpp's in order to compose quick tests the structures will need to be combined into one file.

6 Key and Templates

6.1 Templates

In C++ templates are an outlet that allows one to create functions and classes that operate with generic types. The whole goal of this program is to be versatile and taking in generic types and using templates is a big step towards this. The generic types were extremeley simple to implement and can be seen in the BinaryImplementation header files as well as the Composition head file.
Composition.h

```
template<class ID>
class Group{
private:
int level{0};
double gSum{0};
std::vector< std::pair<double, ID >> elements{};

public:
//initializes according to bracket constructors in private
Group() = default;
Group(int l){
    level = l;
}
//adds element to vector and updates the group sum
void add(std::pair<double,ID > p){
    gSum+=p.first;
    elements.push_back(p);
}
//The rejection algorithm which takes in the random num generator die
std::pair<double,std::pair<int, int>> find(double compMin,  std::function<double()> die){
    updateGSum();
    double levelHeight = pow(2,level)*compMin;
    std::pair<double,ID > current = elements[floor(die()*(elements.size()))];
    double place = die() * levelHeight;
    COUNT_THIS_SCOPE(__PRETTY_FUNCTION__);

    while(current.first<place){
        Counter::ScopeCounter<> sc("main_loop");
        current = elements[floor(die()*(elements.size()))];
        place = die() * levelHeight;
    }
    return current;
}

double getGSum(){return gSum;}

//returns true if there is a creature with the identifier fir the second.first elementx of the pair
bool hasCreature(int identifier){
    return std::find_if(std::begin(elements),std::end(elements),[&](std::pair<double, ID > element){return
}
bool hasRate(ID identifier){
    return std::find_if(std::begin(elements),std::end(elements),[&](std::pair<double, ID > element){return
}
//removes the first element with the group num for the second.first element
void remove(int group){
    elements.erase(std::remove_if(elements.begin(),elements.end(),[&](std::pair<double,ID > element){return
}
//removes element that has the same identifier (.second) as identifier
void remove(ID identifier){
    elements.erase(std::remove_if(elements.begin(),elements.end(),[&](std::pair<double, ID > element){retu
}
void updateGSum(){
    gSum = 0;
    for (int i = 0; i<elements.size();i++){
        gSum+=elements[i].first;
    }
}
```

```
};


template<class ID>
class Composition{
private:
std::function<double()> die;
double currentTime{0};
double deltaT{0};
std::pair<double,ID > min{0,{}};
double groupSums{0};
std::vector< Group<ID> >  groups;
public:
Composition() = default;
Composition(std::vector< std::pair<double,ID > > r,std::function<double()> d){
    die = d;
    //finds the minimum element entered
    min = *std::min_element(std::begin(r),std::end(r),[]( std::pair<double, ID > r1, std::pair<double,std:
    Group<ID> a(1);
    groups.push_back(a);
    for (int x = 0; x < r.size(); x++){
        for (int y = 0; y < groups.size();y++){
            if (r[x].first<pow(2,y+1)*min.first){
                groups[y].add(r[x]);
                break;
            }
            else if(y == groups.size()-1)
            {
                groups.emplace_back(y+2);
            }
        }
    }
    for (int i = 0; i<groups.size(); i++){
        groupSums += groups[i].getGSum();
    }
}

void updateGroupSums(){
    groupSums = 0;
    for (int i = 0; i<groups.size(); i++){
        groupSums += groups[i].getGSum();
    }
}
//adds rate to structure
void addRate(std::pair<double, ID> p){

    for (int y = 0; y < groups.size();y++){
        if (p.first<pow(2,y+1)*min.first){
            groups[y].add(p);
            break;
        }
        else if (y == groups.size()-1)
            groups.emplace_back(y+2);
    }
    updateGroupSums();
}
//selects rate with Gillespie steps
std::pair<double,ID > selectRate(){
    updateGroupSums();
    deltaT = (-log(die()))/groupSums;
    currentTime+=deltaT;
    double place = die()*groupSums;
    for (int i = groups.size()-1; i>=0;i--){
        if (place<groups[i].getGSum()){
            return groups[i].find(min.first,die);
        }
        else
            place-=groups[i].getGSum();
```

```
    }
    return {-1,{ID()}};
}
//deletes an entire creature with identifier
void deleteC(int identifier){
    for (int x = 0; x<groups.size();x++){
        while(groups[x].hasCreature(identifier)){
            groups[x].remove(identifier);
            groups[x].updateGSum();
        }
    }
    updateGroupSums();
}
//deletes one rate
void deleteR(ID identifier){
    for (int x = 0; x<groups.size();x++){
        if(groups[x].hasRate(identifier)){
            groups[x].remove(identifier);
            groups[x].updateGSum();
            break;
        }
    }
    updateGroupSums();
}

double getGroupSums(){return groupSums;}
double getCurrentTime(){return currentTime;}
};
```

The key to this structure is the template¡class ID¿ at the top of each class. This allows an unknown or generic class ID to be passed in by the user and processed by this class that stores the ID passed in a vector of groups.

It looks similiar for the binary structure as well.
BinaryTree.h

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <cmath>
#include <random>

template<class ID>
class Node{
private:
    Node<ID>* parent{nullptr};
    Node<ID>* left{nullptr};
    Node<ID>* right{nullptr};
    double rate{0};
    ID key{};
public:
    Node(void) = default;
    Node(double r, ID val)
    {
        key = val;
        rate = r;
    }
    //Creates a new node that is the sum of the two inputted
    //Children are the inputted nodes
    Node(Node* l, Node* r){
        rate = l->getRate() + r->getRate();
        l->setParent(this);
        r->setParent(this);
        left = l;
        right = r;
    }
    ID getKey(){return key;}
    double getRate(){return rate;}
    Node* getLeft(){return left;}
    Node* getRight(){return right;}
    Node* getParent(){return parent;}
    void setRate(double r){rate = r;}
    void setParent(Node* x){parent = x;}
    void setLeft(Node* x){left=x;}
    void setRight(Node* x){right=x;}
    void setKey(ID a){key = a;}
};

//Structure made from the nodes that only has the head
template<class ID>
class BinaryTree{
private:
    double currentTime{};
    double deltaT{};
    std::function<double()> die;
    Node<ID>* head;
    Node<ID>* conductor;
public:
    //create an empty binary tree.
    BinaryTree(){
        head = conductor = nullptr;
    }
    //construct special tree for Gillespie such that leaves are rates and parents are sums
    BinaryTree(std::vector< std::pair<double, ID> > r, std::function<double()> d){
        die = d;
        head = conductor = new Node<ID>(r[0].first,r[0].second);
        for (int i = 1; i<r.size();i++)
            insert(r[i]);
    }
```

```
    void insert(std::pair<double, ID> a){
        conductor = head;
            // head = conductor = new Node(head, new Node(a.first ,a.second));
            while(!isLeaf()){
                // a's␣rate␣is␣bigger␣than␣the␣right␣and␣left␣nodes
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(a.first>conductor->getLeft()->getRate()&&a.first>conductor->getRight()->getRate()){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//if␣it␣is␣the␣first␣node␣make␣the␣new␣head␣a␣new␣node␣that␣has␣the␣current␣head␣and␣a
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(conductor␣==␣head){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣head␣=␣conductor␣=␣new␣Node<ID>(head,␣new␣Node<ID>(a.first ,a.second));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//otherwise␣figure␣out␣if␣the␣current␣node␣is␣of␣the␣left␣or␣right␣parents
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//and␣construct␣a␣new␣node␣from␣the␣results␣in␣place␣of␣current...␣update␣the␣parents
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Node<ID>*␣nPP␣=␣conductor->getParent();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Node<ID>*␣nParent␣=␣new␣Node<ID>(conductor,␣new␣Node<ID>(a.first ,a.second));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(equals(conductor,␣nPP->getLeft()))
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nPP->setLeft(nParent);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nPP->setRight(nParent);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nParent->setParent(nPP);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣update(nPP,(nPP->getLeft()->getRate()+nPP->getRight()->getRate())-nPP->getRate());
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣conductor␣=␣head;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//otherwise␣follow␣the␣example␣shown␣at␣the␣beggining␣of␣the␣func
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(conductor->getRight()->getRate()␣>=␣conductor->getLeft()->getRate())
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣conductor␣=␣conductor->getLeft();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣conductor␣=␣conductor->getRight();
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣//␣Node*␣temp␣=␣new␣Node(new␣Node(conductor->getRate(),conductor->getKey()),new␣Node(a.first ,a
␣␣␣␣␣␣␣␣␣␣␣␣conductor->setLeft(new␣Node<ID>(conductor->getRate(),conductor->getKey()));
␣␣␣␣␣␣␣␣␣␣␣␣conductor->getLeft()->setParent(conductor);
␣␣␣␣␣␣␣␣␣␣␣␣conductor->setRight(new␣Node<ID>(a.first ,a.second));
␣␣␣␣␣␣␣␣␣␣␣␣conductor->getRight()->setParent(conductor);
␣␣␣␣␣␣␣␣␣␣␣␣double␣difference␣=␣conductor->getRate()␣-␣(conductor->getRight()->getRate()+conductor->getLeft
␣␣␣␣␣␣␣␣␣␣␣␣conductor->setKey({0,0});
␣␣␣␣␣␣␣␣␣␣␣␣update(conductor,-difference);
␣␣␣␣␣␣␣␣␣␣␣␣conductor␣=␣head;
␣␣␣␣}


//This␣remove␣Function␣removes␣all␣instances␣of␣an␣index␣entered
␣␣␣␣void␣removeAll(ID␣creatureNum,␣Node<ID>*␣temp){
␣␣␣␣␣␣␣␣//␣conductor␣=␣head;
␣␣␣␣␣␣␣␣//std::cout<<"\n\n";
␣␣␣␣␣␣␣␣//prettyPrint(head,id);
␣␣␣␣␣␣␣␣if␣(temp␣==␣nullptr)
␣␣␣␣␣␣␣␣␣␣␣␣return;
␣␣␣␣␣␣␣␣if((temp->getLeft()==NULL␣&&␣temp->getRight()==NULL)␣&&␣temp->getKey().first==creatureNum.first){
␣␣␣␣␣␣␣␣␣␣␣␣//␣std::cout<<creatureNum<<"\n";
␣␣␣␣␣␣␣␣␣␣␣␣//␣std::cout<<"found:␣"<<temp->getRate()<<"␣"<<temp->getKey().first<<"␣"<<temp->getKey().second
␣␣␣␣␣␣␣␣␣␣␣␣if␣(temp->getParent()==nullptr)␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣head␣=␣conductor␣=␣nullptr;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣//else␣std::cout<<"ok"<<std::flush;
␣␣␣␣␣␣␣␣␣␣␣␣if␣(equals(temp,␣temp->getParent()->getLeft())){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//std::cout<<"switching␣right␣to␣parent";
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Node<ID>*␣parentR␣=␣temp->getParent()->getRight();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣parentR->setParent(temp->getParent()->getParent());
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(parentR->getParent()==nullptr){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣head␣=␣conductor␣=␣parentR;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(equals(temp->getParent(),␣temp->getParent()->getParent()->getLeft()))
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣parentR->getParent()->setLeft(parentR);
```

```
                else
                    parentR->getParent()->setRight(parentR);

                update(parentR->getParent(),-temp->getRate());
            }
            else{
                //std::cout<<"switching left to parent\n";
                Node<ID>* parentL = temp->getParent()->getLeft();
                //std::cout<<"parentL: "<< parentL->getRate()<<" "<<parentL->getKey().first <<"\n";
                parentL->setParent(temp->getParent()->getParent());
                if (parentL->getParent()==nullptr){
                    head = conductor = parentL;
                    return;
                }
                if (equals(temp->getParent(), temp->getParent()->getParent()->getLeft()))
                    parentL->getParent()->setLeft(parentL);
                else
                    parentL->getParent()->setRight(parentL);

                update(parentL->getParent(),-temp->getRate());
            }
        }
        removeAll(creatureNum,temp->getLeft());
        removeAll(creatureNum, temp->getRight());
    }


    //removes one creature with this key
    void remove(ID creature, Node<ID>* temp){
        if (temp == nullptr)
            return;
        if(temp->getLeft()==NULL && temp->getRight()==NULL && temp->getKey()==creature){
            if (temp->getParent()==nullptr) {
                head = conductor = nullptr;
                return;
            }
            // if temp is a left node
            if (equals(temp, temp->getParent()->getLeft())){
                Node<ID>* parentR = temp->getParent()->getRight();
                parentR->setParent(temp->getParent()->getParent());
                if (parentR->getParent()==nullptr){
                    head = conductor = parentR;
                    return;
                }
                if (equals(temp->getParent(), temp->getParent()->getParent()->getLeft()))
                    parentR->getParent()->setLeft(parentR);
                else
                    parentR->getParent()->setRight(parentR);

                update(parentR->getParent(),-temp->getRate());
            }
            // if temp is a right node
            else{
                //std::cout<<"switching left to parent\n";
                Node<ID>* parentL = temp->getParent()->getLeft();
                //std::cout<<"parentL: "<< parentL->getRate()<<" "<<parentL->getKey().first <<"\n";
                parentL->setParent(temp->getParent()->getParent());
                if (parentL->getParent()==nullptr){
                    head = conductor = parentL;
                    return;
                }
                if (equals(temp->getParent(), temp->getParent()->getParent()->getLeft()))
                    parentL->getParent()->setLeft(parentL);
                else
                    parentL->getParent()->setRight(parentL);

                update(parentL->getParent(),-temp->getRate());
            }
```

```
            return;
        }
        remove(creature,temp->getLeft());
        remove(creature, temp->getRight());
    }
    //checks to see if two nodes rates and keys are equal
    bool equals(Node<ID>* a, Node<ID>* b){
        return a->getRate() == b->getRate() && a->getKey() == b->getKey();
    }
    //will update all parent nodes by adding the increment to their rate val
    void update(Node<ID>* updatePoint, double increment){
        if (updatePoint == nullptr) return;
        updatePoint->setRate(updatePoint->getRate()+increment);
        if (updatePoint->getParent()!=NULL)
            update(updatePoint->getParent(),increment);
    }

    //checks to see if the conductor is a leaf
    bool isLeaf(){return conductor->getLeft()==NULL&&conductor->getRight()==NULL;}

    //Gillespie step
    std::pair<double,ID> find(){
        conductor = head;
        deltaT = (-log(die()))/rSum();
        currentTime+=deltaT;
        double place = die()*rSum();
        while (place<conductor->getRate() && conductor->getRight() != NULL){
            //std::cout<<conductor->getRate()<<'\n';
            if (place>conductor->getLeft()->getRate()){
                place -= conductor->getLeft()->getRate();
                conductor = conductor->getRight();
                //std::cout<<"went right" <<'\n';
            }
            else{
                conductor = conductor->getLeft();
                //std::cout<<"went left" <<'\n';
            }
        }
        return {conductor->getRate(),conductor->getKey()};
    }


    double getCurrentTime(){return currentTime;}

    Node<ID>* getHead(){return head;}
    double rSum(){return head->getRate();}
};
```

Now the user can enter any type and the compiler will handle the rest from there.

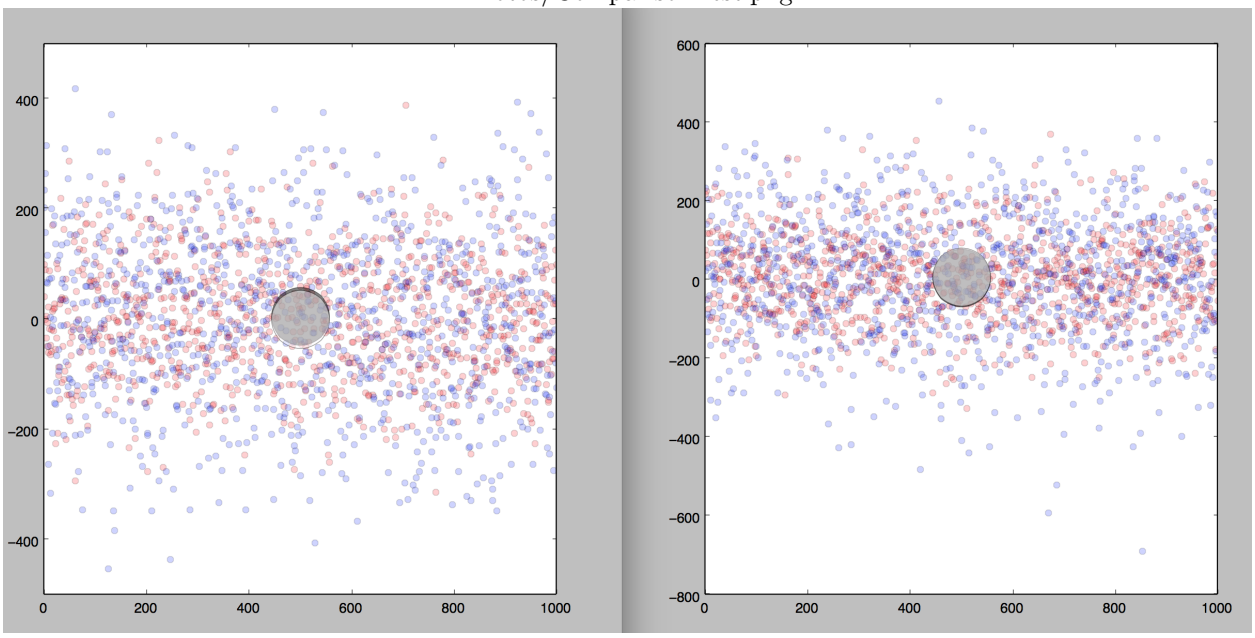
7 Simulations

Simulations proved challenging. The first step was to reorient the one dimensional walker to where it could no longer die or rebirth. Following this, the main.cpp had to be reconfigured so that it could run both simulations for both structures for over a thousand times. This was necessary because the simulations were going to be performed like so.

1. First generate exact times that you want to record for each simulations. I chose to make these times start at .1 end at 4 and take a step of .1 each time
2. Next one has to find the average position for each simulation for the given timesteps
3. after take the average of the averages for all simulations for each data structure
4. compare the results at the end

The end result produced this:

Photos/ComparisonTest.png



All of the red points are the averages for the binarystructure simulations and all of the blue are for the averages of the composition and rejection structures. Finally the black and white opaque dots represent the final averages of the averages for each structure. As one can observe the final results show that the statistical results of both structures prove to be very close to eachother. In fact the position difference between the two proved to be no greater than 1 in most cases.

The code here was what was used to find the averages of the averages as well as graph the results:

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
import os
import sys
#open text file
def findAverage(file):
    count =0;
    s = 0;
    averages = []
    nums = []
    for line in file:
        if line == '\n':
            averages.append(s/count)
            count=0
            s = 0
        else:
            nums = line.split(" ")
            count+=1
            s += float(nums[1])
    print np.mean(averages)
    return (averages)
print "How many simulations would you like to compare?"
numSimulation = int(raw_input())
print "performing simulations"
for i in range(0,numSimulation):
    plt.figure("Binary vs Composition "+str(i+1))
    print "running simulation",i+1
    os.system("./GillespieAlgorithm")
    file = open("outputBT.txt",'r')
    #put columns 0, 1, 2 in variables t, x, y
    bt = findAverage(file)
    file.close
    file = open("outputC.txt",'r')
    c = findAverage(file)
    file.close
    plt.plot(bt,"ro",c,"bo",alpha=.2)
    plt.plot(500,np.mean(bt),"ko",ms=50,alpha=.5)
    plt.plot(500,np.mean(c),"wo",ms = 50,alpha=.5)
plt.show()
```

8 Combined

This had to actually be accomplished in the middle of the last section, but it is important enough to have its own section. Before now the main.cpp that implemented the structures was actually two seperate ones for each structure and in order to perform combined speed and data tests, these structure would have to be combined to one. Here it shows how it was accomplished. It was fairly simple by taking advantage of the ternary operator and a simple boolean.

```
#include <time.h>
#include "counter.hh"
#include <iostream>
#include <random>
#include <vector>
#include <algorithm>
#include <fstream>
#include <stdio.h>
#include <utility>
#include "BinaryImplementation.h"
#include "States.h"
#include "CompRejStruct.h"
#include <chrono>

using entry = std::pair< double, std::pair<int,int> >;
//Declare Gillespie class
class Gillepsie{
private:
    bool structIsaBTree{true};
    std::vector< States > creatures;
    std::vector< std::string > rStrings;
    int limit;
    Composition<std::pair<int,int>> c;
    BinaryTree<std::pair<int,int>> bt;
    std::vector<double> timesteps;
    std::vector< std::vector< double > > data;

public:

    //initializers
    Gillepsie()
    :bt()
    {
        limit = 100;
    }


    Gillepsie(bool s,int numCreatures, std::vector<std::string> states, std::vector< entry > r,std::functio
    : bt(r,d), c(r,d) , creatures(numCreatures)
    {
        structIsaBTree = s;
        rStrings = states;
        for (int i = 0; i<numCreatures;i++){
            creatures[i].initialize(states);
        }
        limit = 100;
    }


    Gillepsie(bool s, int numCreatures, std::vector<std::string> states, std::vector< entry > vec, std::fu
    : Gillepsie(s,numCreatures, states, vec, d)
    {
        limit = l;
    }
    Gillepsie(bool s, int numCreatures, std::vector<std::string> states, std::vector< entry > vec,std::vec
    : Gillepsie(s,numCreatures, states, vec, d)
    {
```

```
        timesteps = myIncrements;
        limit = l;
    }

    //generates output and puts it in data vector
    void run(){
        int currentTimeStep = 0;
        while( ((structIsaBTree) ? bt.getCurrentTime()<limit : c.getCurrentTime()<limit) && ((structIsaBTre
            entry vecPos = (structIsaBTree) ? bt.find() : c.selectRate();
            creatures[vecPos.second.first -1].increment(vecPos.second.second);

            if (timesteps.size()>0){
                    // std::cout<<"creature "<<x+1<<" position "<<creatures[x].get("positionX")<<"\n";
                while(currentTimeStep<timesteps.size()&&(structIsaBTree)?bt.getCurrentTime()>timesteps[curr
                    // std::cout<<"current time: "<<((structIsaBTree)?bt.getCurrentTime():c.getCurrentTime(
                    data.push_back({timesteps[currentTimeStep],creatures[0].get("positionX")});
                    currentTimeStep++;
                }
            }
            else{
                for (int x = 0; x<creatures.size();x++){

                   // std::cout<<"creature "<<x+1<<" position "<<creatures[x].get("positionX")<<"\n";
                     data.push_back({(structIsaBTree) ? bt.getCurrentTime() : c.getCurrentTime() ,creatures
                }
            }
        }
        // std::cout<<"final size: "<<creatures.size()<<"\n";
    }
```