Temperature Query Design

- Premise:
    - The program should be able to take in data from an outside source
        - Use instream
    - The program should save the data in nodes
        - Location, Year, Month, and Temperature are all valid data
    - The nodes should be saved in a linked list
    - The linked list should be organized
        - First, organize data by location.
        - If the data is the same, organize by the time it was taken
            - Starting from the earliest reading and going to the most recent.
    - Test if the values coming from the instream are valid
        - If the data types don't match, throw errors.
            - The months should be 1-12
            - The years should be 1800-2017
            - The temperature should be from -50 celsius to 50, and -99.99 as a null value.
            - The location should have 7 integers.
            - It should be in location -> year -> month -> temperature exclusively
            - If ANY of those criteria aren't met, throw an error. (test cases)
    - Implement the node/list etc as classes.
        - Add the rule of 3 (or 5) in order to better handle the data.
        - Overload the output operator to display data.
- If necessary, make an outstream possible
- Implement the program incrementally
- If necessary, I'll write out the code for placing a node in place.

Pseudocode on next page

Pseudocode:

```
Instream inFS = ("data.txt");

while(inFS != EOF){
    tempLoc  = inFS;
    if(tempLoc == 7 integers long){
        tempYear = inFS;
        if(tempYear >= 1800 && tempYear <= currYear){
            tempMon = inFS;
            if(tempMon > 0 && tempMon < 13){
                Temp      = inFS;
                if((temp > -51 && temp < 51) || temp == -99.99){
                    Node new = createNode(tempLoc, tempYear, tempMon, temp);
                    insertNode(new);
                }
                Else{
                    Throw runtime_error("Invalid temperature. Node not cre*");
                }
            Else{
                Throw runtime_error("Invalid month. Node not created.");
            }
        Else{
            Throw runtime_error("Invalid year. Node not created.");
        }
    Else{
        Throw runtime_error("Invalid location. Node not created.");
}
```

Design Pt 2:

Required:

- Algorithms for determining the average and mode.
- Test cases for both.

Left off from last design (node placement algorithm):

```
void LinkedList::insert(int location, int year, int month, double temperature) {
    Node* n    = new Node(location, year, month, temperature);
    Node* temp;
    Node* curr = head;
```

```
if(isEmpty()){head = n; tail = n; return;}
if(n->location < curr->location){
        n->next = curr;
        head    = n;
}
while(n->location > curr->location){
        if(curr->next == NULL){
                curr->next = n;
                n->next = NULL;
                return;
        }
        temp = curr;
        curr = curr->next;
}
if(n->location == curr->location){
        while(n->year > curr->year){
                if(curr->next == NULL){
                        curr->next = n;
                        n->next = NULL;
                        return;
                }
                temp = curr;
                curr = curr->next;
        }
        if(n->year == curr->year){
                while(n->month > curr->month){
                        if(curr->next == NULL){
                                curr->next = n;
                                n->next = NULL;
                                return;
                        }
                        temp = curr;
                        curr = curr->next;
                }
                if(n->month == curr->month){
                        while(n->temp > curr->temp){
                                if(curr->next == NULL){
                                        curr->next = n;
                                        n->next = NULL;
                                        return;
                                }
                                temp = curr;
                                curr = curr->next;
                                }
                        }
                        else{
                                temp->next = n;
                                n->next = curr;
                return;
                        }
                }
                else{
                        temp->next = n;
                        n->next = curr;
        return;
                }
        }
        else{
                temp->next = n;
                n->next = curr;
        return;
        }
 return;
}
```

What the above algorithm does is sorts the linked list almost like cracking the combination of a lock on a safe. It moves around the list until it finds the correct location to place the node.

Algorithm for Average:

- Criteria for algorithm:
  - Has to take a location integer
  - Has to take a range from X year to Y year.
  - If the location is unavailable, throw answer “Unknown”
  - If the location is available, but any of the date range is not, throw “Unknown”
    - For example, if range is 2004-2007, and all exists but 2005 is missing, throw the error.
  - Purpose:
    - Tallies the number of nodes over the period, and adds up the temperature data. At the end, divides total of temperature data by number of nodes processed.

Algorithm for Mode:

- Criteria for algorithm:
  - Has to take a location integer
  - Has to take a range from X year to Y year.
  - If the location is unavailable, throw answer “Unknown”

    - If the location is available, but any of the date range is not, throw “Unknown”
        - For example, if range is 2004-2007, and all exists but 2005 is missing, throw the error.
    - Purpose:
        - Keeps track of the (rounded) temperatures over a period of time. Uses parallel vectors to keep track of the most frequent number over the period, and outputs the highest. If two numbers have the same frequency, output the one closest to 0.

Skeleton Code:

- void LinkedList::query(const LinkedList& data, LinkedList& query){
    - Node* quer = query.head;
    - Node* dat   = data.head;
    - while(quer != nullptr){
        - if(quer->type == “AVG”){
            - Int start = findStart(quer->start, data); //finds where the data starts
            - Int end =findEnd(quer->end, data); //finds the last node in range
            - for(int i = 0; i < end; i++){ //data collection stops at the end range.
                - if(dat->year >= start){
                    - Total += dat->temp;
                    - Count++;
                - }
                - dat = dat->next;
            - }
            - quer->data = total / count; //assigns result to query’s data.
            - quer = quer->next;   //iterates to next query
            - dat = data.head;   //resets pointer to the beginning for next Q.
        - }
        - //Below code will be for MODE query. I did not forget the code. xD
        - if(quer->type == “MODE”){
            - Int start = findStart(quer->start, data); //finds where the data starts
            - Int end =findEnd(quer->end, data); //finds the last node in range
            - if(end > start){
                - //skips the for loop and applies “unknown” to data.

              }
            - for(int i = 0; i < end; i++){ //data collection stops at the end range.
                - if(dat->year >= start){
                    - Num = round(curr); //rounds the number up or down
                    - if(num > 0){
                        - nums.at(num+50) += 1;
                    - Else{ //vector of size 101 to hold temps -50 to 50
                        - nums.at(num) += 1;
                - }

            - dat = dat->next;
        - }
        - quer->data = findMode(nums); //assigns node to data.
            - //Simply searches the list for the highest count value.
        - quer = quer->next; //iterates to next query
        - dat = data.head; //resets pointer to the beginning for next Q.
    - }

Things I have left out:

- The findStart and findEnd will function like the insert node, but rather than changing any of the data, it will simply find the first instance of the range from the query, and then the last node of the end of the range within query.
    - Within this, I will have it return some arbitrary number (for findEnd) if any sequential data is missing. That is, if the range is from 2005 to 2007 and 2006 is completely missing, the “findEnd” will return a negative number which will not allow the range to be found and will circumvent the for-loop because 0 is greater than any negative number.
- The round function will simply modify the passed double as the integer version of itself (which always rounds down for all intents and purposes. It may to the opposite, in which case I would convert to integer and subtract 1).
    - If the remainder is greater than .5, add 1 to the integer and return.
    - If the remainder is less than .5, return the typecasted integer itself.
- findMode will simply scan through the vector of data, figure out which temperature has the highest number of pings, then wipes the vector and returns the highest value found to where the function was called.
    - The function will be able to discern positive and negative numbers.
    - The array given will be 101 pieces of data long.
        - Array values from 0 to 49 are negative (-50 to -1)
        - Array value at 50 is 0.
        - Array values from 51 to 100 are positive(1 to 50)
    - After the highest is found, all numbers are reset to zero (passed by reference)
    - if(spot >= 50){
        - Return spot-50;
    - Else{
        - Return -(spot+1);

Test Cases:

- Main things:
    - Send an illegal query type != AVG or MODE
    - Send a range query that doesn’t exist within your data
    - Send a range query without a certain range of years
    - Send a location query that doesn’t exist within your data