Assignment #4 CMPUT 291

```
Name: Wyatt Johnson
Unix: wyatt
Date: Nov 23, 2012
Language: C/C++
```

Program Description

NOTE: APPLICATIONS AUTOMATICALLY DELETE DATABASE FILES WHEN COMPLETED

indexedsearch.o program

Sorting

This uses the quick sort algorithm to sort an associative struct of:

```
struct indexed
{
    uint32_t index;
    double value;
};
```

Using a custom sorting function as defined:

```
int Index::compare(const void * b, const void * a)
{
    indexed *struct_a = (indexed *) a;
    indexed *struct_b = (indexed *) b;

    if(struct_a->value < struct_b->value)
        return 1;
    else if(struct_a->value == struct_b->value)
        return 0;
    else
        return -1;
}
```

That performs the sorting on a value level, while preserving the indicies as they are carried with the value. This allows simple determination printing of the top 3 entries after the quicksort to display the smallest value for the distance calculation, carried out as:

```
double Index::compare(song *SongA, song *SongB)
{
    double result = 42;

    int matchingEntries = 0;

    int * A;
    int * B;

    if(SongA->rCount >= SongB->rCount)
    {
        A = new int[SongA->rCount];
        B = new int[SongA->rCount];
    }
    else
    {
        A = new int[SongB->rCount];
        B = new int[SongA->rCount];
    }

    for(int i = 0; i< SongA->rCount; i++)
    {
        for(int f = 0; f<SongB->rCount; f++)
        {

            if(SongA->ratings[i].User.compare(SongB->ratings[f].User) == 0)
            {
                A[matchingEntries] = SongA->ratings[i].rating;
                B[matchingEntries] = SongB->ratings[f].rating;

                matchingEntries++;
            }
        }
    }

    if( matchingEntries > 0 )
    {
        uint64_t sum = 0;

        for(int i = 0; i < matchingEntries; i++)
        {
            uint64_t temp = (A[i] - B[i]);
```

```
            sum += temp*temp;
        }

        result = sqrt(sum);

        result /= matchingEntries;
    }

    delete [] A;
    delete [] B;

    return result;
}
```

Which is designed to calculate the distance on two songs given a song struct, as defined:

```
struct rating {
    std::string User;
    uint8_t rating;
};

struct song {
    int id;
    std::string Title;
    std::string Artists;
    rating ratings[255];
    int rCount;
};
```

Through the use of the suggested distance formula:

```
 D(A, B) = SQRT( (R1A - R1B)^2 + (R2A - R2B)^2 + … (R2A - R2B)^2 )/N
```

Indexing/Serilizing Data

Once the main entries of the input data file are filed into the main database file as:

```
Key: SONG ID
Data: SONG STRING

Ex.

For entry: {[5], [When I approach], [Travie McCoy Feat, Livin, Joe Budden],
[(Ethan, 6), (Michael, 4), (Mason, 5)] }

Key: 5
Data: {[5], [When I approach], [Travie McCoy Feat, Livin, Joe Budden], [(Ethan,
6), (Michael, 4), (Mason, 5)] }
```

Where the string is parsed when needed. The indexed database, calculated after, is stored as:

```
Key: USER NAME
Data: SONG ID CSV

Ex.

For entrys:
{[4], [I Wanna Be A Billionaire], [Travie McCoy Feat, Bruno Mars], [(Michael,
6), (Mason, 2), (Sophia, 1)] }
{[5], [When I approach], [Travie McCoy Feat, Livin, Joe Budden], [(Ethan, 6),
(Michael, 4), (Mason, 5)] }

We would store:

Key: Michael
Data: 4, 5,

Key: Mason
Data: 4, 5,

Key: Sophia
Data: 4

Key: Ethan
Data: 5
```

This would allow a list to be generated using each of the users whe have rated a given song to be directly pulled from the database rather then linearly scanning the database via the inverted indexed database.