Final Project Description:

CSCI 2270

PURPOSE

The purpose of this project is to measure the efficiency of algorithms that detect collisions between a structured data set. We are evaluating a set of data from players built in a manner that is with any kind of data structure of our own choosing (BST, graphs, LL). This is important because data can have problems when records in the structure can have the same hash value and multiple keys assigned to same index spot, so there is no way of identifying separate data sets.

Procedure:

We are using two approaches to handle collisions:

- **Chaining** -> when every hash code is added to a table. Built as a linked list where each index of the hash table is linked next. Element is added, pointer is updated to the head of the linked list, and additional elements to it.

  Hash Table Size: 5072

  Data**:** Hash Table contains an array of player node types that is made of data from just the player. Next and previous pointers for chained linked list. And its own vector to be linked for the team data on that specific index also.

- **Open addressing** -> You go search and look at table if any location is empty and evaluate if it qualifies to store value or keep moving forward. Until again you are able to find an open spot and so on so forth. All records are store directly in the hash table. You use linear probing logic to find first available spot and traverse the table linearly until open location available. Then record at that location.

  Hash Table Size: 20000

  Data: Hash Table contains an array of player node types that is made of data from the player also. It uses the same player node

struct as the chaining. No need for a different struct, but it is built differently so it needs to be a separate hash table.

Data:

Our data set are baseball players. For each player we represent them with a Player Node. This class struct has unique data in the set.

Player class: first name, last name, player ID, birth year, birth country, weight, height, bats and throws.

We created a separate class for team data of each player. As a vector because we identified that the collisions happen when a player is part of different team, league and obviously different salary:

Team class: year, team ID, league ID and salary.

A perfect hash function assigns all records to unique locations in hash table with no wasted space, and all spots are assigned a unique hash code. While an imperfect has function has multiple keys assigned to same index, so we need some awareness of the data being hashed. Just increasing the table size won't necessarily work.

- **Vectors:**
    - We used a struct vector type to store the team data of each player node.
    - We wrote a reference in the header as a Team vector to create a vector once we encounter that index spot and link other to that team.
    - Either chained with linked list or open address as vertically linear with linear probing iteration.

- **Linked List:**
    - We used linked list structure to chain player nodes on the hash table. Using also an array of pointers.

Results:

Chained collision performance:

Contains a linked list of player nodes at each index, and are hashed to the table. The hash table indexes into an array of pointer of the heads of linked lists.

Problem:

Having to follow pointers to search in the linked list can have lower efficiency. Making the load factor much slower and its search linearly gets slower and slower.

Open address collision function:

On the other hand, open address indexes an array of pointers to slot in directly into the array, by pairing the data with their key and values. It uses linear probing where you look at the next slot after the one you have chosen. Then the next slot, and so on so forth. This will be happening until you find a slot that matches or hit an empty slot.

We have found that open-addressing is usually faster than chained linked list when you are trying to find a smaller loading factor. This is important because an algorithm's efficiency would depend on its speed and loading factor to take the data in and out.