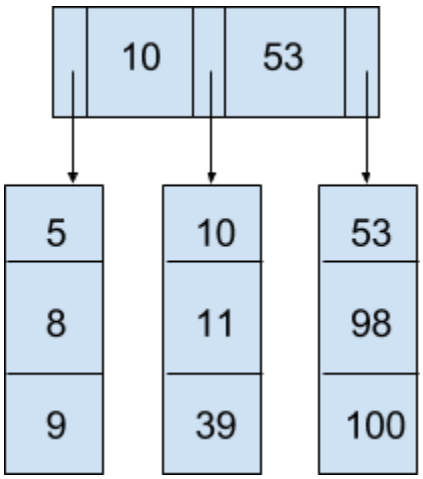
CptS 223 – Advanced Data Structures in C++

Written Homework Assignment 5: B+ Trees, Hashing, and Hash Tables

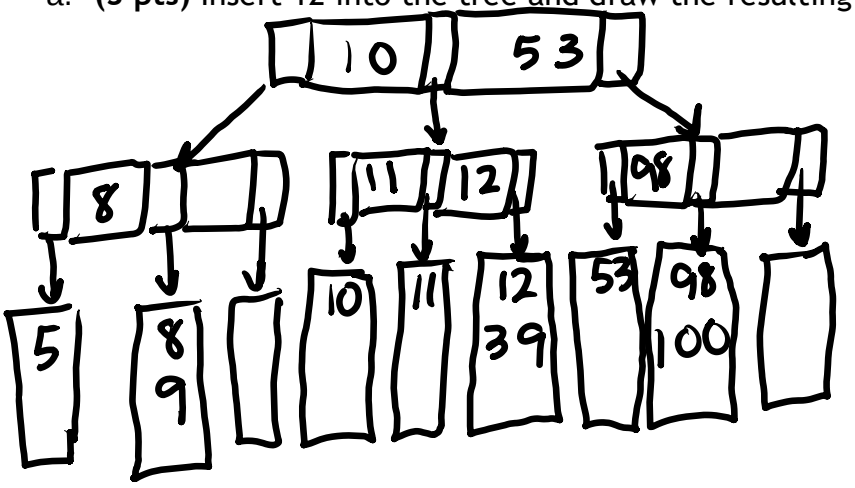
Assigned: Wednesday, October 28, 2020
Due: Wednesday, November 4, 2020

I. Problem Set:

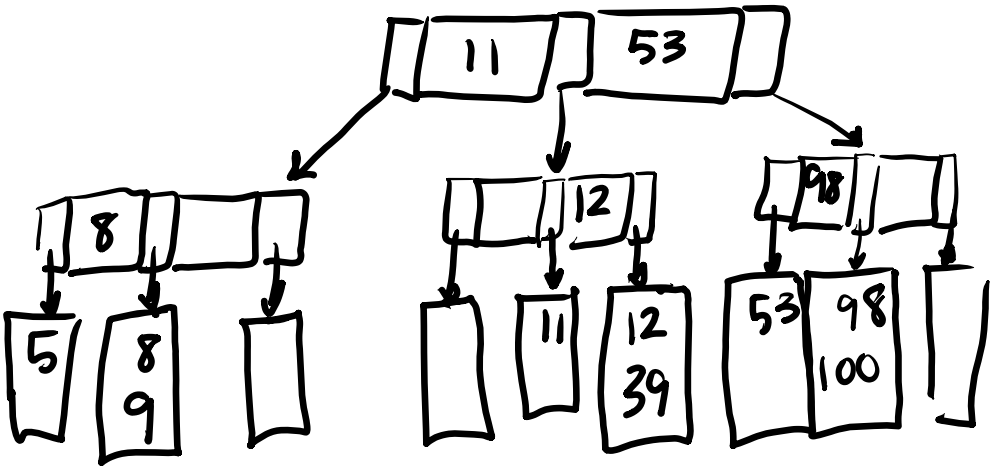
1. **(6 pts)** Given the following B+ tree (M = 3, L = 3):



a. **(3 pts)** Insert 12 into the tree and draw the resulting B+ Tree:



b. **(3 pts)** Based on the tree resulting from part (a), now remove 10 and draw the new tree:



2. **(16 pts)** We are going to design our B+ Tree to be as optimal as possible for our hard drives. We want to keep the tree as short as we can, and pack each disk block in the filesystem as tightly as possible. We also want to access our data in sorted order for printing out reports, so each leaf node will have a pointer to the next one. See figure #1 on next page for a visualization of our tree.

   CPU architecture: Intel Xeon with 64 bit cores
   Filesystem: Ext4 with 4KB (4096 byte) blocks
   The customer records are keyed by a random UUID of 128 bits
   Customer's Data record definition from the header file:

```
#include <uuid>
struct CustomerData
{
     uuid_t uuid;         // Customer 128 bit key
     char[32] name;  // Customer name (char is 1 byte each)
     uint32_t ytd_sales; // Customer year to date sales
};
```

a. **(4 pts)** Calculate the size of the internal nodes (M) for our B-tree:

   128 / 8 = 16 bytes
   (m * pointer) + ((m-1) * 16) <= 4096 bytes
   (m * 4) + ((m-1) *16) <= 4096
   4m + 16m – 16 <= 4096
   20m <= 4112
   m <= 205.6
   m <= 205 bytes

b. **(4 pts)** Calculate the size of the B-tree leaf nodes (L) for this tree make sure to include the pointer (note CPU architecture, which impacts the size of pointers!) to keep the list of leaf nodes:

   L = floor(B/D) where B = block size and D = data size

   floor((4096 – 4) / 52) = 78.38 records

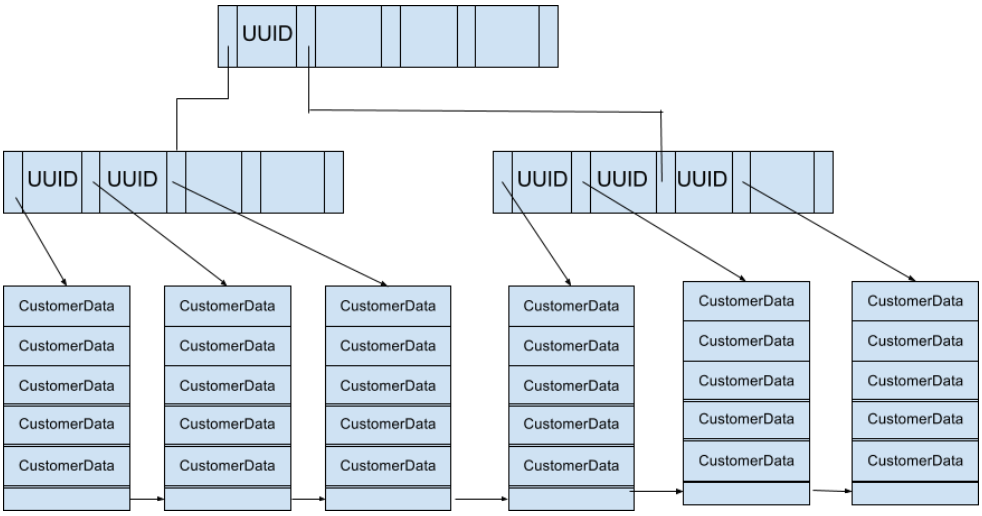


Figure #1: Visualization of our B+ Tree of height 2, customer data records, and pointers between the leaf nodes.

c. **(4 pts)** How tall (on average) will our tree be (in terms of M) with N customer records?

Log205(N/78) = height of the tree

d. **(4 pts)** If we insert 2,500,000 customers how tall will the tree be?

Log205(2500000/78) = 1.949
The height of the tree will be no more than 2.

3. **(12 pts - 4 pts/table)** Starting with an empty hash table with a fixed size of 11, insert the following keys in order into three distinct hash tables (one for each collision mechanism): {12, 9, 1, 0, 42, 98, 70, 3}. You are only required to show the final result of each hash table. In the **very likely** event that a collision resolution mechanism is unable to successfully resolve, simply record the state of the last successful insert and note that collision resolution failed. For each hash table type, compute the hash as follows:

```
hash(key) = (key * key + 3) % 11
```

Separate Chaining (buckets)

0	1	2	3	4	5	6	7	8	9	10
	3		0	12 1 98			9 42	70		

To probe on a collision, start at hash(key) and add the current probe(i') offset. If that bucket is full, increment i until you find an empty bucket.

Linear Probing: probe(i') = (i + 1) % TableSize

0	1	2	3	4	5	6	7	8	9	10
	3		0	12	1	98	9	42	70	

Quadratic Probing: probe(i') = (i * i + 5) % TableSize

0	1	2	3	4	5	6	7	8	9	10
	42		0	12		3	9	70	1	98

4. **(2 pts)** If you wanted to implement a hash table, then which of these would probably be the best initial table size to pick?

   Table Sizes:

   7 100 101 27 525

   Why did you choose that one?

   The best initial hash table size to pick would be size 101. 101 is the best size to pick because it will have a lower load factor and is a prime number.

5. **(6 pts)** For our running hash table, you'll need to decide if you need to rehash. You just inserted a new item into the table, bringing your data count up to 53491 entries. The table's vector is currently sized at 106963 buckets.

   - **(2 pts)** Calculate the load factor (λ):

   N = number of elements in T
   M = size of T
   λ = N/M

   λ = 53491 / 106963 = 0.5

   - **(2 pts)** Given a linear probing collision function should we rehash? Why?

     We should rehash for the linear probing collision because the load factor has exceeded 0.5.

   - **(2 pts)** Given a separate chaining collision function should we rehash? Why?

     We do not need to rehash with separate chaining collision function because the element will be inserted to the front of the linked list.

6. **(4 pts - 1 pt/blank)** What is the Big-O of these actions for a well-designed and properly loaded hash table with N elements?

Function	Big-O complexity
Insert(x)	O(1), Worst-case O(n)
Rehash()	O(n)
Remove(x)	O(1), Worst-case O(n)
Contains(x)	O(1), Worst-case O(n)

7. **(6 pts - 3 pts/each)** Enter a reasonable hash function to calculate a hash value for these function prototypes:

```
int hashit( int key, int tablesize )
{
        return key % tablesize;
}
```

```
int hashit( std::string key, int tablesize )
{
   int hashVal = 0;

   For(int i = 0; I < key.length(); i++)
   {
        hashVal = 37 * hasVal + key[i];
   }

   hasVal %= tablesize;
   if(hashVal < 0)
        hashVal += tablesize;

   return hashVal;
}
```

8. **(3 pts)** The hash table below works, but once we put more than a few thousand entries, the whole thing starts to slow down. Searches, inserts, and contains calls start taking *much* longer than O(1) time. It is problematic because it is slowing down the whole application services backend. We think the performance issue stems from the rehash code, but we are not sure where. Any ideas why the hash table starts to perform poorly as it grows bigger?

```
/**
* Rehashing for linear probing hash table.
*/
void rehash( )
{
      vector<HashEntry> oldArray = array;

      // Create new double-sized, empty table
      array.resize( 2 * oldArray.size( ) );

      for( auto & entry : array )
            entry.info = EMPTY;

      // Copy table over
      currentSize = 0;
      for( auto & entry : oldArray )
            if( entry.info == ACTIVE )
                  insert( std::move( entry.element ) );
}
```

The reason why the hash table starts to perform poorly as it grows bigger is because the rehash function doesn't rehash the elements back into the larger vector. Instead it doubles the size of the array and puts the elements back into their original spots.

II. Submitting Written Homework Assignments:

1. On your local file system, create a new directory called HW5. Create/Move your HW5.pdf file in the directory. In your local Git repo, create a new branch called HW5. Add your HW5 directory to the branch. Add and commit HW5.pdf. Merge HW5 branch with the Master Branch and push the Master branch to the remote origin.

III. Grading Guidelines:

This assignment is worth 55 points. We will grade according to the following criteria:

- See above problems for individual point totals.