CSCI 6620 Spring 2016

Program 4: A Bucket Sort Using Queues

1 Overview.

A bucket sort[1] is the fastest sort for a vary large number of data items[2]. It is an O(n) sort that works by using groups of bits from the sort keys rather than by comparing key values to each other. It is the fastest way to sort a large number of items. However, it *does* involve considerable setup time and data structures, and works only with fixed-length sort keys. A data item could be any type of object (or pointer to object) that has one data member that acts as a sort key.

The bucket sort described here is applicable to any large data set, but the presentation is specific to the one you will be using. In this assignment, you will implement an array of queues. You will build your own queue class, not use the C++ STL queue.

The Data to Sort

- We will be sorting objects with two data fields: (1) the time (type time_t), a 4-byte unsigned integer. (2) the temperature, a 4-byte float.
- The data file will consist of a very large number of time/temperature measurements, recorded in January at an apartment in Tuckahoe, New York, arranged by time.
- There will be three files, one from each of three sensors on different sides of the building. You only need to process one of the files, but different people should choose different files. Handle file opening and EOF properly.

Overview of the Sort The algorithm divides the sort key into k fields, based on its binary representation and makes k passes through the entire data list. On pass 0, the least significant (rightmost) field of the key is isolated and used to assign the data item to a bucket-queue. On pass p, the pth field is used. At the end of each pass, all the data is collected again into a single queue, ready for re-distribution. After k passes, the data is sorted.

For this program:

- First, all of the data must be read from a file, and each data item installed in a new cell. The cells must each be linked onto the end of a queue called the master queue. Data cells will be distributed into buckets.
- An array of 256 buckets must be created. Each bucket is a linked queue, initially empty.
- When the data and buckets are ready, sorting begins. It will work in four passes (numbered 0, 1, 2, and 3), where each pass consists of a distribute operation (next slide) and a collect operation (slide 26) .
- After distributing all the data into the buckets. the queues from the buckets must be collected and joined back into a single master queue.
- At the end of four passes, the data will again be in the master queue, and will be sorted. Output the sorted data to a file.
- The number of fields in the key determines the number of passes.
- The number of bits in each field determines the number of buckets needed: for k bits, we need 2^k buckets.
- The algorithm is simpler to implement if each field is a half-byte, a byte, or two bytes, resulting in 16 or 256 or 35536 queues.

[1] This technique has been in use since the days of punched cards and card-handling machines.

[2] Google's map-reduce is based on a bucket sort.

2 The Bucket Sort Algorithm.

One distribution pass. Distribute each cell from the master list into a bucket as follows:

- Remove it from the front of the master queue.
- Access one byte of the temperature and use the result as a subscript to select a bucket. Isolate that byte using shifting and masking. Define `mask = 0xff;` to isolate one byte (8 bits).
  - Start with the temperature value from the cell you are distributing.
  - Compute `subscript = mask & (temp >> n)`
    where `n` is $8 * k$, and k is the pass number.
  - The computed subscript is the number of the bucket to use for this data item.
  - Detach the cell from the master queue and push it onto the selected bucket queue.
- In pass k, use the kth byte from the right.
- Attach the cell to the end of the queue in that bucket.
- In this process, the actual sort key must not be changed.

Collecting the data into the master queue. At the end of each distribution pass, your master queue is empty. Now collect all the cells, as follows:

- Find the first non-empty bucket, (k), in the bucket array.
- Start at bucket k and process buckets, in order, through number 255.
- If the bucket is not empty,
  - Attach the last cell of the master queue to the first non-dummy cell in the bucket.
  - Set the back pointer of the master list equal to the back pointer of the bucket.
  - All the cells have now been removed from the bucket, so reset it to empty.

Illustration of a bucket sort process. In these diagrams, we show a bucket sort of 14 short integers, written here in hex. We make 4 passes, sorting on one nybble each time. The original data list is shown on the left:

Data:	BUCKETS for distribution pass 0: (rightmost hex digit)																Collect 0	Collect 1	Collect 2	Collect3
9123	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Distr.1	Distr.2	Distr.3	Sorted
438B	FA10	4341		9123		84C5		C437	63A8			438B	973C	A18D	245E		FA10	F00D	F00D	1743
1743				1743										F00D			4341	FA10	9123	245E
C437														BEAD			9123	9123	A18D	4341
A18D														DEAD			1743	C437	4341	438B
F00D	BUCKETS for distribution pass 1: (thrid hex digit)																84C5	4341	438B	63A8
BEAD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	C437	1743	63A8	84C5
FA10	F00D	FA10	9123	C437	4341	245E			438B		63A8	A18D	84C5				63A8	245E	C437	9123
245E					1743						BEAD		973C				438B	438B	245E	973C
63A8											DEAD						973C	63A8	84C5	A18D
DEAD	BUCKETS for distribution pass 2: (second hex digit)																A18D	BEAD	1743	BEAD
84C5	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	F00D	DEAD	973C	C437
973C	F00D	9123		4341	C437			1743			FA10				BEAD		BEAD	A18D	FA10	DEAD
4341		A18D		438B	245E			973C							DEAD		DEAD	84C5	BEAD	F00D
				63A8	84C5												245E	973C	DEAD	FA10
	BUCKETS for distribution pass 3: (first hex digit)																			
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				
		1743	245E		4341		63A8		84C5	9123	A18D	BEAD	C437	DEAD		F00D				
					438B					973C						FA10				