**Lab 4 – Queues, RadixSort**

Answers to questions \*before\* writing, or running, code:

1. The theoretical Big-O execution time \*should\* be linear but, as we learned in class, our code won’t because of the implementation.
2. The internet says the same thing – that it should be linear.

Answers:

Overall, the graph shape is looking like it’s O(n)^2 with the way it shoots up as it runs through the sorting. This is because we’re doing everything from the front of the vector and having those ghost operations happen when every element of the vector must be shifted. We want to be able to avoid this and make it as close to linear as possible.

For the second iteration of the queue (the one we made ourselves) I didn’t have to change a ton, but enough that the functionality of the output was much closer to linear timing like expected. I added the “beg” to track the beginning of the index which lets us use pop() as closer to a linear output. I also initially allocated at least 1000 slots, so it won’t resize on every operation. The “beg” and “end” keep track of the “active” part of the vector. This means I’m keeping the elements in place until the vector needs to be resized.

I probably tinkered a little too much with the code because, in reality, the biggest change was just changing how pop() works so we don’t have to shift all the elements over to fill the gap after we pop from the front.

Output of the first iteration of radix\_sort in my terminal: A screenshot of a computer

AI-generated content may be incorrect.

Here is the timing result for the first iteration of the radix\_sort algorithm

Terminal output for the radix algorithm where I used my own class: A screenshot of a computer

AI-generated content may be incorrect.

And here is the graph showing the timing result:

Code appendix:

vecRadix.cpp – This is where I implemented the algorithm to use with the queue. The code I’m pasting is from when I used my personal queue. So, the header file that’s included is for that class. When I used the original queue we made from class, I just swapped the included header files, and did a quick search/replace so I could change the class names so I could call everything correctly.  
 1. 1. #include "myQueue.h"

2. 2. #include <cmath>

3. 3. #include <cstdlib>

4. 4. #include <ctime>

5. 5. #include <fstream>

6. 6. #include <iomanip>

7. 7. #include <iostream>

8. 8. #include <vector>

9. 9. #include <chrono>

10. 10.

11. 11. using namespace std;

12. 12.

13. 13. vector<unsigned long long> RadixSort(int sz) {

14. 14. myQueue<unsigned long long> mainBin(sz);

15. 15.

16. 16. vector<unsigned long long> original;

17. 17.

18. 18. for (int i = 0; i < sz; i++) {

19. 19. unsigned long long random\_number = 100000000 + rand() % 900000000;

20. 20. mainBin.push(random\_number);

21. 21. original.push\_back(random\_number);

22. 22. }

23. 23.

24. 24. if (sz <= 10) {

25. 25. cout << "Original list: " << endl;

26. 26. for (unsigned long long num : original) {

27. 27. cout << num << " ";

28. 28. }

29. 29. cout << endl << endl;

30. 30. }

31. 31.

32. 32. myQueue<unsigned long long> digitBins[10] = {

33. 33. myQueue<unsigned long long>(sz), myQueue<unsigned long long>(sz),

34. 34. myQueue<unsigned long long>(sz), myQueue<unsigned long long>(sz),

35. 35. myQueue<unsigned long long>(sz), myQueue<unsigned long long>(sz),

36. 36. myQueue<unsigned long long>(sz), myQueue<unsigned long long>(sz),

37. 37. myQueue<unsigned long long>(sz), myQueue<unsigned long long>(sz)};

38. 38.

39. 39. for (int position = 0; position < 9; position++) {

40. 40. while (!mainBin.empty()) {

41. 41. unsigned long long num = mainBin.pop();

42. 42. int digit =

43. 43. (num / static\_cast<unsigned long long>(pow(10, position))) % 10;

44. 44. digitBins[digit].push(num);

45. 45. }

46. 46.

47. 47. for (int d = 0; d < 10; d++) {

48. 48. while (!digitBins[d].empty()) {

49. 49. mainBin.push(digitBins[d].pop());

50. 50. }

51. 51. }

52. 52. }

53. 53.

54. 54. vector<unsigned long long> results;

55. 55. while (!mainBin.empty()) {

56. 56. results.push\_back(mainBin.pop());

57. 57. }

58. 58. return results;

59. 59. }

60. 60.

61. 61. int main() {

62. 62. srand(static\_cast<unsigned int>(time(nullptr)));

63. 63.

64. 64. cout << "Testing RadixSort with 10 elements:" << endl;

65. 65. vector<unsigned long long> sorted = RadixSort(10);

66. 66.

67. 67. cout << "Sorted list: " << endl;

68. 68. for (unsigned long long num : sorted) {

69. 69. cout << num << " ";

70. 70. }

71. 71. cout << endl << endl;

72. 72.

73. 73. ofstream csvFile("radix\_sort\_timing.csv");

74. 74. csvFile << "Size,Time(seconds)" << endl;

75. 75.

76. 76. cout << "Generating timing data..." << endl;

77. 77.

78. 78. for (int size = 1000; size <= 10000; size += 1000) {

79. 79. auto start = chrono::high\_resolution\_clock::now();

80. 80. RadixSort(size);

81. 81. auto end = chrono::high\_resolution\_clock::now();

82. 82.

83. 83. chrono::duration<double> elapsed = end - start;

84. 84. csvFile << size << "," << fixed << setprecision(5) << elapsed.count() << endl;

85. 85. cout << "Completed size " << size << endl;

86. 86. }

87. 87.

88. 88. csvFile.close();

89. 89. cout << "Timing data saved to radix\_sort\_timing.csv" << endl;

90. 90.

91. 91. return 0;

92. 92. }

93. 93.

94.

Here is the code for my personal queue implementation using the “beg” and “end” to make the queue work closer to a linear timing.  
 1. #include <vector>

2. #include <stdexcept>

3.

4. template <typename T> class myQueue {

5. private:

6. std::vector<T> queue;

7. int capacity;

8. int beg; // Index of first element

9. int end; // Index after last element

10.

11. public:

12. myQueue(int sz);

13. void push(T val);

14. T pop();

15. T front() const;

16. T back() const;

17. int size() const;

18. bool empty() const;

19. };

20.

21. template <typename T>

22. myQueue<T>::myQueue(int sz) : capacity(std::max(sz, 1000)), beg(0), end(0) {

23. queue.resize(capacity);

24. }

25.

26. template <typename T> void myQueue<T>::push(T val) {

27. if (end >= capacity) {

28. // Need to resize

29. int newCapacity = capacity + 1000;

30.

31. // If we have data at the beginning we need to make room

32. if (beg > 0) {

33. // Shift data to the beginning

34. for (int i = 0; i < size(); i++) {

35. queue[i] = queue[beg + i];

36. }

37. end = size();

38. beg = 0;

39. } else {

40. // Just increase capacity

41. queue.resize(newCapacity);

42. capacity = newCapacity;

43. }

44. }

45.

46. queue[end] = val;

47. end++;

48. }

49.

50. template <typename T> T myQueue<T>::pop() {

51. if (empty()) {

52. throw std::underflow\_error("Queue is empty");

53. }

54.

55. T val = queue[beg];

56. beg++;

57. return val;

58. }

59.

60. template <typename T> T myQueue<T>::front() const {

61. if (empty()) {

62. throw std::underflow\_error("Queue is empty");

63. }

64. return queue[beg];

65. }

66.

67. template <typename T> T myQueue<T>::back() const {

68. if (empty()) {

69. throw std::underflow\_error("Queue is empty");

70. }

71. return queue[end - 1];

72. }

73.

74. template <typename T> int myQueue<T>::size() const {

75. return end - beg;

76. }

77.

78. template <typename T> bool myQueue<T>::empty() const {

79. return beg == end;

80. }

81.