알고리즘 설계와 분석 HW1

학번: 20192134

이름: 김은원

1. 실험환경

OS: window 10 home

CPU: Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz

RAM: 8GB 2400Mhz DDR4

Compiler: Visual Studio 2019 Release Mode

1. 실험과정

Insertion sort: 일반적인 insertion sort 알고리즘을 구현하였다. Data size인 N에 따라 시간을 계산하였다.

Heap sort: insert heap, delete heap 함수를 사용하여 heap sort를 구현하였다.

Quick sort: pivot을 데이터 제일 앞과 중간, 그리고 마지막 원소 중 median을 pivot으로 설정하여 구현하였다.

Optimized quick sort: 위 quick sort에서 조금 보완한 정렬 방법이다. pivot으로 나뉜 배열 중 짧은 것을 재귀적으로 구하고 길이가 긴 것은 iterative하게 구현하였다.

1. 실험결과

n개의 data size를 가지고 있는 데이터를 sort하는 데 걸리는 시간을 표로 나타냈다.

각각 5번의 실험의 평균을 구했다. 시간 단위는 milliseconds이다.

1. Entirely random: 적절한 범위의 랜덤 숫자를 생성하여 만들어진 데이터

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | insertion | Heap sort | quick | Quick opt |
| 32 | 0.017900 | 0.025733 | 0.026800 | 0.049900 |
| 64 | 0.020700 | 0.025067 | 0.040200 | 0.057000 |
| 128 | 0.1088 | 0.072867 | 0.0498 | 0.0469 |
| 1024 | 3.5816 | 1.023267 | 0.6665 | 0.717633 |
| 65536 | 10991.639648 | 71.27136 | 52.137199 | 52.036098 |
| 1048576 | X | 875.1458 | 924.8678 | 922.1609 |

1. Descending: key값이 정반대로 정렬되어 있는 데이터

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | insertion | Heap sort | quick | Quick opt |
| 128 | 0.0807 | 0.095633 | 0.0842 | 0.066467 |
| 1024 | 8.8865 | 0.595 | 1.064333 | 0.963867 |
| 65536 | 19751.931641 | 47.0334 | 62.889301 | 82.148399 |
| 1048576 | x | 597.3667 | 1491.343 | 1485.587 |

(c ) Few Swaps : 정렬되어 있는 상태에서 랜덤으로 데이터 몇 개만 바뀌어 있는 데이터.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | insertion | Heap sort | quick | Quick opt |
| 128 | 0.018 | 0.076267 | 0.0627 | 0.078067 |
| 1024 | 0.0488 | 0.532033 | 0.578867 | 0.684933 |
| 65536 | 2.659800 | 27.0316 | 34.255001 | 54.650299 |
| 1048576 | 46.0163 | 507.7889 | 715.4395 | 709.7399 |

1. 실험 결과 분석
2. 과연 수행 시간과 이 방법들의 이론적인 시간 복잡도간에 깊은 연관성을 발견할 수 있는가?

다음은 nlogn과 heap sort, quick sort, quick optimized sort의 n 크기에 따른 시간 변화율을 계산한 결과이다. 데이터는 random인 경우로 계산했다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | nlogn 변화율 | heap변화율 | quick변화율 | quickopt변화율 |
| 32 | 2.4 | 0.974119 | 1.5 | 1.142285 |
| 64 | 2.333333 | 2.90689 | 1.238806 | 0.822807 |
| 128 | 11.42857 | 14.04294 | 13.38353 | 15.30134 |
| 1024 | 102.4 | 69.65079 | 78.22535 | 72.51074 |
| 65536 | 20 | 12.27907 | 17.73912 | 17.72156 |

Insertion sort의 경우는 시간복잡도가 O(N2)으로 데이터 크기가 커질수록 다른 sort 알고리즘에 비해 큰 폭으로 증가했다. random에서는 O(N2)에 가까운 시간 복잡도를 보였지만 Few swaps 에서는 O(N)에 가까운 시간 복잡도를 보였다.

Heap sort의 경우 이론적으로 시간복잡도가 O(nlogn) 이다. 하지만 수행결과는 nlogn의 변화율보다 적게 나타났다. 즉 증가속도가 이론상보다 느리게 나타났다. 그러나O(nlogn)과 유사한 증가속도를 보이고 있다.

Quick sort와 Quick opt sort의 경우 이론상의 O(nlogn)보다는 변화율이 적게 나타났다. 하지만 O(nlogn)과 유사한 증가속도를 보이고 있다.

따라서 heap sort, quick sort, optimized quick sort 모두 O(nlogn)과 연관성이 높다는 것을 알 수 있다.

1. 이때 서로 다른 종류의 데이터에 대하여 수행 시간에 차이가 있는가? 있다면 그 이유는 무엇으로 추정되는가?

Insertion의 경우 서로 다른 종류의 데이터에 대해 수행 시간 차이가 분명하다. 가장 빠른 데이터는 Few swaps이다. Insertion sort 특징으로 거의 정렬된 데이터에 대해서는 비교하여 데이터를 바꿔줄 필요가 줄어들기 때문에 속도가 빠른 것이다. Descending에서는 반대로 비교하여 데이터 원소 값을 교환할 양이 많기 때문에 시간이 오래 걸렸다.

Heap의 경우는 insertion 만큼 차이가 나진 않지만 Few swaps 가 제일 시간이 적게 나타났다. Heap 구조상 descending으로 데이터가 배열되면 작은 수가 앞으로 정렬되는 tree이기 때문에 매번 작은 수가 맨 위 node로 올라가야 한다. 따라서 descending 이 few swaps 보다 시간이 오래 걸린다.

Quick sort, Quick opt sort의 경우는 Few swaps 가 제일 빨랐다. Pivot을 기준으로 나눌 때 Few swaps는 pivot이 대체로 중간 값이 배열 가운데에 있어 속도가 빨랐다.

1. quick sort 방법은 그 이름이 의미하는 바와 같이 정말 빠른가?

Few swaps에서는 데이터 크기가 128일 때 insertion sort가 0.018이고 quick sort가 0.0627로 insertion sort가 더 빠른 것으로 나타났다. 또한 모든 Few swaps에서는 quick sort보다 insertion sort가 속도가 더 빨랐다. 따라서 항상 quick sort가 빠른 것은 아니다.

1. 어떤 부류의 데이터에 대해 insertion sort 방법의 수행 시간이 heap sort 또는 quick sort 방법에 비해 빠르거나 그리 느리지 않은가? 과연 insertion sort 방법이 거의 선형적인 시간에 작동하는 경우를 발견하였는가?

Few swaps의 경우 insertion sort 방법이 heap sort, quick sort, quick optimized sort보다 빠르거나 그리 느리지 않았다. 거의 정렬되어 있는 배열에 대해선 insertion sort가 거의 선형적인 시간에 작동하여 다른 sort보다 수행 속도가 빠른 것을 알 수 있다.

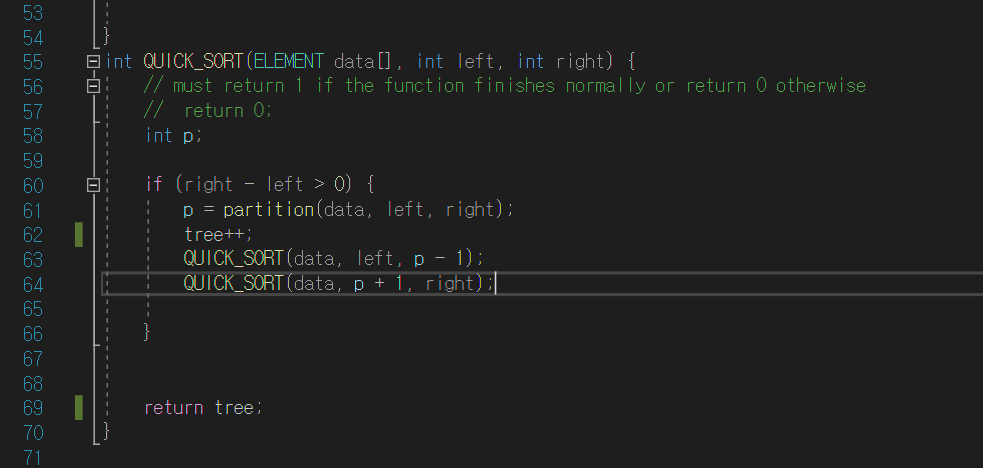
1. n 값이 작을 경우 굳이 quick sort 방법을 사용할 필요없이 insertion sort 방법을 사용해도 크게 문제 가 되지 않는다. 과연 이는 어느 정도 범위의 n 값까지 일지 Entirely random 데이터에 대한 실험을 통하여 자신의 값을 제시하라

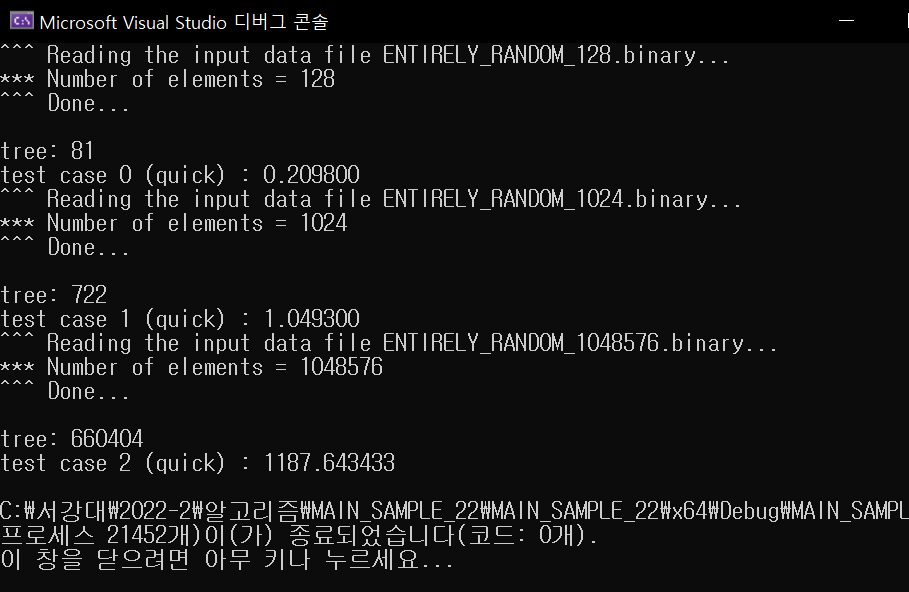
실험 결과, n값이 64일때까지는 insertion sort 방법이 다른 sort 방법보다 빠르지만 n =128 부터는 다른 모든 sort 방법보다는 속도가 느린 것으로 나타났다. 따라서 데이터가 128보다 작으면 insertion sort 방법이 더 빠르다는 것을 알 수 있다.

1. 입력 데이터의 크기 n에 quick sort 방법을 수행할 때 재귀적으로 생성되는 트리의 깊이가 어느 정도 되는지 각 리프 노드의 레벨에 대한 최소/최대/평균을 적절히 크기가 큰 Entirely random 데이터에 대한 실험을 통하여 자신의 값을 제시하라. 평균적으로 최악의 경우에 해당하는 O(n)에 가까운지 아니면 수업 시간에 이론적으로 증명한 바와 같이 O(n log n)에 가까운지 실험적으로 분석할 것.

Partition 함수를 통해 몇 번 pivot이 발생했는지 계산하였다. N=128이면 이진 트리 공식에 따라 n=128=27 이고 depth가 k 일 때 최대 노드가 2k+1-1인 공식을 통해 2k+1-1 = 128이고 계산결과 k = 6.011이다. K=6이면 최대 노드의 개수는 127이고 따라서 n=128이기 때문에 이상적으로 나뉘었을 때 depth가 7인 것을 알 수 있다.

Quick sort.cpp 에 전역변수 tree를 선언하고 다음처럼 tree를 계산하여 main.cpp에 출력해보았다.





Tree는 클수록 pivot을 많이 찾는다는 점에서 효율이 좋지 않다. 예를 들어 N=128일 때 최소 tree는 26인 64이다. 실험 결과는 다음과 같이 나타났다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | N=2m | Tree | Log2(Tree) | 결과 depth |
| 7 | 128 | 81 | 6.339 | 7 |
| 10 | 1024 | 722 | 9.495 | 10 |
| 20 | 1048576 | 660404 | 19.332 | 20 |

Depth는 결과적으로 O(log2n)에 가깝다는 점을 알 수 있다.

1. 각 방법을 최대한 효율적으로 구현하기 위하여 어떤 노력을 기울였는가?

Quick sort 알고리즘에서 pivot을 구할 때 최대한 중간 값을 구하기 위해 첫번째 원소와 중간 원소와 마지막 원소를 비교하였다. 이때 원소가 3개보다 큰 경우만 median 값을 구하도록 했다. 2개인 경우 median을 구하는 것보다 바로 정렬하는 것이 효율적이기 때문이다. Optimized Quick sort 알고리즘에서는 pivot이 가운데 인덱스보다 작으면 pivot 왼쪽 부분을 recursive하게, pivot 오른쪽 부분을 iterative하게 구현하여 보다 recursive를 적게 하기위해 노력했다.

1. 기타 이번 숙제를 하면서 적용한 창의적인 방법이나 경험적으로 알게 된 중요 사항을 기술하라.

Insertion sort 알고리즘이 거의 정렬된 데이터에서는 막강한 알고리즘이라는 사실을 알게 되었다. 평균적으로 O(n2)인 알고리즘이 어떤 경우에서는 O(n)으로 선형 시간 내에 계산이 가능하다는 사실이 인상 깊었다. 또한 Heap sort는 이론상 O(nlogn)이고 실험 결과, 비슷한 결과를 나타냈다. Quick sort의 경우 pivot이 가장 왼쪽 또는 오른쪽인 경우인 worst 경우 heap sort보다 속도가 느린 점을 실험 결과에서도 경험적으로 알 수 있었다.