External Sorting

대용량 데이터를 처리할 수 있는 알고리즘. 정렬 해야하는 데이터를 메모리에 올릴 수 없을 때 사용함. 보통 merge sort를 반복적으로 수행함 (External merge sort)

<방법>

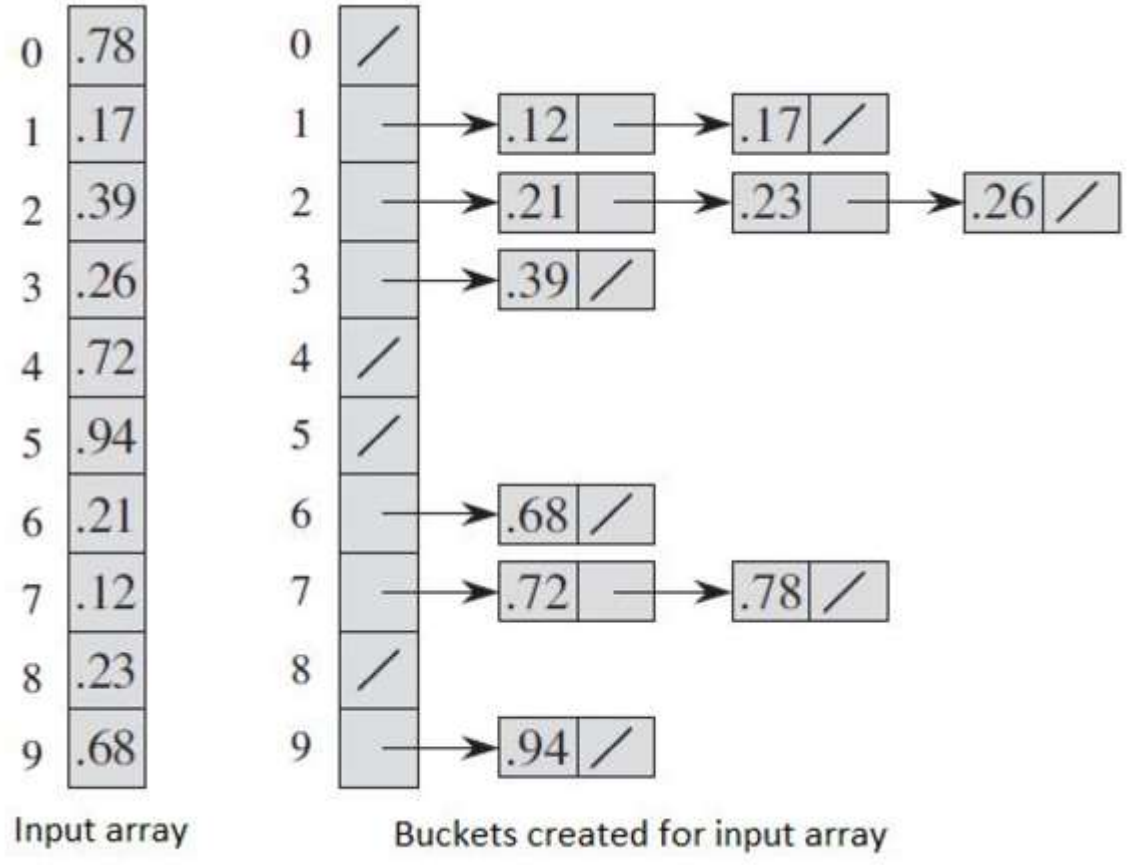
900MB를 100MB가 가능한 메모리를 이용해서 정렬하는 경우

9번의 sort를 메모리에서 하고 디스크에 저장함.

디스크에 저장된 각 9개에 대해서 10MB씩 빼서 정렬하고...

물론 bucket sort를 사용해도 External sorting을 할 수 있음.

Bucket sort : O(n)



In place sort

정렬에서 추가적인 메모리가 필요 없는 정렬

Stable sort

동일한 값의 데이터가 정렬 후에도 순서를 유지하는 sorting

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Radix

http://blog.naver.com/PostView.nhn?blogId=markmarine&logNo=220657572364&parentCategoryNo=&categoryNo=&viewDate=&isShowPopularPosts=false&from=postView

eg) 우리나라 인구 나이로 정렬할 때 씀

Bubble sort : Stable, In place, Best o(n)

Selection sort : In place, Unstable

Insertion sort : Best o(n), Stable, In place

Adaptive; efficient for data sets that are already substantially sorted. Time complexity is O(nk) when each element in the input is no more than k places away from its sorted position

Online; can sort a list as it receives it

Merge sort : Stable, Not In place

Quick sort : Unstable, Not In place

Merge sort VS Quick sort

Merge sort는 모든 경우에 n log n이고 Quick은 최악에 n^2임

하지만 보통 quick sort가 merge sort보다 좋다고 평가함.

Why? 실제 런타임에서 걸리는 시간에 영향을 미치는 요소 중 swap 횟수가 중요함.

이유는 swap을 하려면 메모리에서 데이터를 읽어야하는데 메모리에서 데이터를 읽는 행위 자체가 cpu의 성능 대신 메모리의 성능에 의존하기 때문임.

Quick sort는 little additional space만 필요하고 cache locality도 있어서 merge 보다 좋음.

Quick sort는 left랑 right를 한칸 씩 움직이니까 locality, merge는 나누니까 안되즤

또한 실제로 현대 quick sort는 최악의 케이스가 n log n임 (피봇만 잘 고르면 되니까)

Java collection interface (Set / List / Map)

Map, Set, List는 interface임 이걸 구현한게 hashMap그런거..

List : 중복 O, 순서 유지(들어간 순), null 허용

LinkedList는 값을 자주 추가하거나 제거할 때 쓰임

Vector는 동기화를 제공해서 multithread에 쓰임

Set : 중복 X, 순서 X (LinkedHashSet은 순서유지, TreeSet은 정렬 순서 유지)

null 1개만 허용(중복x)

Map : value만 중복 O, 순서 X (TreeMap은 정렬 순서 유지), null은 key 1개 value 여러 개

HashMap도 null key 1, value 여러 개, 비동기

Hashtable은 null X, 동기화

When to use?

- index로 접근할 때 : List
- 입력한 순서를 유지하고 싶을 때 : List
- 중복 없는 거 : Set

LinkedHashSet : 입력순서 유지

TreeSet : 정렬순서 유지

HashMap vs Tree

둘다 DB indexing이나 데이터 저장할 때 쓰임

속도 : Map은 search가 O(1), Tree는 O(log n) = height of tree

Map은 1개를 찾을 때, Tree는 범위를 찾을 때(leaf에서 정렬 되있으니까)

메모리 : Hash는 쏠림 현상이 있고 Tree는 부모 node를 저장해야함

하지만 쏠림으로 인한 체이닝 등이 더 영향이 큼

Hash

해시 버킷 <-> 해시 충돌 : trade off 관계

해시 충돌 해결법

- Open addressing : 충돌 시 다른 버킷에 저장
- Separate Chaining : 링크드리스트

  데이터 개수가 적을 때는 Open addressing이 유리함.

  Remove가 자주 일어날 때는 Separate chaining이 유리함

  Java에서는 separate chaining을 사용. (Java8은 내부적으로 red-black tree 사용)

Heap

힙은 special tree-based 자료구조임. 대개 완전 이진 트리를 기반으로 max, min heap이 있음.

Head sort는 빈 heap에 데이터를 넣고 재구성하면서 root를 배열에 하나씩 넣고 다시 재구성하는 방식으로 진행됨.

Min heap 기준 복잡도 / merge는 힙 2개를 합치는거

Operation	Binary[6]	Binomial[6]	Fibonacci[6][7]	Pairing[8]	Brodal[9][b]	Rank-pairing[11]	Strict Fibonacci[12]
find-min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$
insert	$O(\log n)$	$\Theta(1)$[c]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[c]	$o(\log n)$[c][d]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
merge	$\Theta(n)$	$O(\log n)$[e]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Priority queue

일반 queue와 달리 넣은 순서가 아니라 우선순위가 빠른놈부터 나옴

우선순위 큐를 구현하는 방법은 대게 array / linked list / heap

우선순위 큐는 우선순위에 따라서 정렬되야 하니까 재배치 할 때 다른 데이터와 비교해야 하는데 heap은 tree기반이라 자식들과 비교하면 되지만 나머지는 다른 모두랑 비교해야함.

어떤 tree를 기반으로 하느냐에 따라 복잡도가 다른데 바로 위 그림이랑 똑같음.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)