Completed on 2018.7.5

2018.5.21 BubbleSort.java, SelectionSort.java; InsertionSort.java

Array

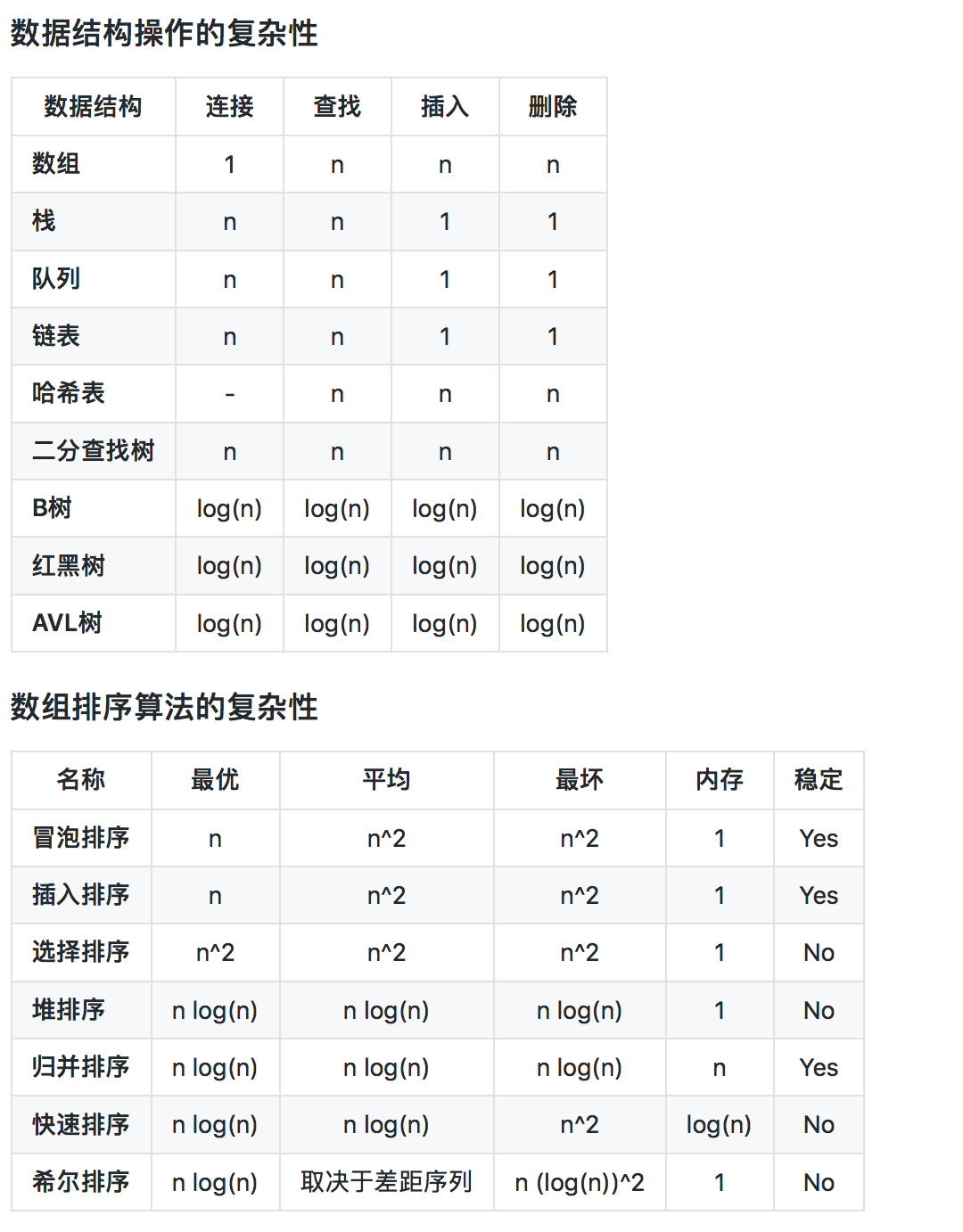
Sort:

Unstable sort is the relative of duplicate items will not be preserved.

Stable sort is the relative ordering was preserved.

Bubble sort： O(n^2); Stable sort

Selection sort: O(n^2); Unstable sort



2018.5.22: InsertionSort.java; ShellSort.java; Factorial.java

Insertion sort: O(n^2); Stable sort

Shell sort: O(n^2) worst case, but can perform better than that; Unstable algorithm

Merge Sort: O(nlogn)

2018.5.23: MergeSort.java; QuickSort.java

Quick sort: in-place sort; O(nlogn); unstable;

Counting sot: not in-place sort; O(n); Works when the number of values want to sort= number ranges;

2018.5.28: Radix Sort

Radix: the number of unique digits or letters;

Width: the number of digits or letters;

Data must have same radix and width

Must use a stable sort algorithm at each stage. 必须stable sort

Sort 1’s , 10’s, 100’s position….Sorting from the least significant digit to the most significant digit.

Counting sort if often used as a sort algorithm for radix sort. O(n). Often runs slower than O(nlogn) because of the overhead involved.

2018.6.1: radixSort.java

2018.6.2 InsertionSort2.java; mergeSortDescend.java

List: ordered sequence; Abstract data type

ArrayList:

2018.6.3 LinkedList.java

LinkedList: singly linked list; Doubly linked list

A->B->C….Z->NULL

2018.6.4: DoublyLinkedList.java; jdkLinkedList.java

DoublyLinkedList:

NULL<-A<->B<->C<->…<->Z->NULL

Circular LinkedList: a variation of linkedlist, but the last node in the list points to the first node.

2018.6.5: LinkedList chanllenge 1&2 .java; stacks.java…valid palindrome.java(leetcode)

Stack: LIFO: last in first out

Stack: three operations: push, pop, peek

It’s just like singly linked list

Define a stack using linkedlist

LinkedList<E> stack = new LinkedList<>();

…

Queue: FIFO: first in first out. 排队一样

Three operations: add/enqueuer: add items at the end of the queue;

Remove/dequeuer: remove item at the front of the queue;

Peek: get the item at the front of the queue

2018.6.6: Queue.java

circular queue

2018.6.8: Queue.challenge.java

finish queue. Continuing on hashtable

2018.6.10: Evaluate Reverse Polish Notation.java;

Reverse Polish Notation: <http://en.wikipedia.org/wiki/Reverse_Polish_notation>

2018.6.12: HashTable:

Hashing: Maps keys of any data type to integers. Hash function maps keys to integers.

In java, hash function is Object.hashcode(). Collision occurs when more than one value has the same hashed value.

Load Factor: tell how full a hash table is: =#of items/capacity

Handle collisions: 1. Chaining; 2. Linear Probing

2018.6.13: Hash table

|  |  |  |  |
| --- | --- | --- | --- |
| Hashmap | Hashtable | Arraylist | Vector |
| Not synchronized | Synchronized | Not synchronized | Synchronized |
| Allow 1 null key and null values | No null key or value |  |  |

Bucket sort: O(n).

- not in-place

- stability: depends on the sort algorithm

- To achieve O(n), one value per bucket

- Often use insertion sort (fast on small size data)

LinkedHashmap: hashtable and linkedlist

2018.6.14: Binary Search

Binary Search:

* Data must be sorted!
* Choose element in the middle of the array, compare against the search value
* If element = value, done
* If element > value, search the left half of the array, do from step2 again
* If element < value, search the right half of the array, do from step2 again
* O(logn)

2018.6.18: Tree

Trees:

* DS & ADT
* Every node has one and only one parent
* Root node: doesn’t has parent
* Every item in the tree is a node
* Leaf node has no children
* Subtree tree: the tree starts from the given node and all of its descendants
* No cyclic path, no cycles
* Depth of node: # of edges from the node to the root
* Height of the node: # of edges on the longest path from the node to a leaf
* Height of tree: # of edges on the longest path from the root to a leaf
* Root are level 0
* Level = depth

Binary tree:

* Every node has 0/1/2 children
* Children: left and right
* Complete tree: every level except the last level is completely filled. All of the interior nodes have 2 children, last level of all nodes have to be the left as much as possible
* Full binary tree: every node except leaf node has 2 children

Binary Search tree:

* Perform insertion, deletion & retrievals in O(logn) time
* Left child < its parent < right child
* Everything to the left of the root is less than the root. Everything to the right of the root is greater than root.value
* Can do a binary search
* Duplicate value: 1. Not allowed 2. Store left/right subtree 3. Counter++ for each node
* Minimum: follow left to leaf node
* Maximum: follow right to leaf node
* 25,20,15,10: 25

20

15

10

10

15

20

25

This is a linked list. O(n)

\*Self-balancing BST: red-black tree, AVL

2018.6.20: Tree Traversal

25

20 27

15 22 26 30

29 32

1. Level Traversal

visit nodes on each level;

25->20->27->15->22->26->30->29->32

2. Pre-order: root-left-right

visit the root of every subtree first (visit node before node’s children)

25->20->15->22->27->26->30->29->32

3. Post-order: left-right-root

visit the root of every subtree last;

15->22->20->26->29->32->30->27->25

4. In-order: left-root-right

visit left child, then root, then right child.

15->20->22->25->26->27->29->30->32

the data is sorted

2018.6.23: Tree delete

1. find the target in BST

a. current.value > key, go left

b. current.value < key, go right

c. current.value ==key, find

2. delete target from tree

a. current node doesn’t have left or right child: return null

b. current node has only left or right child: return the child.subtree

c. current node has both right and left child: find the Min/Max in the right/left subtree, return

2018.7.2 Heap

Heap

1. A complete binary tree
2. Must satisfy heap property
3. Max heap: Every parent >= its children
4. Min heap: Every parent <= its children
5. children are added at each level from left to right
6. The maximum or minimum value will always be at the root of the tree: advantage
7. Heapify: process of converting a Binary Tree into a heap –often has to be done after insertion /deletion

Store Binary Heaps as Arrays

1. root at Array[0]
2. traverse each level from left to right, so the left child of the root would go into Array[1], its right child would go into Array[2], etc.
3. For the node at Array[i]:
   1. left child = 2\*i+1, right child = 2\*i+2, parent = floor((i-1)/2)

Insert into Heap

1. Always add new items at end of the array
2. Then fix the heap(heapify)
3. Compare new item against its parent
4. If the item is greater than its parent, we swap it with its parent
5. rinse and repeat

Delete

1. Must choose a replacement value
2. Will take the rightmost value, so that the tree remains complete
3. Must heapify the heap
4. When replacement value is greater than parent, fix heap above. Otherwise, fix heap below.
5. Fix heap above: Same as insert, swap the replacement value with its parent
6. Fix heap below: Swap the replacement value with the largest of its to children
7. Rinse and repeat in both cases until the replacement value is in its correct position.

80 delete: 75

75 60 replacement: 67

68 55 40 52

67

80

67 60

68 55 40 52

80

68 60

67 55 40 52

2018.7.5 Priority Queue and Heap sort

Priority Queue:

* Use Max/Min heap as implementation
* The item with highest priority stays on the root
* Insert, Remove/Poll, Peek

Heap Sort

* The root has the largest value
* Swap the root with last element in the array
* Heapify the tree, but exclude the last node
* After heapify, second largest element is at the root
* Rinse and repeat