Contents

1 Data Structures **4**

1.1 Arrays . . . 4
1.1.1 Runtime . . . 4
1.1.2 Advantage . . . 4
1.1.3 Disadvantage . . . 4
1.1.4 Implementation in Java . . . 5
1.1.5 Typical Algorithm . . . 5
1.2 ArrayList . . . 5
1.2.1 Runtime . . . 5
1.2.2 Advantage . . . 5
1.2.3 Disadvantage . . . 5
1.2.4 Implementation in Java . . . 6
1.2.5 Typical Algorithm . . . 6
1.3 Single LinkedList . . . 6
1.3.1 Runtime . . . 6
1.3.2 Advantage . . . 6
1.3.3 Disadvantage . . . 6
1.3.4 Implementation in Java . . . 6
1.3.5 Typical Algorithm . . . 7
1.4 Double LinkedList . . . 7
1.5 Stack . . . 7
1.5.1 Runtime . . . 7
1.5.2 Advantage . . . 7
1.5.3 Disadvantage . . . 7
1.5.4 Implementation in Java . . . 7
1.5.5 Typical Algorithm . . . 7
1.6 Queue . . . 8
1.6.1 Runtime . . . 8
1.6.2 Advantage . . . 8
1.6.3 Disadvantage . . . 8
1.6.4 Implementation in Java . . . 8
1.6.5 Typical Algorithm . . . 8
1.7 PriorityQueue . . . 8
1.7.1 Runtime . . . 9
1.7.2 Advantage . . . 9
1.7.3 Disadvantage . . . 9
1.7.4 Implementation in Java . . . 9
1.7.5 Typical Algorithm . . . 10
1.8 SkipList . . . 10

1.8.1 Runtime . . . 10
1.8.2 Advantage . . . 10
1.8.3 Disadvantage . . . 11
1.8.4 Implementation in Java . . . 11
1.8.5 Typical Algorithm . . . 11
1.9 HashTable . . . 11
1.9.1 Runtime . . . 11
1.9.2 Advantage . . . 11
1.9.3 Disadvantage . . . 11
1.9.4 Implementation in Java . . . 12
1.9.5 Typical Algorithm . . . 12
1.10 Binary Search Tree . . . 12
1.10.1 Runtime . . . 13
1.10.2 Advantage . . . 13
1.10.3 Disadvantage . . . 13
1.10.4 Implementation in Java . . . 13
1.10.5 Typical Algorithm . . . 13
1.11 Cartesian Tree . . . 13
1.11.1 Runtime . . . 14
1.11.2 Advantage . . . 14
1.11.3 Disadvantage . . . 14
1.11.4 Implementation in Java . . . 14
1.11.5 Typical Algorithm . . . 14
1.12 B-Tree . . . 15
1.12.1 Runtime . . . 15
1.12.2 Advantage . . . 15
1.12.3 Disadvantage . . . 15
1.12.4 Implementation in Java . . . 15
1.12.5 Typical Algorithm . . . 15
1.13 Red-Black-Tree . . . 15
1.13.1 Runtime . . . 16
1.13.2 Advantage . . . 17
1.13.3 Disadvantage . . . 17
1.13.4 Implementation in Java . . . 17
1.13.5 Typical Algorithm . . . 17
1.14 Splay-Tree . . . 17
1.14.1 Runtime . . . 17
1.14.2 Advantage . . . 17
1.14.3 Disadvantage . . . 17
1.14.4 Implementation in Java . . . 17
1.14.5 Typical Algorithm . . . 17

1.15 AVL-Tree 17
1.15.1 Runtime 17
1.15.2 Advantage 17
1.15.3 Disadvantage 17
1.15.4 Implementation in Java 17
1.15.5 Typical Algorithm 17
1.16 KD-Tree 17
1.16.1 Runtime 17
1.16.2 Advantage 17
1.16.3 Disadvantage 17
1.16.4 Implementation in Java 17
1.16.5 Typical Algorithm 17

1 Data Structures

Access means that your are given the index and you have to return the element on this index. Search means, that your are given an element and you to find it in an Data Structure. Deletion means, that the element is deleted from the Data Structure.

1.1 Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Y

1.1.1 Runtime

Average: Access: $\theta(1)$ Search: $\theta(n)$ Insertion: $\theta(n)$ Deletion: $\theta(n)$

Worst Case: Access: $O(1)$ Search: $O(n)$ Insertion: $O(n)$ Deletion: $O(n)$

Deletion is in $O(n)$, because we have to copy the rest of the array to a new array. Or in an unsorted array, we swap the element to the end of the array and decrease the size of the array by one.

1.1.2 Advantage

- Fast Acces.
- It is used to represent multiple data items of same type by using only single name.
- More dimensional Data Structure.
- Less memory footprints than i.e. a List, because no Pointer to the next/previos element needed (in Java).

1.1.3 Disadvantage

- We must know in advance that how many elements are to be stored in array.
- Waste of memory, if more space allocated than needed.
- The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

1.1.4 Implementation in Java

java.util.Arrays provides useful methods for arrays such as binarySearch or sort.

1.1.5 Typical Algorithm

Used for Matrices and Vectors.

1.2 ArrayList

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

1.2.1 Runtime

Average: Access: $\theta(1)$ Search: $\theta(n)$ Insertion: $\theta(1)$ if the element is inserted at the end of the array Deletion: $\theta(n)$

Worst Case: Access: $O(1)$ Search: $O(n)$ Insertion: $O(1)$ if the element is inserted at the end of the array Deletion: $O(n)$

1.2.2 Advantage

- variable length
- Fast access
- Insertion in average in $\theta(1)$

1.2.3 Disadvantage

- Insertion may cause copying of the ArrayList.
- Fast access.
- Insertion in average in $\theta(1)$. Insertion/Deletion may cause copying of the ArrayList.

1.2.4 Implementation in Java

java.util.ArrayList

1.2.5 Typical Algorithm

1.3 Single LinkedList

1.3.1 Runtime

Average: Access: $\theta(n)$ Search: $\theta(n)$ Insertion: $\theta(1)$ if the element is inserted at the end of the array or a pointer to the previous element is given. Deletion: $\theta(n)$ if last element is deleted or a pointer to the element is given.

Worst Case: Access: $O(n)$ Search: $O(n)$ Insertion: $O(1)$ if the element is inserted at the end of the array or a pointer to the previous element is given. Deletion: $O(1)$ if the last element is deleted or a pointer to the element is given.

1.3.2 Advantage

- No limited size (except for the available memory)
- Fast insertion.
- No memory waste (except for pointers to the next element)

1.3.3 Disadvantage

- Slow Access to elements. (i.e. no binary Search possible)
- Another advantage of arrays in access time is special locality in memory. Arrays are defined as contiguous blocks of memory, and so any element will be physically near its neighbours.This greatly benifits from modern CPU caching methods.
- Extra memory for a pointer is required.

1.3.4 Implementation in Java

java.util.LinkedList

1.3.5 Typical Algorithm

1.4 Double LinkedList

Nearly the same as Single Linked List. But every element holds a pointer to the previos element. So backward traversion is possible and more memory is needed.

1.5 Stack

In computer science, a stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed. The order in which elements come off a stack gives rise to its alternative name, LIFO (for last in, first out). Additionally, a peek operation may give access to the top without modifying the stack.

1.5.1 Runtime

Average: Access: $\theta(n)$ Search: $\theta(n)$ Insertion: $\theta(1)$ Deletion: $\theta(n)$
Worst Case: Access: $O(n)$ Search: $O(n)$ Insertion: $O(1)$ Deletion: $O(1)$

1.5.2 Advantage

- No limited size (except for the available memory)
- Fast insertion.
- No memory waste

1.5.3 Disadvantage

- Slow Access to elements. (i.e. no binary Search possible)

1.5.4 Implementation in Java

java.util.Stack
public class $Stack < E >$ extends $Vector < E >$

1.5.5 Typical Algorithm

- Polish Notation
- Backtracking

- Java Virtual Machine

1.6 Queue

In computer science, a queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as enqueue, and removal of entities from the front terminal position, known as dequeue. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed.

1.6.1 Runtime

Average: Access: $\theta(n)$ Search: $\theta(n)$ Insertion: $\theta(1)$ Deletion: $\theta(n)$
Worst Case: Access: $O(n)$ Search: $O(n)$ Insertion: $O(1)$ Deletion: $O(1)$

1.6.2 Advantage

- No limited size (except for the available memory)
- Fast insertion.
- No memory waste

1.6.3 Disadvantage

- Slow Access to elements. (i.e. no binary Search possible)

1.6.4 Implementation in Java

Interface $Queue < E >$ java.util All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

1.6.5 Typical Algorithm

1.7 PriorityQueue

In computer science, a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high

priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.
Note: Implementation is often done with heaps, a data structure that uses arrays (fixed size or dynamic array). java.util.PriorityQueue uses also an array, that grows by 50 percent for huge arrays, 100 percent for small arrays)

1.7.1 Runtime

Binary PriorityQueue:
Average: Access: $\theta(logn)$ Search: $\theta(logn)$ Insertion: $\theta(logn)$ Deletion of min element: $\theta(logn)$

Worst Case: Access: $O(logn)$ Search: $O(logn)$ Insertion: $O(logn)$ Deletion of min element: $O(logn)$

Fibonacci PriorityQueue:
Average: Access: $\theta(logn)$ Search: $\theta(logn)$ Insertion: $\theta(1)$ Deletion of min element: $\theta(logn)$

Worst Case: Access: $O(logn)$ Search: $O(logn)$ Insertion: $O(1)$ Deletion of min element: $O(logn)$

Fibonacci Heap is faster in InsertMin $\theta(1)$ and in decrease-key $\theta(1)$ versus$O(logn)$ for Binary Priority Queue.

1.7.2 Advantage

- Elements are sorted
- Inserting new elements is faster than in an sorted array or sorted List
- Sorting takes time $\theta(n)$ with a fibonacci Priority Queue.

1.7.3 Disadvantage

- Slower Insertion, Deletion, Access

1.7.4 Implementation in Java

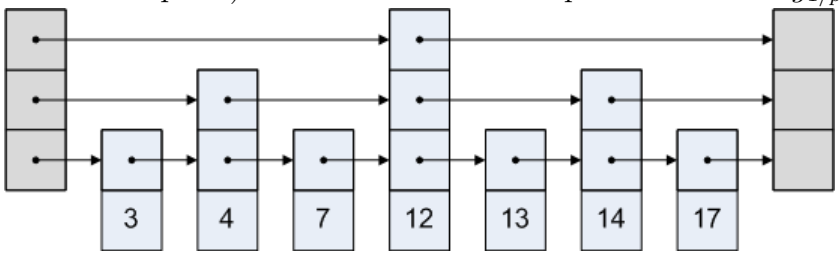
java.util.PriorityQueue (Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).)

1.7.5 Typical Algorithm

- Dijkstra's algorithm
- Best-first search algorithms
- Prim's algorithm for minimum spanning tree
- Bandwidth management

1.8 SkipList

SkipList is a ordered list.
A skip list is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly used values for p are $\frac{1}{2}$ or $\frac{1}{4}$). On average, each element appears in $\frac{1}{1-p}$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists. The skip list contains $log_{1/p}n$ lists.



Decide how many lists will a node be part of. With probabily of $\frac{1}{2}$ node will be part of the lowest level only, with $\frac{1}{4}$ node will be part of the two lowest list and so on.
See http://igoro.com/archive/skip-lists-are-fascinating/ for more details.

1.8.1 Runtime

Average: Access: $\theta(logn)$ Search: $\theta(logn)$ Insertion: $\theta(logn)$ Deletion: $\theta(logn)$
Worst Case: Access: $O(n)$ Search: $O(n)$ Insertion: $O(n)$ Deletion: $O(n)$

1.8.2 Advantage

- Faster search and insertion of elements (as fast as balanced AVL-Trees)
- Elements are sorted
- Easy to implement

1.8.3 Disadvantage

- Space Complexity $O(n * log(n))$

1.8.4 Implementation in Java

There is no implementation in java. But java.util.concurrent.ConcurrentSkipListSet¡E¿ and java.util.concurrent.ConcurrentSkipListMap¡K,V¿ are using the same logic.

1.8.5 Typical Algorithm

1.9 HashTable

In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash collisions where the hash function generates the same index for more than one key. Such collisions must be accommodated in some way.

1.9.1 Runtime

Average: Access: N/A
Search: $\theta(1)$ Insertion: $\theta(1)$ Deletion $\theta(1)$
Worst Case: Access: N/A
Search: $O(n)$ Insertion: $O(n)$ Deletion of min element: $O(n)$

1.9.2 Advantage

- Fast operations
- If the number of entries can predicted in advance, it is particulary efficient (no need to resize buckets)

1.9.3 Disadvantage

- Cost of a good hash function can slow the programm (if number of entries is small)
- Not ordered

- Slow for applications like spell-checking
- Dynammic Resizing of the hash table
- Poor locality of reference, data to be accessed is distributed seemingly at random in memory
- Inefficient when there are many collisions

1.9.4 Implementation in Java

java.util.HashSet¡E¿ (Implemantation in java uses an array that contains Entry. Entry has an pointer to its next element)
java.util.HashMap¡K,V¿
LinkedHashSet¡E¿ (Hash table and linked list implementation of the Set interface. An insertion-ordered Set implementation that runs nearly as fast as HashSet.)
java.util.LinkedHashMap¡K,V¿ (Hash table and linked list implementation of the Map interface. An insertion-ordered Map implementation that runs nearly as fast as HashMap. Also useful for building caches (see removeEldestEntry(Map.Entry))).
java.util.WeakHashMap¡K,V¿ (An implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows key-value pairs to be garbage-collected when the key is no longer referenced outside of the WeakHashMap. This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.)
ConcurrentHashMap (A highly concurrent, high-performance ConcurrentMap implementation based on a hash table. This implementation never blocks when performing retrievals and allows the client to select the concurrency level for updates. It is intended as a drop-in replacement for Hashtable: in addition to implementing ConcurrentMap, it supports all of the "legacy" methods peculiar to Hashtable.)

1.9.5 Typical Algorithm

1.10 Binary Search Tree

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies

the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and not greater than any key in the right sub-tree.[1]:287 (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.)

1.10.1 Runtime

This chapter examines basic Binary Search Trees (that are not full)
Average: Access: $\theta(logn)$ Search: $\theta(logn)$ Insertion: $\theta(logn)$ Deletion $\theta(logn)$
Average means, that the inserted elements are sorted randomly. If they are sorted in decreasing Order all this operations take $O(n)$ time.x

Worst Case: Access: $O(n)$ Search: $O(n)$ Insertion: $O(n)$ Deletion: $O(n)$

1.10.2 Advantage

- Sorting and Search can be very fast.
- Easy to implement.
- Useful for other data structure like Priority Queues.

1.10.3 Disadvantage

- Can be slow if order of elements is not randomly.

1.10.4 Implementation in Java

There exists no implementation of basic Binary Search Tree in Java api.

1.10.5 Typical Algorithm

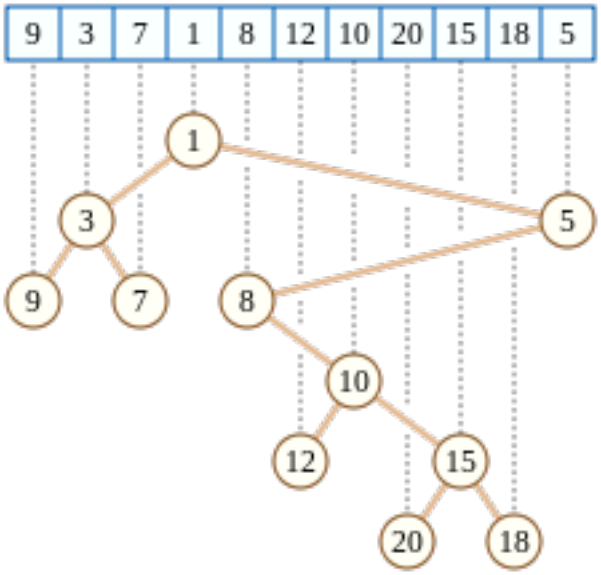
Search and Sort.

1.11 Cartesian Tree

A Cartesian tree is a tree data structure created from a set of data that obeys the following structural invariants:

- The tree obeys in the min (or max) heap property – each node is less (or greater) than its children.

- An inorder traversal of the nodes yields the values in the same order in which they appear in the initial sequence.



1.11.1 Runtime

Average: Access: N/A
Search: $\theta(logn)$ Insertion: $\theta(logn)$ Deletion $\theta(logn)$
Average means, that the inserted elements are sorted randomly. If they are sorted in decreasing Order all this operations take $O(n)$ time.x
Worst Case: Access: N/A
Search: $O(n)$ Insertion: $O(n)$ Deletion: $O(n)$

1.11.2 Advantage

- Elements are sorted like they were inserted
- Min value between to node is fast to determine

1.11.3 Disadvantage

- No Access to a specific element.

1.11.4 Implementation in Java

1.11.5 Typical Algorithm

- Cartesian Tree Sorting

- Used for Treap data structure

1.12 B-Tree

nochhmal durchlesen, sehr interessant im Algorithms Buch

1.12.1 Runtime

1.12.2 Advantage

1.12.3 Disadvantage

1.12.4 Implementation in Java

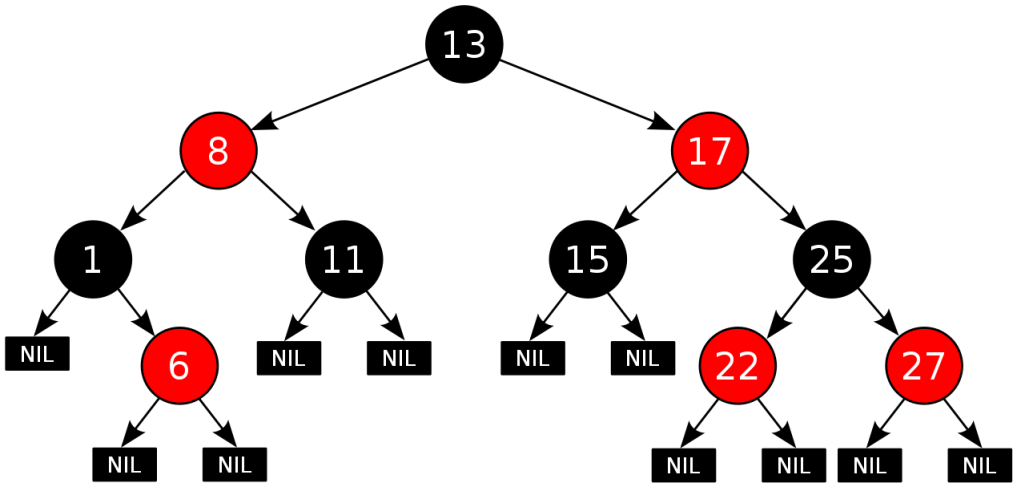
1.12.5 Typical Algorithm

1.13 Red-Black-Tree

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions. There are 5 properties that must be satisfied by a Red-Black-Tree:

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. Some definitions: the number of black nodes from the root to a node is the node's black depth; the uniform number of black nodes in all paths from root to the leaves is called the black-height of the red–black tree.

No path from the root to a leaf is more than twice as long as any other path from the root to a leaf.



1.13.1 Runtime

Average: Access: $O(logn)$ Search: $O(logn)$ Insertion: $O(logn)$ Deletion $O(logn)$
Average means, that the inserted elements are sorted randomly. If they are sorted in decreasing Order all this operations take $O(n)$ time.x

Worst Case: Access: $O(logn)$ Search: $O(logn)$ Insertion: $O(logn)$ Deletion: $O(logn)$

1.13.2 Advantage

1.13.3 Disadvantage

1.13.4 Implementation in Java

1.13.5 Typical Algorithm

1.14 Splay-Tree

1.14.1 Runtime

1.14.2 Advantage

1.14.3 Disadvantage

1.14.4 Implementation in Java

1.14.5 Typical Algorithm

1.15 AVL-Tree

1.15.1 Runtime

1.15.2 Advantage

1.15.3 Disadvantage

1.15.4 Implementation in Java

1.15.5 Typical Algorithm

1.16 KD-Tree

1.16.1 Runtime

1.16.2 Advantage

1.16.3 Disadvantage

1.16.4 Implementation in Java

1.16.5 Typical Algorithm