What are Data Structures?

Data Structure[1]

A way in which data are stored for efficient search and retrieval. Different data structures are suited for different problems.

- Data type $\neq$ data structure
- `java.util.HashSet` vs. hash table
- `array` vs. `array`

[1] Encyclopædia Britannica

Array

Definition

- An indexed set of related elements.[2]
- An assemblage of items that are randomly accessible by integers, the index.[3]

- Example: linear array

[2] Oxford Dictionary
[3] National Institute of Standards & Technology

Array

Operations

- `get`
- `get` $O(1)$
- `set`
- `set` $O(1)$
- `indexOf`
- `indexOf` $O(n)$

`get(1)` `set(2)` `indexOf(object)`

Linked List

Definition

A linked list is a data structure in which the objects are arranged in a linear order. Unlike arrays in which the linear order is determined by indices, the order is determined by a pointer in each object.[4]

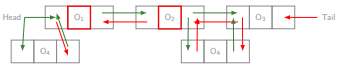
- Different types: singly, doubly, multiply, circular, ...
- Example: doubly linked list

[4] Introduction to Algorithms By Cormen, Leierson, Rivest & Stein

Linked List

Operations

- `add/remove first/last`
- `add/remove first/last` $O(1)$
- `get/insertAt`
- `get/insertAt` $O(n)$
- `indexOf`
- `indexOf` $O(n)$



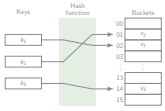
`addFirst(`O_4`)` `insertAt(2)` `indexOf(`O_2`)`

Hash Table

Definition

A dictionary in which keys are mapped to array positions by hash functions.[5]

- Hash functions: determinism, uniformity, defined range, data normalisation, non-invertible, perfect, ...
- Collisions resolution: chaining, open addressing, ...
- Example:

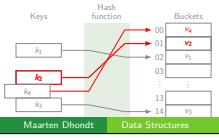


[5] National Institute of Standards & Technology

Hash Table

Operations

- `put`
- `put` $O(1)$ / $O(n)$
- `remove`
- `remove` $O(1)$ / $O(n)$
- `get`
- `get` $O(1)$ / $O(n)$

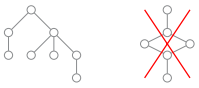


Tree

Definition

A data structure made up of nodes or vertices and edges without having any cycle. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy.

- Depth, binary, (nearly) complete, ...
- Example:



Binary Heap

Definition (Heap)

A complete tree where every node has a key more extreme (greater or less) than or equal to the key of its parent.[6]

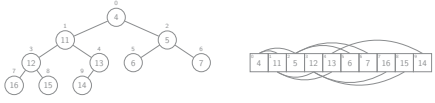
Definition (Binary Heap)

A binary heap data structure is an array object that we can view as a nearly complete binary tree that satisfies the min-heap or max-heap property.[7]

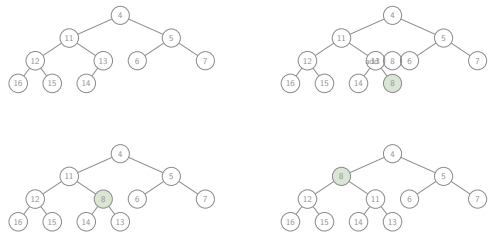
[6] National Institute of Standards & Technology
[7] Introduction to Algorithms By Cormen, Leierson, Rivest & Stein

Binary Min-Heap

- `Parent(n)` $\lfloor \frac{n-1}{2} \rfloor$
- `Left(n)` $2n+1$
- `Right(n)` $2(n+1)$

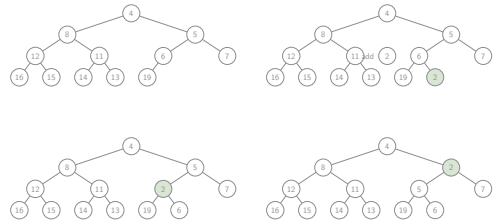


Binary Min-Heap

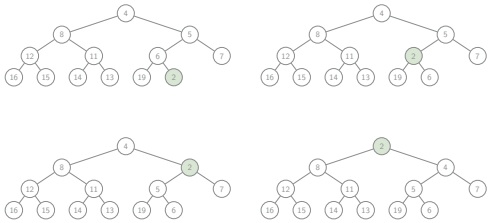


Binary Min-Heap

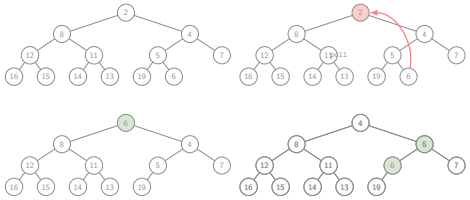
Binary Min-Heap



Binary Min-Heap



Binary Min-Heap



Binary Min-Heap

Operations

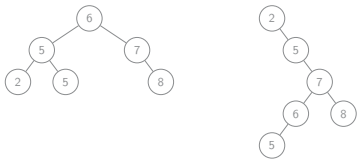
- `insert`
- `insert` $O(\log n)$
- `removeAt`
- `removeAt` $O(\log n)$
- `peek`
- `peek` $O(1)$
- `poll`
- `poll` $O(\log n)$

- Heapsort
- Frequently used in Priority Queues

Binary Search Tree

Definition

A binary tree in which the left child $\leq$ the parent and the right child $\geq$ the parent.



Binary Search Tree

Operations

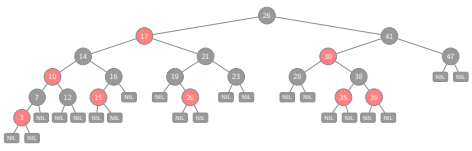
- `insert` $O(\log n)$ / $O(n)$
- `delete` $O(\log n)$ / $O(n)$
- `search` $O(\log n)$ / $O(n)$

Red-Black Tree

- Binary search tree
- Approximately balanced
- NIL leaves
- Red-black properties
  - Every node is either red or black
  - Root is black
  - Every leaf is black
  - If a node is red, its children are black
  - For each node, all paths to its descendant leaves contain the same number of black nodes

Red-Black Tree

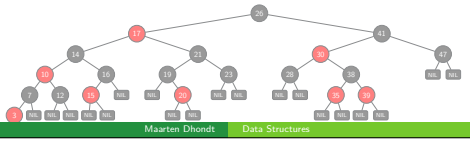
- Node is either red or black
- Root is black
- Every leaf is black
- If red, children are black
- $\forall$ node: all paths to its leaves have the same number of black nodes



Red-Black Tree

Operations

- `insert`
- `insert` $O(\log n)$
- `delete`
- `delete` $O(\log n)$
- `search`
- `search` $O(\log n)$



Java Collection API

List Interface

	Impl	`add`	`remove`	`contains`	`get`
LinkedList	linked list	$O(1)$	$O(1)$	$O(n)$	$O(n)$
ArrayList	array	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Vector	array	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Stack	array	$O(1)$	$O(n)$	$O(n)$	$O(1)$

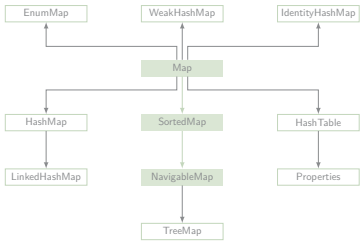
Set Interface

	Impl	`add`	`contains`	`next`
HashSet	hash table	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashSet	hash table linked list	$O(1)$	$O(1)$	$O(1)$
TreeSet	red-black tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

Queue Interface

	Impl	`offer`	`peak`	`poll`
PriorityQueue	binary heap	$O(\log n)$	$O(1)$	$O(\log n)$
ArrayDeque	array	$O(1)$	$O(1)$	$O(1)$
LinkedList	linked list	$O(1)$	$O(1)$	$O(1)$

Java Map API



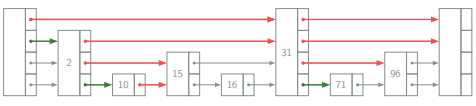
Map Interface

	Impl	`get`	`containsKey`	`next`
HashTable	hash table	$O(1)$	$O(1)$	$O(h/n)$
Properties	hash table	$O(1)$	$O(1)$	$O(h/n)$
HashMap	hash table	$O(1)$	$O(1)$	$O(h/n)$
LinkedHashMap	hash table linked list	$O(1)$	$O(1)$	$O(1)$
TreeMap	red-black tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
IdendityHashMap	array	$O(1)$	$O(1)$	$O(h/n)$
WeekHashMap	hash table	$O(1)$	$O(1)$	$O(h/n)$
EnumMap	array	$O(1)$	$O(1)$	$O(1)$

Skip list

- Balanced binary tree alternative
  - `insert`, `delete` & `search` in $O(\log n)$
- Probabilistic balancing rather than strictly enforced balancing
- Insertion and deletion $\rightarrow$ simpler and faster
- Linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one
  - Hierarchy has $\approx \log n$ levels

Skip list

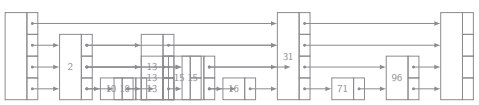


`find 71` `find 12`

Skip list

Insertion

- `insert at level=1`
- `while (coinflip() == HEADS)`
  `insert at ++level`



`insert 13` `insert 13` — Coinflips: H
`insert 13` — Coinflips: HH `insert 13` — Coinflips: HHT

Skip list

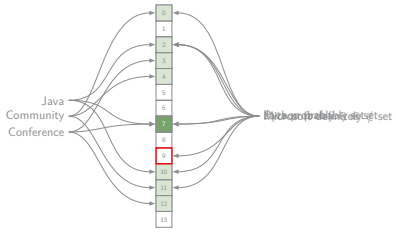
Operations

- `insert`
- `insert` $O(\log n)$ / $O(n)$
- `delete`
- `delete` $O(\log n)$ / $O(n)$
- `search`
- `search` $O(\log n)$ / $O(n)$

Bloom Filter

- Data structure to indicate if an element is contained in a set
- Bit array → space efficient
- Probabilistic
  - False positives possible
  - False negatives impossible
- Elements can be added to the set, but not removed

Bloom Filter



Bloom filter

Math

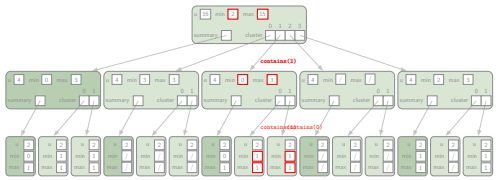
- m: array length
- k: # hash functions
- n: # of inserted elements
- p: false positive probability
- # bits needed: $m = \frac{-n \ln p}{(\ln 2)^2}$
- Optimal # of hash functions: $k = \frac{m}{n} \ln(2)$

- Supports unions and intersections
- Fits arbitrarily # of elements (false positive rate increases)
- Common application: caching

van Emde Boas Tree

- Dynamic ordered set/map
  - `insert`, `delete`, `contains`, `minimum`, `maximum`, `successor`, `predecessor` operations in $O(\log \log U)$
- Integer keys
  - No duplicates
- $O(U)$ space → only for large collections

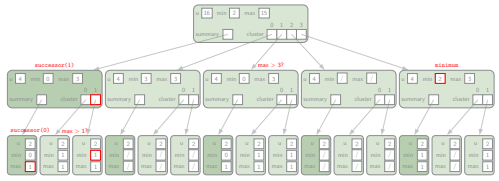
van Emde Boas Tree



`{2,3,4,5,7,14,15}`

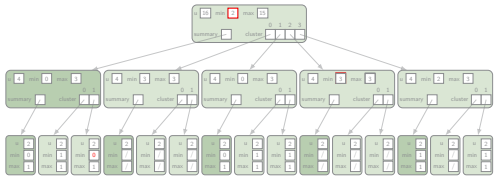
`minimum maximum contains(5) contains(6)`

van Emde Boas Tree



`{2,3,4,5,7,14,15}`
`successor(7)`

van Emde Boas Tree



`{2,3,4,5,7,14,15}`
`insert(11)`

van Emde Boas Tree

Operations

- `min`, `max`, `isEmpty` $O(1)$
- `insert`, `delete`, `contains`, `succ`, `pred` $O(\log \log U)$

- $O(U)$ space
  - X-fast trie: $O(n \log U)$ space
  - Y-fast trie: $O(n)$ space
- $U = 2^k, k \in \mathbb{N}$