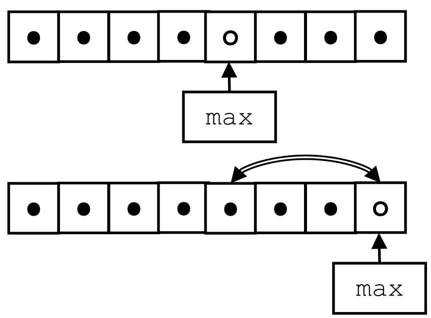
Copyright @Snoide

Sort

Selection Sort

> Best: O(n^2)
> Av.: O(n^2)
> Worst: O(n^2)
> non-stable

repeat:

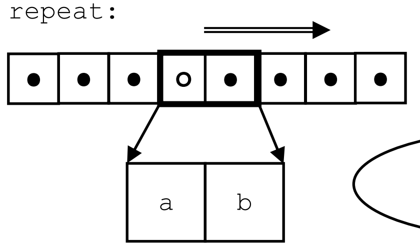


Bubble Sort

> Best: O(n^2)
> Av.: O(n^2)
> Worst: O(n^2)

If improved by using isSorted:
> Best: O(n)
> Av.: O(n^2)
> Worst: O(n^2)

repeat:

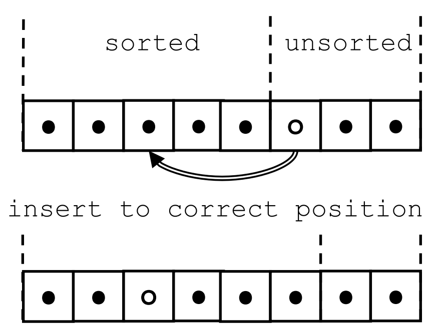


if a < b then swap

Insertion Sort

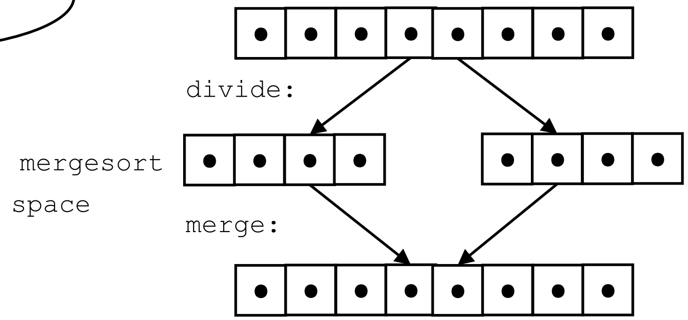
> Best: O(n)
> Av.: O(n^2)
> Worst: O(n^2)

repeat:



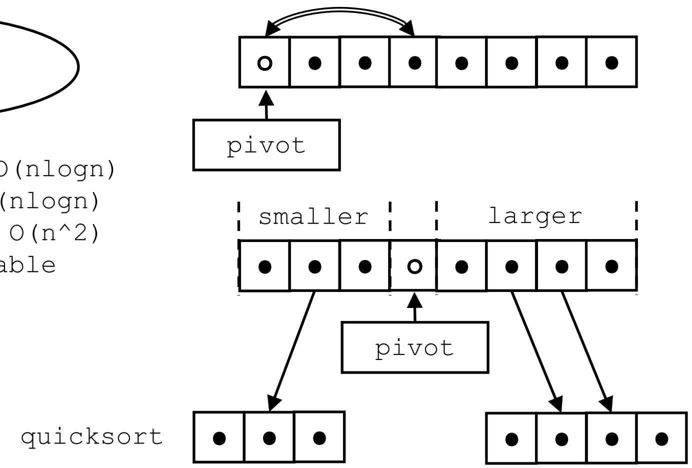
Merge Sort

> Best: O(nlogn)
> Av.: O(nlogn)
> Worst: O(nlogn)
> additional O(n) space
(not in-place)



Quick Sort

> Best: O(nlogn)
> Av.: O(nlogn)
> Worst: O(n^2)
> non-stable

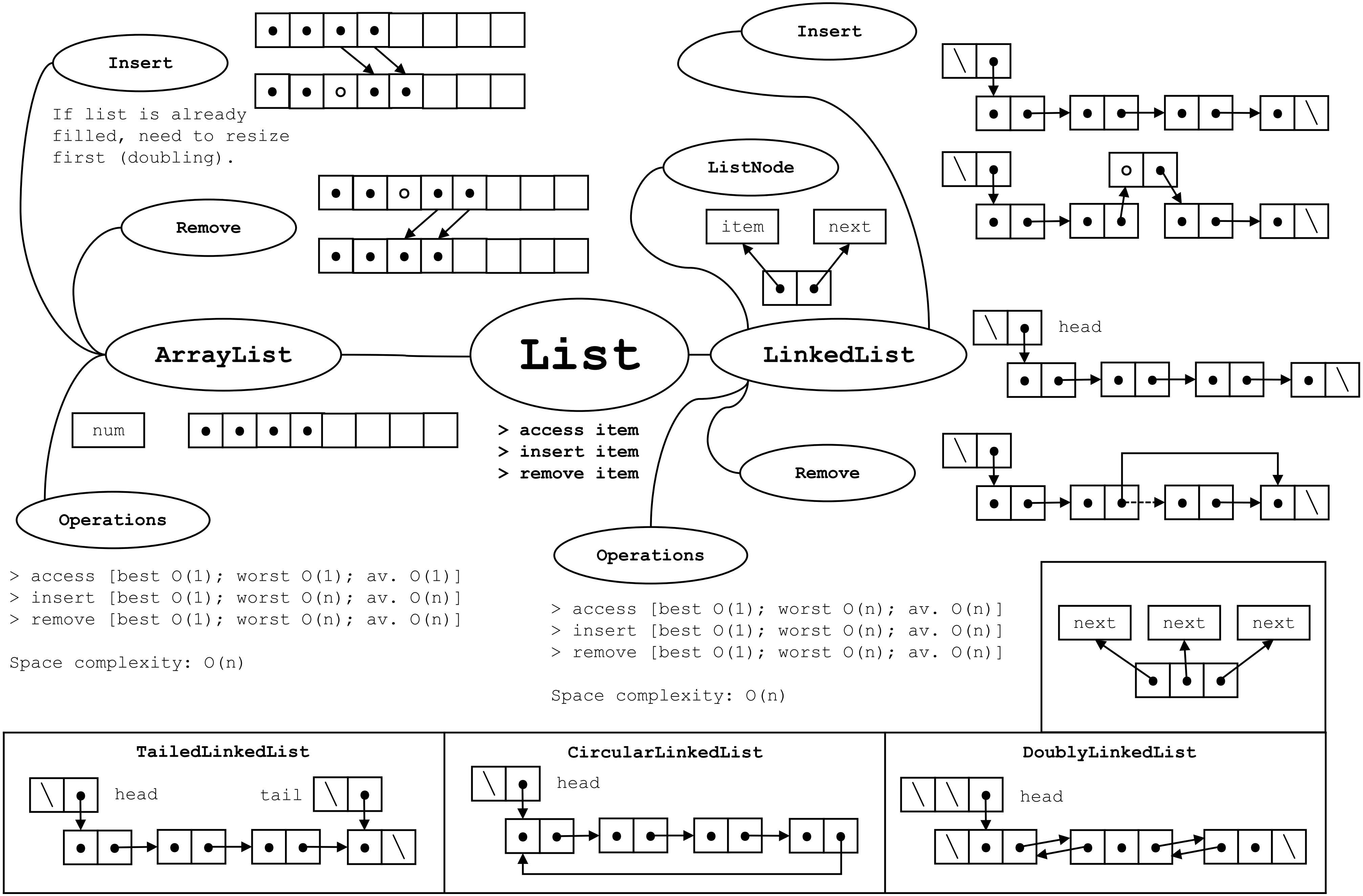


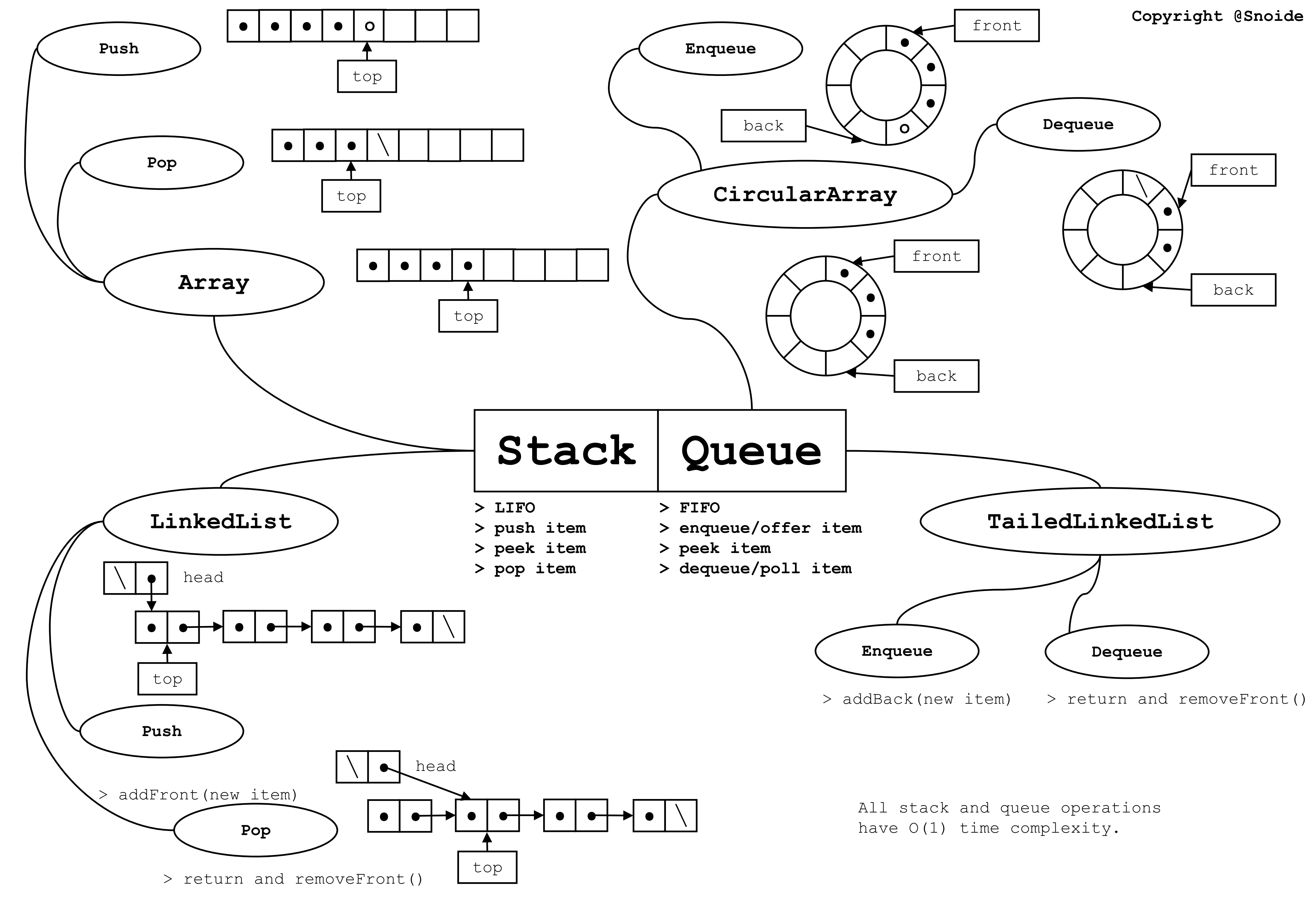
Radix Sort

> Best: O(nd)
> Av.: O(nd)
> Worst: O(nd)
> not in-place

> non-comparison
For n-digit objects:
1. Group by n-th digit.
2. Ungroup.
3. Group by n-th digit.
4. Ungroup
…
2n - 1: Group by first digit.
2n. Ungroup.

Copyright @Snoide




Copyright @Snoide
Stack
Queue
> LIFO
> push item
> peek item
> pop item
> FIFO
> enqueue/offer item
> peek item
> dequeue/poll item
Array
Push
Pop
top
top
top
CircularArray
Enqueue
Dequeue
front
back
front
back
front
back
LinkedList
head
top
Push
> addFront(new item)
Pop
> return and removeFront()
head
top
TailedLinkedList
Enqueue
Dequeue
> addBack(new item)
> return and removeFront()
All stack and queue operations
have O(1) time complexity.

Copyright @Snoide

HashTable

> retrieve value [av. O(1)]
>> return a[h(key)]
> insert value [av. O(1)]
>> a[h(key)] = value
> delete value [av. O(1)]
>> a[h(key)] = null

Hash Function

> fast
> scatter keys evenly
> less collisions (1-1 if perfect)
> less slots required

Uniform Hash Functions

> distribute keys evenly

Division Method
$hash(k) = k \% m$
Multiplication Method
$hash(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$

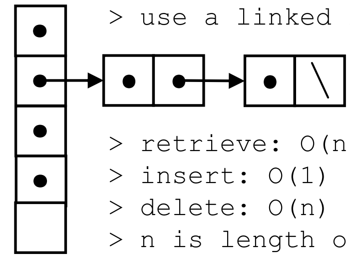
HashMap

> construct:
HashMap(*capacity, *loadFactor)
> put(K key, V value)
> get(K key)
> containsKey()
> containsValue()
> clear()

Collision Resolution

> minimise clustering
> always find an empty slot
> no secondary clustering (give different probe sequence when two initial probes are the same)
> fast

Separate Chaining



> use a linked list to store collided keys

> retrieve: O(n)
> insert: O(1)
> delete: O(n)
> n is length of linked lists

Probing

> looks for the next empty space (where)
> delete: mark as deleted

linear probing	modified linear probing	quadratic probing
$hash(k)$	$hash(k)$	$hash(k)$
$[hash(k) + 1] \% m$	$[hash(k) + 1d] \% m$	$[hash(k) + 1] \% m$
$[hash(k) + 2] \% m$	$[hash(k) + 2d] \% m$	$[hash(k) + 4] \% m$
$[hash(k) + 3] \% m$	$[hash(k) + 3d] \% m$	$[hash(k) + 9] \% m$
…	…	…

> primary clustering

> secondary clustering (two keys have same hash value)

Double hashing
$hash(k)$
$[hash(k) + 1 * hash_2(k)] \% m$
$[hash(k) + 2 * hash_2\,(k)] \% m$
$[hash(k) + 3 * hash_2\,(k)] \% m$
…

> secondary hash function (!= 0)

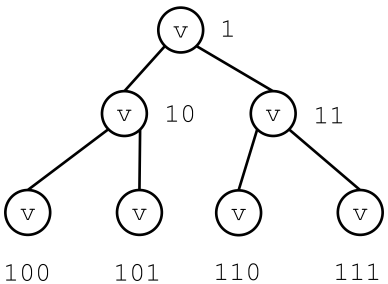
Copyright @Snoide

Heap

> insert value
> extract max

BinaryTree

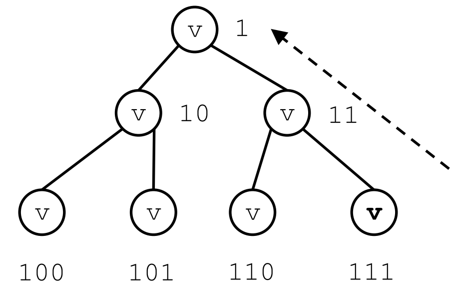
> Each node has two attributes:
value and **index**



> parent(i) = i / 2
> left(i) = 2 * i
> right(i) = 2 * i + 1

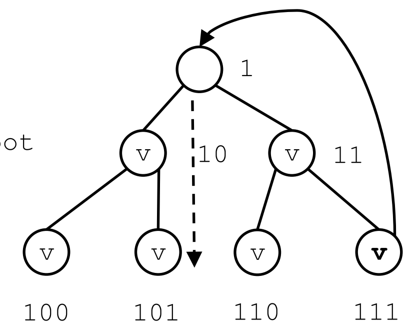
Insert

> insert at the back
[O(1)]
> shift up if required
[O(log n)]



ExtractMax

> assign the back to the root
[O(1)]
> shift down if required
[O(log n)]



Create

> Method 1: insert each value to an empty heap
[O(nlog n)]
> Method 2: copy the content and shift down from parent of the lowest nodes
[O(n)]

HeapSort

> call ExtractMax n times
O(nlog n)

Copyright @Snoide

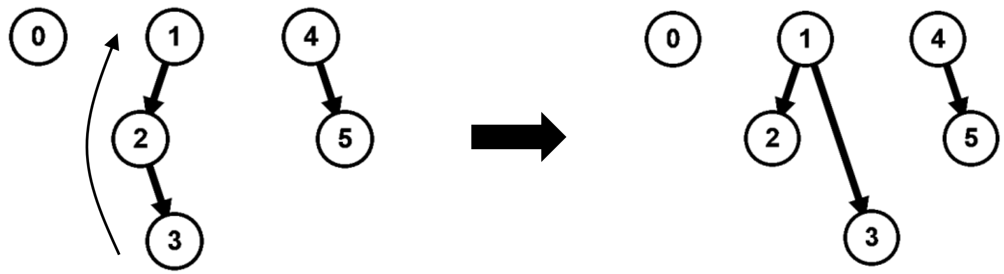
UFDS

Constructor

> rep[i] = i
> rank[i] = 0

FindSet

> find until rep[i] == i
> path-compression: rep[i] = findSet(rep[i])

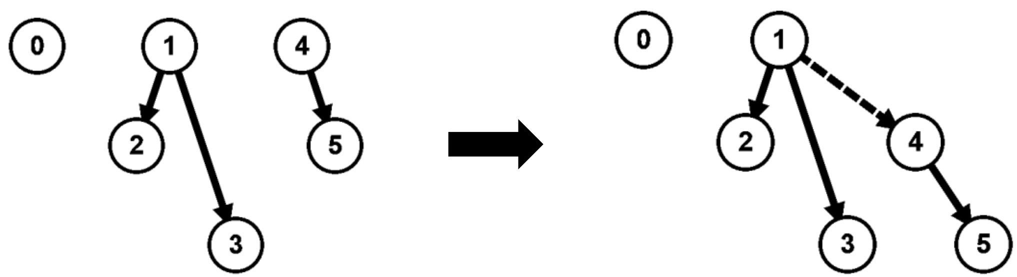


IsSameSet

> check whether rep[a] == rep[b]

UnionSet

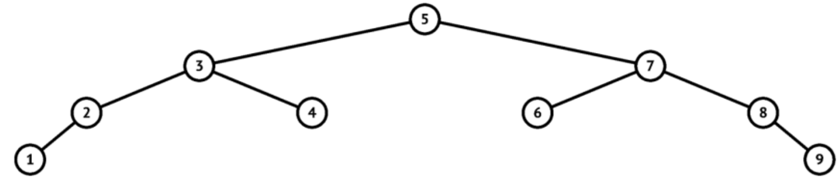
> if rank[a] > rank[b], then rep[b] = a
> If rank[a] == rank[b], then rep[b] = a and rank[a]++



Copyright @Snoide

Binary Search Tree

> item
> parent
> left
> right

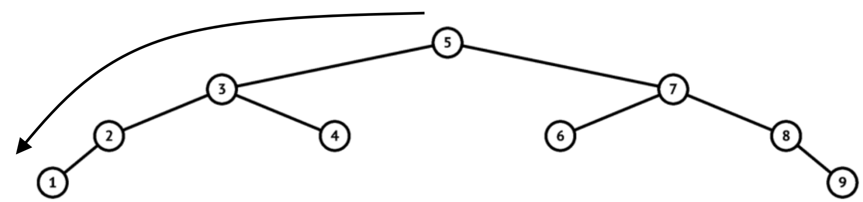


Search

> if v < node, search(left, i)
> if v > node, search(right, i)
> O(h)

FindMax/FindMin

> O(h)



Insert

> similar to search
> O(h)

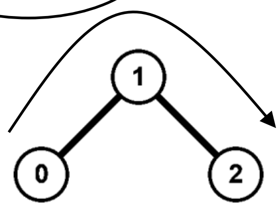
Delete

> similar to search
> O(h)

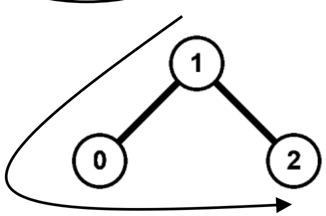
Predecessor/ Successor

> leftmost child of right child (successor)
> O(h)

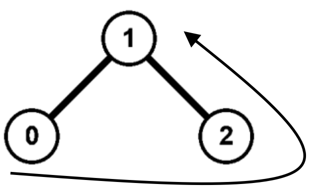
In-order



Pre-order



Post-order



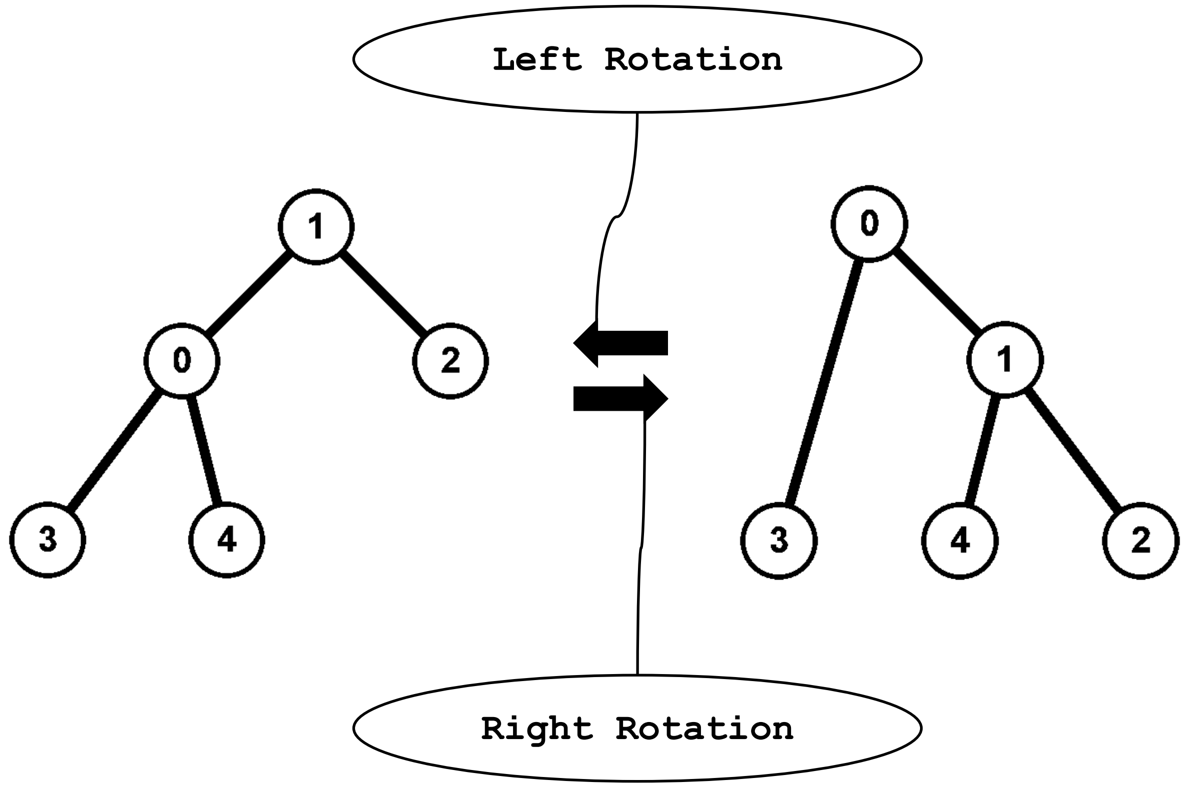
Copyright @Snoide

AVL Tree

```
> item
> parent
> left
> right
> height
> size
```

Rebalance

```
> if bf == 2 and bf(left) == 0/1,
  then right rotate
> if bf == 2 and bf(left) == -1,
  then left rotate left and right rotate
> if bf == -2 and bf(right) == 0/-1,
  then left rotate
> if bf == -2 and bf(right) == 1,
  then right rotate right and left rotate
```



Copyright @Snoide

Graph

Adjacency Matrix

> 2D array AdjMatrix
> Space complexity: $O(V^2)$

Pros	Cons
> Check edge existence in O(1) > Good for dense graph/Floyd Warshall's	> O(V) to enumerate neighbours > $O(V^2)$ space

Adjacency List

> Array of list which stores neighbours
> Space complexity: O(V + E)

Pros	Cons
> O(k) to enumerate k neighbours > Good for sparse graph/Dijkstra's/DFS/BFS > O(V+E) space	> O(k) to check existence

Edge List

> Array of all edges
> Space complexity: O(E)

Pros	Cons
> Good for Kruskal's/Bellman Ford	> Hard to find vertices

Copyright @Snoide

Graph Traversal

BFS

> Queue
> for SSSP

```
for all v in V:
    visited[v] = 0 // avoid cycle
    p[v] = -1 // parent
Q = {s} // start
visited[s] = 1
```

O(V)

```
while Q is not empty:
    u = Q.dequeue()
    for all v adjacent to u:
        visited(v) = true
        p[v] = u
        Q.enqueue(v)
```

O(V + E)

DFS

> Stack
> for SCC

```
for all v in V:
    visited[v] = 0 // avoid cycle
    p[v] = -1 // parent
DFS(s)
```

O(V)

```
def DFS(u):
    visited[u] = 1
    for all v adjacent to u:
        if v is unvisited:
            p[v] = u
            DFS(v)
```

O(E)

Applications

> Reachability test
> Shortest path in unweighted graph
> Counting component
> Topological sort (Kahn's/DFS)
> Counting SCCs (Kosaraju's)

Minimum Spanning Tree

Prim's

> Start from any vertex
> PriorityQueue
> O(ElogV)

```
1. T = {s} // start

2. enqueue edge connected to s into PQ

3. while PQ is not empty:
       e = pq.dequeue()
       if e.v not in pq:
           T.append(v)
           enqueue edge connected to v into PQ

4. T is an MST
```

Kruskal's

> Start from shortest edge
> Sorted EdgeList and UFDS
> O(ElogV)

```
1. Sort E

2. T = {}

3. while there are unprocessed edges in E:
       pick the smallest edge e
       if adding e to T does not form a cycle:
           add e to T

4. T is an MST
```

Single-Source Shortest Pathway

Unweighted

> Modified BFS

Weighted

> Bellman Ford O(VE)

```
initSSSP(s)

for i = 1 to |V| - 1:
    for each edge(u, v) in E:
        relax(u, v, w(u, v))
```

Tree

> Every path is a shortest path.
> DFS/BFS O(V)

No Negative Weight Edge

> Dijkstra's O((V+E) log V)

```
PQ.enqueue((0, s))
for all other vertices v:
    PQ.enqueue((INF, v))
```

```
while PQ is not empty:
    u = PQ.dequeue()
    Solved.append(u)
    relax all outgoing edges of u
```

```
def relax(E(u, v, w)):
    find v in PQ and update d[v]
```

No Negative Weight Cycle

> Modified Dijkstra's O((V+E) log V)

```
initSSSP(s)

PQ.enqueue((0, s))
while PQ is not empty:
    (d, u) = PQ.dequeue()
    if d == D[u]:
        for each v adjacent to u:
            relax(u, v)
            PQ.enqueue((D[v], v))
```

Directed & Acyclic

> one-pass Bellman Ford O(E)

All-Pairs Shortest Pathway

Floyd Warshall's O(V³)

```
for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            D[i][j] = min(D[i][j], D[i][k] + D[k][j])
        }
    }
}
```