SEARCHING	Algorithm	Avg	Worst	Space
V verticies	DFS	-	O(E+V)	O(V)
E edges	BFS	-	O(E+V)	O(V)
Sorted Array	Binary Search	O(log n)	O(log n)	O(1)
	Bruteforce	O(n)	O(n)	O(1)
	Dijkstra-MinHeap	O(V+E log V)	O(V+E log V)	O(V)
	Dijkstra-Unsorted Array	$O(V^2)$	$O(V^2)$	O(V)
Unsorted list	Quickselect	O(n)	$O(n^2)$	O(?)

SORTING	Algorithm	Best	Avg	Worse
	Mergesort	O(n·log n)	O(n·log n)	O(n·log n)
	Insertion Sort	O(n)	$\Theta(n^2)$	$\Theta(n^2)$
	Heapsort	O(n·log n)	O(n·log n)	O(n·log n)
	QuickSort	O(n·log n)	O(n·log n)	$O(n^2)$

STRUCTURES	Avg Index	Avg Search	Avg Insert	Avg Delete	W index	W search	W insert	W delete	Space
Basic Array	O(1)	O(n)	-		O(1)	O(n)	-	-	n
Dynamic Array	O(1)	O(n)	$*^{1}$	$*^{1}$	O(1)	O(n)	$*^{1}$	$*^{1}$	
Singly Linked	O(n)	O(n)	$*^{2}$	$*^{2}$	O(n)	O(n)	$*^{2}$	$*^{2}$	
Doubly Linked	O(n)	O(n)	$*^{2}$	$*^{2}$	O(n)	O(n)	$*^{2}$	$*^{2}$	
Hashtable	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	
Binary ST	O(log n)	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)	O(n)	
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	
Red Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	
B-Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	

O(1) w/ perfect hashing.

$*^{1}$: Insert/Delete at beginning O(n). Insert/Delete at end O(1) ammortized.

$*^{2}$: Insert/Delete at beginning O(1). Insert/Delete at end, O(1) if last elem is known, O(n) if unknown.

; beginning != front

HEAPS	Heapify	Find Max	Extract Max	Increas Key	Insert	Delete	Merge
Linked List (sorted)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	-	O(n)	O(n)		O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log n)		O(log n)	O(log n)	O(m+n)
Binomial Heap	-	O(1)	O(log n)		O(log n)	O(log n)	O(log n)
Fibonacci Heap	-	O(1)	O(log n)		O(1)	O(log n)	O(1)
Leftist Heap			O(log n)		O(log n)	O(log n)	O(log n)

GRAPHS	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency List	O(V+E)	O(1)	O(1)	O(V+E)	O(E)	O(V)
Incidence List	O(V+E)	O(1)	O(1)	O(E)	O(E)	O(E)
Adjacency Matrix	$O(V^2)$	$O(V^2)$	O(1)	$O(V^2)$	O(1)	O(1)
Incidence Matrix	O(V·E)	O(V·E)	O(V·E)	O(V·E)	O(V·E)	O(E)

Leftist Heap: npl(x) is the length of the shortest path from x to a node w/o two children.
Property: For every node x in the heap the null path length of the left child is at least as large as that of the right.

QuickSelect: Find the k:th smallest element in an unordered list.

Tree Traversal: Preorder - Root -> Left -> Right   Inorder - Left -> Root -> Right
Postorder - Left -> Right -> Root   Level-Order - Left to right, Top to bottom

GRAPH: Sparse graph => Adjacency list   Dense graph => Adjacency Matrix.

Recreation : BST Ok for : Pre ~~+in/lvl~~ and Post ~~+in/lvl~~. NOT OK for only inorder.
BT OK for Pre+in/lvl and Post+in/lvl.

BinaryTree: A tree in which no node can have more than two children.

BST : For every node x in the tree the values of all the items in its left subtree are smaller than the item in x. The analog opposit is true for the right subtree.

AVL Tree : For every node in the tree the height of the left and right subtrees can differ at most by 1.

Binary Heap : A BH is a BT that is completely filled with the possible exception of the bottom lvl. For anny element in position i the left child is in position 2i, the right child in 2i+1 and the parent @ $\lfloor i/2 \rfloor$. In a heap, for every node x the key in the parent of x is smaller than the key in x, with the exception of the root.

RED/BLACK : 1. Every node is colored either red or black. 2. The root is black. 3. If a Node is red, its child must be black. 4. Every path from a node to a null reference must contain the same number of black nodes.

BFS: Maintain a queue of nodes to visit next. The queue contains the start node initially. Repeat: Remove node from queue. Visit it. Find all nodes adjacent to the node and add them to the queue IF they have not already been visited or is not already in the queue.

DFS: Maintain a stack of nodes to visit next. Repeat: Add node to stack. Visit it. Pop stack and add that nodes adjacent nodes to the stak IF they have not already been visited or is not already on the stack.

PRIM: S is a set of all the nodes that are in the tree so far. Pick one arbitrary node. While there is a node in S: Pick the lowest-weight edge between a node in S and a node not in S. Add that ~~edge~~ to the spanning tree and the node to S. (edge) (Use a PQ)

KRUSKAL: Start w/ a set of all nodes and no edges. At each point choose an edge w/ the lowest cost which doesn't create a cycle.

TOPSORT: Keep an init empty list that will contain the sorted elems. Let S be a set of all nodes w/ no incoming edges. While{S is non empty: remove a node n from S, add n to the end of L. For each node m with an edge e from n to m: remove edge from the graph. IF m has no other incoming edges: insert m into S. If graph has edges "Cycle error", else return L.

BUCKET: Set up an array of init empty buckets. Scatter: Go over the original array and put each object in ~~this~~ its bucket. Sort each non-empty bucket. (insertion-sort) Gather: Visit the buckets in order and put all elements back.

RADIX: Perform BUCKET by "least significant". O(kN), N keys which have k or fewer digits.

Doubly w tail!!

```
Public void reverse(){
  Node curr = first;
  while(curr != null){
    Node next = curr.next;
    curr.Prev = next;
    curr = next;
  }
  Node tmp = first;
  first = last;
  last = tmp;
}
```

```
DFS(G, v){
  label v as visited
  for all edges v to w
    if w is not visited
      DFS(G, w)
  Push v to stack
}
```

```
class MultiMap<K,V>{
  Map<K,List<V>> map;
  multimap(){map = new AVL<K,List<V>>}
  insert(K k, V v){
    List<V> Vs = map.lookup(k)}
    if(vs==null){
      vs= new LL<V>();}
    map.insert(k,vs.insert(v));
  }
  delete(k){
    List<V> vs = map.lookup(k)
    V v = vs.head;
    vs = vs.tail;
    if(vs.empty())
      map.delete(k)
    map.insert(k, vs)
    return v;
  }
```

Singly

```
Public void reverse(){
  Node Prev=null;
  Node curr= head;
  Node next;
  while(curr != null){
    next = curr.next;
    curr.next = Prev;
    prev = curr;
    curr = next;
  }
  head = prev;
}
```

Riktad, oviktad → byt riktn

```
Public void reverse(){
  List<Integer>[] r = new LL[nodes]
  for(i:0 → nodes){
    r[i] = new LL<Integer>();
  }
  for(int i:0 → nodes){
    for(Integer j:adjacent[i]}
      r[j].add(i);
  }
  adjacent = rev;
```

Dijkstra: Sätt alla noder till not visited. Sätt avståndet till alla noder till ∞. Sätt avståndet till startnoden till 0. Lägg de närliggande noderna i en min-heap. Beräkna avståndet från start till "extractmin()" ur pq. Repeat.

```
d, P, k : arrays of size |V| init to ∞, null, false.
q = new pq. d[s]=0. q.insert(s,0).
while(q is not empty){ V = q.deletemin();
  if(!k[v]){ k[v] = true. for each adjacent v' to v{
    if(!k[v'] and d[v'] > d[v] + c(v,v') {
      d[v'] = d[v] + c(v.v')   ← cost(v,v')
      P[v'] = V
      q.insert(v',d[v'])
return (d,P)
```

```
append(List ys){   Dubbellänkade!
  tail.prev.next = Ys.head.next;
  Ys.head.next.Prev = tail.Prev;
  tail = Ys.tail;
  Ys.tail = new Node(null, Ys.head, null);
  Ys.head.next = Ys.tail;
}
```

Array → AVL givet sorterat arr

```
Public AVLTree(List ns){
  root = fromArray(ns, 0, ns.size-1);
}
Private TreeNode fromArray(List ns, int f, int t){
  if(f<= t){
    int m = f+(t-f)/2;
    return new TreeNode(
      fromArray(ns, f, m-1),
      ns.get(m),
      fromArray(ns, m+1, t));
  } else { return null }
```

Array Träd → Pekarträd

```
Tree(A[] t){   Bygg träd
  if(t== null)
    return;
  root = fromArray(t, 0);
}
Node fromArray(A[] t, int pos){
  if(pos >= t.length)
    return null;
  return new Node(
    t[pos],
    fromArray(t, Pos*2+1),
    fromArray(t, Pos*2+2));
}
```

Tree W/O parent point → Tree w

```
Public TreeWith(TreeWithout<A> t){
  root = addParents(t.root, null)
}
Private TreeNode addParents(
  TW/O<A>.TreeNode wo,
  TreeNode Parent) {
  if wo == null return null;
  TreeNode w = new TreeNode(
  wo.contents, null, null,
  Parent);
  w.left = addParents(wo.left, w)
  w.right = addParents(wo.right, w)
  return w
}
```