	`vector<T>`	`sorted_vector<T>`	`linked_list<T> (single, head)`	`linked_list<T> (single, head, tail)`	`linked_list<T> (double, head)`	`linked_list<T> (double, head, tail)`
push_back	$O(1)$ (amortized)	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
pop_back	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
push_front	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
pop_front	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
remove	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
find	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Note: `sorted_vector<T>` trade input time for find time, and never use a `doubly_linked_list<T>` that only has head.

	`stack<T>`	`queue<T>`
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$

	`binary_search_tree<T>`	`avl_tree<T>`	`b_tree<T, M>`
push	avg: $O(h = \log n)$ worst: $O(h = n)$	$O(\log n)$	$O(m \log_m n = \log n)$
remove	avg: $O(h = \log n)$ worst: $O(h = n)$	$O(\log n)$	$O(m \log_m n = \log n)$
find	avg: $O(h = \log n)$ worst: $O(h = n)$	$O(\log n)$	$O(\log_m n = \log n)$

	`hash_map<T>`
set	$O(1)$
get	$O(1)$
remove	$O(1)$

	`heap<T>`
create	$O(n)$
heapify up	$O(\log n)$
heapify down	$O(\log n)$
insert	$O(\log n)$
remove	$O(\log n)$

	`disjoint_set<T>`

	`disjoint_set<T>`
union	$O(\log^* n = 1)$
find	$O(\log^* n = 1)$

	Edge List	**Adjacency Matrix**	**Adjacency List**
`space`	$O(V+E)$	$O(V^2)$	$O(V+E)$
`insert_vertex(v)`	$O(1)$	$O(V)$	$O(1)$
`remove_vertex(v)`	$O(E)$	$O(V)$	$O(\deg V)$
`insert_edge`	$O(1)$	$O(1)$	$O(1)$
`remove_edge`	$O(1)$	$O(1)$	$O(1)$
`incident_edge(v)`	$O(E)$	$O(V)$	$O(\deg V)$
`are_adjacent(v, w)`	$O(E)$	$O(1)$	$O(\min(\deg V, \deg W))$

```
function prim(graph, s) {
    let arr = graph.vertices.map((vertex, i) => vertex == s
                ? [vertex, 0]
                : [vertex, Infinity])
    let pq = container(arr); // O(V)
    let distance_map = map(arr);
    let precessor_map = map();

    let new_graph = Graph.new();

    while (pq.is_not_empty()) { // O(V) * Inner
        let [least, _] = pq.smallest() // heap: O(log V), array: O(V);
        new_graph.add_vertex(least); // adjacency_matrix : O(V), adjacency_list : O(1);
        new_graph.add_edge(least, processor_map[least]) // O(1)

        for (let [neigh, neigh_edge] in least.neighbors_with_edge()) { // O(deg V)
            if (new_graph.has(neigh)) continue;
            if (neigh_edge.weight < distance_map[neigh]) {
                distance_map[neigh] = neigh_edge.weight; // O(1)
                pq[neigh] = neigh_edge.weight; // heap : O(log V), array: O(1)
            }
        }
    }

    return new_graph
}

// total time:
// adjacency_matrix + array : O(V) * (O(V) + O(V) + O(deg V) * O(1)) = O(V^2 + E)
// adjacency_matrix+heap: O(V) * (O(log V) + O(V) + O(deg V) * O(log V)) = O(V^2 + Elog V)
```

```
// adjacency_list + array : O(V) * (O(V) + O(log V) + O(deg V) * O(1)) = O (V^2 +
E)
// adjacency_list + heap : O(V) * (O(log V) + O(deg V) * O(log V)) = O(Vlog V +
Elog V)
// adjacency_list + fibonacci_heap : O(V) * (O(log V) + O(deg V) * O(1)) =
O(VlogV + E)
```

Prim	`adjacency_matrix`	`adjacency_list`
`array`	$O(V^2 + E)$	$O(V^2 + E)$
`heap`	$O(V^2 + E \log V)$	$O((V + E) \log V)$

```
function kruskal(graph) {
    let edges = graph.edges.sort((a, b) => a.weight < b.weight); // O(ElogE)
    let dset = disjoint_set(graph.vertices);
    let new_graph = Graph.new();
    for (let edge in edges) { // O(V)
        if (new_graph.vertices.length >= n - 1) break;
        new_graph.add_vertex(edge.vertexA);
        new_graph.add_vertex(edge.vertexB);
        new_graph.add_edge(edge.vertexA, edge.vertexB);
        if (dset.find(edge.vertexA) != dset.find(edge.vertexB)) {
            dset.union(edge.vertexA, edge.vertexB);
        }
    }
    return new_graph;
}
```

Kruskal	
`heap`	$O(E \log E)$
`sorting`	$O(E \log E)$

```
function dijkstra(graph, start, destin) {
    let arr = graph.vertices.map((vertex, i) => vertex == s
                   ? [vertex, 0]
                   : [vertex, Infinity])
    let prev_map = {};
    let distance_map = map(arr);
    let pq = priority_queue(arr);
    let visited = {};

    distance_map[start] = 0;
    let current = start;

    while (pq.top != destin) { // O(V)
        let current = pq.pop(); // O(log V)
        for (let [neigh, edge] in current.neighbor_with_edge()) { // O(deg V)
            if (visited.neigh !== undefined) return;
            if (distance_map[neigh] < distance_map[current] + edge.weight) {
```

```
                distance_map[neigh] = distance_map[current] + edge.weight; //
fibonacci_heap : O(1), heap : O(log V)
                prev_map[neigh] = current;
            }
        }
        visited[current] = true;
    }
    let path = []
    while (prev_map[current] !== undefined) {
        path.push(current);
        current = prev_map[current];
    }
    return [path, distance_map[destin]];
}
// time complexity:
// heap : O(V) * (O(log V) + O(deg V) * O(log V)) = O(Vlog V) + O(Elog V) = O((V
+ E)log V)
// fibo_heap : O(V) * (O(log V) + O(deg V) * O(1)) = O(Vlog V) + O(E) = O(E +
Vlog V)
```

time complexity is $O(E + V\log V)$ (fibonacci heap)