* hash function = translation + compression

* Two ways of compressing the output of the hash function $h(t)$ into range $[0, M)$. m: table size
  - $|h(t)| \bmod M$ (M should be a prime) $a \% m \neq 0$
  - $|a \times h(t) + b| \bmod M$ (a and b are prime numbers → used if you can't controle m

* seperate chaining: 1. (erasing) searching for a key is $\theta(\frac{N}{M})$ (# key →)
  2. inserting a key is $\theta(1)$ if duplicate keys allowed
  $\theta(\frac{N}{M})$ if duplicates are not allowed

* open addressing: linear, quadratic probing + double hashing
  ($i+1, i+2 \cdots$; $i+1^2, i+2^2, i+3^2 \cdots$)

* linear probing disadvantage: primary clustering

* evenly distribute $\theta(1)$ cluster (half, $\frac{1}{3}$ ...) $\theta(n)$

* quadratic hashing disadvantage: secondary clustering.

* quadratic residues ⇒ $(e + j^2) \bmod M$ only map to some

* double hashing: $t(\text{key}) + j \times t'(\text{key})$ (j: collision number; t': second hash that's used if there's collision)

* if linear probing is used and the hash function distributes keys evenly, the expected number of probes to successfully search for an existing element is $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

* load factor $\alpha = \frac{N}{M}$ (N ← # keys, M ← size of table)

* complexity of std::unordered_map<>
  for all insert, access, erase, find:
  average case $\theta(1)$, worst case $\theta(n)$

* Dynamic Hashing
  - double the size of the hash table if the load factor > 0.5
  - Rehash every element (deleted positions don't get rehashed)

* Adjacency Matrix / Adjacency List

Adjacency Matrix	Adjacency List
A $\lvert V\rvert \times \lvert V\rvert$ matrix, uses $\theta(V^2)$ space ($\lvert E\rvert = O(V^2)$)	uses $\theta(V+E)$ space ($\lvert E\rvert = O(V)$)
represent dense graphs	represent sparse graphs
$\theta(1)$ check edge exists between	to check edge: worst case $\theta(V)$, average case $\theta(\frac{E}{V})$ / $\theta(1+\frac{E}{V})$
$\theta(V^2)$ iterate over all edges	$\theta(V+E)$ iterate all edges
$\theta(1)$ add edge to graph	$\theta(1)$ add edge to the graph
$\theta(1)$ remove edge from graph	to remove: worst case $\theta(E)$, average case $\theta(\frac{E}{V})$

* DFS and BFS adjacency list $\theta(V+E)$, adjacency matrix $\theta(V^2)$

* Prim's Algorithm: Building "forest"
  Matrix (linear search) $\theta(V^2)$ dense
  list (binary heap) $\theta(E \log V)$ sparse
  works on negative edges
  fails on directed graphs

* Kruskal's Algorithm: $\theta(E \log E)$ / $\theta(E \log V)$ sparse
  * $\theta(\log E) = \theta(\log V)$

0-1 knapsack
* Brute force $\theta(n2^n)$ DP $\theta(nC)$ (n: # items, C: capacity)
fractional knapsack
greedy $\theta(n \log n)$

$$F(i,c) = \begin{cases} 0, & \text{if } i = 0 \\ \max(F(i-1,c),\ F(i-1,c-w_i)+v_i), & \text{if } i > 0,\ c > 0 \end{cases}$$

* Dijkstra's Algorithm: recording v, k_v, d_v, p_v
  Adjacency matrix (linear search) = $\theta(V^2)$ dense
  Adjacency list (binary heap) = $\theta(E \log V)$ sparse

optimization problem: looking for largest, shortest... (max, min.)

* Branch and Bound:
  1. minimization: keep upper bound, change lower bound (our current best; estimate of partial sol; optimistic underestimate)
  2. maximization: keep lower bound, change upper bound (optimistic overestimate)

* Backtracking ⇒ solving N-Queens
  (constraint satisfaction: satisfying constraint is not necessarily optimal)

* For seperate chaining, collisions only occur between elements that have the same hash value. But that's not true for open addressing

* array-based binary tree		* pointer-based binary	
insert (Best Case)	$\theta(1)$	insert (Best Case)	$\theta(1)$
assuming no duplicate elements			
insert (Worst Case)	$\theta(n)$	insert (Worst Case)	$\theta(n)$
Remove key (Worst Case)	$\theta(n)$	remove (Worst Case)	$\theta(n)$
Find Parent	$\theta(1)$ *	Find Parent	$\theta(n)$
Find child	$\theta(1)$	Find child	$\theta(1)$
Space Best Case	$\theta(n)$	Space Best Case	$\theta(n)$
Space Worst Case	$\theta(2^n)$	Space Worst Case	$\theta(n)$

Space Worst Case $\theta(2^n)$: Depth = n, space = $2^0 + 2^1 + 2^2 + \cdots + 2^{n-2} + 2^{n-1} = 2^n - 1 = \theta(2^n)$
A tree with at most k children: $1\left(\frac{1-k^n}{1-k}\right) = \theta(k^n)$

* Insertion into AVL tree = $\theta(1)$ rot (fixing one node)   * Deletion from AVL: $\theta(\log n)$ rot

AVL:

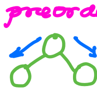
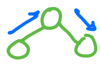
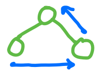
	Best-Case	Average	Worst-Case
Finding a value	$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$ ($\theta(1)$)
Inserting a value	$\theta(\log n)$ ($\theta(1)$ BST)	$\theta(\log n)$	$\theta(\log n)$ ($\theta(1)$)
Deleting a value	$\theta(\log n)$ ($\theta(1)$ BST)	$\theta(\log n)$	$\theta(\log n)$ ($\theta(1)$)

* std::map<>, for search, insert, delete: $\theta(\log n)$ average and worst-case time

* A "unique" shortest path must be included in the MST

As α increases, the performance of separate chaining does not deteriorate as quickly as the performances of open addressing methods

only knowing preorder and postorder traversals is not enough

preorder  inorder  postorder 

* Bottom-up DP can be used any time top-down is used

* DP can not be used to solve a problem that can also solved using divide and conquer

* A proper or full binary tree where every node either has 0 or 2 children. (all non-leaf nodes in a proper binary tree have 2 children)

* A complete binary tree is a tree in which every depth, except possibly the last, is completely filled. If the last depth (the bottom row) is not completely filled, then all nodes that do exist at that depth must be filled from left to right with no gaps (vertices can have 1 child)

* General trees, where each node can have any number of children, can be stored as a binary tree

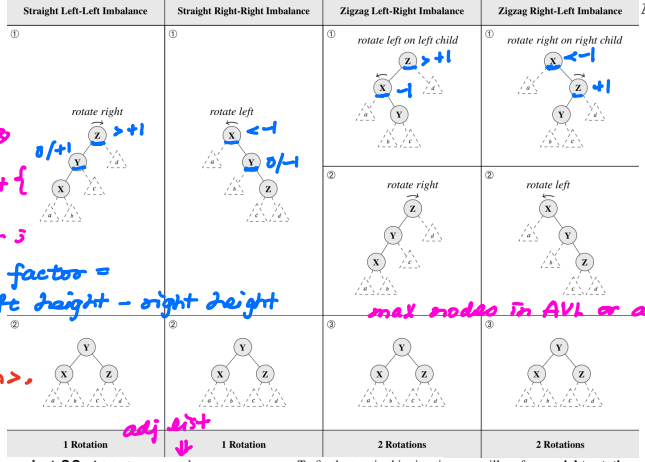
* Shortest path
  1. DFS: single-source shortest path problem in $O(V+E)$ or $O(V^2)$ time if acyclic graph
  2. BFS: unweighted or same-weight graphs
  3. Dijkstra's: non-negative weighted/unweighted graphs (can also be directed)

* Only connected graphs have a MST

* Given a connected graph with n vertices, $n-1$ edges are in the MST of the graph

* 2D vector:
```
vector<vector<T>> vec_2d(m, vector<T>(n, data));
```

```
class PersonComparator {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.get_age() < p2.get_age();
    }
};
priority_queue<Person, vector<Person>, PersonComparator> myPQ;
```



balance factor = left height - right height

adj list

```
Algorithm check_and_balance(Node* n):
  if balance_factor(n) > +1:
    if balance_factor(n->left) < 0:
      n->left = rotate_left(n->left)
    n = rotate_right(n)
  else if balance_factor(n) < -1:
    if balance_factor(n->right) > 0:
      n->right = rotate_right(n->right)
    n = rotate_left(n)
```

minimum nodes in AVL tree:
$f(h) = f(h-1) + f(h-2) + 1$
$f(1) = 1 \quad f(0) = 0$

max nodes in AVL or any binary tree: $2^h - 1$

height(node) = max{height(children)} + 1
height(empty) = 0
depth(node) = depth(parent) + 1
depth(empty) = 0

To fix the tree in this situation, we will perform a **right rotation** on the node rooted at the node with the balance factor greater than +1 (i.e., the darker shaded node). To conduct a right rotation, complete the following steps:

1. Take the root of the subtree that you want to rotate (the darker shaded node) and set its left child to the right child of its original left child.
2. Take the original left child (the lighter shaded node) and set its right child to its original parent (the darker shaded node).
3. Return the new root of the rotated subtree to the caller that invoked the rotation (which should be the parent of the initial imbalanced node, if there is one). If the node you want to rotate (the darker shaded node) has a parent, reset its parent's child so that it points to the new root of the rotated subtree (which ends up being the lighter shaded node).

To fix the tree in this situation, we will perform a **left rotation** on the node with the balance factor less than -1 (i.e., the darker shaded node). To conduct a left rotation on a node, complete the following steps:

1. Take the root of the subtree you want to rotate (the darker shaded node) and set its right child to the left child of its original right child.
2. Take the original right child (the lighter shaded node) and set its left child to its original parent (the darker shaded node).
3. Return the new root of the rotated subtree to the caller that invoked the rotation (which should be the parent of the initial imbalanced node, if there is one). If the node you want to rotate (the darker shaded node) has a parent, reset its parent's child so that it points to the new root of the rotated subtree (which ends up being the lighter shaded node).

```
void find_all_paths_helper(const std::vector<std::vector<int32_t>>& graph,
                           int32_t curr, int32_t dest, std::vector<bool>& visited,
                           std::vector<int32_t>& current_path,
                           std::vector<std::vector<int32_t>>& solution) {
  visited[curr] = true;
  current_path.push_back(curr);

  if (curr == dest) {
    // we have reached the destination vertex, so append the current path to the s
    solution.push_back(current_path);
  } // if
  else {
    // recurse into all adjacent vertices that are not visited
    const std::vector<int32_t>& neighbors = graph[curr];
    for (int32_t neighbor : neighbors) {
      if (!visited[neighbor]) {
        find_all_paths_helper(graph, neighbor, dest, visited, current_path, solution);
      } // if
    } // for i
  } // else

  // once you are done processing current vertex, remove it from path and mark it as unvisited
  current_path.pop_back();
  visited[curr] = false;
} // find_all_paths_helper()
```

DFS - recursive

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \binom{n}{0} = 1 \quad \binom{n}{n} = 1$$

```
unordered_map<key_type, value_type> map;
map.insert({key, value}); map.emplace(key, value);
value_type value = map[key];
auto it = map.begin(); it->first/second...
clear(), empty(), size(),
map.erase(key);
auto it = map.find(key);
if (it == map.end()) {
    // not found
} else { // found }
for (auto& pair : map) {
    // pair.first
    // pair.second
}
```

```
std::vector<std::vector<int32_t>> find_all_paths(const std::vector<std::vector<int32_t>>& graph,
                                                 int32_t source, int32_t dest) {
  std::vector<bool> visited(graph.size(), false);
  std::vector<int32_t> current_path;
  std::vector<std::vector<int32_t>> solution;
  find_all_paths_helper(graph, source, dest, visited, current_path, solution);
  return solution;
} // find_all_paths()
```

```
int32_t num_devices(std::vector<std::vector<bool>>& adj_matrix) {
  std::vector<bool> visited(adj_matrix.size(), false);
  std::stack<size_t> dfs;
  int32_t network_count = 0;
  for (size_t i = 0; i < visited.size(); ++i) {
    if (!visited[i]) {
      dfs.push(i);
      while (!dfs.empty()) {
        size_t curr = dfs.top();
        dfs.pop();
        for (size_t j = 0; j < adj_matrix[curr].size(); ++j) {
          if (adj_matrix[curr][j] && !visited[j]) {
            visited[j] = true;
            dfs.push(j);
          } // if
        } // for j
      } // while
      ++network_count;
    } // if
  } // for i
  return network_count;
} // num_devices()
```

DFS - iterative

```
std::vector<int32_t> find_shortest_path(const std::vector<std::vector<int32_t>>& graph,
                                        int32_t source, int32_t dest) {
  // this map stores the predecessor of each vertex discovered along our BFS
  // e.g., if backtrace[2] = 1, then vertex 1 directly precedes 2 along path
  std::unordered_map<int32_t, int32_t> backtrace;
  backtrace[source] = -1;  // source vertex has no predecessor

  // perform BFS
  std::queue<int32_t> bfs;
  bfs.push(source);
  bool path_found = false;
  while (!path_found && !bfs.empty()) {
    int32_t curr = bfs.front();
    bfs.pop();
    // iterate over vertices directly connected with current vertex
    // if any are unvisited (i.e., has no predecessor), push into queue
    for (int32_t neighbor : graph[curr]) {
      if (backtrace.find(neighbor) == backtrace.end()) {
        backtrace[neighbor] = curr;
        // if neighbor is solution, break out of loop
        if (neighbor == dest) {
          path_found = true;
          break;
        } // if
        bfs.push(neighbor);
      } // if
    } // for neighbor
  } // while

  // backtrace from dest back to source
  // this is done by following the predecessors from dest to source
  std::vector<int32_t> path;
  int32_t prev_vertex = dest;
  if (path_found) {
    path.push_back(prev_vertex);
    while (backtrace[prev_vertex] != -1) {
      path.push_back(backtrace[prev_vertex]);
      prev_vertex = backtrace[prev_vertex];
    } // while
    std::reverse(path.begin(), path.end());
  } // if
  return path;
} // find_shortest_path()
```

BFS

```
int32_t weld_pipes(const std::vector<int32_t>& pipes){
 std::vector<std::vector<int32_t>> weights(pipes.size(), std::vector<int32_t>(pipes.size()));
 std::vector<std::vector<int32_t>> costs(pipes.size(), std::vector<int32_t>(pipes.size(), -1
 return weld_helper(pipes, costs, weights, 0, pipes.size() - 1);
 // weld_pipes()

int32_t weld_helper(const std::vector<int32_t>& pipes, std::vector<std::vector<int32_t>>& cos
                    std::vector<std::vector<int32_t>>& weights, int32_t begin, int32_t end){
 if (costs[begin][end] != -1) {
   return costs[begin][end];  // no need to redo work
 } // if
 if (begin == end){  // base cases
   costs[begin][end] = 0;
   weights[begin][end] = pipes[begin];
   return 0;
 } // if.
 int32_t min_cost = std::numeric_limits<int32_t>::max();
 for (int32_t k = begin; k < end; ++k) {
   int32_t cost_k = weld_helper(pipes, costs, weights, begin, k) +
                    weld_helper(pipes, costs, weights, k + 1, end);
   cost_k += std::max(weights[begin][k], weights[k + 1][end]);
   min_cost = std::min(min_cost, cost_k);
 } // for k
 costs[begin][end] = min_cost;
 weights[begin][end] = weights[begin][begin] + weights[begin + 1][end];
 return costs[begin][end];
 // weld_helper()
```

$$F(x,y) = \begin{cases} 0 & \text{if } x = y \\ \min\left(\max\left(\sum_{i=x}^{k} w_i, \sum_{j=k+1}^{y} w_j\right) + F(x,k) + F(k+1,y)\right) & x \le k < y \end{cases}$$

Finding subset with sum x:

$$F(n,x) = \begin{cases} \text{true} & \text{if } n=0,\ x=0 \\ \text{false} & \text{if } n=0,\ x \neq 0 \\ F(n-1,x) \text{ or } F(n-1, x-v_{n-1}) \end{cases}$$

Robbing house (not adj):

$$F(i) \begin{cases} \max(F(i-2) + \text{house}[i],\ F(i-1)) \\ 0 \text{ if } i < 0 \end{cases}$$