Common Errors to Check

Division by 0

Confuse the time complexity with 1 operation with the time complexity of the total process

Recursion has no terminating condition

Accidentally breaking invariants

Try to access nullptr after modifying codes

Assume two input source are different (Assume they are the same all the time to avoid tricky corner cases)

Does not add idiot check

Does not consider that the input can be empty range

Does not use `delete` & `delete[]` to free memory, causing memory leak

Off-by-One Error

```
const size_t SIZE = 5;
int x[SIZE];
size_t i;

// Acessing x[SIZE]
for (i = 0; i <= SIZE; ++i)
    x[i] = i;

// Accessing x[SIZE - 1 + 1]
for (i = 0; i <= SIZE - 1; ++i)
    x[i] = x[i + 1];

// Not accessing x[SIZE - 1]
```

```
for (i = 1; i < SIZE; ++i)
    x[i - 1] = i;
```

Use reserve and index at the same time or use resize and push at the same time

```
// would double the size
vector<int> tmp(k);
while (...) {
    tmp.push_back(...);
}

// would cause the size to be empty
vector<int> tmp;
tmp.reserve(k);
for(int i = 0; i < k; ++k) {
    tmp[i] = ...;
}43 , 58 , 73 , 23 , 38 , 41 , 26 , 15
```

'\0' at the front

```
const  char *s = "hello";
char ss[20];int  length = strlen(s);
for (int i = 0; i < length; ++i)
ss[i] = s[length  - i];printf("%s", ss);
```

Mistake `memory` complexity with `time` complexity

Swap An Array of N Elements (True no matter N even or odd)

`while (i = 0; i < n / 2; ++i) { swap(ar[i], ar[n - i -1]); }`

Hash table

std::map is implemented with a binary search tree under the hood

std::unordered_map is implemented with hash table

```
unordered_map<string, string> dict;
dict["first"] = "second";
for (auto& item : dict) {
    cout << item.first << " : " << item.second 43 , 58 , 73 , 23 , 38
, 41 , 26 , 15<< endl;
}
// # On console:
// First : Second
```

Ghost is not counted as a the load, so load factor = occupied / total size

Find all pairs such that their sum is equal and there can be `at most 2 pairs` for a fixed sum

```
using namespace std;

string convert(int a, int b) {
    return "(" + to_string(a)+ ", " + to_string(b) + ")";
}

void two_pair_sums(const vector<int>& nums, ostream& os) {
    // TODO

    std::unordered_map<int, string> dict;

    vector<int>::const_iterator last = nums.end();
    vector<int>::const_iterator curr = nums.begin();
    vector<int>::const_iterator iter;
    for (; curr != last; ++curr) {
        for (iter = curr + 1; iter != last; ++iter) {
            auto found = dict.find(*curr + *iter);
            if (found == dict.end()) {
                dict[*curr + *iter] = convert(*curr, *iter);
            } else {
                os << dict[*curr + *iter] + " and " +
convert(*curr, *iter) << endl;
            }
```

```
        }
    }
}
```

More advanced hash table concepts

When 2 elements hash to the same position in an empty hash table. Their difference in hash value is a `multiple` of the table size.

Separate Chaining

> ### The load factor of separate chaining can be greater than 1

Create some kind of `table` to simulate the result

Index	0	1	2	3	4	5	6	7	8	9
Key										
Value										

l07 Insertion Sequence

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12
n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
13	14	15	16	17	18	19	20	21	22	23	24	25

> **1. ("banana", "yellow")**
>
> **2. ("blueberry", "blue")**

3. ("blackberry", "black")

4. ("cranberry", "red")

5. ("apricot", "orange")

6. ("lime", "green")

Linear Probing

Index	0	1	2	3	4	5	6	7	8	9
Key	apricot	banana	blueberry	blackberry	cranberry	lime				
Value	orange	yellow	blue	black	red	green				

Quadratic Probing

Advantage:

> Quadratic probing may insert keys into positions of a hash table that are far from the index that they normally hash to.

Disadvantage:

> Depending on the size of the hash table involved, it is possible for quadratic probing to never consider specific indices while searching for an open position.

Index	0	1	2	3	4	5	6	7	8	9
Key	apricot	banana	blueberry	cranberry		blackberry		**lime**		
Value	orange	yellow	blue	red		black		**green**		

Very easy to miscalculate the position of of limeAVL

$$
\begin{aligned}
h(l) &\rightarrow \text{bucket}_{1,1,occupied} && \rightarrow \text{bucket}_{1+1^2,2,occupied} \\
&\rightarrow \text{bucket}_{1+2^2,5,occupied} && \rightarrow \text{bucket}_{1+3^2,0,occupied} \\
&\rightarrow \text{bucket}_{1+4^2,7,allocable}
\end{aligned}
$$

Double Hashing

> **Step $h^{'}(k) = q - (h(k)\%q)$ ahead with $q = 5$ each time when facing a occupied bucket**
>
> **It is important to remember that different hash integers yield different steps $h^{'}(k)$. As such, `be careful not to advance with the step calculated for the previous hash int`**

Index	0	1	2	3	4	5	6	7	8	9
Key	apricot	banana	cranberry	lime		blueberry				blackberry
Value	orange	orange	red	green		blue				black

Separate Chaining

Hash Table Implementation, ag07

Tree

In order traversal of BST is sorted while that of BT is not guaranteed.

The first number in a postorder traversal `may not` be the smallest element in the tree.

```
#    __1__
#   /     \
# null     2
```

Number of worst Tree Possible: 2^{node-1}

> Worst case: all nodes (12) form a stick
>
> 11 children have the possibility to switch left or right from their parents 2^11

Height is counted from the bottom instead of the top.

> Nodes on the same level may not have the same height

Complete binary trees

> Such tree has every level fully filled, with the possible exception of the lowest level, which is filled left to right.

Full binary trees

> Proper/full binary trees have every level fully filled

Traversal

> Pre-order traversal
>
> In-order traversal
>
> Post-order traversal

Write down pre/in/post order traversal for a given tree

> Method:

> **Write down you answer just for nodes with height $\in [0, H]$**
>
> **Insert the nodes from height $H + 1$ into the answer**
>
> **This is like how you would approach the hashing locations for a sequence of input**

Example:

```
#      __ 3 __
#     /       \
#    6         7
#     \       /   \
#      1     5     4
#     /             \
#    2               8
```

Pre-order:

> **Height $\leq$ 0: [3]**
>
> **Height $\leq$ 1: [3 , 6 , 7]**
>
> **Height $\leq$ 2: [3 , 6 , 1 , 7 , 5 , 4] node**
>
> **Height $\leq$ 3: [3, 6 , 1 , 2 , 7 , 5 , 4 , 8]**

In-order:

> **Height $\leq$ 0: [3]**
>
> **Height $\leq$ 1: [6 , 3 , 7]**
>
> **Height $\leq$ 2: [6 , 1 , 3 , 5 , 7 , 4]**
>
> **Height $\leq$ 3: [6, 2 , 1, 3, 5, 7, 4, 8]**

Post-order:

> #### Height $\leq$ 0: [3]
>
> #### Height $\leq$ 1: [6 , 7 , 3]
>
> #### Height $\leq$ 2: [1 , 6 , 5 , 4 , 7 , 3]
>
> #### Height $\leq$ 3: [2 , 1 , 6 , 5 , 8 , 4 , 7 , 3]

Given pre-order and in-order traversals, reconstruct post-order traversal

Pre-order : [3, 4, 0, 1, 5, 2, 6, 7]

In-order : [4, 3, 1, 5, 0, 2, 7, 6]

Try to reconstruct line by line (height by height)

```
# 3 is first in Pre-order

# 3

# 4 is on the left of 3 in In-order with no children and 0 is only
2 steps away from 3

#   3
#  / \
# 4   0

# From In-order, 1 & 5 is 0's left subtree and from Pre-order 1 is
the parent of 5, so
node
#    0
#   /
#  1
#   \
#    5

# From In-order 2, 7 & 6 is 0's right subtree and from Pre-order,
6 & 7 is the right subtree of 2
#    0
#   / \
#  1   2
#   \
```

```
#    5

#    3
#   / \
#  4   0
#     / \
#    1   2
#    \
#     5

# As 6 & 7 is the right subtree of 2, then by In-order, 7 is 6's
left child

#    3
#   / \
#  4   0
#     / \
#    1   2
#     \   \
#      5  6
#        /
#       7 the time complexity with 1 operation with the time
complexity of the total process
Recursion has no terminating condition
```

Post-order : [4 , 5 , 1 , 7 , 6 , 2 , 0 , 3]

Given just pre-order and post-order tree, we can't uniquely identify the tree.

AVL Tree

Min_Node(height + 2) = Min_Node(height + 1) + Min_Node(height) + 1

`Don't overlook` instructions !!! Insert and remove all the nodes.

The question may ask you to replace with `Inorder predecessor` instead of successor

Binary Search Tree has $O(log(n))$ average but $O(n)$ `worst` search complexity if not AVL

AVL Tree guarantee $O(log(n))$ search complexity for a tree size of n

Balance factor =
$height(node \rightarrow left) - height(node \rightarrow right)$

AVL can't have triple rotation

Inorder successor : current->right->left->...->left

> **Be careful when current->right->left = nullptr, then `current->right` is the successor**
>
> **Remove current with successor**

```
#     __43__                  __58__
#    /      \                /      \
#   23      58       ->     23      73
#  /  \       \             /  \
# 15   38      73          15  38
#     /  \                    /  \
#    26  41                  26  41
```

Inorder predecessor : current->left>right->...->right

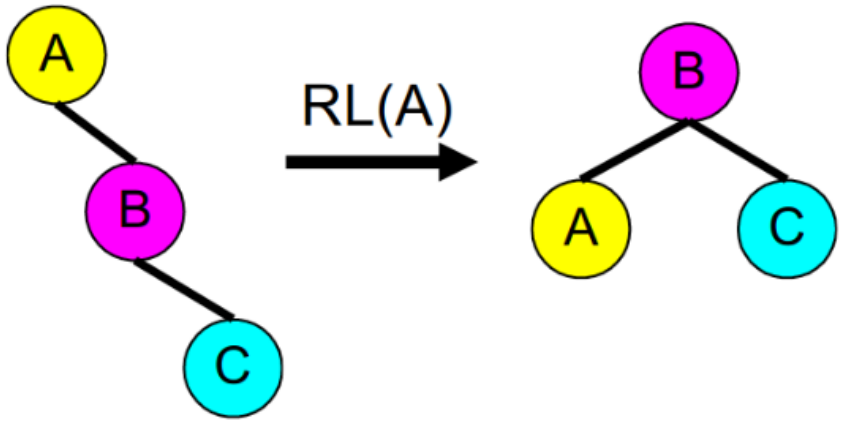
> **Be careful when current->left->right = nullptr**
>
> **Then `current->left` is the predecessor**

Remove current with predecessor

```
#      __43__                  __41__
#     /      \                /      \
#    23       58       ->    23       58
#   /  \        \            /  \        \
# 15   38       73         15   38       73
#      /  \                    /
#     26  41                  26
```

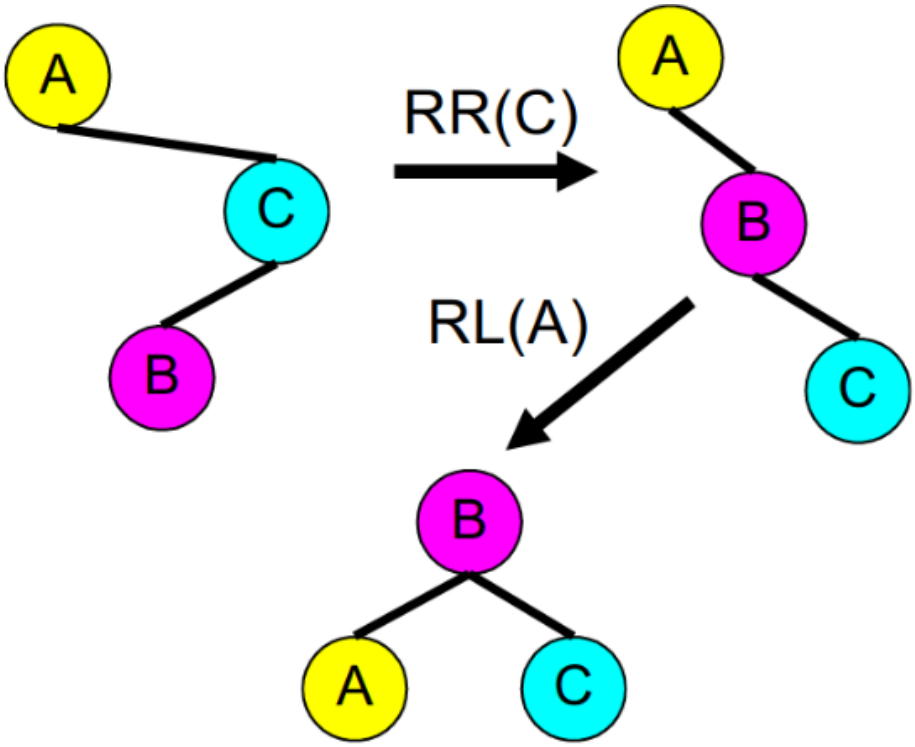
Rotation

Condition 1 (Taking the right part for illustration)



1. A is not balanced
2. The height of B's right node $\geq$ the height of B's left node
3. Then a single rotation is needed.
4. B's left child becomes A's right child

Condition 2 (Taking the right part for illustration)



bottom to top

1. A is not balanced
2. The height of C's right node $<$ the height of C's left node
3. Then a double rotation is needed.
4. B's right node becomes C's left node
5. B's left node becomes A's right node

Remember to label the height of nodes, including nullptr so that we don't forget to fix the AVL

Graph

Going through a maze within allowed areas run in $O(n^2)$, need to judge at most n - 1 cells for each node whether they have been visited before. Also see P1.

It takes $O(1 + E/V)$ on average to find the closest node to a given one with adjacency list.

Concepts

> ### Weighted
>
> ### Unweighted
>
> ### Directed
>
> ### Undirected
>
> ### Sparse, eg : Facebook network is sparse
>
> ### Dense, eg: fully connected
>
> ### Adjacent : there exist path from node i to node j

MST (Only works on Undirected Graph)

Prim ($O(V^2)$ or $O(Elog(E))$)

> #### 1. Keep a table in the form

Already added to the finished graph	Current nearest distance to the finished graph	Current Previous element
...	...	...
...	...	...

2. Mark node[0] as already added to the finished graph

3. Whenever add a node into a finished graph, update the current nearest distance of all nodes outside the finished graph

4. Select the node outside the finished graph with the minimum current nearest distance and mark it as already added to the finished graph

5. If less than n iterations, go to step 3

Kruskal ($O(Elog(E)) \approx O(Elog(V))$**) as** $O(log(V)) = O(log(V^2) = O(log(E))$

1. Sort all edges from smallest to largest

2. Add the smallest edge that has not been added to the solution

3. If this edge forms a circle in the solution, then remove it

4. Proceed until all nodes are connected.

Dijkstra ($O(V^2)$ **or** $O(Elog(E))$**)**

Similar to MST but finds the minimum distance from a given node to `all other nodes`.

DFS & BFS (Taking $O(|V| + |E|)$ time)

Remember when using DFS with `stack` by pushing nodes from left to right

> Then you will `investigate from right to left` during next round
>
> Moreover, the traversal will `print value from top to down`

```
#     __43__
#    /      \
#   23      58     -> print from the right
#  /  \       \
# 15   38      73
#     /  \
#    26  41
```

> Traversal : [43 , `58 , 73` , `23 , 38 , 41` , 26 , 15]

If we terminate immediately when we first encounter an element

> if an element at a depth of 121 is popped in `BFS`, this element `can't` be found at depth $\leq$ 121.

When using BFS with a queue

> We will still investigate from left to right during next round
>
> And the traversal will print value level by level

```
#     __43__
#    /      \
#   23      58     -> print from the right
#  /  \       \
# 15   38      73
#     /  \
#    26  41
```

> Traversal : [43 , 23 , 58 , 15 , 38 , 73 , 26 , 41]

Can DFS and BFS guarantee that we can find the shortest path between 2 points ?

> DFS
>
> > Unweighted Graph : May not be the case
> >
> > Weighted Graph : May not be the case
>
> BFS
>
> > Unweighted Graph : Can guarantee
> >
> > Weighted Graph : May not be the case

Backtracking

Only care about whether there can be a solution

Greedy

If Knapsack problem accept fractional taking, greedy is the best approach

> Finishing remaining food with your stomach. However, if you begin a dish, you do NOT need to finish it in its entirety.

Dynamic Programming

Knapsack

```
uint32_t knapsackDP(const vector<Item> &items, const size_t m) {
    const size_t n = items.size();
    vector<vector<uint32_t>>
    memo(n + 1, vector<uint32_t>(m + 1, 0));  // +1, +1

 for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < m + 1; ++j) {  // +1 for memo[][m]
            if (j < items[i].size)
            memo[i + 1][j] = memo[i][j];
        else
            memo[i + 1][j] = max(memo[i][j], memo[i][j –
items[i].size] + items[i].value);
    }  // for j
 }  // for i
 return memo[n][m];
}
```

Seam Carving (removing a zigzag column from the board with the lowest sum)

> We need to calculate the best value for each cell at row k, then we proceed to row k + 1

Branch and Bound

> Unlike backtracking, always try to find the optimal solution
>
> The branch and bound solution to the 0-1 Knapsack problem runs in $O(n2^n)$ time if you are given n items to choose from

Heuristic only find sub-optimal solution, not `optimal` ones.

Largest Subset

You are given a set of N integers and a target integer K. Suppose you want to find the subset of these N integers with the largest size, such that the total sum of all elements in the subset is less than or equal to K.

> **If log(N) < K, choose greedy, otherwise choose dynamic programming**
>
> > **Greedy:** $Nlog(N) + N$
> >
> > **Dynamic:** NK

Unordered Map

Unordered map can't be sorted

When the key exist before, any `.insert()` method would fail to insert value

Map

Map is sorted by key with a underling Red Black Tree structure

(--(someMap.end()->second) returns the value of that item with the max key

But this will not be the case when it is an `unordered_map`, don't confuse !!!

Index Sorting

```
> class SortByCoord {
    const vector<double> &_coords;
public:
    SortByCoord(const vector<double> &z) : _coords(z) {}
    bool operator()(size_t i, size_t j) const {
        return _coords[i] < _coords[j];
    }  // operator()()
};  // SortByCoord{}


vector<size_t> idx(100);
vector<double> xCoord(100);
for (size_t k = 0; k != 100; ++k) {
    idx[k] = k;
    xCoord[k] = rand() % 1000 / 10.0;
}  // for
SortByCoord sbx(xCoord);  // sbx is a function object!
sort(begin(idx), end(idx), sbx);
```

Priority Queue

```
std::priority_queue<T, std::vector<T>, std::less<T>> defaultPQ;
std::priority_queue<T, std::vector<T>, COMP> customPQ;
// Range build constructor, copy the vector elements and build a heap
inplace, tight_bound = O(n).
std::priority_queue<T, std::vector<T>, COMP>
rangePQ(sourceVector.begin(), sourceVector.end(), COMP_OBJECT);
```

Read in

```
int i;
string buffer
while (!(cin >> i) {
    // Keep reading unless encounter a integer
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

while (getline(cin, buffer)) {
    // Keep reading until reaching cin.eof()
}
```

Write out

```
template <typename T>
ostream &operator<<(ostream &out, const stack<T> &s) {
    // display the contents of a stack on a single line
    // e.g., cout << my_stack << endl;
    stack<T> tmpStack = s;  // deep copy
    while (!tmpStack.empty()) {
        out << tmpStack.top() << ' ';
        tmpStack.pop();
    }  // while
    return out;
}  // operator<<()

template <typename T>
ostream &operator<<(ostream &out, const stack<T> &s) {
    // display the contents of a stack on a single line
    // e.g., cout << my_stack << endl;
    for (auto &el : v) {
        out << el << ' ';
    }
    return out;
}  // operator<<()
```

Lambda

There are several ways to capture variables in a lambda:

[] captures no variables from the surrounding scope

[=] captures all variables in the surrounding scope by value (i.e., a copy of each variable is made)

[&] captures all variables in the surrounding scope by reference

[foo] captures only the variable foo, by value

[&foo] captures only the variable foo, by reference

[=, &foo] captures all variables in the surrounding scope by value except for foo, which is captured by reference

[&, foo] captures all variables in the surrounding scope by reference except for foo, which is captured by value

[this] captures the current object – needed if a lambda defined in an object's member function needs access to its member variable