Dict implementation (HashMap)

1. **Array of Buckets (or Slots):** At the core of the hash table, there is an array. Each element in this array is called a bucket (or slot). The size of this array usually increases as more elements are added to the hash table to reduce the likelihood of collisions and ensure constant-time average complexity for retrieval.

2. **Hash Function:** This is a function that takes a key as input and returns an integer, which is then used as the index to place the corresponding value in the array of buckets. This function is designed to distribute keys evenly across the array to minimize collisions (where two keys end up with the same index).

3. **Handling Collisions:** If two keys hash to the same index (a collision), we need a strategy to handle this. The two most common methods are separate chaining and open addressing.
   * **Separate Chaining:** Each bucket holds a linked list (or another kind of data structure). All the values that hash to the same bucket will be stored in this list. A disadvantage of this method is that if many keys hash to the same bucket, retrieving a value can devolve to a linear search through the list.
   * **Open Addressing:** Instead of storing multiple values at each index, open addressing proactively searches for another free slot in the array when a collision occurs. There are several methods of open addressing, such as linear probing (look for the next free slot in a linear fashion), quadratic probing (look for slots based on a quadratic function of the distance from the original hash), or double hashing (use a second hash function to determine how to probe for free slots).

4. **Rehashing:** As the hash table grows or shrinks, it may be necessary to adjust the size of the underlying array to ensure good performance. This process involves creating a new, ⟳ Regenerate (or smaller) array, and re-inserting all the existing keys/values into it. This is an expen

```cpp
unordered_map<string, int> map; // Declare a hashmap
map["a"] = 10; map["b"] = 20; map["c"] = 30; // Insertions

if(map.count("a")) cout << "Key 'a' exists!" << endl; // Check if Key Exists

map.erase("b"); // Deletion

for(auto& pair : map) cout << pair.first << ": " << pair.second << endl; // I
```

Binary search

```cpp
std::pair<int, int> find_bounds(const std::vector<int>& vec, int value) {
    int left = 0;
    int right = vec.size() - 1;
    int mid;

    // Binary search loop
    while (left <= right) {
        mid = left + (right - left) / 2;

        if (vec[mid] == value) {
            return {mid, mid}; // Value found
        } else if (vec[mid] < value) {
            left = mid + 1; // Narrow search to right half
        } else {
            right = mid - 1; // Narrow search to left half
        }
    }

    // When value is not found
    int lower_bound = right;
    int upper_bound = left;

    return {lower_bound, upper_bound};
}
```

Assigning just `mid` to `right` or `left` would be incorrect in the context of a standard binary search for a target value within a sorted array. Here's why:

* If you set `right = mid` when `vec[mid] > value`, you are not excluding the midpoint itself from the search range, even though you know that the target value cannot be at the midpoint (because `vec[mid] > value`). This could lead to an infinite loop if the target value is not in the array.
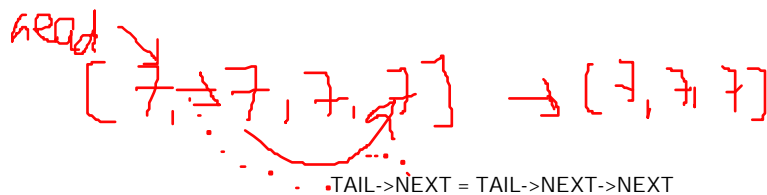
LinkedList

removeElements

1)

```
ListNode *dummy = new ListNode(0, head);
// tail->next = head;
ListNode *tail = dummy;
return dummy->next;
```

VS

```
ListNode *tail = new ListNode(0, head); return head
```

EDGE CASE : REMOVING FIRST ELEMENT

TAIL->NEXT = TAIL->NEXT->NEXT

2)

```
if (tail->next->val == val)
{
    // ListNode *nodeToRemove = tail->next;
    // tail->next = nodeToRemove->next;
    // delete nodeToRemove;
    tail->next = tail->next->next;   ①
}
// tail = tail->next->next;
// else
tail = tail->next;   ②
```

VS

```
if (tail->next->val == val)
{
    ListNode *nodeToRemove = tail->next;
    tail->next = nodeToRemove->next;
    delete nodeToRemove;
    // tail->next = tail->next->next;
}
// tail = tail->next->next;
else
    tail = tail->next;
```

nodeToRemove
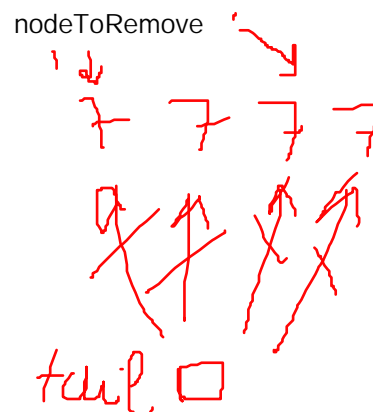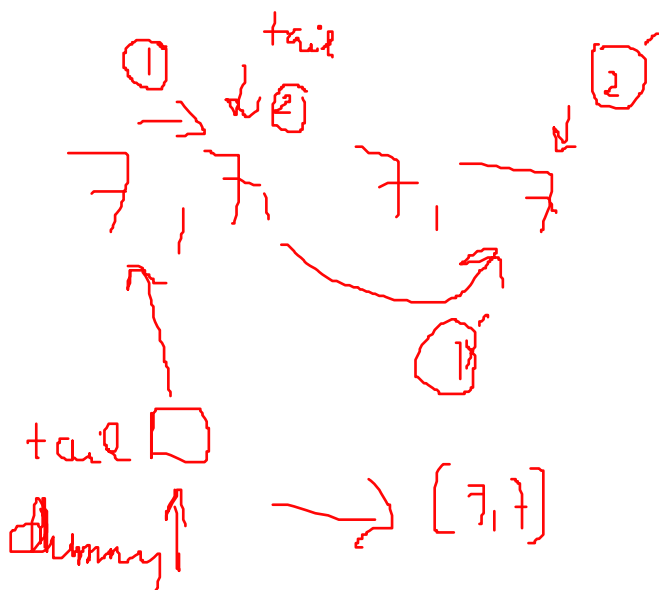
Backtracking

1. **Choose**: Select an option from the remaining choices.
2. **Explore**: Move forward with that choice and explore further options.
3. **Check**: If a solution is found, save it. If the current path leads to an invalid solution or if you've explored all possibilities, go to step 4.
4. **Backtrack**: Undo the last choice (or choices) and explore the next option in the previous step.
5. **Repeat**: Continue the process until all possible solutions have been explored.

In the subsets problem:

```cpp
void generateBinaryStrings(int n, vector<int>& current) {
    // Base Case: If the current binary string is of length n, print it
    if (current.size() == n) {
        for (int x : current) cout << x << ' ';
        cout << '\n';
        return;
    }

    // Choose, Explore, and Backtrack for 0 and 1
    for (int i = 0; i <= 1; i++) {
        // Choose: Add the current bit
        current.push_back(i);

        // Explore: Recursively call the function to explore further bits
        generateBinaryStrings(n, current);

        // Backtrack: Remove the last added bit to explore the next possibil
        current.pop_back();
    }
}
```

1. `(3, [])`
   1.1. `(3, [0])`
      1.1.1. `(3, [0, 0])`
         1.1.1.1. `(3, [0, 0, 0])`
         1.1.1.2. `(3, [0, 0, 1])`
      1.1.2. `(3, [0, 1])`
         1.1.2.1. `(3, [0, 1, 0])`
         1.1.2.2. `(3, [0, 1, 1])`
   1.2. `(3, [1])`
      1.2.1. `(3, [1, 0])`
         1.2.1.1. `(3, [1, 0, 0])`
         1.2.1.2. `(3, [1, 0, 1])`
      1.2.2. `(3, [1, 1])`
         1.2.2.1. `(3, [1, 1, 0])`
         1.2.2.2. `(3, [1, 1, 1])`

generateSubsets

main

generateSubsets()

subset

generateSubsets()

idx=0

idx=1

idx=2

idx=3

['1']

['1','2']

('1','2','3']

['1','2'] ['1','3'] ['1']

['1']

idx == nums.size()) V

$8 = 2^3$

idx=0, has '1'?

idx=1, has '2'?

idx=2, has '3'?

[1,2]

[1,2,3] [1,2] [1,3] [1] [2,3] [2] [3] []

idx=3
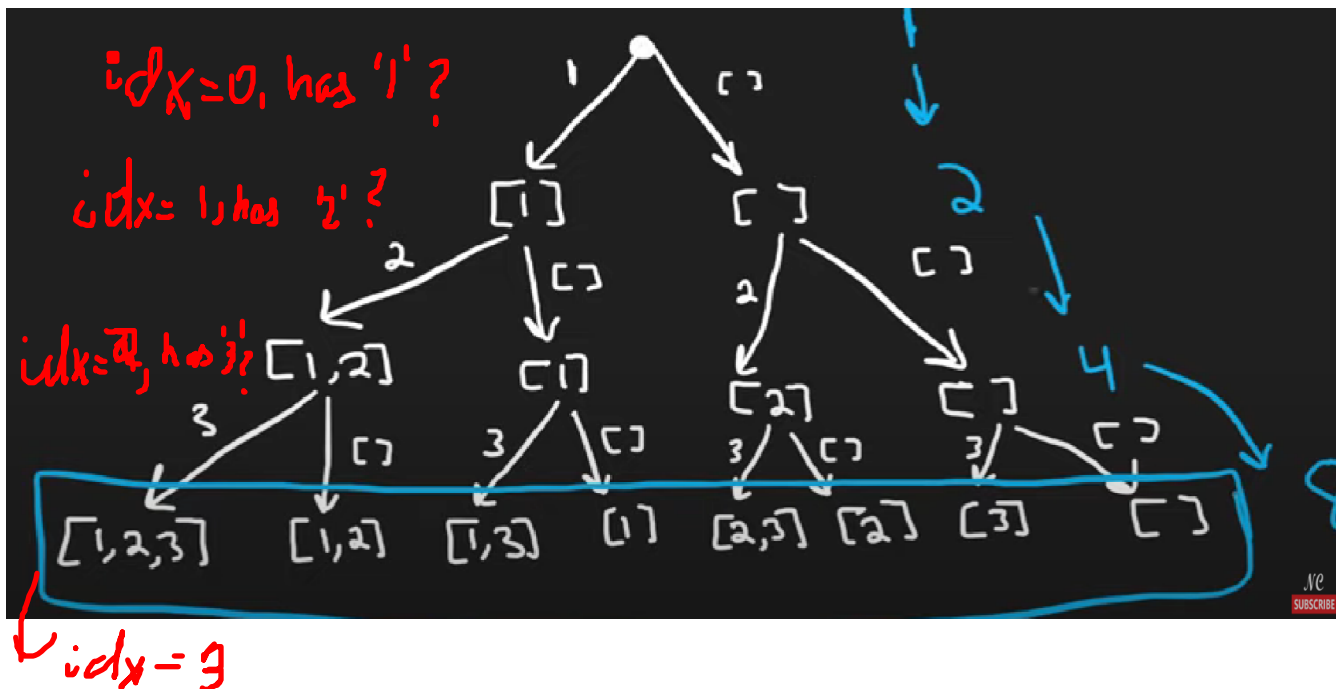
Dynamic Programming

Fibonacci


```
memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
return memo[n];
```

fib(6)

memo
{
→ 3: 2,
  4: 3
}

2n

fib memoized complexity
O(n) time
O(n) space

CanSum

m = target sum
n = array length

$O(n^m)$ time
$O(m)$ space

brute force
$O(n^m)$ time
$O(m)$ space

→

memoized
$O(m * n)$ time
$O(m)$ space

2 in fib

3) Implement Priority Queue Using Heaps:

Binary Heap is generally preferred for priority queue implementation because heaps provide better performance compared to arrays or LinkedList. Considering the properties of a heap, The entry with the largest key is on the top and can be removed immediately. It will, however, take time O(log n) to restore the heap property for the remaining keys. However if another entry is

Binary Heap	insert()	remove()	peek()
Time Complexity	O(log n)	O(log n)	O(1)

https://www.geeksforgeeks.org/priority-queue-using-binary-heap/

A priority queue can be implemented using a binary heap, and a binary heap can be efficiently represented as an array rather than using a tree with separate TreeNode objects.

Here's why using an array can be advantageous, with an example in C++ for implementing a max-heap as a priority queue:

1. **Space Efficiency**: Arrays eliminate the need for pointers, which can save space.
2. **Cache Locality**: Arrays provide better cache locality, making access faster.
3. **Easy Indexing**: In a binary heap, the parent, left child, and right child of any node can be found using simple arithmetic on the indices, making the operations efficient.