Dict implementation (HashTable)

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, (or smaller) array, and re-inserting all the existing keys/values into it. This is an exper

↻ Regenerate

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.
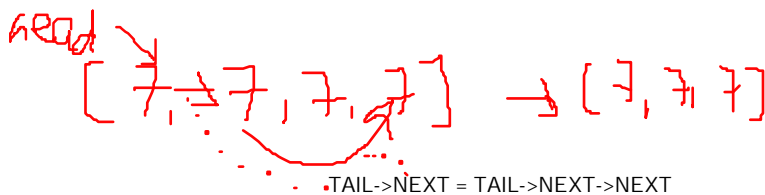
LinkedList

removeElements

1)

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

∨S

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

head

[7, 7, 7, 7] → [7, 7, 7]

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

EDGE CASE : REMOVING
FIRST ELEMENT

tail dummy

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
```

∨S

```
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;  3
```
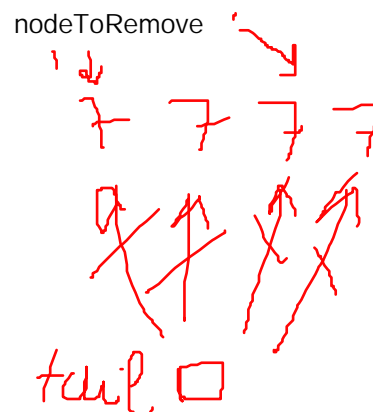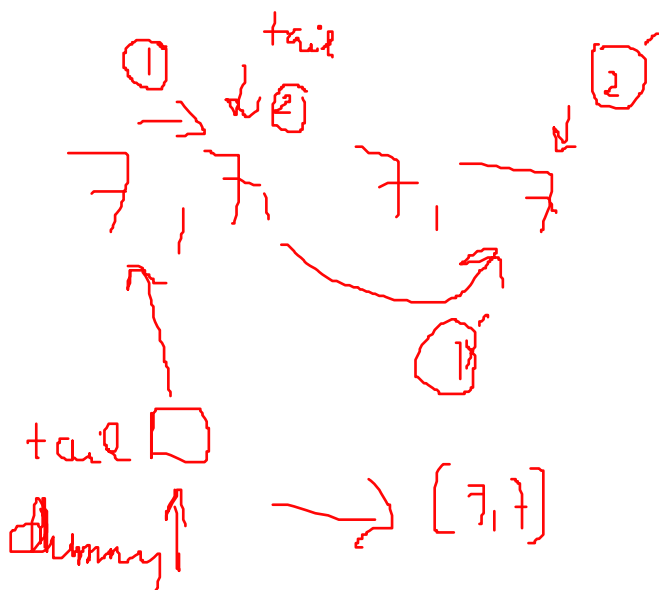
nodeToRemove

tail dummy → [7, 7]

tail
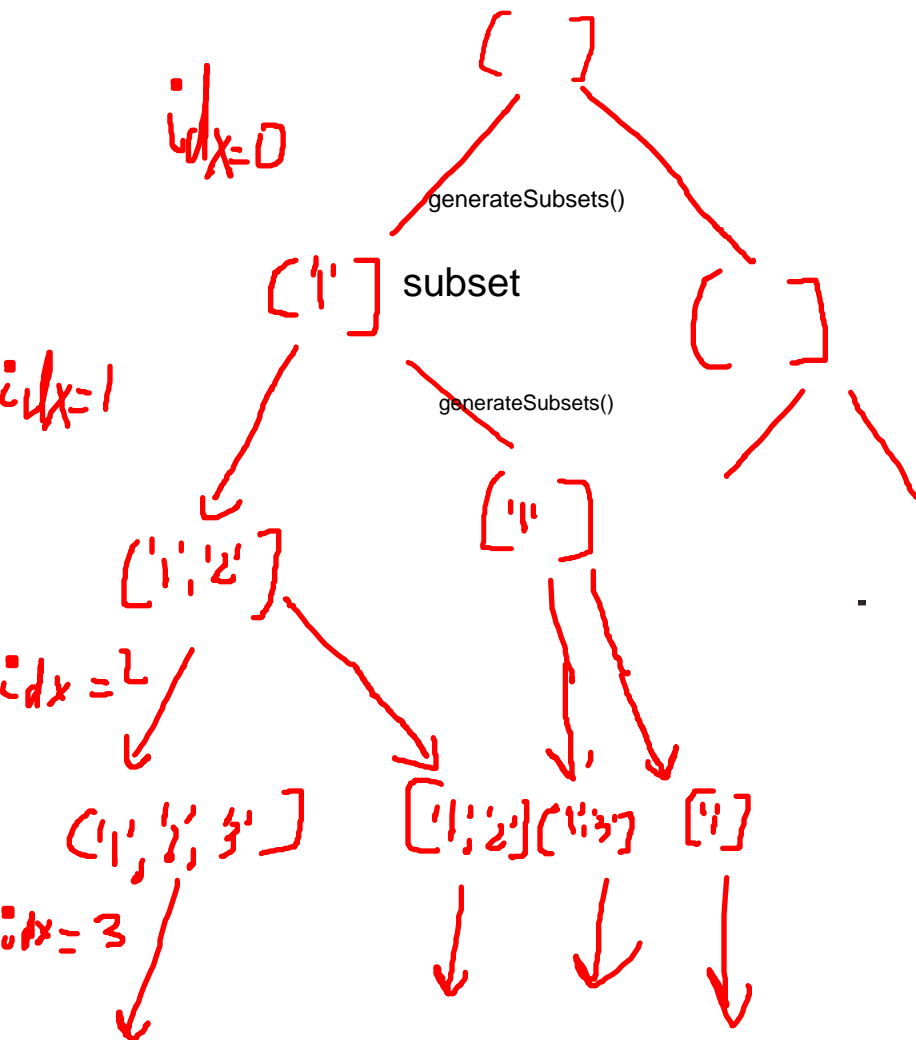
Backtracking

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])`
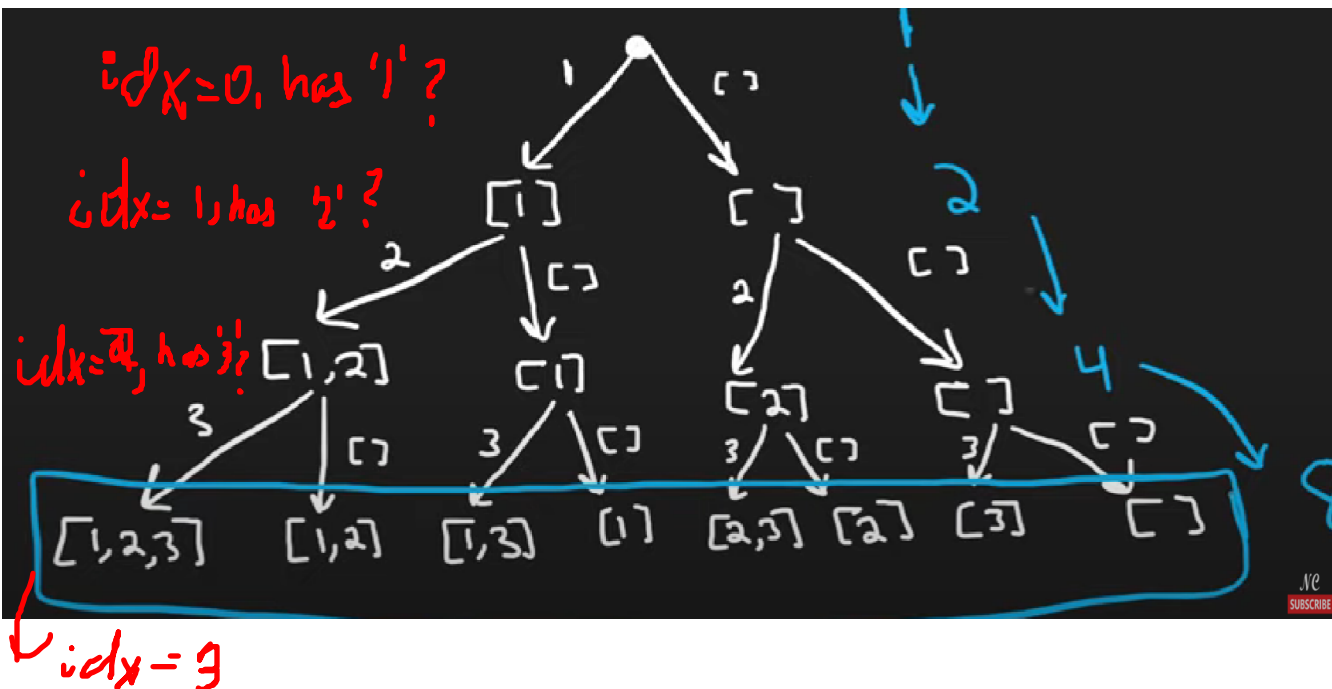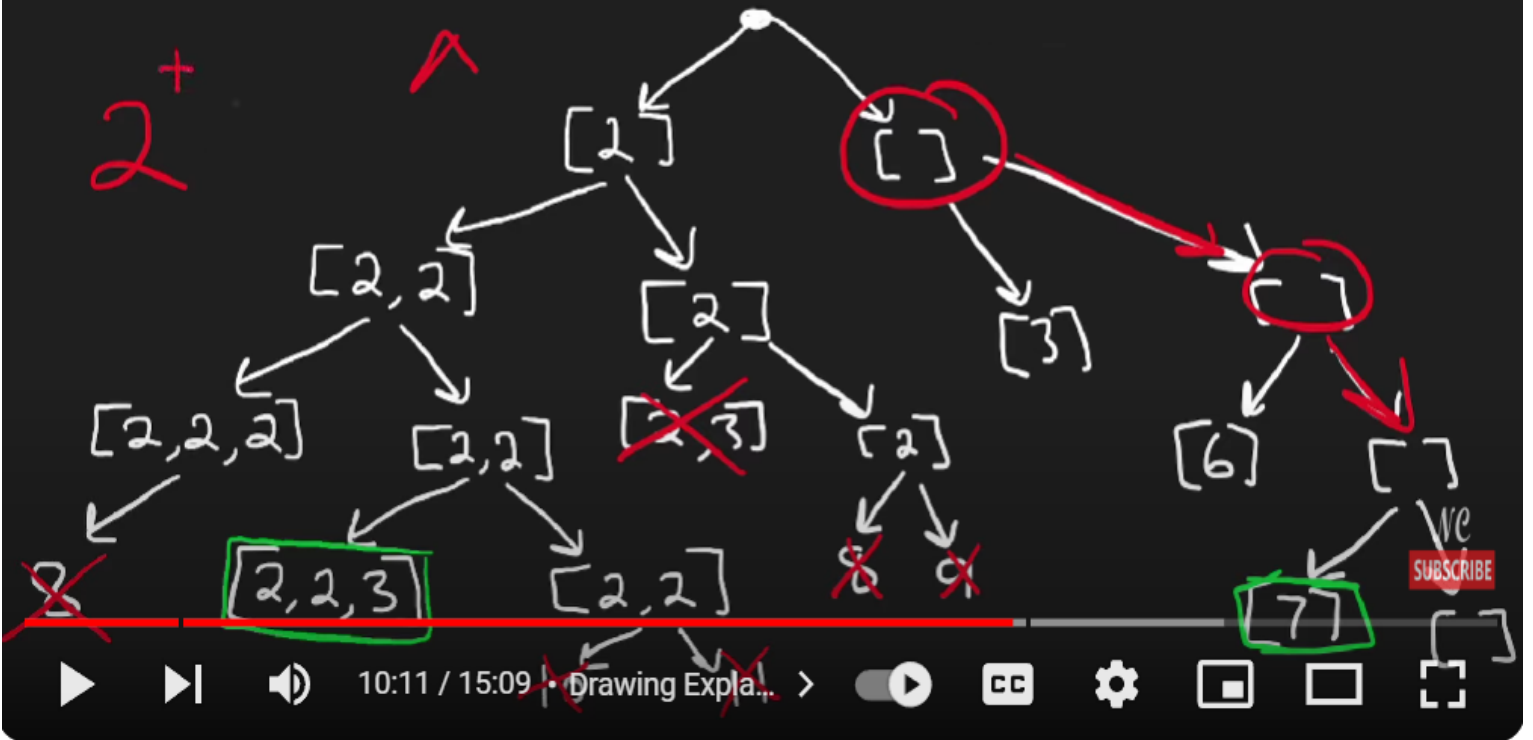
generateSubsets

main

generateSubsets()

subset

generateSubsets()

$idx=0$

$idx=1$

$['1']$

$['1','2']$

$idx=2$

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

$idx=3$

$['1']$

$idx == nums.size()$ V

1. `(0, [])`
   1.1. `(1, [1])`
      1.1.1. `(2, [1, 2])`
         1.1.1.1. `(3, [1, 2, 3])` V
         1.1.1.2. `(3, [1, 2])` V
      1.1.2. `(2, [1])`
         1.1.2.1. `(3, [1, 3])` V
         1.1.2.2. `(3, [1])` V
   1.2. `(1, [])`
      1.2.1. `(2, [2])`
         1.2.1.1. `(3, [2, 3])` V
         1.2.1.2. `(3, [2])` V
      1.2.2. `(2, [])`
         1.2.2.1. `(3, [3])` V
         1.2.2.2. `(3, [])` V

$8 = 2^3$

$idx=0$, has '1'?

$idx=1$, has '2'?

$idx=3$, has '3'?

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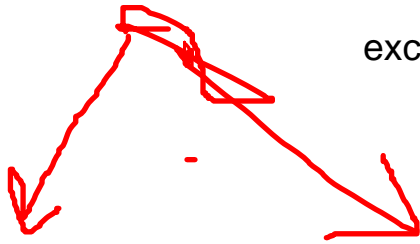[1,2,3]   [1,2]   [1,3]   [1]   [2,3]   [2]   [3]   [ ]

$idx=3$

Combination Sum - Backtracking - Leetcode 39 - Python

include i   exclude i

Dynamic Programming

Fibonacci


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

fib(6)

memo
{
→ 3: 2,
   4: 3
}

2 n

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

CanSum

m = target sum
n = array length

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

*n

m

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

→

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

2 in FiB

Priority Queue

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

	PUSH insert()	POP remove()	TOP peek()
Binary Heap			
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.

A binary tree can be efficiently represented using an array, where each element of the array represents a node in the tree. Here's an example of how you can do this:

1. The root is stored at index 0.
2. For any given node at position `i`, its:
   * Left Child will be at position `2*i + 1`
   * Right Child will be at position `2*i + 2`
   * Parent Node will be at position `(i-1)/2`

Heap

```cpp
class MaxHeap {
private:
    vector<int> heap; // Array representation of the heap

public:
    // Inserts a value into the heap
    void push(int value) {
        heap.push_back(value); // Add value to the end
        int i = heap.size() - 1;

        // Bubble up the new value to its proper position        logn
        while (i > 0 && heap[parent(i)] < heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    // Removes the max value from the heap
    void pop() {
        heap[0] = heap.back(); // Replace the root with the last value
        heap.pop_back();       // Remove the last value
        heapify(0);            // Heapify to restore max-heap property
    }

    // Returns the max value (root of the heap)
    int top() {
        return heap[0];
    }

private:
    // Returns the parent index of a given index
    int parent(int i) { return (i - 1) / 2; }

    // Returns the left child index of a given index
    int left(int i) { return 2 * i + 1; }

    // Returns the right child index of a given index
    int right(int i) { return 2 * i + 2; }

    // Recursive function to restore the max-heap property at a given index  // bubble down    logn
    void heapify(int i) {
        int largest = i;
        int l = left(i);
        int r = right(i);

        // Find the largest of the current node, left child, and right child
        if (l < heap.size() && heap[l] > heap[largest]) largest = l;
        if (r < heap.size() && heap[r] > heap[largest]) largest = r;

        // If the largest isn't the current node, swap and continue heapify;
        if (largest != i) {
            swap(heap[i], heap[largest]);
            heapify(largest);
        }
    }
};

int main() {
    MaxHeap pq;
    pq.push(20);
    pq.push(15);
    pq.push(10);
    cout << "Max: " << pq.top() << endl; // Output will be 20

    return 0;
}
```

```c
void heapify(int arr[], int N, int i)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}
```

```c
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```
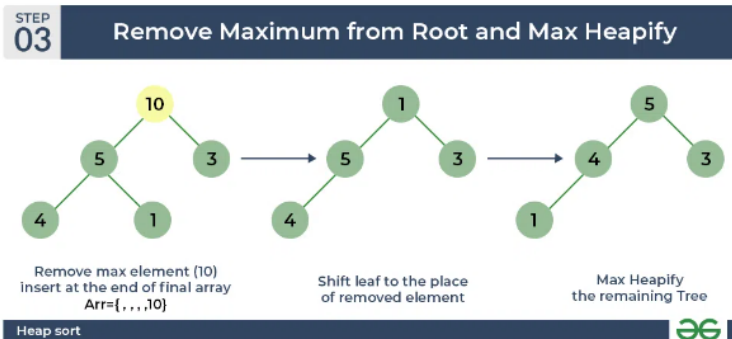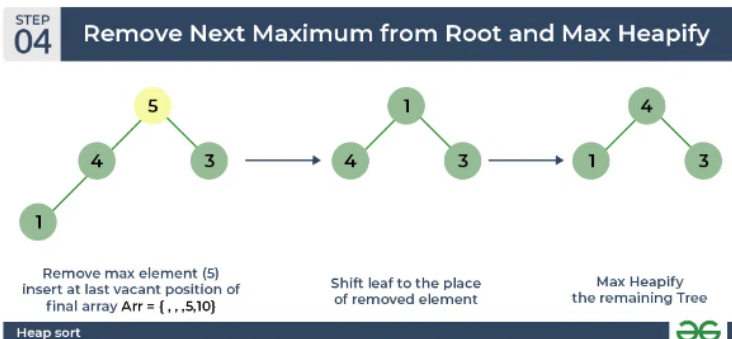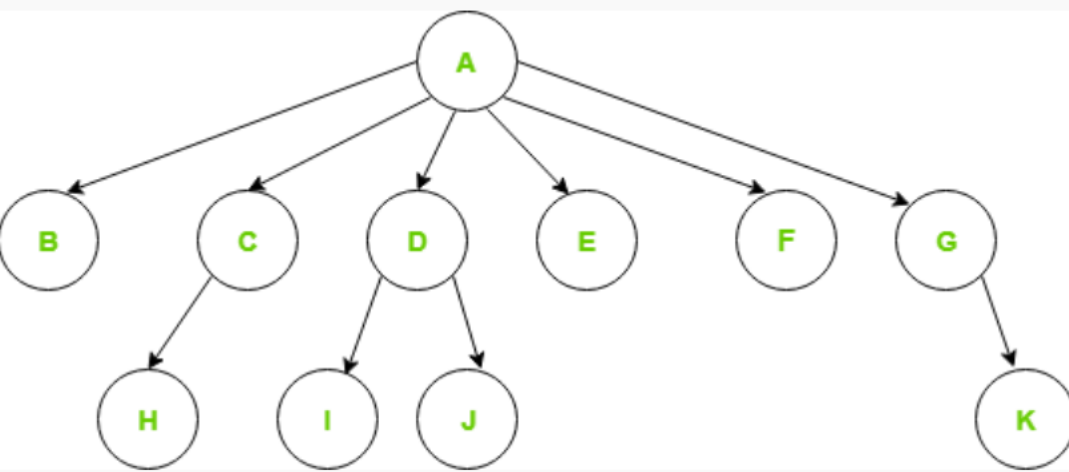

STEP 03 — Remove Maximum from Root and Max Heapify

Remove max element (10) insert at the end of final array Arr={ , , , ,10}

Shift leaf to the place of removed element

Max Heapify the remaining Tree

Heap sort

Heap sort algorithm | Remove maximum from root and max heapify

nd it will look like the following:

STEP 04 — Remove Next Maximum from Root and Max Heapify

Remove max element (5) insert at last vacant position of final array Arr = { , , ,5,10}

Shift leaf to the place of removed element

Max Heapify the remaining Tree

Heap sort

N-ary Trees
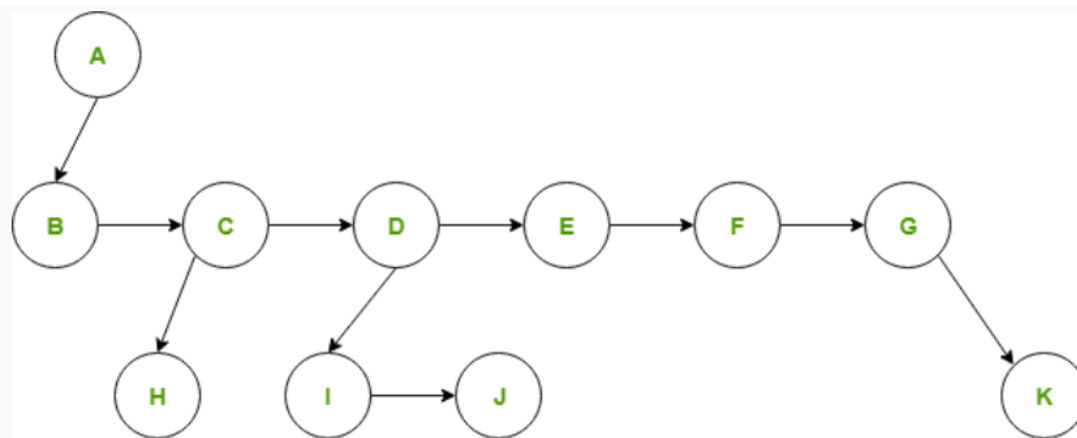

Generic Tree

```cpp
#include <vector>

class Node {
public:
    int data;
    std::vector<Node*> children;

    Node(int data)
    {
        this->data = data;
    }
};
```


FIRST CHILD/NEXT SIBLING REPRESENTATION

```cpp
struct Node {
    int data;
    Node *firstChild;
    Node *nextSibling;
};
```

Treated as binary trees – Since we are able to convert any generic tree to binary representation, we can treat all generic trees with a first child/next sibling representation as binary trees. Instead of left and right pointers, we just use firstChild and nextSibling.

Trie

https://www.geeksforgeeks.org/trie-memory-optimization-using-hash-map/