

## Table of Contents

<i>Introduction</i> .....	4
<i>Big O Notation</i> .....	6
<i>Abstraction (Low Level Abstraction)</i> .....	8
<i>Static &amp; Dynamic Arrays</i> .....	10
<i>Linked lists</i> .....	13
Singly Linked List.....	13
Doubly Linked List .....	14
Singly Linked List Inserting Operation .....	15
Doubly Linked List Inserting Operation .....	15
Singly Linked List Removing Operation.....	16
Doubly Linked List Removing Operation.....	16
<i>Stack</i> .....	18
Stack Pushing.....	19
Stack Popping.....	20
<i>Queue</i> .....	21
A Queue .....	22
Enqueue .....	22
Dequeue .....	23
Adding Queue .....	24
Removing Queue .....	24
<i>Priority Queue in Data Structure &amp; Algorithm</i> .....	26
Priority Queue & Its Instructions.....	28
What Is A Heap.....	29
<i>Priority Queue with Heap (Part 1)</i> .....	30
What Is Binary Heap .....	31
Binary Heap Representation .....	32
Binary Heap Representation .....	33
Adding Elements To Binary Heap .....	34
Removing Elements from Binary Heap (Polling).....	36
Removing Elements from Binary Heap .....	40

<b>Priority Queue With Heap (Part 2) .....</b>	<b>42</b>
Heap Data Presentation With Hash Lookup table .....	43
Inserting Data With Lookup Table.....	44
Removing Data With Lookup Table.....	47
Polling Data With Lookup Table.....	49
<b>Sorting.....</b>	<b>52</b>
<b>Union Find.....</b>	<b>53</b>
Find.....	53
Union.....	53
Union Find With Magnet Examples.....	54
Union Find With Path Compression .....	56
<b>Greedy Algorithm .....</b>	<b>58</b>
Divide and Conquer Algorithm.....	60
Divide.....	60
Conquer.....	60
Combine .....	60
<b>Dynamic Programming.....</b>	<b>62</b>
<b>What is a tree &amp; binary tree? .....</b>	<b>64</b>
<b>Binary tree's operations .....</b>	<b>66</b>
Insert Operation.....	66
Search Operation .....	67
Removing operation .....	68
Binary Tree's traversal .....	74
In-order Traversal.....	74
Pre-order Traversal.....	75
Post-order Traversal .....	76
Level Order Traversal .....	76
<b>Binary Search Algorithm .....</b>	<b>78</b>
<b>Hash table (hash function) .....</b>	<b>80</b>
<b>Hash table (separate chaining).....</b>	<b>83</b>
<b>Hash table (Open addressing) .....</b>	<b>87</b>
<b>Hash table - Open addressing (Linear Probing).....</b>	<b>89</b>
<b>Hash table - Open addressing (Quadratic Probing).....</b>	<b>92</b>

<b><i>Hash table - Open addressing (Double Hashing) .....</i></b>	<b>94</b>
<b><i>Hash table - Removing elements (open addressing).....</i></b>	<b>96</b>
<b><i>Fenwick tree (Binary Indexed Tree) .....</i></b>	<b>98</b>
<b><i>Fenwick tree (Point Update).....</i></b>	<b>101</b>
<b><i>Fenwick Tree Construction .....</i></b>	<b>103</b>
<b><i>AVL Tree.....</i></b>	<b>105</b>
<b>Tree Rotation Sample.....</b>	<b>107</b>
<b>Tree Rotation Example 1.....</b>	<b>107</b>
<b>Tree Rotation Example 2.....</b>	<b>108</b>
<b>Left Rotation.....</b>	<b>109</b>
<b>Right Rotation.....</b>	<b>110</b>
<b>Left Right Rotation .....</b>	<b>111</b>
<b>Right Left Rotation .....</b>	<b>112</b>
<b><i>References.....</i></b>	<b>114</b>

အခု series PDF လေးသည် ကျန်းမာရ်တိုင်လွှဲလာရင်းနဲ့ သိလာတာတွေ ရေးထားတာ  
ဖြစ်တဲ့အတွက် လိုအပ်တာလေးတွေပါကောင်းပါနေနိုင်သေးပါတယ်။

## Introduction

Applications တွေ system တွေက တစ်ဖြည့်ဖြည့်နဲ့ ကျယ်ပြန်လာတဲ့ အချိန်၊ data တွေများပြားလာတဲ့ အချိန်မှာ အဲဒီ များပြားလာတဲ့ data တွေကို manage လုပ်ဖို့ဆိုတာကလဲ challenging တစ်ခုဖြစ်လာပါတယ်။ manage လုပ်တယ်ဆိုတာထက် ဒီလို million , billion, trillion ချိတဲ့ data တွေကို performance ကောင်းစွာ ဘယ်လို handle လုပ်မလဲ process လုပ်မလဲဆိုတာက တစ်ကယ်ကို အရေးကြီးတဲ့ အပိုင်းတစ်ခုအနေနဲ့ ရှိလာပါတယ်။ ဒါကြောင့်မို့လဲ data structures and algorithms ဆိုပြီးတော့ ဖြစ်ပေါ်လာတာပါ။

Data structures and algorithms တွေကိုအသုံးပြုပြီးတော့ data တွေအများကြီးကို ကောင်းမွန်စွာ handle ပြုလုပ်လာနိုင်ပါတယ်။ ကျနော်တို့ လက်ရှိအသုံးပြုနေတဲ့ systems တွေ applications တွေအားလုံးက data structures တွေနဲ့ ဖွဲ့စည်းထားတဲ့ဖြစ်ပါတယ်။

Algorithms ဆိုတာက အရှင်းဆုံးပြောရရင် step by step procedure ပဲ။ operation တစ်ခု run နိုင်ဖို့အတွက် တစ်ဆင့်ခြင်းဆီ instructions တွေနဲ့ တည်ဆောက်ထားတဲ့ သဘောတရားပါပဲ။ ဥပမာ ကျနော် facebook ပေါ်article တွေတင်တာကို algorithm အနေနဲ့ ပြောရရင်

1. ကိုယ်ရေးချင်တဲ့စာကို သေချာအောင် research လုပ်တယ်။
2. Short notes တွေပြန်ထုတ်တယ်။
3. စာစီတယ် । စာရေးတယ်
4. အမှားစစ်တယ်
5. Facebook ပေါ်တင်တယ်။ ဒါဆို ရင် facebook ပေါ်ကျနော် article ရောက်သွားဖို့ အတွက် ဒီ instructions တွေကို တစ်ဆင့်ခြင်းဆီ ကျနော်လုပ်ပါတယ်။ ခြင်းချက်ရှိတာကတော့ အမြဲတမ်း step by step ကြီးသွားနေတာမျိုးလဲမဟုတ်ဘူး၊ တစ်ခါတစ်လေကျ problem concept ပေါ်မှုတည်ပြီး ပြောင်းလဲရတာလဲရှိနိုင်ပါတယ်။

Algorithm တွေကို ဘယ် programming နဲ့မှ ရေးလို့ရမယ်ဆိုပြီး ကန့်သတ်ထားတာမရှိဘူး၊ programming တိုင်းမှာ fundamental code construction တွေပါပြီးသားဖြစ်တဲ့အတွက် ဘာနဲ့ဖြစ်ဖြစ်ဆောက်လို့ရပါတယ်။

Algorithm တွေရဲ complexity ကိုကြည့်မယ်ဆို ၂ မျိုးခဲ့ခြားထားလို့ရတယ်။ - Number of operation ပေါ်မှုတည်ထားတဲ့ time complexity နဲ့ - Memory space ပေါ်မှုတည်ထားတဲ့ space complexity ဆိုပြီးတော့ပါ။ Time complexity ကတော့ algorithm တစ်ခု run ပြီဆို operation ဘယ်နှစ်ကြိမ်လုပ်သွားလဲဆိုတာပေါ်မှုတည်ပြီးတွက်တယ်။ space complexity ဆိုတာကတော့ algorithm runtime မှာ system ရဲ့ memory အသုံးပြုပုံမှုတည်ပြီးတွက်ပါတယ်။

နောက် article မှာ Big O နဲ့အတူ complexity အကြောင်းထပ်ရေးပေးထားပါတယ်။

## Big O Notation

Big O Notation ဆိုတာက program/algorithm တစ်ခုရဲ့ performance run time ကိုဆိုလိုတာပါ။ ဒီ algorithm တစ်ခုက ဘယ်လောက်မြန်လဲ၊ ဘယ်လောက်နေးလဲစသည်ဖြင့်ပေါ့။ တစ်နည်းအားဖြင့် ပြောရမယ်ဆိုရင် algorithm တစ်ခုရဲ့ခက်ခဲမှု (complexity) လိုလဲပြောလိုရပါတယ်။ ဒီနေ့မှာ ကျနော် Big O notation ကို နားလည်ရလွယ်အောင် real world example တွေနဲ့နှင့်ယူဉ်ပြီး ပြောပြပေးသွားပါမယ်။

စာကြည့်တိုက်ကြီးတစ်ခုရှိတယ်ဆိုပါစို့၊ စာအုပ်တွေလည်း တော်တော်စုတယ်။ စာအုပ်ပေါင်း အမျိုးအစား အားလုံးပေါင်းလိုက်ရင် 1million လောက်တောင် ရှိတယ်ပဲထားလိုက်ရအောင်။ ဒါပေမဲ့ စာကြည့်တိုက်မှာ စာအုပ်လာတွားတဲ့ လူက ဘယ်စာအုပ်လေးလိုချင်ပါတယ် ဆိုပြီး နာမည်တပ်ပြောလိုက်ရင် အဲ စာအုပ်ကို လိုက်ရာဖို့က အချိန်တော်တော်ယူနေရတယ်။ ဒါနဲ့ပဲ စာအုပ်တွေကို ရှာမယ့် searching algorithm တစ်ခုရေးဖို့စဉ်းစားကြတယ်၊ မြန်နိုင်သမှု မြန်မြန်လိုအပ်တဲ့စာအုပ်ကို ရရှိနိုင်ဖို့ပေါ့။ algorithm မှာ နှစ်ခုရှိတယ်၊ simple search နဲ့ binary search ဆိုပြီး (main topic က Big O အကြောင်းဆိုတော့ binary search ကိုမသိလဲရပါတယ်)။ Algorithm နှစ်ခုလုံး က တိကျပြီး မြန်မြန်ဆန်ဆန် နဲ့ ထွက်လာဖို့လိုပါတယ်။ binary search ကတော့ ပိုမြန်တယ်။

Algorithm တွေရဲ့ run time ကို သိဖို့အတွက် စာအုပ် ၁၀၀ ပေါ်မှာ အခြေခံပြီးတွက်ကြည့်ရအောင်။ စာအုပ်တစ်အုပ် ကို 1millisecond (1000ms = 1 sec) ကြားတယ်လို့ define လုပ်လိုက်ရအောင်။ simple search က စာအုပ် ၁၀၀ လုံးအတွက် လိုက်စစ်စရာလိုတဲ့ အတွက် simple search နဲ့ ရှာမယ်ဆို စာအုပ် ၁၀၀ အတွက် 100ms ကြားပါမယ်။ binary search နဲ့ဆို စာအုပ် ၁၀၀ မှာ လိုချင်တဲ့ result ထွက်လာဖို့ အတွက် စာအုပ် ၇ အုပ်စာလောက်ပဲ စစ်ရပါတယ်။ binary search ရဲ့ algorithm အရ ( $\log_2 100 \approx 7$  လောက်ရပါတယ်)။ ဆိုတဲ့အတွက် ကြောင့် binary search နဲ့ run မယ်ဆို 7ms ပဲကြားပါမယ်။ တစ်ကယ်တော့ ၁၀၀ ပဲ run ရမှာ မဟုတ်ဘူး။ 1million record ကို run ရမှာပါ။

Record 100 base အရ ထွက်လာတဲ့ run time (milliseconds) ပေါ်မှုတည်ပြီး binary search က simple search ထက်စာရင် 14 ဆလောက်မြန်တယ်လို့မှတ်ယူထားပါတယ်။ Binary search နဲ့ record တစ်သန်းကို run လိုက်တဲ့ အချိန်မှာ 20sec လောက်ကြားပါတယ်။ simple search နဲ့ဆို 14 ဆိုတော့ 280sec လောက်ကြားမယ်လို့ခန့်မှန်းပါတယ်။

အဲမှာ စမှား တာပါ တစ်ကယ်တော့ simple search နဲ့ဆို 16 mins ကျော်ကွာသွားမှာပါ။  
 ဘာလို့လဲဆိုတော့ algorithm နှစ်ခု က rate ခြင်းမတူကြပါဘူး။ ဆိုလိုတာက input size  
 များလာတာနဲ့အမျှ run time လဲပိုပြီးကွာသွားမှာပါ။ 14 ဆိုမြန်တယ်ဆိုပြီး ပုံသေတွက်လို့မရပါဘူး။  
 ဆိုလိုချင်တာက ဒါက record 1million ပဲရှိသေးတယ်။ 1 billion ဆို binary search နဲ့ဆို 30sec  
 လောက်ပဲကြာပြီး simple search နဲ့ဆို 11 ရက်လောက်ကြာမှာပါ။ rate ကလည်း 33million  
 လောက်ပိုမြန်မှာပါ။

Big O ရဲ့အဓိက ဆိုလိုရင်းက လည်း အဲဒါပါပဲ algorithm တစ်ခုရဲ့ run time ကိုပြတယ်။ ဒါပေမဲ့  
 အချိန် တွေ seconds တွေနဲ့ မဟုတ်ပဲနဲ့ operation နဲ့ပြတယ်။ simple search algorithm နဲ့ဆို  
 အုပ်ရေး၁၀၀ မှာ အခေါက် ၁၀၀ run ရပေမဲ့ binary search algorithm နဲ့ဆို ၂ ခေါက် (7  
 operations) ပဲ run ရမယ်။ input size ကြီးလာတာနဲ့ အမျှ သူတို့ကြားထဲက ကွာခြားချက်ကလဲ  
 ကြီးလာမှာပဲ။ ဘယ်လောက် ကွာခြားသွားလဲဆိုတာကို Big O notation က formula တွေနဲ့ define  
 လုပ်ပေးတယ်။

သတိချပ်စရာရှိတာက Big O calculation တွက်ပြီးဆို wrost case တွေအထိစဉ်းစားပြီး  
 တွက်ရတယ်။ ဥပမာ record 100 မှာရှာမယ်ဆို ကိုယ်ရှာတဲ့ key ကို ပထမဆုံးတင်တွေ့တယ်ဆိုပါစို့။  
 algorithm က entry တိုင်းကို သွားဖတ်စရာမလိုတော့တဲ့အတွက် ကိုယ့် algorithm က n time လား  
 1 time လားဆိုတာ သတိပြန်ချပ်သင့်ပါတယ်။ entry အကုန်လုံးကို run မှု Big O of n ရလာမှာပါ။  
 ဒါကြောင့် လည်း အဆိုးဆုံး worst case အထိစဉ်းစားခိုင်းတာပါ။

Big O ရဲ့ run times တွေ အမျိုးမျိုးရှိတယ်။ အသုံးများတာတွေကို အောက်မှာ ရေးပေးလိုက်တယ်။  
 fastest to slowest order နဲ့။

$O(\log n)$ , log time (eg binary search)  $O(n)$ , linear time (eg simple search)  $O(n * \log n)$ ,  
 (fast sorting algorithm)  $O(n^2)$ , (slow sorting algorithm)  $O(n!)$  (pretty slow algorithm)

Run time တွေကို စမ်းကြည့်ချင်တယ်ဆို ကိုယ့်ဘာသာ calculator နဲ့တွက်ပြီး စမ်းကြည့်လို့ရတယ်။  
 ဥပမာ 1billion records မှာ log time နဲ့ဆိုဘယ်လောက်ကြာမလဲ၊ linear နဲ့ဆိုဘယ်လောက်လဲ  
 စသည်ဖြင့်ပေါ့။

## Abstraction (Low Level Abstraction)

CS50 က Professor David J. Malan ပြောတဲ့ presentation တစ်ခုနားထောင်နေမိရင်းနဲ့  
သဘောကျလို့ ကိုယ်နားလည်သလောက် ဘာသာပြန်ရင်း sharing ပြန်လုပ်ပေးလိုက်ပါတယ်။  
ပြောသွားတဲ့ အကြောင်းအရာ ကတော့ Abstraction အကြောင်းကို ပြောသွားတာ။

Abstraction ဆိုတာ က ရှုပ်ထွေးတဲ့ အရာတစ်ခုကနေပြီးတော့ တစ်ဆင့်ခြင်းတစ်ဆင့်ခြင်းလွယ်ကူတဲ့  
အခြေအနေတစ်ခု (နားလည်ရလွယ်ကူနိုင်တဲ့ အခြေအနေ) တစ်ခု အဖြစ် ခွဲထုတ်လိုက်တာမျိုး  
ကိုဆိုလိုတာပါ။ ဥပမာ တစ်ခု အနေနဲ့ပြောရမယ်ဆိုရင် ကားတစ်စီး ဘယ်လိုအလုပ်လုပ်လဲဆိုတာ  
အသေးစိတ် ကျနော်တို့မသိဘူး၊ ဒီကားဘီး ကြီးကို ဘယ်လိုတွေလုပ်ထားလဲ၊ engine ကိုရော  
ဘယ်လိုအစိတ်အပိုင်းတွေနဲ့ ဖွဲ့စည်းထားလဲ၊ ဒီကားတစ်ခုခဲ့ ပျက်သွားရင်ရော ဘယ်လိုပြင်မလဲ  
စသည်ဖြင့် အသေးစိတ်ကို ကျနော်တို့ မသိဘူး။ ဒါပေမဲ့ ဒီကားကို ကျနော်တို့ operate လုပ်တတ်တယ်၊  
ဆိုလိုချင်တာ က ကားကို စက်နှီးတတ်တယ်၊ မောင်းတတ်တယ်၊ ရပ်မယ်ဆို ဘာကို နင်းရမယ်၊  
ဒီကားကို energy ဘယ်လို ပြန်ဖြည့်ရမယ်စသည်ဖြင့် ကျနော်တို့ Interfacing ပိုင်းကို  
ကောင်းကောင်းသိပြီး ကားကို အသုံးပြုနိုင်တယ်။ ဘာကြောင့်လဲ ဆိုတော့ abstract လုပ်လိုက်လို့ပဲ၊  
ခက်ခက်ခဲ့ ပိုင်းတွေကို အသေးစိတ် မစဉ်းစားတော့ ဘဲ ကားတစ်စီး ကို operate လုပ်နိုင်မယ့်  
Interface အပိုင်းကိုပဲ abstract လုပ်လိုက်တဲ့ အတွက် ကျနော်တို့ ကားကို လွယ်လင့်တစ်ကူ  
အသုံးပြုလို့ရသွားပါတယ်။

နောက်ထပ် ဥပမာတစ်ခု အနေနဲ့ပြောရရင် what is a country ဆိုပြီး filter လုပ်ကြည့်မယ်ဆို  
country ဆိုတာ states & divisions တွေစားတာ၊ states & divisions တွေဆိုတာ မြို့၊  
နယ်တွေစုံတာ၊ မြို့နယ်တွေ ဆိုတာ ရပ်ကွက်တွေ၊ ရပ်ကွက်တွေဆိုတာ အိမ်တွေစုံတာ၊  
အိမ်ဆိုတာ နံရုံတွေပါတယ် အမိုးတွေပါတယ်၊ နံရုံတွေဆိုတာ သစ်သားတို့ အုတ်တို့နဲ့ လုပ်တာ  
စသည်ဖြင့် တစ်ဆင့် ခြင်း ဆီ filter လုပ်သွားလို့ရပါတယ်။ ဒါပေမဲ့ ကျနော်တို့တွေက ဒီလိုတွေ  
အောက်ခြေ အစကနေ ပြီးတော့ construct လုပ်ထားတဲ့ အကြောင်းအရာတွေသိစရာမလိုဘူး။  
နိုင်ငံလို့ပြောလိုက်ပြီဆိုလဲ နားလည်တယ်။ ရန်ကုန်တိုင်းလို့ပြောလိုက်ပြီ ဆိုလဲနားလည်ပါတယ်။

ထိုနည်းလည်းကောင်းပဲ computer မှာဆိုလိုရှိရင်လဲ ကျနော်တို့ resource အနေနဲ့သူကို ပေးရတာ  
power ပဲ ရှိတယ်။ ကျနော်တို့ ဒီ computer ကို Input အနေနဲ့ electricity တစ်ခုပဲ ပေးရှုနဲ့  
နေစဉ်ကျနော်တို့လုပ်နေတဲ့ program ရေးတာတို့ design ဆွဲတာတို့ စသည်ဖြင့် complex ဖြစ်တဲ့

task တွေကို ဘယ်လိုလုပ်သွားလဲပေါ့။ ကျနော်တို့သိတော်တာ က computer တွေက binary system နဲ့ အလုပ်လုပ်တယ်။ အိုကေ ဒါဆိုရင် Microsoft word, note software တွေမှာ HI ဆိုတဲ့ message ကို ရနိုင်ဖို့ decompose လုပ်မယ်ဆိုရင် binary system အရ အလုပ်လုပ်သွားတာက 01001000, 01001001။ အဲဒါကို decimal system ကို convert လုပ်မယ်ဆို 72, 73 ရတယ်။ ဒီ decimal conversion အရဆိုလိုရှိရင် Microsoft word တို့ notes တို့လို software တွေမှာ HI ဆိုတဲ့ message ရတယ်။ Graphic software တွေ browser တွေမှာဆိုရင် 72 က အနီရောင်၊ 73 က အစိမ်းရောင်စသည်ဖြင့် သတ်မှတ်ထားပြီး mapping လုပ်ထားကြတယ်။ ဒါပေမဲ့လည်း ကျနော်တို့ ဒါတွေသိစရာမလိုဘူး HI ကဘယ်လို ဖြစ်လာတာလဲ color တွေဘယ်လိုထွက်လာလဲ စသည်ဖြင့် သိစရာမလိုဘူး။ ကျနော်တို့က တစ်ကယ့် work done ဖြစ်တာကိုပဲလိုချင်တဲ့ အတွက် low level abstraction လုပ်ပြီးတော့ အသုံးပြုနေကြတာပါ။

ဆိုတော့ abstraction က ဘယ်လောက်အရေးပါတယ် ဆိုတာတော့ သိလောက်ပြီထင်ပါတယ်။ Abstraction မရှိရင်လဲ ကျနော်တို့ရဲ့ daily operation တွေမှာ တော်တော်အခက်အခဲ တွေ့မှာ ကျိုန်းသေပါတယ်။

ဒါပေမဲ့ လည်း computer science ကို လေ့လာတဲ့သူတွေအတွက် abstraction ဘယ်လိုလုပ်သွားလဲဆိုတာ အရေးပါတဲ့ အကြောင်းကို stage presentation တစ်ခုနဲ့ prove လုပ်ပြသွားတယ်။ ဘာလိုလဲဆိုတော့ stage အောက်က လူတွေကို ကုပ်တုံးပဲဆွဲခိုင်းတာ ဒါပေမဲ့ကုပ်တုံးလို့ အစက မပြောဘူး၊ မျဉ်း တစ်ကြောင်းဆွဲပါ၊ ပြီးရင် ညာဘက်ကိုဆက်ဆွဲ၊ ဘယ်လို ဘယ်ပုံ ဆိုပြီး တော့ တစ်ဆင့်ခြင်းဆီ instruction ပေးသွားတယ်။ Instruction ပြီးသွားတဲ့ အချိန်မှာ အကုန်လုံးလိုလို က ကုပ်တုံးလိုလို ပုံစံတွေထွက်တယ်၊ တစ်ချို့ဆို ကွက်တိပုံအတိုင်းထွက်တယ်။

ဒီ scenario က ဘာကိုဆိုလိုချင်တာလဲဆိုတော့ ကုပ်တုံးတစ်ခုကို ဘယ်လိုဆွဲရမလဲဆိုတဲ့ ဆွဲပုံဆွဲနည်း (high level instruction) ကို မသိဘူးဆိုရင် ကိုယ်က လုံးဝ ground up အစကနေ ဆွဲရမှာ ဖြစ်ပါတယ်။ အဲအတွက်ကြောင့် ကုပ်တုံးတစ်ခုကိုဘယ်လို ဆွဲသွားတယ်၊ low level အထိ ဘယ်လို abstract လုပ်သွားလဲ ဆိုတာကို နားလည်သွားလိုရှိရင် ဆွဲနိုင်သွားမှာ ဖြစ်ပါတယ်။

ထိုနည်းလည်းကောင်းပဲ computer science major မှာလည်း system တွေက low level abstraction အထိ ဘယ်လိုလုပ်သွားတယ်ဆိုတဲ့ အချက်က သိဖို့လိုအပ်တယ်၊ အရေးကြီးတယ်ဆိုတဲ့ အကြောင်းပြောရင်း conclude လုပ်သွားပါတယ်။

## Static & Dynamic Arrays

Static တွေ dynamic တွေ မပြောခင် မှာ array အကြောင်းကို အရင်ပြောရအောင်။ array ကတော့ အားလုံးနဲ့ လည်း မစိမ်းလောက်ဘူး၊ data value တွေကို collection လုပ်ထားတဲ့ sequence လိုလဲပြောလို့ရတယ်။ ဆိုလိုချင်တာ က data တွေ store လုပ်ဖို့ပြန်ပြီး retrieve လုပ်ဖို့အတွက် အဆင်ပြေ အောင်ဖန်တီးထားတဲ့ data structure တစ်ခုလို့ဆိုနိုင်တယ်။ array တစ်ခုမှာ element ရှိတယ်၊ index ရှိတယ်။ element ဆိုတာကတော့ array ထဲမှာ သိမ်းထားတဲ့ data value ၏ အဲ့ array ထဲမှာ သိမ်းထားတဲ့ element တစ်ခုခြင်းရဲ့ memory ထဲမှာ numeric key တစ်ခု တွဲပေးထားတယ်။ အဲ့ဒါ key ကို index လို့ခေါ်တယ်။ element value ကို index key သုံးပြီး လှမ်းဆွဲထုတ်လို့ရတယ်။

Array နဲ့ မသိမ်းဘဲ value တစ်ခုခြင်းဆို ကို variable တစ်ခုပေးပြီးလည်း သိမ်းလို့ရတယ်။ ဒါပေမဲ့ value တွေများလာတဲ့ အခါ တစ်ခုခြင်းဆိုကို variable set လုပ်ပေးဖို့လည်း မလွယ်ဘူး၊ space လည်း ပိုယူတယ်။ တစ်ဖက်က coding ပိုင်းက ကြည့်မယ်ဆိုရင်လည်း data တွေ manage လုပ်ရတာ အဆင်မပြေတော့ဘူး။ array ထဲကို value တွေအများကြီး ထည့် set လုပ်လိုက်ခြင်းအားဖြင့် declare လုပ်ရတာ က လည်း တစ်ခါပဲ လုပ်ရမယ်။ ပြီးရင် data တွေ ပြန်ဆွဲထုတ် တဲ့ နေရာမှာလည်း index key နဲ့ အဆင်ပြေပြေဆွဲထုတ်လို့ရတယ်။

Static array ကတော့ fixed length ဖြစ်တယ်၊ ပြောချင်တာက array အခန်း င့် ခု ဆို ငါးခုကိုပဲ memory မှာ preset လုပ်ထားလိုက်တယ်။ value တွေဆွဲထုတ်တဲ့နေရာမှာ ထုံးစံအတိုင်းပဲ index key သုံးပြီးဆွဲထုတ်တယ်။ static array မှာ element value တွေကို manage လုပ်လို့ မရဘူးတော့ မဟုတ်ဘူး၊ လုပ်လို့ရတယ်။ ဒါပေမဲ့သူအလုပ်လုပ်တဲ့ ပုံစံကာ၊ ဆိုပါတော့ element value င့် ခုပါတဲ့ array တစ်ခုရှိတယ်၊ နောက်ထပ် value တစ်ခုထပ်ထည့်ချင်တယ်ဆိုရင် fixed length ဖြစ်နေတဲ့အတွက်ကြောင့် direct သွားထည့်လို့မရဘူး၊ memory ပေါ်ကို static array အသစ် တစ်ခုထပ်ကြည်ာတယ်၊ ထပ်ထည့်ချင်တဲ့ value အတွက် array တစ်ခန်းအပိုနဲ့ပေါ့၊ ပြီးတော့ မှ အရင် array ကို copy paste လာလုပ်တယ်၊ နောက်ထပ် value တစ်ခု ကို တစ်ခါတည်း ထပ်ထည့်တယ်၊ အဲ့ဒါ process ပြီးမှ previous array ကို remove လုပ်တယ်။ ဆိုလိုချင်တာက static array ပေါ်မှာ element value တွေကို manage လုပ်လာပြီဆို များလာတဲ့အတွက် မကောင်းဘူး။ အဲ့ဒါ issue ကိုဖြေရှင်းနိုင်ဖို့အတွက် dynamic array ကို သုံးလာကြတယ်။

Dynamic array ကလည်းစစချင်းတော့ static array လိုပဲ array တစ်ခု create လုပ်ရတာပဲ။ ဒါပေမဲ့ သူက assign လုပ်ထားတဲ့ element value တွေကို manage လုပ်နိုင်တယ်။ သူ manage လုပ်ပုံက array ငါန်းရှိတယ်။ assign လုပ်ထားတဲ့ element တွေကို ငါ ခုလုံးရှိတယ်။ အခန်းလွတ်မရှိဘူး။ နောက်တစ်ခုထပ်ထည့်ချင်တယ်ဆို နေရာလွတ်မရှိတော့တဲ့ အတွက် လက်ရှိရှိပြီးသား array size ရဲ့ နှစ်ဆယူပြီး အရင် array ကို copy လုပ်တယ်။ element တွေထပ်ထည့်တယ်။ array resize လုပ်တယ်လိုလည်း ခေါ်တယ်။ array အခန်းပြည့်ပြီး နောက်ထပ် value assign လုပ်ချင်တဲ့ အချိန်ရယ်၊ assign value တွေ remove လုပ်ရင်း array အခန်းလွတ်တွေများပြီး space တွေယူနေတယ်လို့ ယူဆရတဲ့ အချိန်တွေရယ် resize လုပ်ပါတယ်။

### Static array ကတော့

- လိုအပ်တဲ့ memory amount ကိုပဲသုံးပြီး data manage လုပ်တဲ့ နေရာမှာကောင်းတယ်။
- Dynamic array ထက်စာရင် element တွေ access လုပ်ရတာပုံမြန်တယ်။
- Data တွေ manage operation (CRUD) လုပ်ရမယ်ဆို resource waste ဖြစ်တယ်။
- Fixed size ဖြစ်ပြီးတော့ 1 array စဆောက်တဲ့ အချိန်မှာ assign value တွေမပါသေးရင်တော်မှ မျှော်ဆောင်မှု ယူထားရတယ်။

### dynamic array ကတော့

- data တွေ ခဏခဏ manage လုပ်တဲ့မယ်ဆို အသုံးဝင်တယ်။
- size က flexible ဖြစ်တယ်။
- array resize လုပ်တဲ့ အချိန်မှာ လိုအပ်တဲ့ resources တွေကို runtime မှာပဲ တစ်ခါတည်းသုံးသွားတဲ့ အတွက် resources တွေ waste မဖြစ်တော့ဘူး။
- static array နဲ့ယုံ့ရင်တော့ element တွေ access လုပ်ရတာပုံနေးတယ်။

# Static Array

A =	44	12	-5	17	6	0	3	9	100
	↓	↓	↓	↓	↓	↓	↓	↓	↓
	0	1	2	3	4	5	6	7	8

Elements in  $A$  are referenced by their index. There is no other way to access elements in an array. Array indexing is zero-based, meaning the first element is found in position zero.

# Dynamic Array

Suppose we create a dynamic array with an initial capacity of two and then begin adding elements to it.

$\emptyset$	$\emptyset$
-------------	-------------

<b>7</b>	$\emptyset$
----------	-------------

7	<b>-9</b>
---	-----------

7	-9	<b>3</b>	$\emptyset$
---	----	----------	-------------

7	-9	3	<b>12</b>
---	----	---	-----------

7	-9	3	12	<b>5</b>	$\emptyset$	$\emptyset$	$\emptyset$
---	----	---	----	----------	-------------	-------------	-------------

7	-9	3	12	5	<b>-6</b>	$\emptyset$	$\emptyset$
---	----	---	----	---	-----------	-------------	-------------

## Linked lists

Linked list ဆိတ်ကကျ data node တွေကို sequence အတိုင်း link လုပ်ထားတာ။ မြင်အောင်ပြောရမယ်ဆိုရင် list စစချင်းမှာ head ရှိမယ်၊ အဆုံးမှာ tail ရှိမယ်၊ ကြားထဲမှာ data node တွေနဲ့သူတို့ကို ချိတ်ဆက်ထားတဲ့ link တွေရှိမယ်။ attach တဲ့ထားတဲ့ ပုံတွေ **ကြည့်လိုက်ရင်တော့ပိုရှင်းသွားလိမ့်မယ်။**

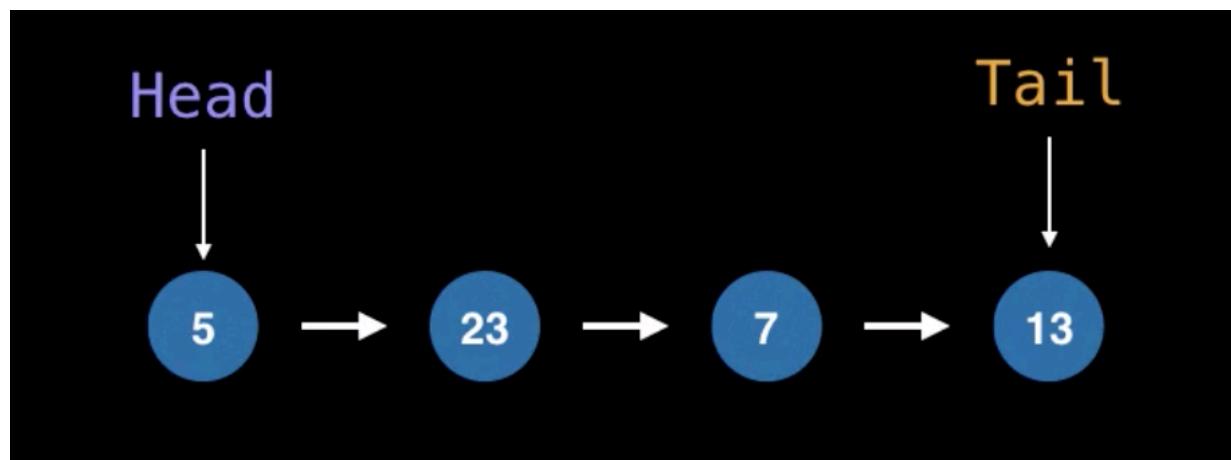
Linked lists တွေကို**ပြန်ခွဲကြည့်မယ်ဆို** - Singly linked list - Doubly linked list - Circular linked list ဆိုပြီးရှိတယ်။

Singly linked list ကကျတော့ data node တွေကြားမှာ ချိတ်ဆက်ထားတဲ့ link တွေက node တစ်ခုခြင်းဆိုရဲ့ forward ကိုပဲ ချိတ်လို့ရတယ်၊ backward (previous node) ကို link ပြန်လုပ်ထားတာမရှိဘူး။

Doubly linked list ကတော့ node တွေအကုန်လုံးကို forward ရော့ backward ရော့ link လုပ်ထားတယ်၊ ဆိုတော့ ပိုပြီးတော့ အသို့ဝင် တယ်လို့ဆိုရမယ်။ previous data ကိုပါလှမ်း access လုပ်လို့ရတာကိုး။

Circular linked list က သူ့နာမည်အတိုင်းပဲ circular ပြန်လုပ်ပြီး link တယ်၊ singly linked list မှာဆို tail ကနေ head ဆိုကိုလှမ်းပြီး link လုပ်ထားတယ်။ doubly linked မှာက ကျ tail ကနေ head ကိုလည်း link တယ်၊ doubly ရဲ့ထုံးစံအတိုင်း head ကနေလည်း tail ကို ပြန်ပြီး link လုပ်တယ်။

### Singly Linked List



## Doubly Linked List

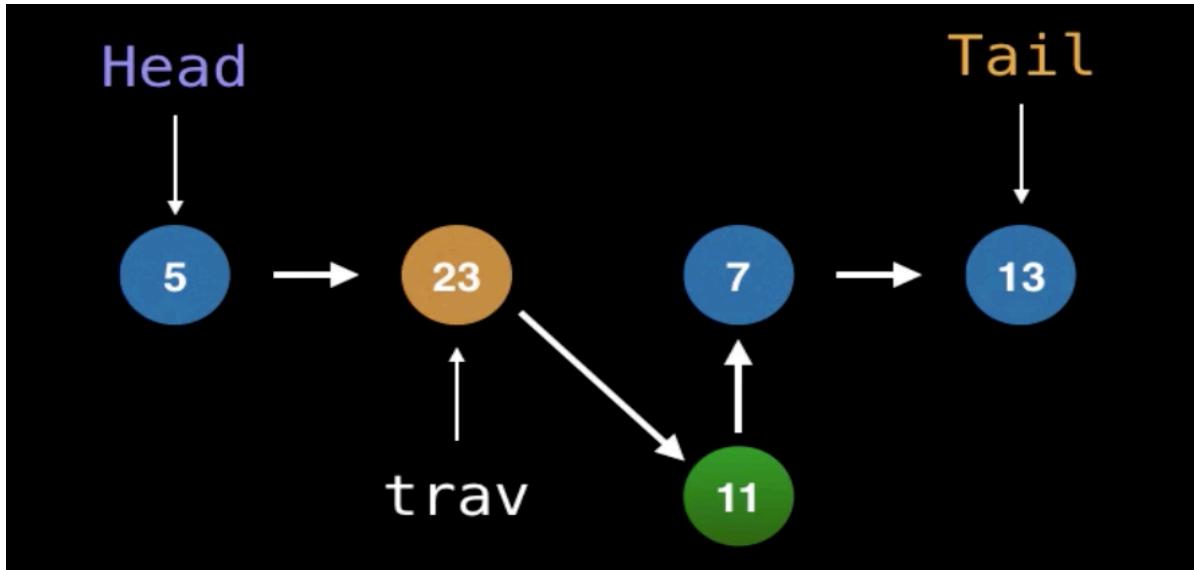


Operation လုပ်ပုံလုပ်နည်း ကလည်း နည်းနည်းဆီကွာတယ်။ insert လုပ်တဲ့ပုံကို အရင်ပြောကြည့် ရမယ်ဆိုရင်

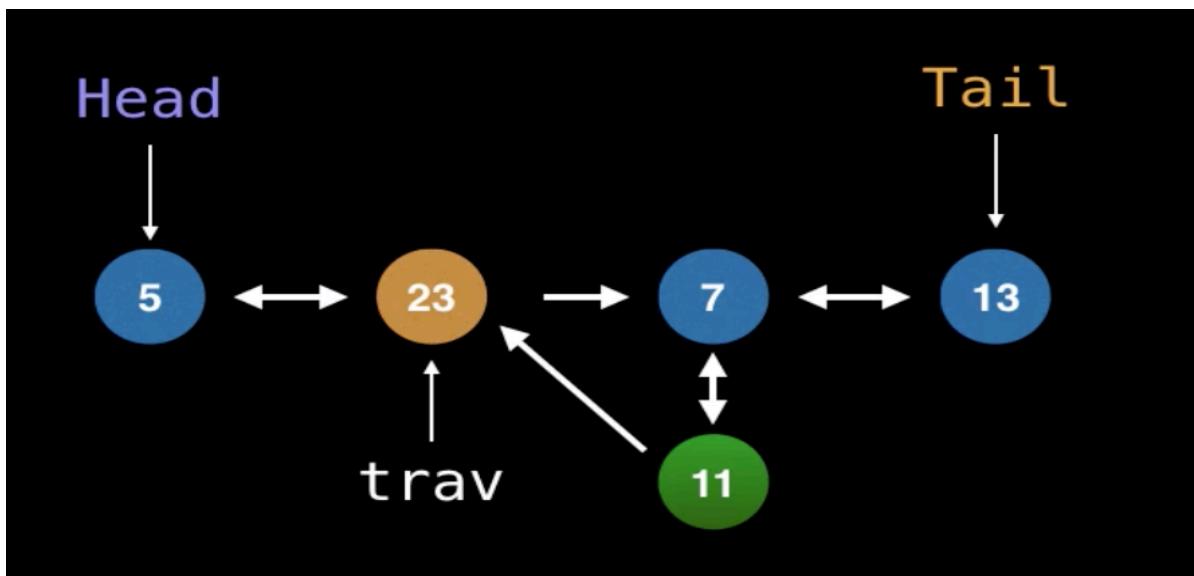
Singly linked list မှာ 3rd node မှာ node အသစ်တစ်ခုထည့်မယ်ဆိုပါစို့။ node တစ်ခုခြင်းဆီကို traverse လုပ်တယ်၊ ဒုတိယပြောက်မှာ ရပ်လိုက်တယ်၊ ထည့်မယ့် value node ကို create လုပ်တယ်၊ တတိယပြောက် node ကို pointer ထောက်တယ်၊ traverser ရပ်ထားတဲ့ ဒုတိယပြောက် element ကို သူ့နဲ့ ချိတ်တယ်၊ ပြီးတာနဲ့ insert လုပ်လိုက်တယ်။

Doubly linked list မှာလည်း insert operation မှာ singly နဲ့ အတူတူပဲ ကွာသွားတာ က singly က link ချိတ်တဲ့ နေရာမှာ forward link ပဲချိတ်ရပေမဲ့ doubly မှာ previous ရော ကြော အတွက်ပါ ချိတ်ပေးရတယ်။

## Singly Linked List Inserting Operation



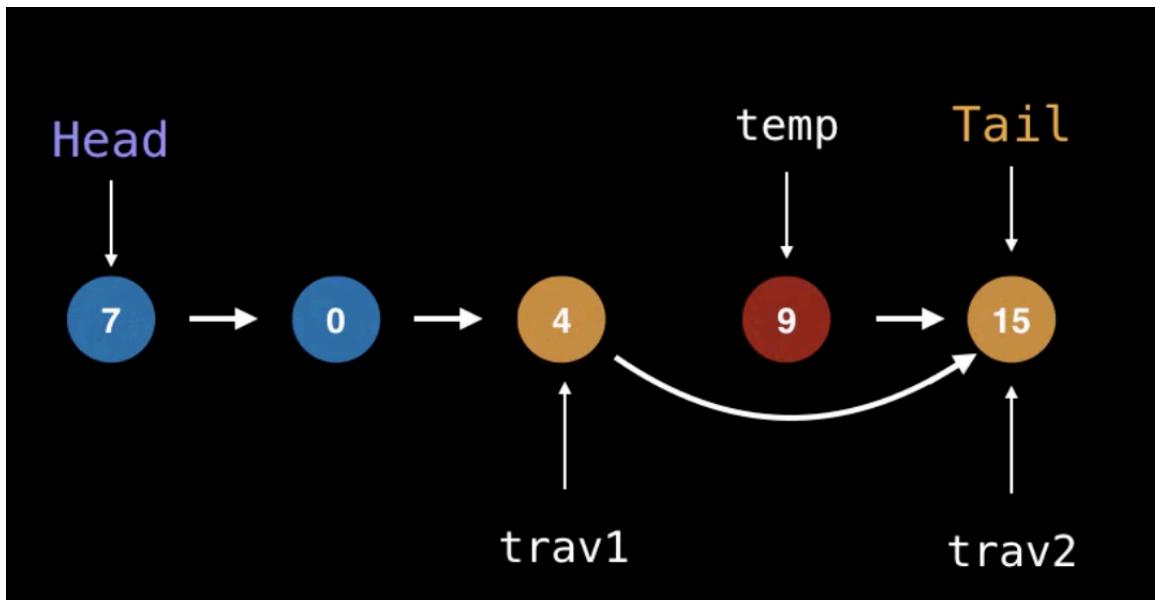
## Doubly Linked List Inserting Operation



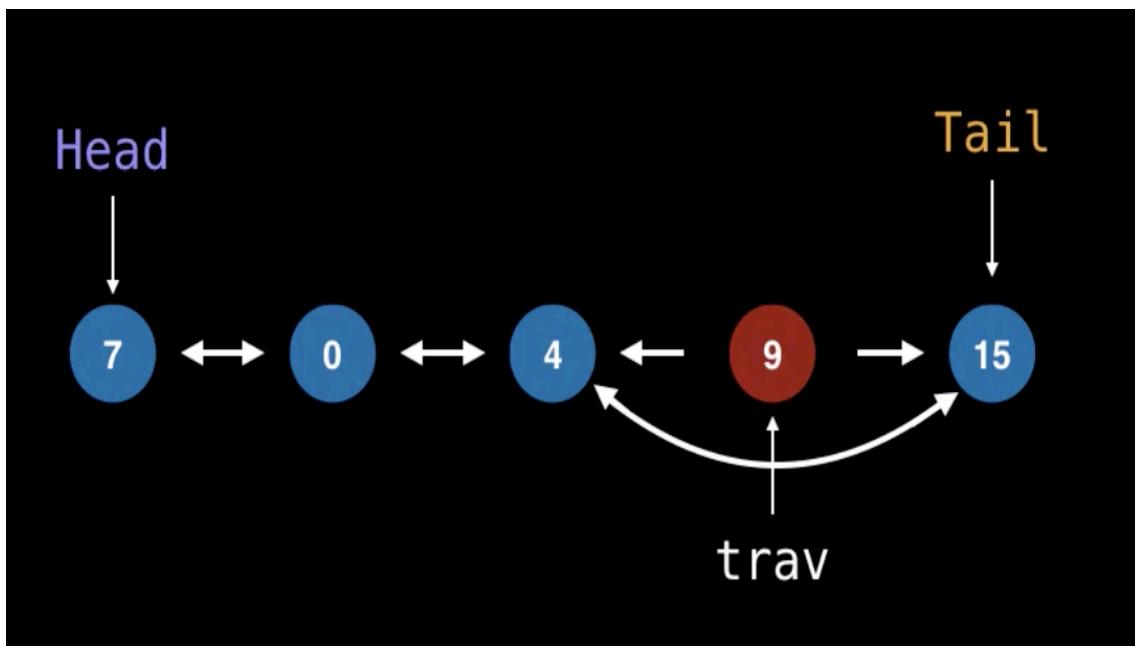
Remove operation မှာတော့ ပိုပြီးကွာတယ်။ singly မှာ remove operation လုပ်တော့မယ်ဆို travserer နှစ်ခုနဲ့သွားရတယ်။ trav1 & trav2 ပေါ့၊ remove လုပ်ရမယ့် node ကို trav2 က တွေ့သွားပြီးဆို တွေ့သွားတဲ့ node ကို memory ထဲမှာ temporary allocate လုပ်လိုက်တယ်။ allocate လုပ်ပြီးတာနဲ့ trav2 က နောက် node တစ်ခုကို ထပ်သွားတယ်၊ ပြီးတော့မှ temp allocate ထားတယ့် node ကိုဖြတ်ချပြီး deallocate ပြန်လုပ်လိုက်တယ်။ trav1 & 2 ကိုလည်း linked ပြန်လုပ်လိုက်တယ်။

Doubly မှာတော့ traverser နှစ်ခုမလိုဘူး။ တစ်ခုနဲ့ ဆိုရတယ်။ ဘာလိုလဲဆိုတော့ သူမှာက previous ရော့ forward ရော့ချိတ်လိုရတယ်။ singly မှာက မရှိတဲ့အတွက် reference key လိုလို traverser နှစ်ခုနဲ့ သွားရတာ။ဖြတ်ရမယ့် node ရောက်ပြုဆို သူ့ရဲ့ရှေ့ node နဲ့ နောက် node ကို forward & backward link ချိတ်ပေးပြီး ဖြတ်လိုက်တယ်။

## Singly Linked List Removing Operation



## Doubly Linked List Removing Operation



နှစ်ခုလုံးမှာတော့ draw back တွေရှိတယ်။ singly က memory use တာနည်းပေမဲ့ previous element ကိုပြန်သွားလို့မရဘူး။ doubly က previous node ပြန်သွားလို့ရတယ်၊ memory လည်းနှစ်ဆိုကုန်တယ်။

ဘယ်လိုနေရာတွေမှာ သုံးလည်း ဆိုတော့ sequentially linked လုပ်ထားတဲ့ process တော်တော် များများမှာ သုံးတယ်။ ဥပမာ file system တစ်ချို့မှာ file locations တွေကို linked လုပ်ပြီး သိမ်းထားတာတို့၊ browser history မှာ သိမ်းထားတဲ့ page တွေမှာ back ပြန်သွားလို့ရတာမျိုးတို့၊ memory block တွေကို linked လုပ်ထားတဲ့ low level memory management system တို့မှာတို့ အစရှိတဲ့ နေရာတော်တော်များများမှာ သုံးထားတယ်။ linked list ရဲ့ သဘောတရားကို နားလည်သွားပြီဆိုရင် ဘယ်လို technology မှာ linked list algorithm သုံးထားလည်းဆိုတာ ခန့်မှန်းလို့ရသွားနိုင်မယ့် အပြင် ကိုယ်ကိုတိုင်လည်း develop လုပ်သွားနိုင်မှပါ။

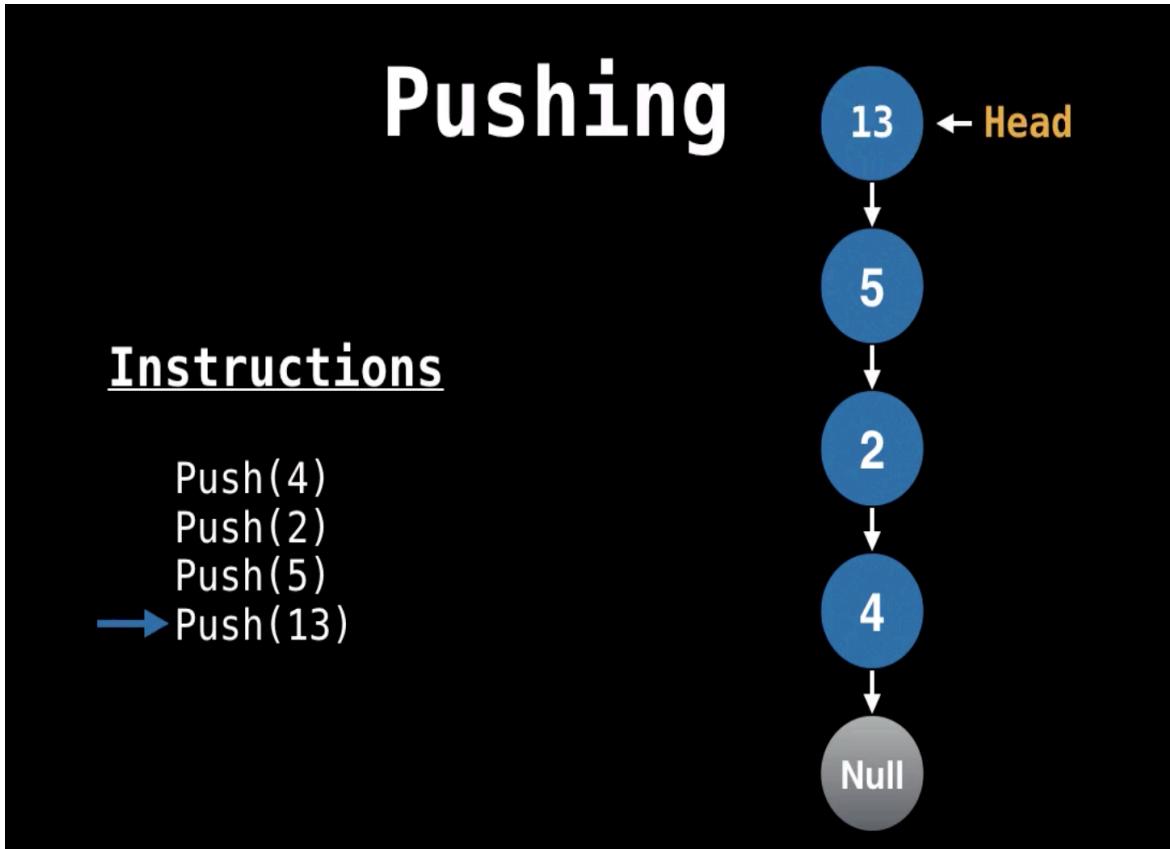
## Stack

Stack ဆိုတာက သူ့ကို ပေးထားတဲ့နာမည် အတိုင်းပဲ အစဉ်ကိုယ်ထပ်ထပ်ဆင့်ထားတဲ့ ပုံစံမျိုးကို ဆိုလိုတဲ့ structure တစ်ခုပဲ၊ LIFO (last in first out) or FILO(first in last out) ဆိုတဲ့ order အတိုင်းအလုပ်လုပ်တယ်။ ပြောရမယ်ဆိုရင် စာအုပ်တွေထပ်ပြီးတင်ထားတယ်၊ နောက်ဆုံးတင်တဲ့ တစ်အုပ်ကိုပဲ အရင်ပြန်ယူတယ် (last in first out)၊ ပထမဆုံး ထားလိုက်တဲ့စာအုပ်ကို နောက်ဆုံးမှ ထုတ်လို့ရတယ် (first in last out)၊ အဲဒီ သဘောတရားပါပဲ။

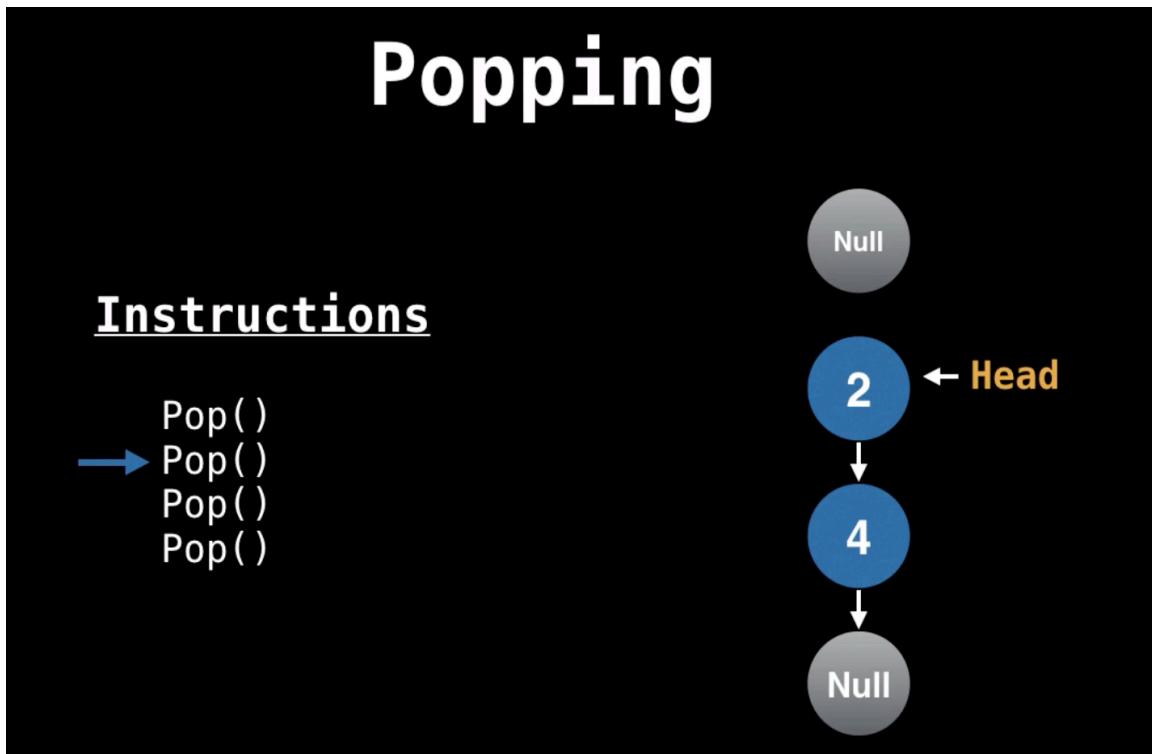
stack ကို array နဲ့ပဲဖြစ်ဖြစ် linked list နဲ့ပဲဖြစ်ဖြစ် ဆောက်လို့ရတယ်၊ fixed sized ရော dynamic size ရော ကြိုက်သလိုဆောက်လို့ရတယ်၊ ဆောက်တဲ့ program ပေါ်မှုတည်တယ်။ stack မှာ operations တွေအမျိုးမျိုးရှိပေမဲ့ အဓိက သုံးတာက တော့ push နဲ့ pop ပဲ။ push ဆိုရင် stack ထဲကို data လာထည့်မယ်။ pop ဆို ထုတ်မယ်၊ push လုပ်လုပ် pop လုပ်လုပ် အပေါ်ဆုံးက data value ကိုပဲ လုပ်လို့ရမယ်။ တစ်ခြား သုံးတဲ့ function တွေလည်း ရှိတယ်၊ ဥပမာ ထိပ်ဆုံး ကို element ကို access လုပ်ဖို့ဆို peek ဆိုတာကိုသုံးတာတို့၊ stack က ပြည့် နေပြီးလား empty ဖြစ်နေလား စတာတွေကို စစ်တဲ့ function တွေလဲရှိတယ်။ ဒီနေရာမှာတော့ ကျနော် push နဲ့ pop ကိုပဲ အဓိက ထားပြီး ပြောလိုက်မယ်၊ ဒါဆိုရင် ကျန်တဲ့ဟာတွေလဲ ခြိုင်ပြီးတစ်ခါတည်း ပြောပြီးသားဖြစ်သွားပါလိမ့်မယ်။

stack တစ်ခုထဲကို data တစ်ခု push (ထည့်) လုပ်တော့မယ်ဆိုရင် - အရင်ဆုံး stack က ပြည့်နေလား အရင်ကြည့်တယ် - Stack က ပြည့်နေတယ်ဆို သက်ဆိုင်တဲ့ msg ပြန်ပြီးတော့ exit လုပ်တယ် (dynamic size အတွက်ဆိုရင်တော့ သူ့ဘာသာ resizing လုပ်သွားတယ်) - Stack က ထည့်လို့ ရသေးတယ်ဆိုရင် top pointer ကို နောက်ထပ်ဝင်လာမယ့်နေရာအတွက်ဦးထားပေးလိုက်တယ် (top pointer ဆိုတာလက်ရှိ stack ထဲမှာ အပေါ်ဆုံးမှာရှိနေတဲ့ data ကိုထောက်ထားတဲ့ pointer, push operation လုပ်တော့မယ်ဆို top pointer ကို +1 ပုံစံယူပြီး နောက်ဝင်လာမယ့် နေရာအတွက် empty space ကိုထောက်ပေးထားတယ်။) - နောက်တစ်ဆင့်အနေနဲ့ top pointer ထောက်ထားတဲ့ နေရာကို data ဝင်လာတယ် - Operation success ဖြစ်တဲ့ အကြောင်း return ပြန်တယ်

## Stack Pushing



## Stack Popping



Stack ထဲကနေ pop operation (stack ထဲကနေ data ထုတ်တော့မယ်ဆိုရင်) - Stack က empty ဖြစ်နေလားအရင်ကြည့်တယ် - Empty ဖြစ်နေတယ်ဆို သက်ဆိုင်တဲ့ msg ပြန်ပြီးတော့ exit လုပ်တယ် - Empty မဖြစ်ဘူးဆိုရင် လက်ရှိ top point ထောက်ထားတဲ့ data ကို access လုပ်တယ်။ - Top pointer ကို အောက်တစ်ထစ်ဆင်းလိုက်ပြီး ခုနက access လုပ်ထားတဲ့ data ကို stack ထဲကနေထုတ်လိုက်တယ် - Operation success ဖြစ်တဲ့ အကြောင်း return ပြန်တယ်

ဒါ algorithm ကို ဘယ်လို နေရာတွေမှာ အသုံးချလဲဆိုတော့ LIFO သဘောတရားရှိနိုင်တဲ့ နေရာမှန်သမျှမှာ apply လုပ်လို့ရတယ်။ ဥပမာ ပြောရမယ်ဆိုရင် IDE တွေမှာ undo လုပ်တဲ့ process တို့၊ browser တွေမှာ back သွားတဲ့ process တို့၊ expressions တွေ convert လုပ်တဲ့နေရာ စတဲ့ နေရာတွေမှာသုံးလို့ရတယ်၊ ဘယ်လိုနေရာတွေမှာ ထပ်သုံးထားလဲသိချင်ရင် အောက်က link မှာ ဝင်ကြည့်လို့ရပါတယ်။ <http://jcsites.juniata.edu/faculty/kruse/cs240/stackapps.htm>

## Queue

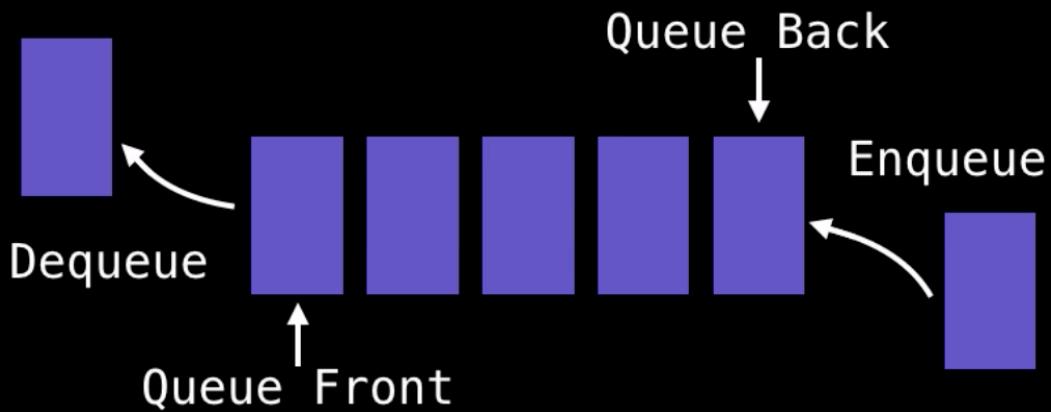
Queue ကတော့ stack နဲ့ဆင်တယ်။ stack နဲ့မတူတာကကျ stack က one ended ကိုပဲ data အသွင်းအထုတ်ရတယ်။ queue ကကျတော့ တစ်ဖက်ကဝ် တစ်ဖက်ကထွက်ပံ့စံမျိုး။ insert လုပ်ဖို့အတွက်ကို enqueue လို့ခေါ် ပြီးတော့ remove လုပ်ဖို့ကို dequeue လို့ခေါ်တယ်။ stack တုန်းက LIFO (last in first out) structure အတိုင်း ဆိုပေမဲ့ queue မှာတော့ FIFO(first in first out) ပဲ၊ အရင်ဝင်လာတဲ့ process က အရင်ပြန်ထွက်တယ်၊ နောက်မှဝင်လာတဲ့ ကောင်က နောက်မှ ထွက်ရမယ်။ real world example အနေနဲ့ ပြောရရင် starbucks မှာ ကော်ဖီ မှာဖို့တန်းစီ ပြီး စောင့်နေတဲ့ လူတွေမျိုး၊ အရင်ရောက်တဲ့ လူက အရင်မှာတယ်။ ပြီးရင် ထွက်သွားတယ်။

ဒါ article မဖတ်ခင် stack algorithm ရဲ့ article ကိုအရင်ဖတ်ဖို့ recommend လုပ်ချင်ပါတယ်။

Stack လို့ queue ကို array တွေ linked lists တွေသုံးပြီးတော့ ဆောက်လို့ရတယ်။ queue process မှာ queue ဆောက်တာတို့ stack မှာလို့ ထိပ်ဆုံး element ကို access လုပ်ဖို့ဆို peek ဆိုတာကိုသုံးတာတို့ queue က ပြည့် နေပြီးလား empty ဖြစ်နေလားစတာတွေကို စစ်တဲ့ function တွေလဲရှိတယ်။ အဓိက အသုံးများတာကတော့ အပေါ်မှာပြောထားတဲ့ function နှစ်ခုဖြစ်တဲ့ enqueue နဲ့ dequeue ပဲ။ queue အတန်းကြီးတစ်ခုရှိတယ်ဆိုပါစို့၊ ညာဘက်(အနောက်ဘက်) ကို queue ထည့်ဖို့ (enqueue) လုပ်ဖို့သုံးတယ်၊ queue ရဲ့ဘယ်ဘက်(အရှေ့ဘက်)ကို queue remove လုပ်ဖို့သုံးတယ်၊ attach လုပ်ထားတဲ့ပုံနဲ့ တွဲကြည့်လိုက်ရင်တော့ ပိုမြင်သွားပါလိမ့်မယ်။

## A Queue

A queue is a linear data structure which models real world queues by having two primary operations, namely **enqueue** and **dequeue**.

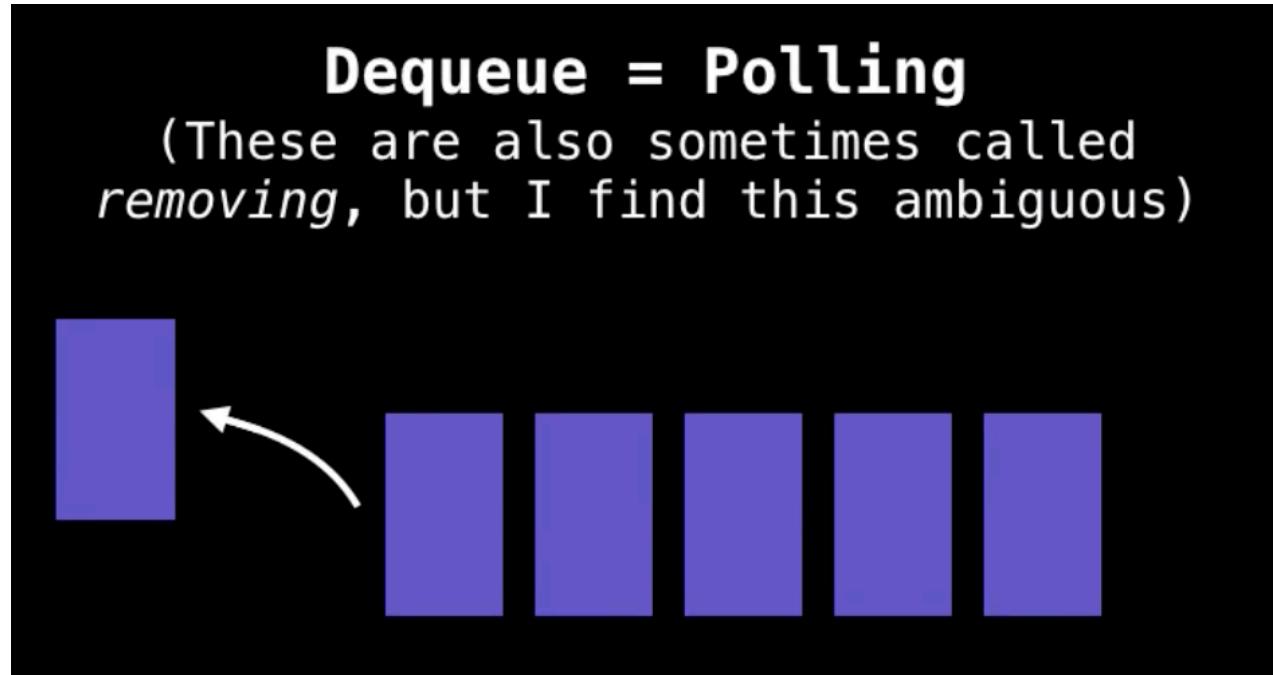


## Enqueue

**Enqueue = Adding = Offering**



## Dequeue



Queue တွေဘယ်လိုထည့်လဲဆိုတော့ (enqueue operation) - အသစ်ထည့်တော့မယ်ဆို queue  
ပြည့်နေပြီလားစစ်တယ် - ပြည့်နေတယ်ဆို သက်ဆိုင်တဲ့ msg ပြန်ပြီးတော့ exit လုပ်တယ် -  
ထပ်ထည့်လို့ရသေးတယ်ဆို stack မှာလိုမျိုးပဲ top element ကို access လုပ်တဲ့ top pointer queue  
မှာလဲ access လုပ်လို့ရတဲ့ rear pointer ဆိုတာရှိတယ်၊ အဲဒီ pointer ကို +1 လုပ်ပြီး  
နောက်ဝင်လာမယ့် process အတွက် empty space ကို ထောက်ပေးထားတယ် - နောက်တစ်ဆင့်  
အနေနဲ့ rear pointer ထောက်ထားတဲ့ empty space ကို value ထည့်တယ် - Operation success  
ဖြစ်တဲ့ အကြောင်း return ပြန်တယ်

## Adding Queue

### Instructions:

Enqueue(12)

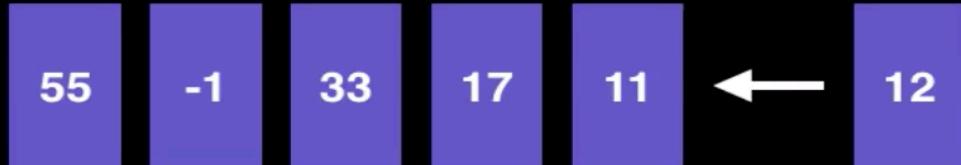
Dequeue()

Dequeue()

Enqueue(7)

Dequeue()

Enqueue(-6)



## Removing Queue

### Instructions:

Enqueue(12)

Dequeue()

Dequeue()

Enqueue(7)

Dequeue()

Enqueue(-6)



ဒီနေရာမှာ stack နဲ့ မတူတာကျ stack က one ended ဆိုတော့ ထည့်တာရောတုတ်တာရော တစ်ဖက် တည်းကသွားတာဖြစ်တဲ့ အတွက်ကြောင့် pointer က တစ်ခုပဲရှိစရာလိုတယ်။ နှစ်ဖက်လုံး အလုပ်လုပ်တဲ့ queue မှာတော့ enqueue အတွက် ဆို rear (back pointer) နဲ့ dequeue အတွက် front pointer ဆိုပြီး ခွဲထားတယ်။

Queue ထွောယ်လိုတုတ်လဲဆိုတော့ (dequeue operation)

- queue က empty ဖြစ်နေလား အရင်ကြည့်တယ်။
- Empty ဖြစ်နေတယ်ဆို သက်ဆိုင်တဲ့ msg ပြန်ပြီးတော့ exit လုပ်တယ်။
- Empty မဖြစ်ဘူးဆိုရင် လက်ရှိ front pointer ထောက်ထားတဲ့ data ကို access လုပ်တယ်။
- နောက်တစ်ဆင့် အနေနဲ့ front pointer ကနေ backward သွားပြီး နောက် data element တစ်ခုကို access လုပ်ပြီး ရှေ့ကဟာကို queue ထဲကနေ ထုတ်လိုက်တယ်။
- Operation success ဖြစ်တဲ့ အကြောင်း return ပြန်တယ်

Queue algorithm ကိုဘယ်နေရာတွေမှာသုံးလဲဆိုတော့ ဥပမာ အနေနဲ့ ပြောရမယ်ဆိုရင် web server ကိုလာတဲ့ request တွေ handle လုပ်တဲ့ နေရာမှာသုံးလို့ရတယ်။ request တစ်သိန်းလာတယ်၊ ဒါပေမဲ့ server က တစ်ခါ process လုပ်ရင် တစ်ထောင်ပဲရတယ်။ ကျွန်ုတဲ့ request တွေကို queue ဆဲ ထည့်ထားပြီးတော့ first come first serve ပုံစံမျိုးနဲ့ အလုပ်လုပ်သွားတယ်။ web server request မှုရယ်မဟုတ်ဘူး၊ ကျွန်ုတဲ့ batch request တွေကို လည်း queue သုံးပြီး manage လုပ်လို့ရပါတယ်။ တစ်ခြားသော scheduling case တွေမှာလဲ သုံးလို့ရတယ်။ round robin scheduling တို့ Job scheduling တို့ စသည်ဖြင့် FIFO (first in first out) order နဲ့ သွားတဲ့ logic မှန်သမျှကို queue နဲ့ implement လုပ်လို့ရတယ်။ အခုနောက်ပိုင်းဆို Redis queue ဆိုရင် program တွေ တော်တော်များများ ထဲမှာ integrate လုပ်လာကြပါတယ်။ တစ်ခြား multitasking လုပ်တဲ့ feature တွေမှာလည်း queue ကိုအသုံးပြုလို့ရပါတယ်။

## Priority Queue in Data Structure & Algorithm

Priority Queue ဆိတာကတော့ ပုံမှန် regular queue data structure ကိုပဲ အခြေခံပြီးလုပ်ထားတဲ့ structure တစ်ခုပဲ၊ process တော်တော်များများကလည်း ပုံစံအတူတူပဲ၊ မတူပဲ ကွဲပြားသွားတာက ဘာလဲဆိတာ Priority Queue (PQ လိုပဲခေါ်သွားပါမယ်) မှာက သူထဲမှာ ပါတဲ့ data node တစ်ခုခြင်းဆို က priority order လေးတွေပါတယ်။ ဆိုလိုချင်တာက ဘယ် node က ဘယ် node ထက် အရေးကြီးတယ်ဆိတာမျိုးကို ပြောနိုင်ဖို့အတွက် ကိုယ်ပိုင် priority value လေးတွေပါတယ်။ PQ ကိုယ်တိုင်ကလည်း comparable model ဖြစ်တယ်၊ အဲလို compare လုပ်နိုင်ဖို့အတွက် က data node တွေမှာ priority value တွေမဖြစ်မနေပါရမယ်၊ ဒါမှာလည်း PQ က ဘယ် node က ပိုအရေးကြီးတယ်ဆိတာခွဲလို့ရမှာဖြစ်တယ်။ (must read queue article first)

PQ အလုပ်လုပ်ပုံက လွယ်မယောင်နဲ့ နည်းနည်းတော့ ခက်ပါတယ်။ ဘာလိုလွယ်တာလဲ ဆိုတော့ သူရဲ့ အလုပ်လုပ်ပုံကရှင်းပါတယ်။ queue တစ်ခုထည့်တော့မယ်ဆို queue ရဲ့ data node နဲ့အတူ သူရဲ့ priority value လည်းပါလာပါတယ်။ PQ က priority value အမြင့်ဆုံး data node ကို အရင် process လုပ်ပေးပါတယ်။ ဥပမာ ပြောရမယ်ဆိုရင် data node တွေရဲ့ priority value တွေက ဒီလိုတွေရှိမယ်ဆိုပါစို့။

4,5,1,3,8,2,1,3,6,9 priority ကေန်း အနည်းဆုံး node ကို priority value အမြင့်ဆုံးလို့ သတ်မှတ်ပြီး process လုပ်ကြည့်လိုက်မယ်ဆို အောက်မှာရေးထားသလိုဖြစ်သွားမှာပါ။

1,1,2,3,3,4,5,6,8,9 . PQ က အဲလိုပြန် order လုပ်ပြီး run သွားမှာပါ။ ပြောချင်တာက priority value အများဆုံး node တွေကို အရင် run သွားတာပါ။ ဘာလို့ခက်တယ် ပြောတာလဲဆိုတော့ ဒီလိုပါ။ ကျနော်တို့ က အခု ဘယ် priority value က မြင့်တယ် ဘာဟာ ဆိုတာကို မျက်စိနဲ့ ကြည့်ပြီး စီလိုက်လိုပါ။ 1 က အမြင့်ဆုံး 9 က အနိမ့်ဆုံး ဆိုတာကို မျက်စိနဲ့ ကြည့်ပြီး တော့ ပဲဆုံးဖြတ်လိုက်တာပါ။ ဒါပေမဲ့ PQ မှာ ဘယ်သူ က မြင့်တယ်၊ နိမ့်တယ် ဆိုတဲ့ order ကို ဘယ်လို ဆုံးဖြတ်မှာလဲ။ အဲလိုဆုံးဖြတ်ဖို့အတွက် data structure နောက်တစ်ခုကိုသုံးပါတယ်။ Heap လို့ခေါ်ပါတယ်။

PQ ဆောက်ဖို့အတွက် heap မှုရယ် မဟုတ်ပါဘူး။ တစ်ခြား structure တွေနဲ့လည်း အသုံးပြုပြီး ဆောက်လို့ရတယ်၊ ဒါပေမဲ့ commonly အသုံးများတာကတော့ Heap ပါပဲ။ Heap ဆိုတာဘာလည်း ဆိုတော့ သူက tree based structure တစ်ခုပဲ၊ အဲဒီ tree based structure မှာကိုမှ သူရဲ့ rules

တွေ့ရှိတယ် (သူကတော့ heap invariant) လိုခေါ်တယ်။ ဘာကိုဆိုလိုချင်တာလဲဆိုတော့ သူမှာ Min Heap & Max Heap ဆိုပြီး နှစ်မျိုးရှိတယ်။ min heap ဆိုရင်လဲ root node ကအသေးဆုံးဖြစ်ပြီး သူအောက် က child node တွေက သူထက် ထပ်သေးသွားလို့မရတော့ဘူး။ max heap လည်း ထိုနည်းလည်းကောင်းပဲ အောက်က child nodes တွေက root ထက် ထပ်ကြီးသွားလို့မရတော့ဘူး။

ဒါဆိုလိုရှိရင် heap ရဲ့ order အလိုက်စိတ္တားတဲ့ structure နဲ့ PQ ရဲ့ priority value ဆက်စပ်ပုံကို တွေးမိလိုရမယ်ထင်ပါတယ်။ heap ကလည်း min ဆိုရင် min အတိုင်းစီတယ်။ max ဆို max အတိုင်းစီတယ်။ PQ မှာလည်း priority value ရဲ့အစဉ်အတိုင်း process လုပ်တယ်။

PQ ကိုသုံးပြီးတော့ implement လုပ်ထားတဲ့ algorithm တွေလည်း ရှိတယ်။ sorting algorithm ဖြစ်တဲ့ heapsort တို့ min key node တွေကို extract လုပ်တဲ့ prim algorithm တို့ load balancing မှာ traffic တွေ manage လုပ်တဲ့ နေရာတို့ စသည်ဖြင့်အသုံးပြုထားတာတွေအများကြီးလည်းရှိတယ်။ တစ်ကယ်လည်း strong ဖြစ်တယ်။

ဒါ article မှာတော့ PQ ကိုအမိကထားပြီးပြောသွားတဲ့အတွက်ကြောင့် Heap အကြောင်းက Introduction အနေနဲ့ပဲပါသေးတယ်။ Heap ရဲ့ data operation အပိုင်းက နည်းနည်းလေး လေချာကြည့်ရတယ်။ နောက် article မှာ PQ မှာသုံးတဲ့ Heap အကြောင်းကို ထပ်ရေးပေးသွားပါမယ်။

## Priority Queue & Its Instructions

# What is a Priority Queue?

### Instructions:

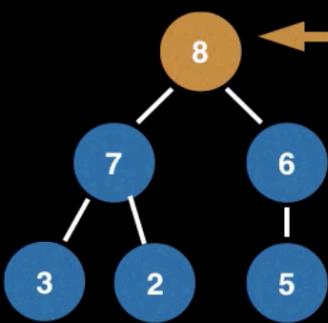
```
poll()  
add(2)  
poll()  
add(4)  
poll()  
add(5)  
add(9)  
→ poll rest
```



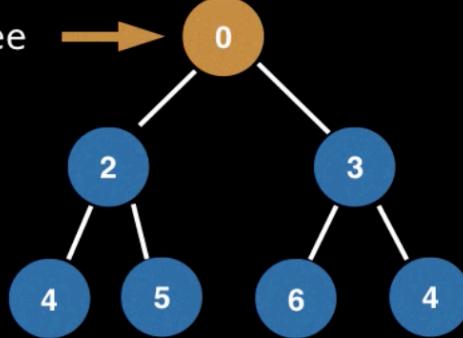
## What Is A Heap

# What is a Heap?

A heap is a **tree** based DS that satisfies the **heap invariant** (also called heap property): If A is a parent node of B then A is ordered with respect to B for all nodes A, B in the heap.



Max Heap



Min Heap

## Priority Queue with Heap (Part 1)

Priority Queue ကို heap နဲ့ ဆောက်ရတဲ့ အကြောင်းအရင်းက လောလောဆယ်မှာ time complexity အတွက် အသင့်တော်ဆုံးဖြစ်နေသေးလိုပါ။ တစ်ခြား ဆောက်လိုရတဲ့ algorithm တွေလဲရှိတယ်၊ ဥပမာ unsorted list တို့ဘာတို့နဲ့ ဆောက်လိုရတယ် ဒါပေမဲ့ time complexity အတွက်အဆင်မပြေသေးဘူး။ ဒါကြောင့်မို့ heap ကိုပဲ သုံးဖြစ်နေကြတာပါ။

ဒီ article ကိုမဖတ်ခင်မှာ Priority Queue ရဲ့ article ကိုအရင်သွားဖတ်ဖို့လိုအပ်ပါတယ်(must)။  
Heap အကြောင်းကို Introduction လုပ်ထားတာလဲပါပါတယ်.

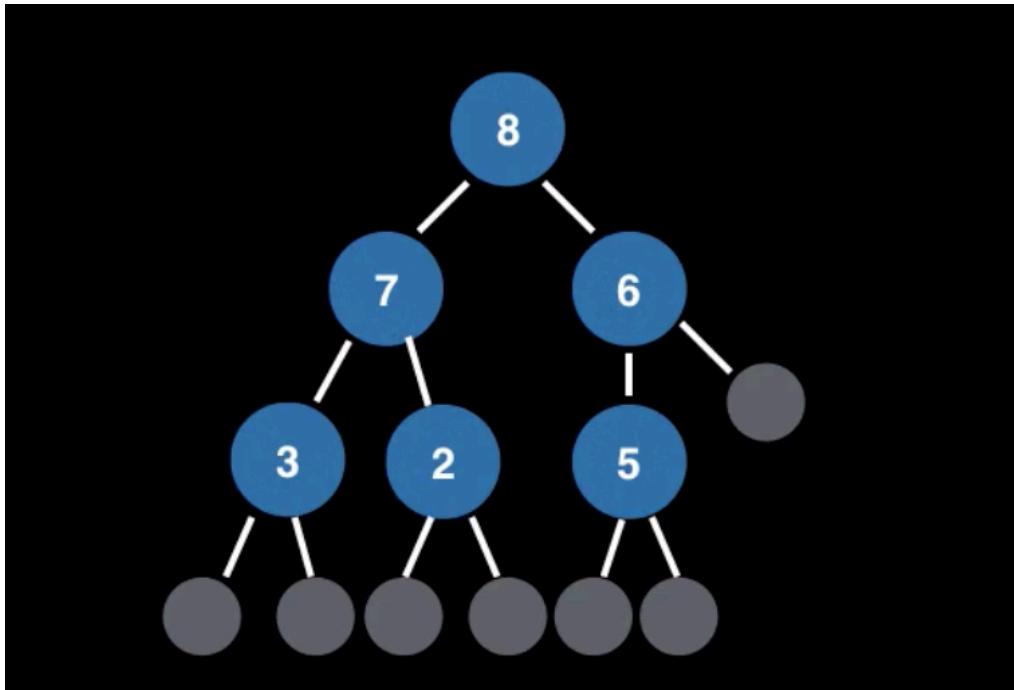
ဒီ article မှာ ပုံတွေလည်း တစ်ဆင့်ခြင်းဆီအတွက် attach တွဲပေးထားတယ်။ စာချည်းပဲ ဖတ်ရင် နားလည်ရခက်မှာဆိုးတဲ့ အတွက် ပုံတွေနဲ့ တွဲကြည့်ဖို့ suggest လုပ်ပါတယ်။

Priority Queue ဆောက်ဖို့ အတွက် Heap ထဲမှာလည်း

- Binary heap - Fibonacci heap
- Binomial heap
- Pairing heap အစရှိသည်ဖြင့် အမျိုးအစား အမျိုးမျိုး ရှိတယ်။ လောလောဆယ်တော့ Binary heap ကိုသုံးပြီးတော့ ရှင်းပြောသွားပါမယ်။

Heap တိုင်းကတော့ tree structure ပဲ သူ့ရဲ့ heap rules တွေကိုလည်းလိုက်နာတယ်။ Binary heap ဆိုတော့ သူနာမည်အတိုင်းပဲ binary tree structure ဖြစ်တယ်။ Node တစ်ခုခြင်းဆီတိုင်းမှာလည်း child element နှစ်ခုစီနဲ့ tree structure ဆောက်ထားတယ်။

## What Is Binary Heap



ကျနောက်မှာပုံတွေလည်း attach တဲ့ပေးထားတယ်၊ တစ်ချို့စာတွေက ကျပုံတွေနဲ့ တွဲပြီးကြည့်မှုပြီးထင်သာမြင်သာရှိမှုမိုလိုပါ။ အခု အရင်ဆုံးအနေနဲ့ binary heap ရဲ့ data represent လုပ်ပုံကို ကြည့်ရမယ်ဆိုရင် သူ့ရဲ့ data node တွေကို array index တွေအတိုင်းမှတ်ထားလိုလည်း ရတယ်၊ ဥပမာ data node value 6 ၏ array index key number 3 မှာရှိတယ်၊ အဲလိုမျိုး present လုပ်လိုလဲရပါတယ်

## Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Let  $i$  be the parent node index

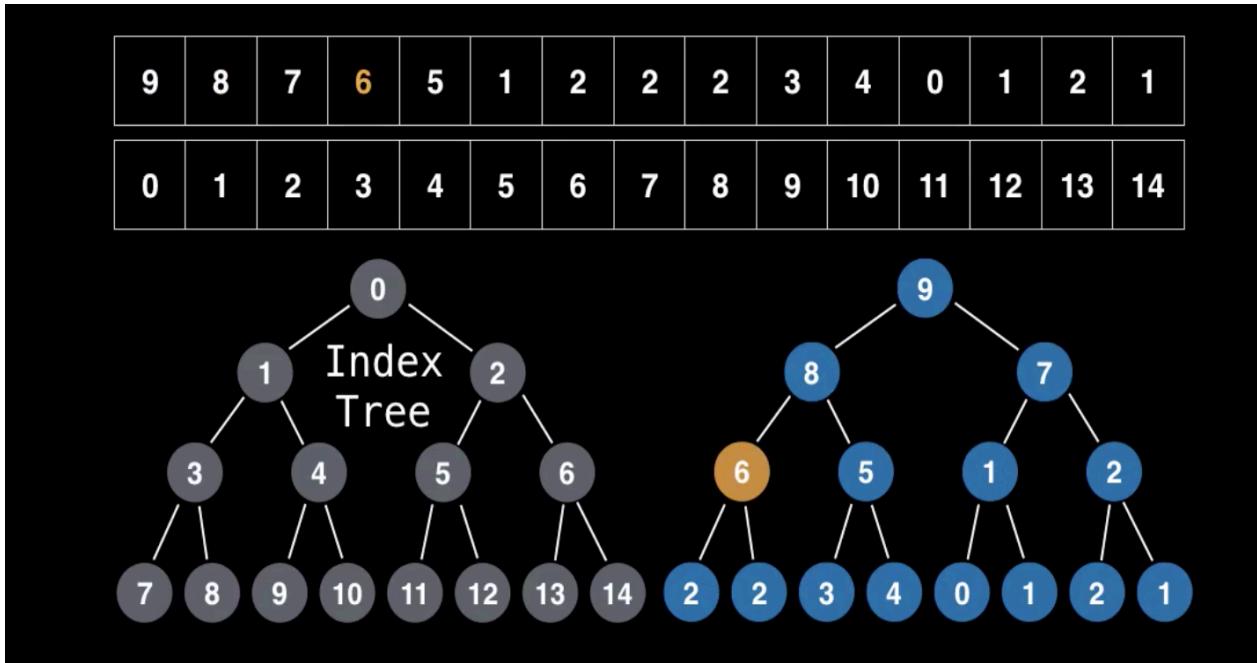
Left child index:  $2i + 1$   
 Right child index:  $2i + 2$  (zero based)

```

graph TD
    9((9)) --> 8((8))
    9((9)) --> 7((7))
    8((8)) --> 6((6))
    8((8)) --> 5((5))
    7((7)) --> 1((1))
    7((7)) --> 2((2))
    6((6)) --> 2_1((2))
    6((6)) --> 2_2((2))
    5((5)) --> 3((3))
    5((5)) --> 4((4))
    1((1)) --> 0((0))
    1((1)) --> 1_1((1))
    2((2)) --> 2_3((2))
    2((2)) --> 1_2((1))
  
```

နောက်တစ်ခုအနေနဲ့ parent node တစ်ခုရဲ့အောက် ၂ child element တွေရဲ့ array index key ကိုလည်း တွက်ထုတ်လို့ရတယ်။ parent node ရဲ့ array key ၁ လို့ဆိုပါစို့။ child element တွေ ဆွဲထုတ်မယ်ဆို binary tree ဖြစ်တဲ့အတွက် left & right child element ဆိုပြီးရှိမယ်။ left child အတွက်ဆို  $(2(\text{current\_node\_key}) + 1) = 5$  ရမယ်၊ right child အတွက်ဆို  $(2(\text{current\_node\_key})+2)=6$  ရပါမယ်။

## Binary Heap Representation



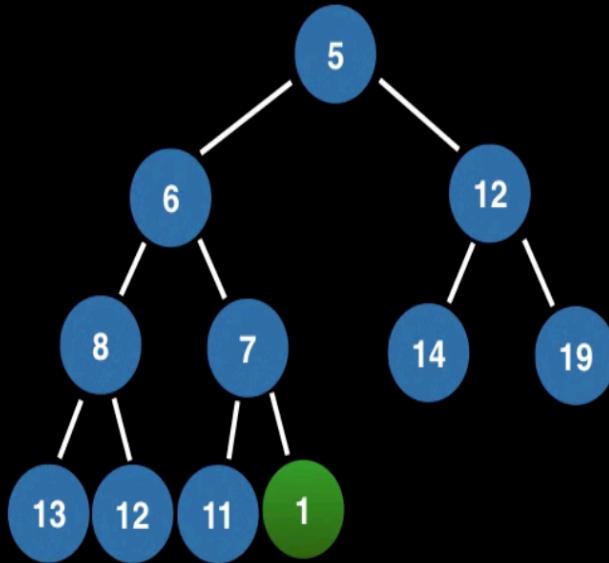
Binary heap မှာ data element insert လုပ်တော့မယ်ဆိုရင် သူရဲ့ tree structure အရ အမြတ်စီး left to right သွားပါတယ်။ insert လုပ်ပြီဆို အောက်ခြေ row ကနဲ စပါတယ်။

အောက်ဆုံးအတန်းကနဲ left to right ပေါ့။ တစ်ခုရှိတာက ထည့်ပြီးတာနဲ့ပြီးမသွားဘူး၊ သူရဲ့ heap invariant (အရင် article မှာရေးထားပါတယ်) အတွက်စစ်ရသေးတယ်။ အဆင်မပြေားဆိုရင် တစ်ဆင့်ခြင်းဆို heap invariant satisfy ဖြစ်တဲ့အထိ parent node နဲ့ replace လုပ်လုပ်သွားတယ်။ အဲလို့ တစ်ဆင့်ခြင်းဆို တတ်တတ်သွားပြီး heap invariant satify လုပ်တာကို bubbling up လုပ်တယ်လို့လဲ ခေါ်ပါတယ်။

## Adding Elements To Binary Heap

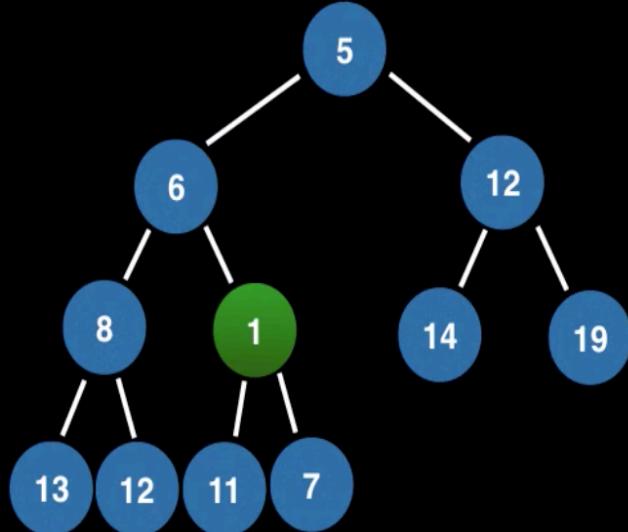
### Instructions:

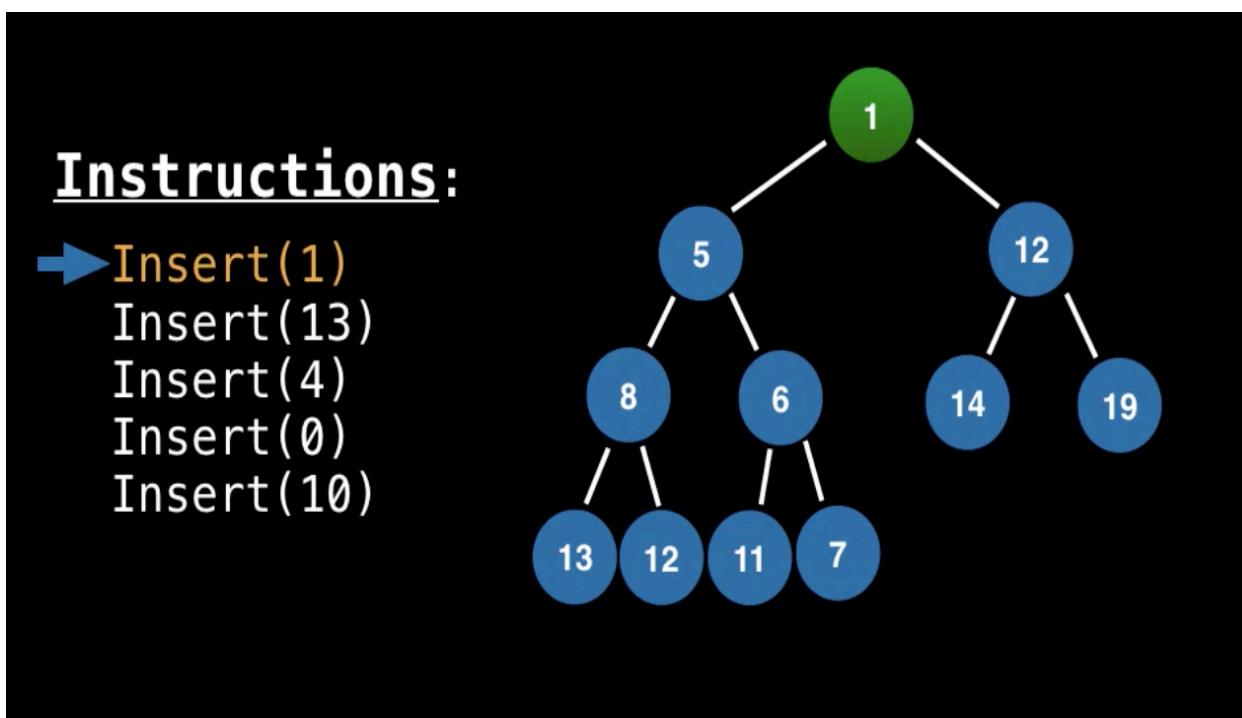
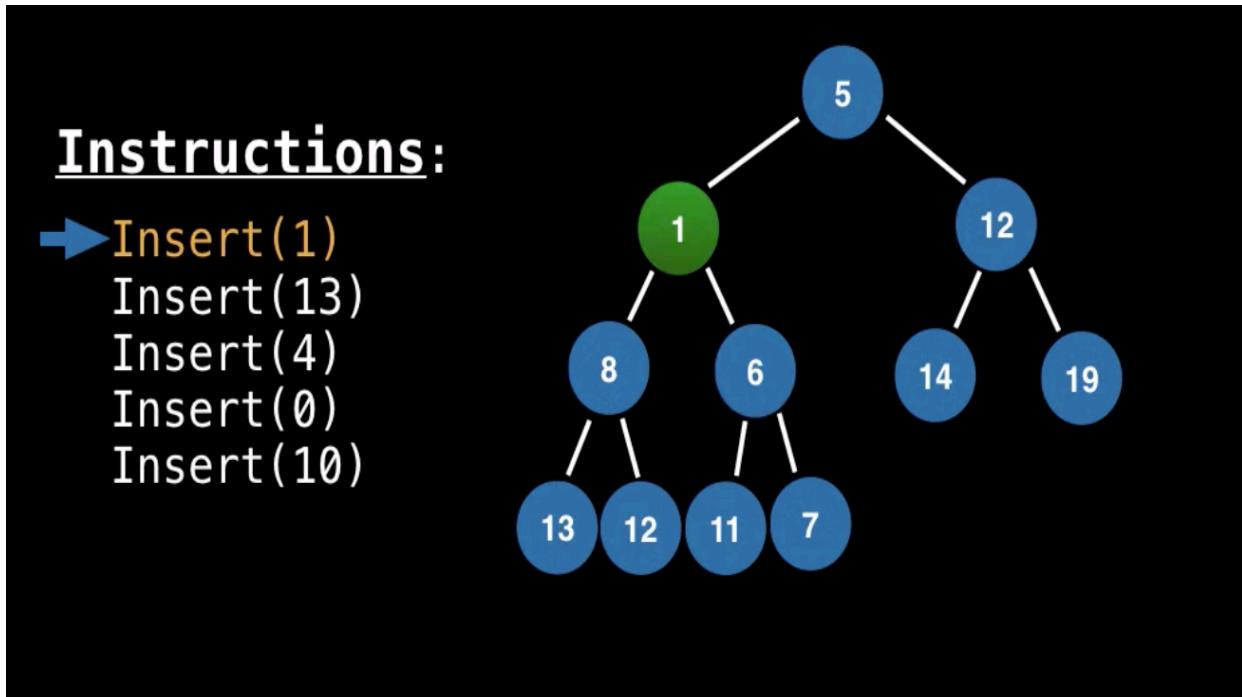
- Insert(1)
- Insert(13)
- Insert(4)
- Insert(0)
- Insert(10)



### Instructions:

- Insert(1)
- Insert(13)
- Insert(4)
- Insert(0)
- Insert(10)

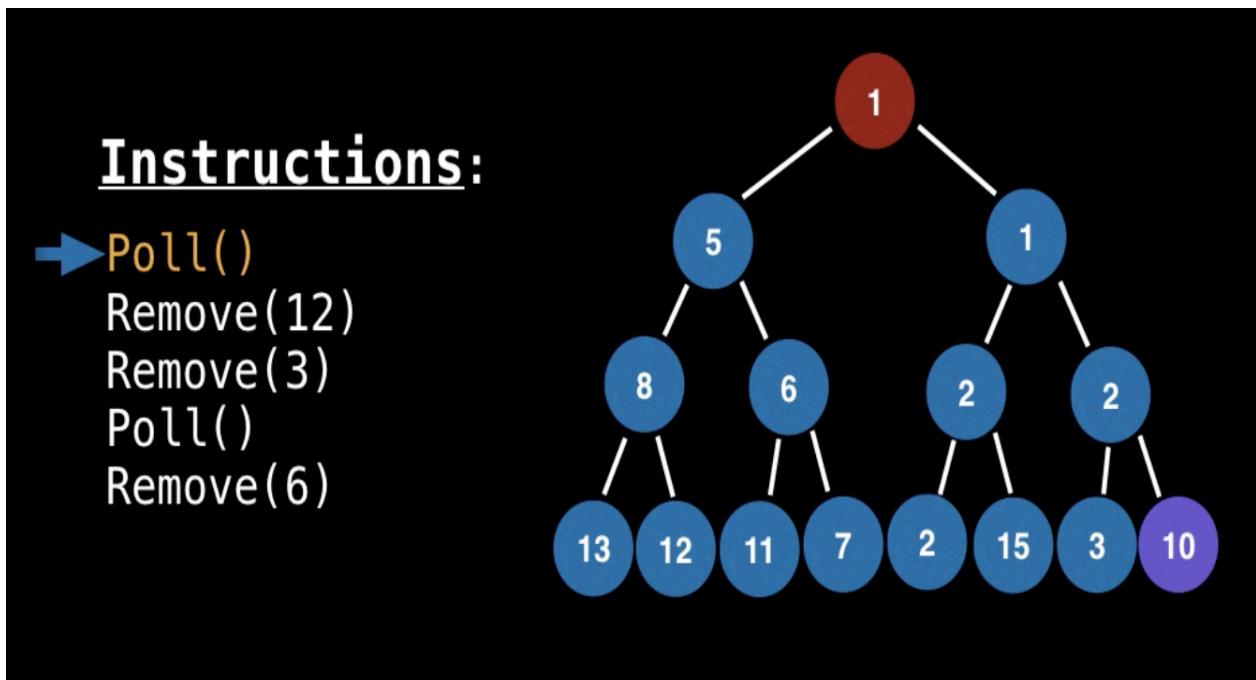


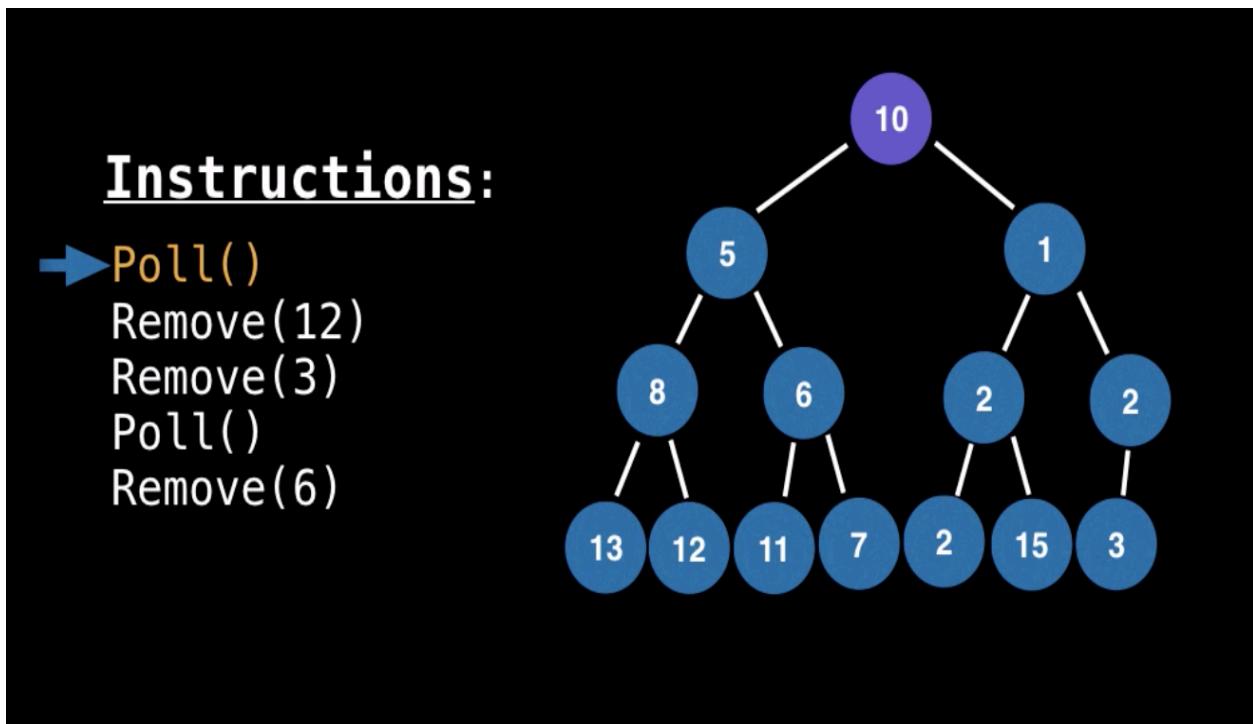
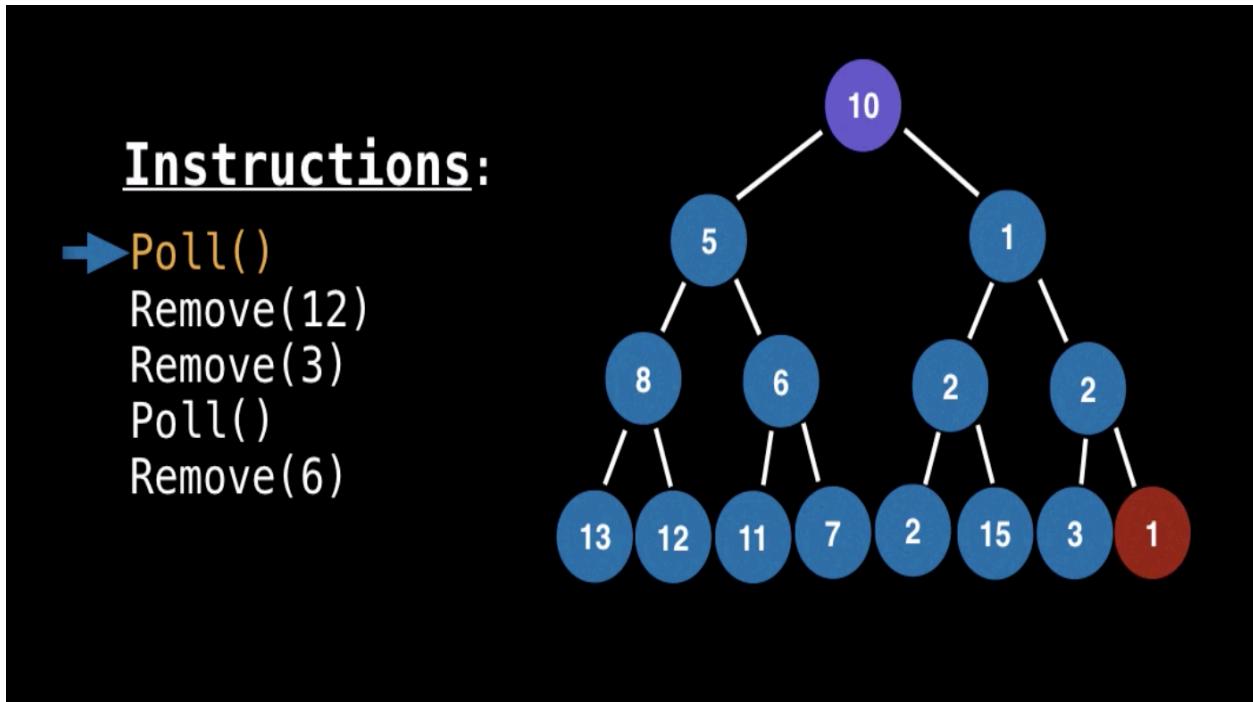


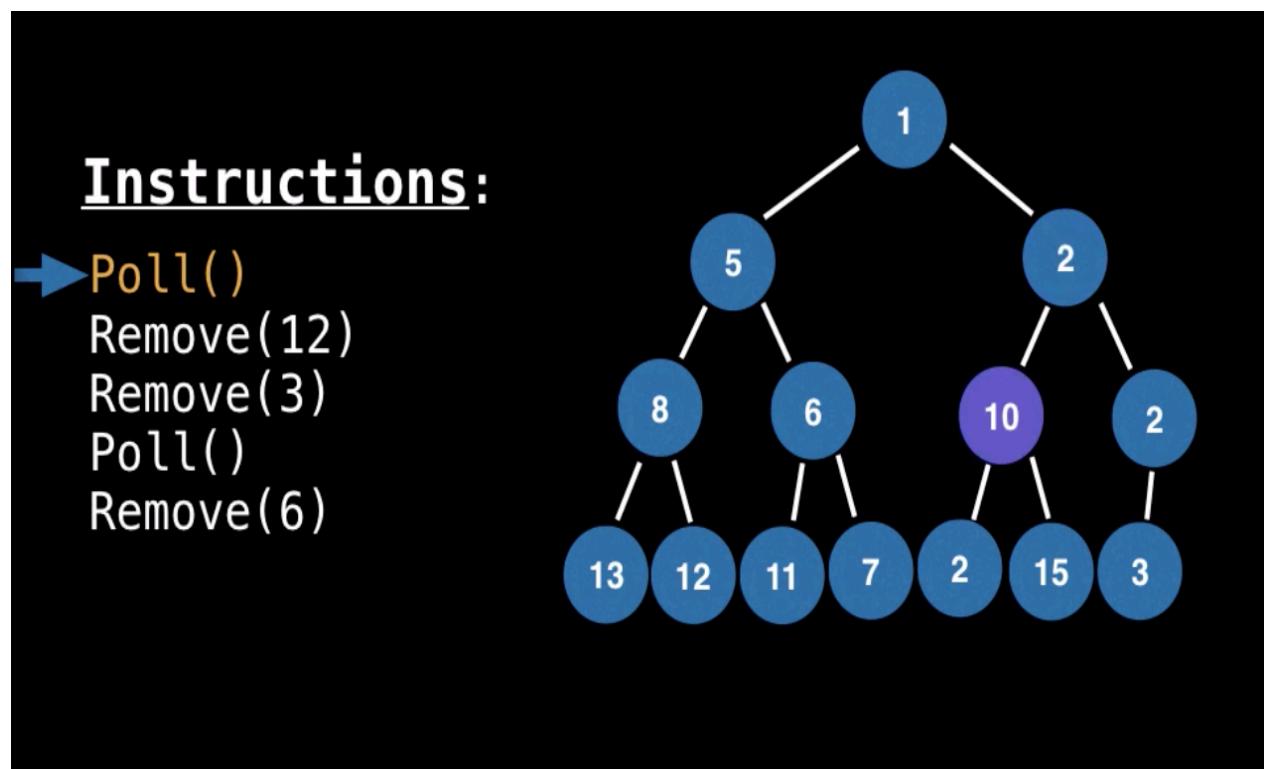
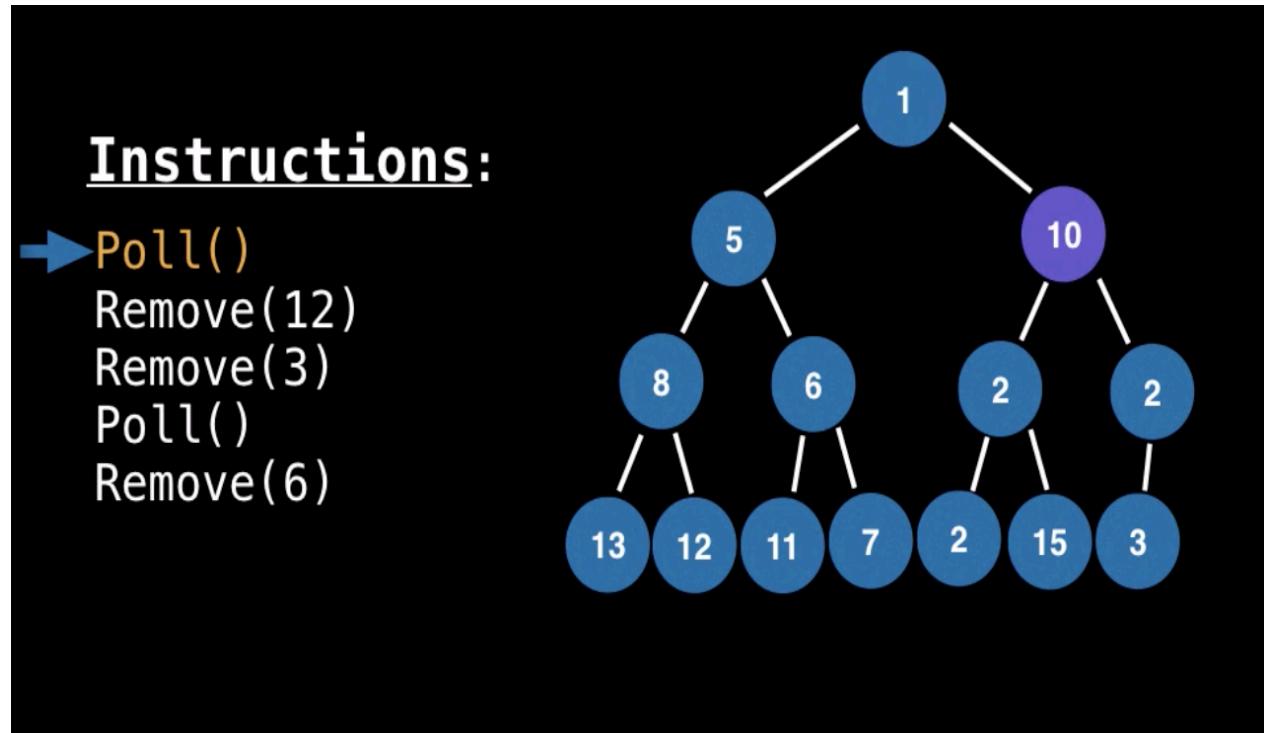
Remove လုပ်တာကတော့ နှစ်မျိုးရှိတယ်။ ရှိုးရိုး poll function နဲ့ ဖျက်ချင်တဲ့ value ကို ထည့်ပေးပြီး ဖျက်ရတဲ့ remove Function ဆိုပြီး ရှိတယ်။ Poll function ကိုအရင်ပြောကြည့်ရအောင်။ ထည့်တုန်းက left to right order နဲ့သွားခဲ့ တာဆိုတော့ ဖျက်မယ်ဆိုရင် right to left ဖြစ်သွားမယ်။

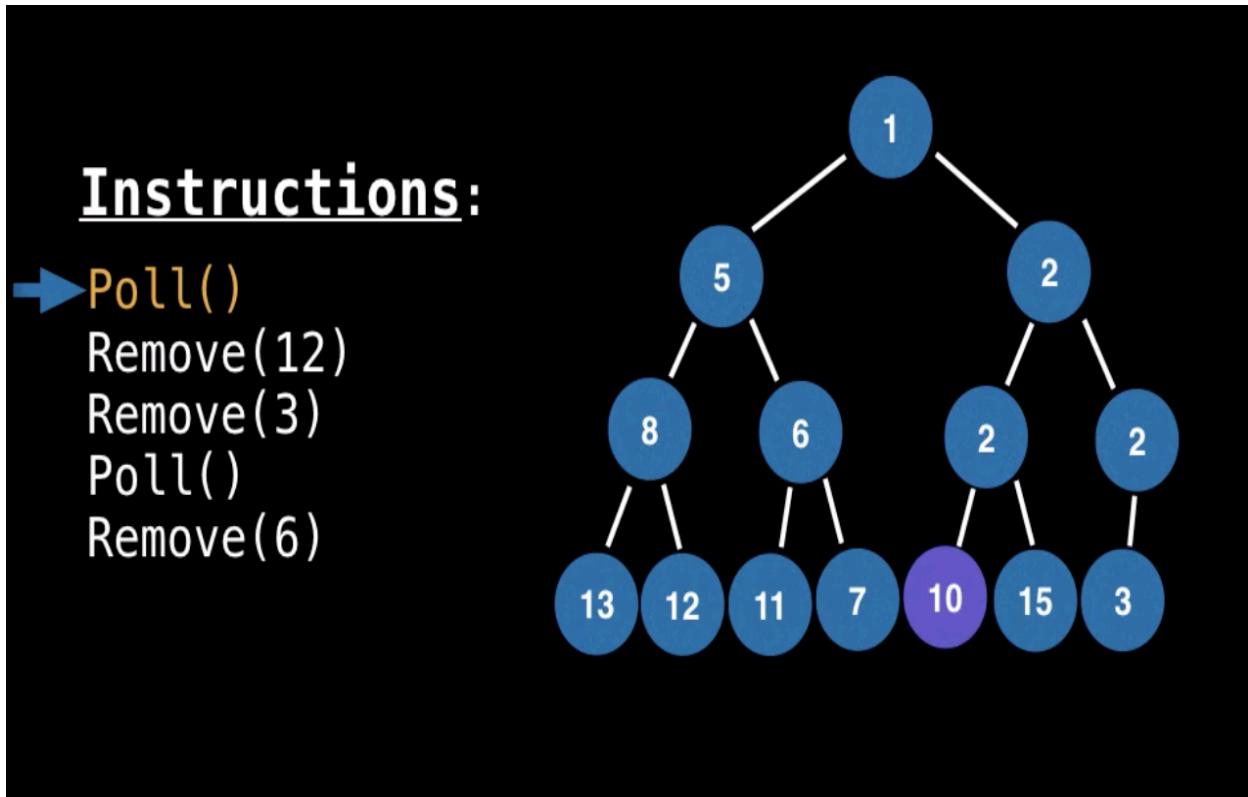
မဖျက်ခင်မှာ ဖျက်ရမယ့် element နဲ့ root node နဲ့ ကို အရင် swap လုပ်တယ်။ ပြီးတော့မှ ဖျက်ချလိုက်တယ်။ root node နေရာမှာ ရောက်သွားတဲ့ element ကို တစ်ဆင့်ခြင်းဆီ heap invariant ဖြစ်အောင် child node တွေနဲ့ replace ပြန်လုပ်တယ်။ bubbling down လုပ်တယ်လိုလဲခေါ်တယ်။ invariant ဖြစ်ပြီးသားဆိုရင်တော့ လုပ်စရာမလိုဘူး။ bubbling down လုပ်ရင်တစ်ခု သတိထားရမှာက parent element တစ်ခုမှာ child ၂ ခုစီ ရှိတယ်။ child ၂ ခုမှာ ငယ်တဲ့ ဘက်ကိုပဲ ရွေးပြီး replace လုပ်ပါတယ်။ အကယ်လို့ နှစ်ခုလုံးက တန်ဖိုးတူနေတယ်ဆို ဘယ်ဘက်ကို ရွေးပါတယ်။

## Removing Elements from Binary Heap (Polling)







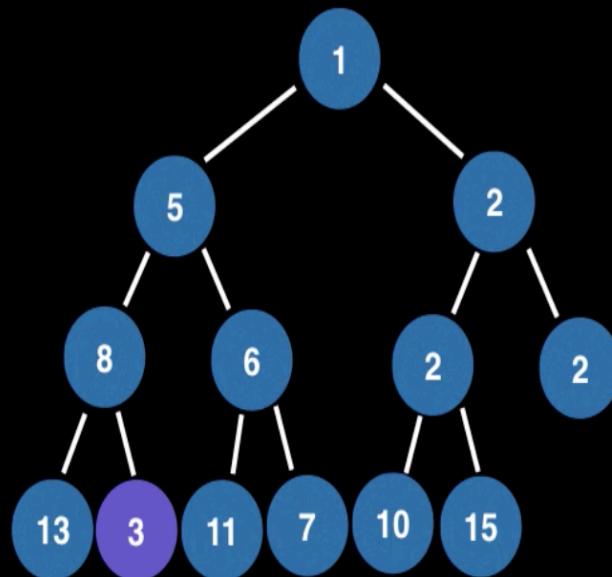


Remove (value param) ကကျတော့ သူမှာ ဖျက်ချင်တဲ့ value parameter ပါလာပြီဖြစ်တဲ့ အတွက် အဲဒီ value ကို tree မှာတစ်ဆင့်ခြင်းဆီ လိုက်ရှာရပါတယ်၊ အပေါ်ဆုံးအတန်းကနေပြီးတော့ အောက်အထိ တစ်ဆင့်ခြင်းဆီ left to right ရှာပါတယ်။ အဲလို ရှာရတဲ့ အတွက် Linear time ကြာတယ်လိုဆိုရမှာပေါ့။ တွေ့ပြီဆိုရင် အဲဒီ element နဲ့ လက်ရှိနောက်ဆုံး မှာရှိနေတဲ့ element ကို swap လုပ်တယ်။ swap လုပ်ပြီး နောက်ဆုံး element ကိုပဲဖြတ်တယ်။ ပြီးရင် heap invariant ဖြစ်အောင် ပြန်စီတယ်။ bubbling up or down ပြန်လုပ်တယ်ပေါ့။

## Removing Elements from Binary Heap

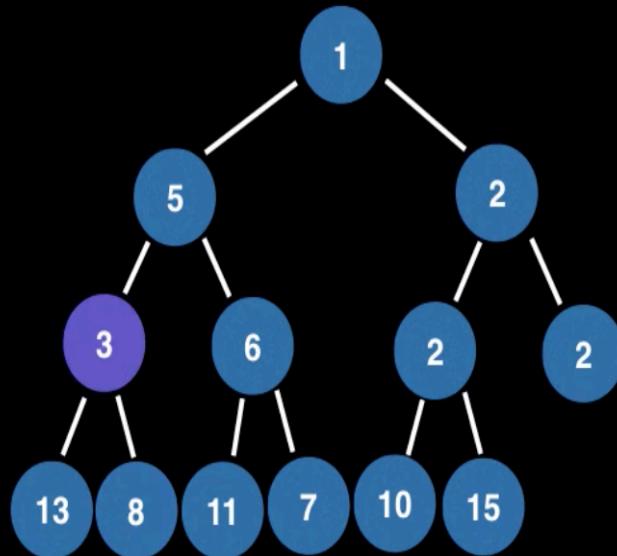
### Instructions:

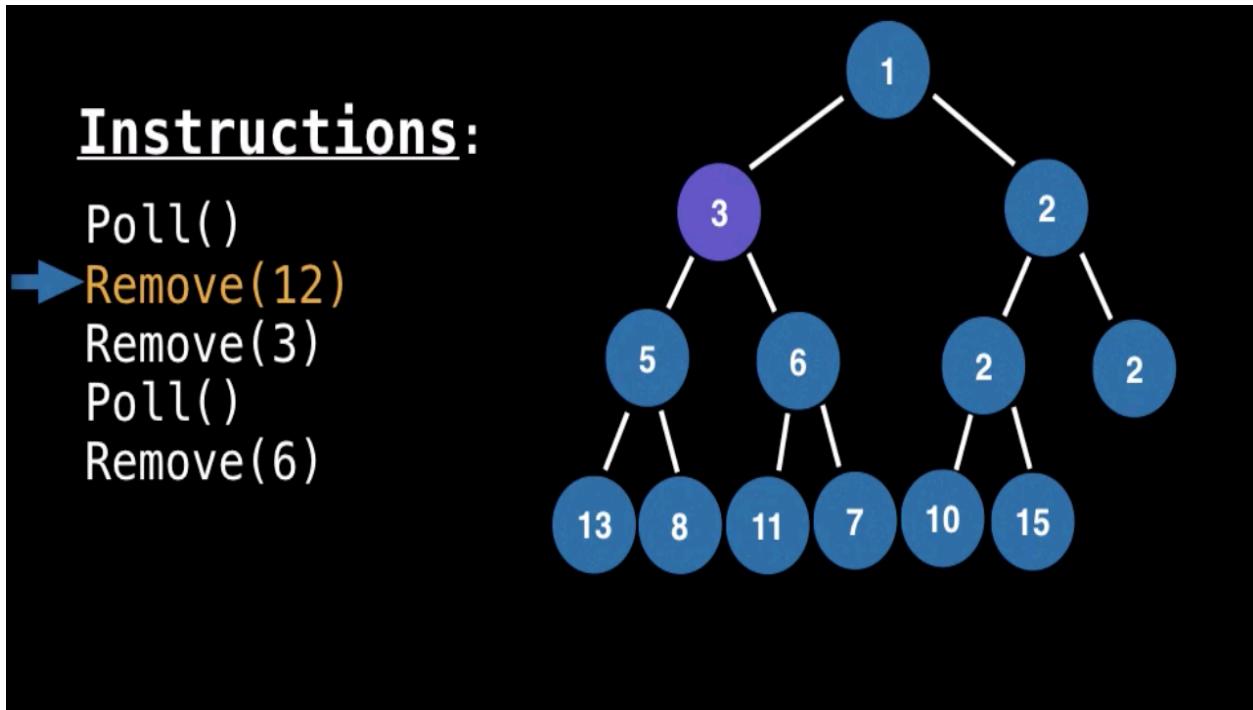
Poll()  
→ Remove(12)  
Remove(3)  
Poll()  
Remove(6)



### Instructions:

Poll()  
→ Remove(12)  
Remove(3)  
Poll()  
Remove(6)





Part 2 မှာ hashtable ကို သုံးပြီးတော့ heap မှာ data ထည့်တာထုတ်တာတွေ ထပ်မြောပြပေးသွားပါမယ်။ လက်ရှိမှာ တစ်ချို့ process တွေ က linear of time ကြာပါတယ်။ ဥပမာ remove operation ဆို ဖျက်ရမယ့် value ကို လိုက်ရှာနေရတယ်။ hashtable မှာတော့ အဲလို လိုက်ရှာနေစရာ မလိုတော့ပဲနဲ့ manage လုပ်သွားနိုင်မှာပါ။

စစဖတ်ဖတ်ခြင်းတော့ နည်းနည်းရှုပ်နိုင်တယ်။ ၂ ခေါက် လောက်သေချာလေးဖတ်လိုက်ရင် ရှင်းသွားမှာပါ။

## Priority Queue With Heap (Part 2)

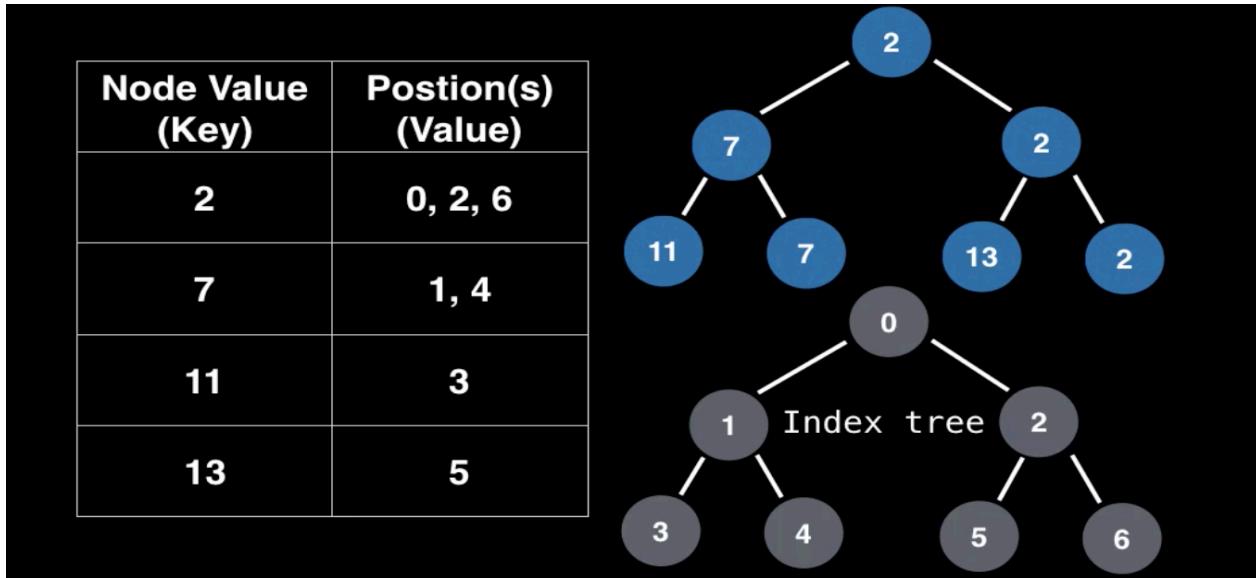
ဒါ part 2 article ကို မဖတ်ခင် part 1 ကို အရင်ဖတ်ထားမှ အဆင်ပြေပါလိမ့်မယ်။

Part 1 မှာရေးခဲ့တဲ့ မူလ remove algorithm က key တစ်ခုရှာပြီဆို tree တစ်ခုလုံးပတ်ရှာနေတဲ့ အတွက် efficiency မကောင်းဘူးလိုအပ်ရမယ်။ အဲအတွက်ကြောင့် hash table ကို သုံးပြီး lookup table ပုံစံမျိုးတစ်ခုဖန်တီးမယ်။ အရှေ့ article မှာတုန်း က heap data represent လုပ်ခဲ့တုန်းက ပြောခဲ့သလိုပဲ element node value တွေကို array index (position value) တွေနဲ့ တွဲပြီးဆွဲထုတ်ပြခဲ့သေးတယ်။ အဲလိုပုံစံမျိုးပါပဲ။

Table မှာ column နှစ်ခုရှိမယ်၊ တစ်ခု က element တွေရဲ့ actual value ကိုပြမယ်။ နောက်တစ်ခုက သူတို့ရဲ့ position key (array index value) တွေပြထားမယ်။ ဆိုတော့ ကျနော်တော့ element တစ်ခု ဖျက်ချင်ပြဆို table ထဲမှာ သူရဲ့ position key ကိုသွားရာလိုက်ခြင်းအားဖြင့် element ဘယ်နေရာမှာ ရှိတယ်ဆိုတာသိလိုက်ရပြီ၊ tree တစ်ခုလုံးတစ်ဆင့်ခြင်းလိုက်ရှာနေစရာမလိုတော့ပါဘူး။

ဒီနေရာမှာ စဉ်းစားစရာတစ်ခု ရှိလာတာက element value တူတာတွေရှိလာရင်ရော ဘယ်လို့ select လုပ်မလဲပေါ့။ လောလေဆယ် ကျနော်တို့လုပ်ထားတဲ့ lookup က element value တစ်ခု ကို position array value တစ်ခု လုပ်ထားတယ်။ ဒီလို scenario မျိုးအတွက် ဆိုရင်တော့ ကျနော်တို့ element value တစ်ခုအတွက် multiple position value ထားလို့ရပါတယ်။ element value က 7 လိုပဲဆိုပါစို့။ 7 => 4,5,9 အဲလို table ထဲမှာ store လုပ်ထားလို့ရပါတယ်။ (fig 1 မှာ sample image ပါပါတယ်)။

## Heap Data Presentation With Hash Lookup table



Hash table အကြောင်းလဲ နားလည်သွားပြီဆိုတော့ သူ့ကိုသုံးပြီးတော့ data operation ဘယ်လိုလုပ်လဲပြောကြည့်ရအောင်။ process တော်တော်များများကတော့ အတူတူပဲဖြစ်တဲ့ အတွက်အရမ်းကြီးတော့ စိမ်းမှာမဟုတ်လောက်ပါဘူး။

Insert ထည့်တော့မယ်ဆို ပုံမှန်အတိုင်းပဲ အောက်ဆုံးအတန်းက နေ left to right သွားတယ်။ ထည့်လိုက်ပြီဆိုတာနဲ့ lookup table မှာလည်း သူ့ရဲ့ element value နဲ့ position value ဝင်သွားမယ်။ နောက်တစ်ဆင့် အနေနဲ့ heap invariant ဖြစ်အောင် bubbling up လုပ်တယ်။ tree မှာ bubbling up လုပ်နေတိုင်းမှာ lookup table မှာလည်း position value တွေ swap လုပ်လုပ်သွားတယ်။ (fig 2.1 to fig 2.4 inserting data using lookup ကပုံတွေနဲ့ တွဲကြည့်ပါ)။

## Inserting Data With Lookup Table

Node Value	Position(s)
2	0, 2, 6
7	1, 4
11	3
13	5
3	7

**Instructions:**

- `insert(3)`
- `remove(2)`
- `poll()`

Node Value	Position(s)
2	0, 2, 6
7	1, 4
11	3
13	5
3	7

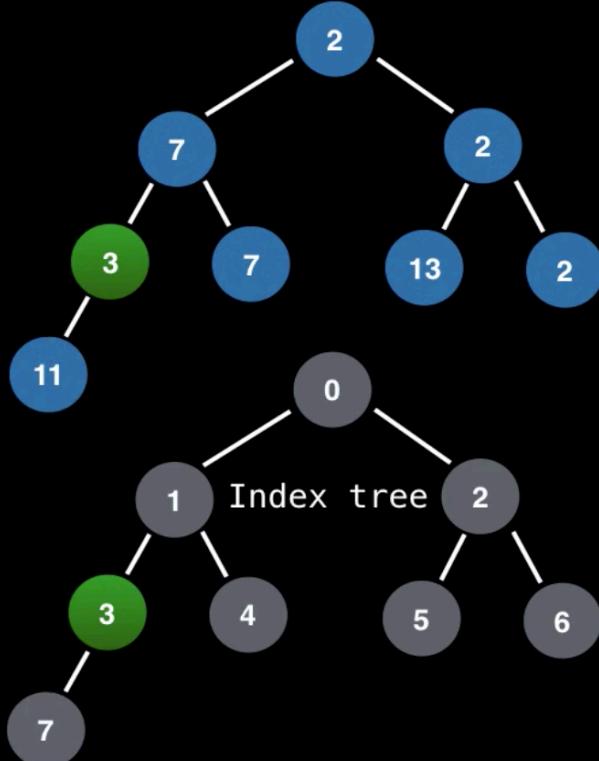
**Instructions:**

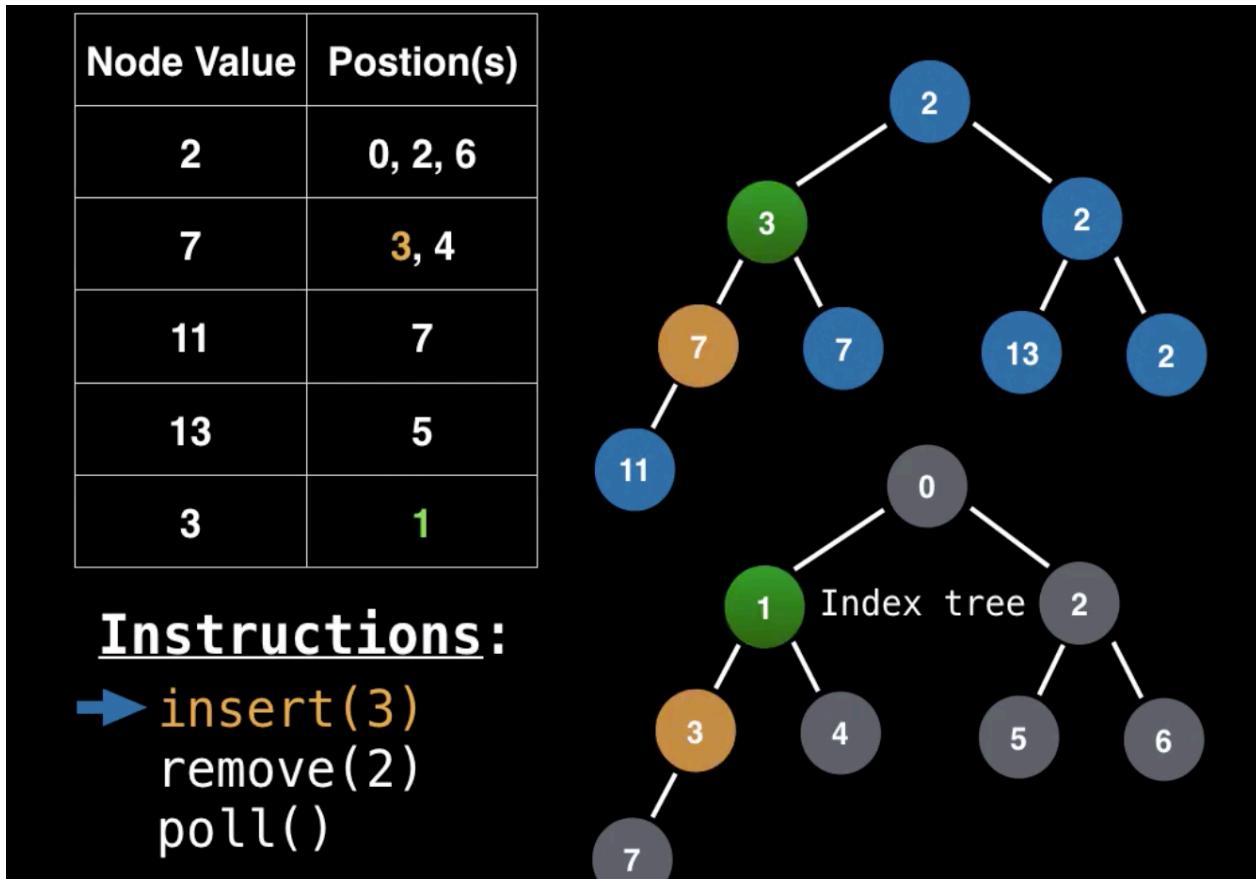
- `insert(3)`
- `remove(2)`
- `poll()`

Node Value	Position(s)
2	0, 2, 6
7	1, 4
11	7
13	5
3	3

**Instructions:**

- `insert(3)`
- `remove(2)`
- `poll()`





Remove operation ကလဲထုံးစံအတိုင်းပဲ ဖျက်မယ့် key ကို lookup table ထဲမှာသွားရှာတယ်။ ဒါပေမဲ့ မေးစရာတစ်ခုရှိတာက same value , multiple position ဖြစ်တဲ့ကောင်တွေ အတွက်ဆိုရင်ရော ဘယ်လိုလုပ်မလဲပေါ့။ အဖြေက တော့ ကြိုက်တဲ့ position ကကောင်ကို ယူဖျက်လိုက်လို့ရပါတယ်၊ ဖျက်ပြီး နောက်ဆုံးမှာ heap invariant ဖြစ်အောင် လုပ်နိုင်ရပါပြီ။ position key ရပြီဆိုရင် ပုံမှန်လုပ်နည်းအတိုင်းပဲ ဖျက်ရမယ့် value နဲ့ tree ရဲ့ နောက်ဆုံး value ကို swap လုပ်တယ်၊ ပြီးတော့ နောက်ဆုံး value ကို ဖျက်တယ်။ swap လုပ်လိုက်တဲ့ value ကို heap invariant ဖြစ်အောင် bubbling up or down ပြန်လုပ်တယ်။ (fig 3.1 to fig 3.4 removing data using lookup ကပုံတွေနဲ့ တွဲကြည့်ပါ)။

## Removing Data With Lookup Table

Node Value	Position(s)
2	0, 2, 6
7	3, 4
11	7
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

Node Value	Position(s)
2	2, 6
7	3, 4
11	0
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

Node Value	Position(s)
2	2, 6
7	3, 4
11	0
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

Node Value	Position(s)
2	0, 2
7	3, 4
11	6
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

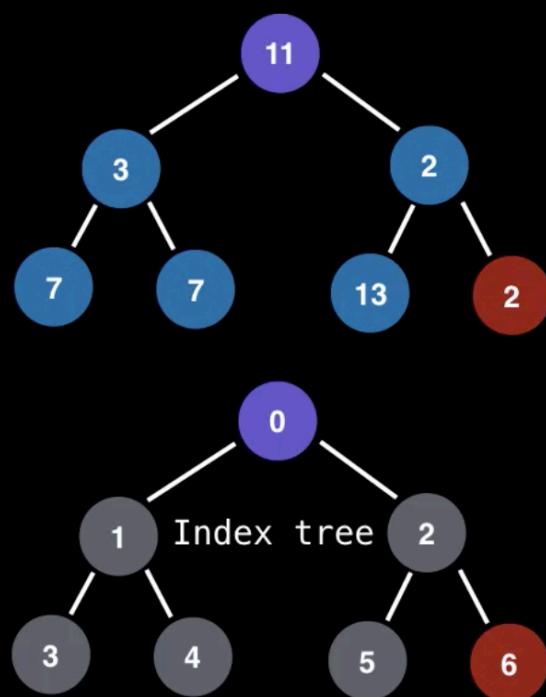
Polling ကတေသ့ ဘာမှပြောင်းလဲတာသိပ်မရှိဘူး။ နောက်ဆုံး value ကို ထိပ်ဆုံးက root value နဲ့ swap လုပ်တယ်။ ပြီးတေသ့ ဖျက်တယ်၊ ထိပ်ဆုံးရောက်သွားတဲ့ value ကို bubble down ပြန်လုပ်တယ်။ tree မှာလုပ်သွားတဲ့ step တွေအတိုင်း lookup မှာလိုက်ပြောင်းတယ်။ (fig 4.1 to fig 4.3 polling data using lookup ကပိုတွေနဲ့ တွေ့ကြည့်ပါ)။

## Polling Data With Lookup Table

Node Value	Position(s)
2	6, 2
7	3, 4
11	0
13	5
3	1

### Instructions:

insert(3)  
remove(2)  
→ poll()



Node Value	Position(s)
2	2
7	3, 4
11	0
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

Node Value	Position(s)
2	0
7	3, 4
11	2
13	5
3	1

**Instructions:**

- insert(3)
- remove(2)
- poll()

ဒီလောက်ဆိုရင်တော့ heap အကြောင်းလည်း တော်တော်ရှင်းသွားပြီ။ hashtable အကြောင်းကိုလဲ intro ဝင်ပြီးသားဖြစ်သွားပါပြီ။ နောက် articles တွေမှာ hashtable အကြောင်းထပ်ရေးပေးသွားပါမယ်။

## Sorting

Sorting ဆိုတဲ့ သဘောတရားကိုတော့ အားလုံးလည်း သိကြမယ်လို့ထင်ပါတယ်။ data structures series ရေးနေပေါ့ sorting နဲ့ ပတ်သတ်တဲ့ article မရှိသေးတာနဲ့ ရေးပေးလိုက်ပါတယ်။ sorting ရဲ့ အဓိက ရည်ရွယ်ချက်ကတော့ data တွေကို အစီအစဉ်တကျ နေရာချထားဖို့ပါပဲ။ ဘာကြောင့် sorting လုပ်ရလဲဆိုတော့ searching algorithm တွေပို့ပြီးတော့ strong ဖြစ်စေဖို့ (ဥပမာ Binary search) ပို့ပြီးတော့ ဖတ်လိုအဆင်ပြေတဲ့ format တွေချထားဖို့အတွက်ပဲဖြစ်ပါတယ် (ဥပမာ dictionary မှာဆို alphabet နဲ့ စီထားသလိုပေါ့)။

Sorting စီတဲ့နေရာမှာလည်း သူ့ဟာနဲ့သူ type တွေအမျိုးမျိုးရှိတယ်။ Sorting စီပြီးတဲ့အခါန်မှာ data value တွေက မစီခင်က sequence အတိုင်းရှိရင် stable sorting . sequence ပျက်သွားရင် unstable sorting । ဒါမျိုးတွေရှိမယ်။ ဥပမာ 3 ပါတဲ့ index က နှစ်ခုရှိတယ်ဆိုပါစို့၊ index3 မှာလဲ 3 । index 6 မှာလဲ 3။ sorting စီပြီးတဲ့ အခါန်မှာ index 3 ရဲ့ 3 ကအရင်လာပြီးမှ index 6 ရဲ့ 3 လာတယ်ဆို sequence အတိုင်းဖြစ်တယ် ဒါဆို stable sorting , index6 က အရင်ရှေ့ရောက် နေတယ်ဆို unstable sorting ပေါ့။

In place နဲ့ out place ကွာတဲ့ sorting သဘောတရားလည်းရှိတယ်။ ဘာကွာသွားလဲဆိုတော့ in place က sorting စီတဲ့နေရာမှာ သူ့ရဲ့ structure ထဲမှာပဲလုပ်တယ်။ ဆိုလိုချင်တာ သူ့အတွက် allocate လုပ်ထားတဲ့ memory space မှာပဲ run လို့ရတယ်။ ဥပမာ bubble sort. out place ဆိုတာကတော့ sorting စီဖို့အတွက် လိုအပ်တာထက်ပို့ပြီးတော့ memory ယူရတယ်။ ဥပမာ tree sort လိုမျိုးပေါ့။

sorting algorithm တွေကတော့ တော်တော်များများကိုရှိပါတယ်။ အဲတဲ့ကမှ bubble sort , quick sort, merge sort , selection sort, insertion sort စသေဖြင့် အသုံးများတဲ့ sorting တွေလည်းရှိပါတယ်။ sorting algorithm တော်တော်များများကို geeksforgeeks မှာသွားဖတ်လို့ရပါတယ်။ တော်တော်စုံပါတယ်။ sorting algorithm အခု 40 လောက်ပါမယ်ထင်တယ်။ အောက်မှ link ပေးထားပါတယ်။

<https://www.geeksforgeeks.org/sorting-algorithms/>

## Union Find

Union Find ကို ရှင်းရှင်းလင်းလင်း ဖွင့်ဆိုရမယ်ဆိုရင် တစ်ကွဲတစ်ပြားစီရှိနေတဲ့ object တစ်ခုခြင်းဆီ ဒါမှုမဟုတ် object အစုလိုက်ကို find လုပ်ပြီးတော့ တစ်စုတစ်စည်းတည်း ဖြစ်အောင် ပေါင်းလိုက်တာကို union process လို့ ပြောလို့ရပါတယ်။ Union Find ကတော့ အများကြီးပြောစရာမရှိပါဘူး၊ သူမှာ အဓိက သုံးတဲ့ function နှစ်မျိုးရှိတယ်၊ သိထားတဲ့ အတိုင်း find & union ပါပဲ။

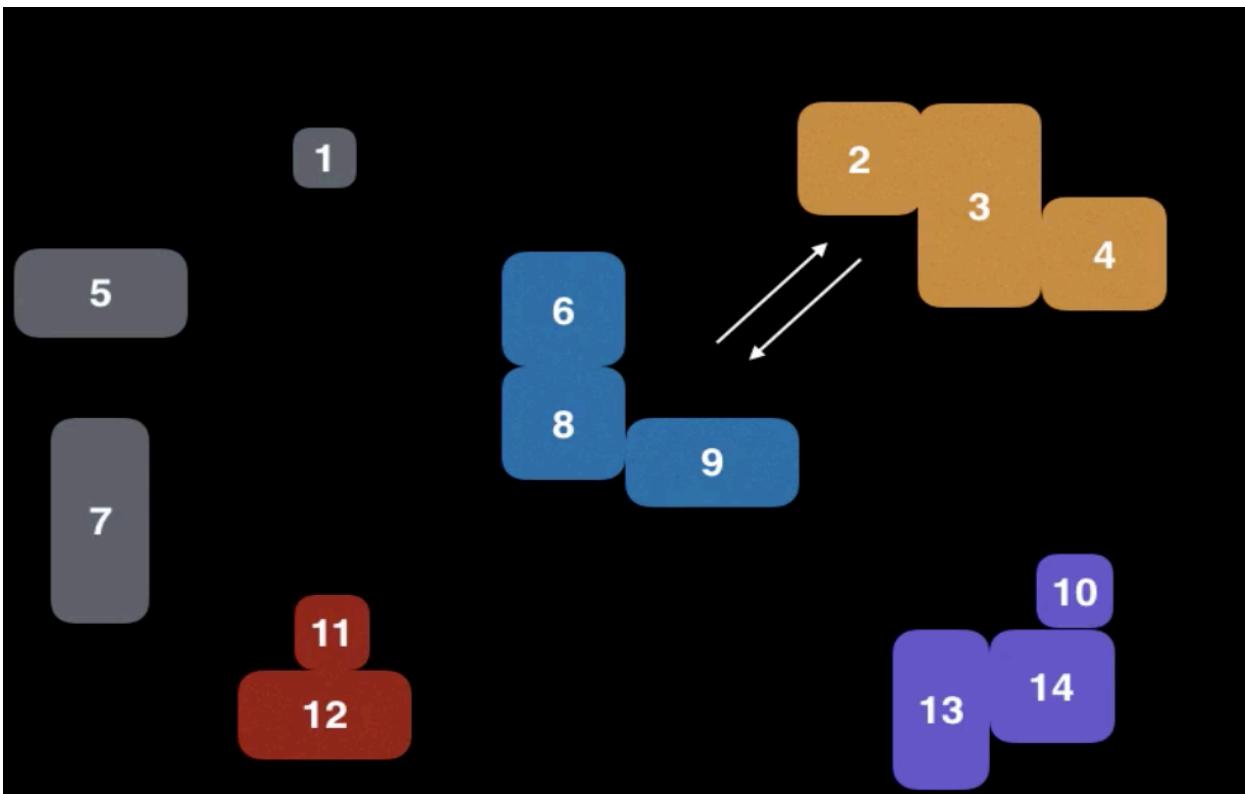
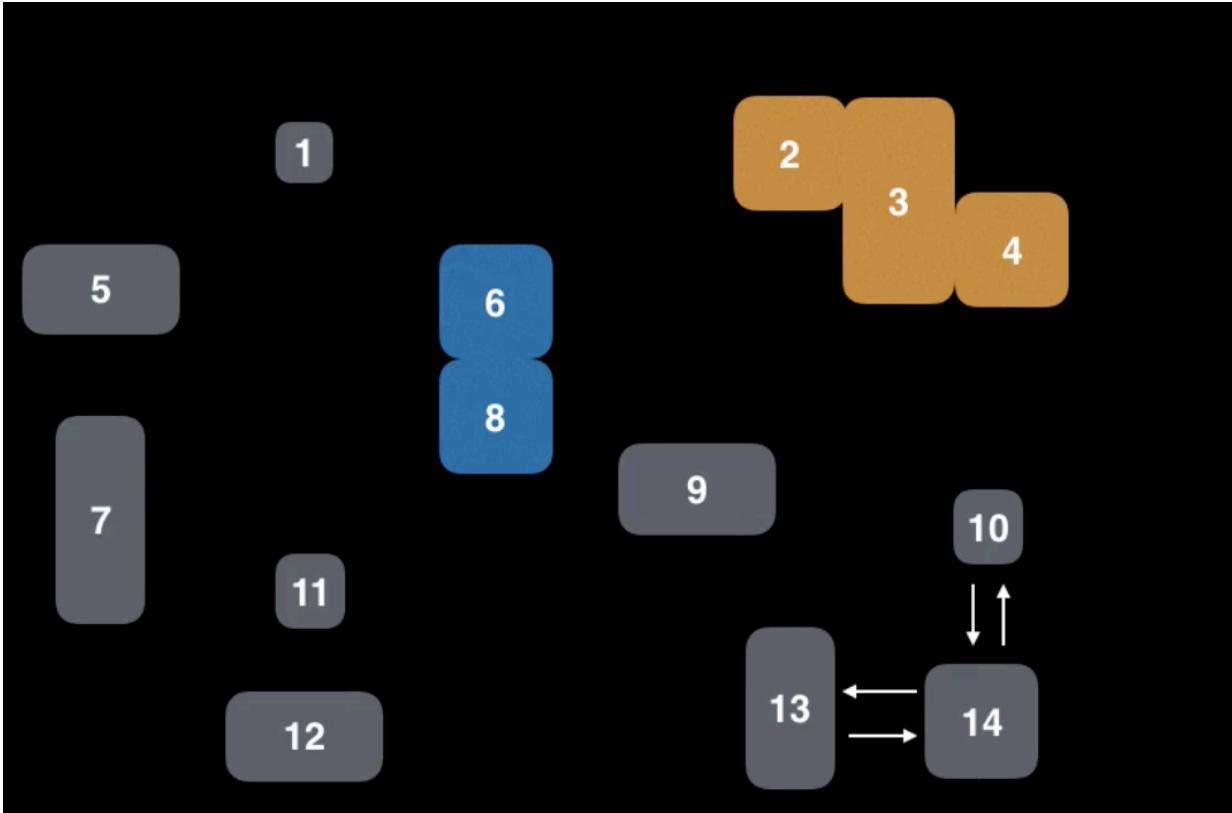
### Find

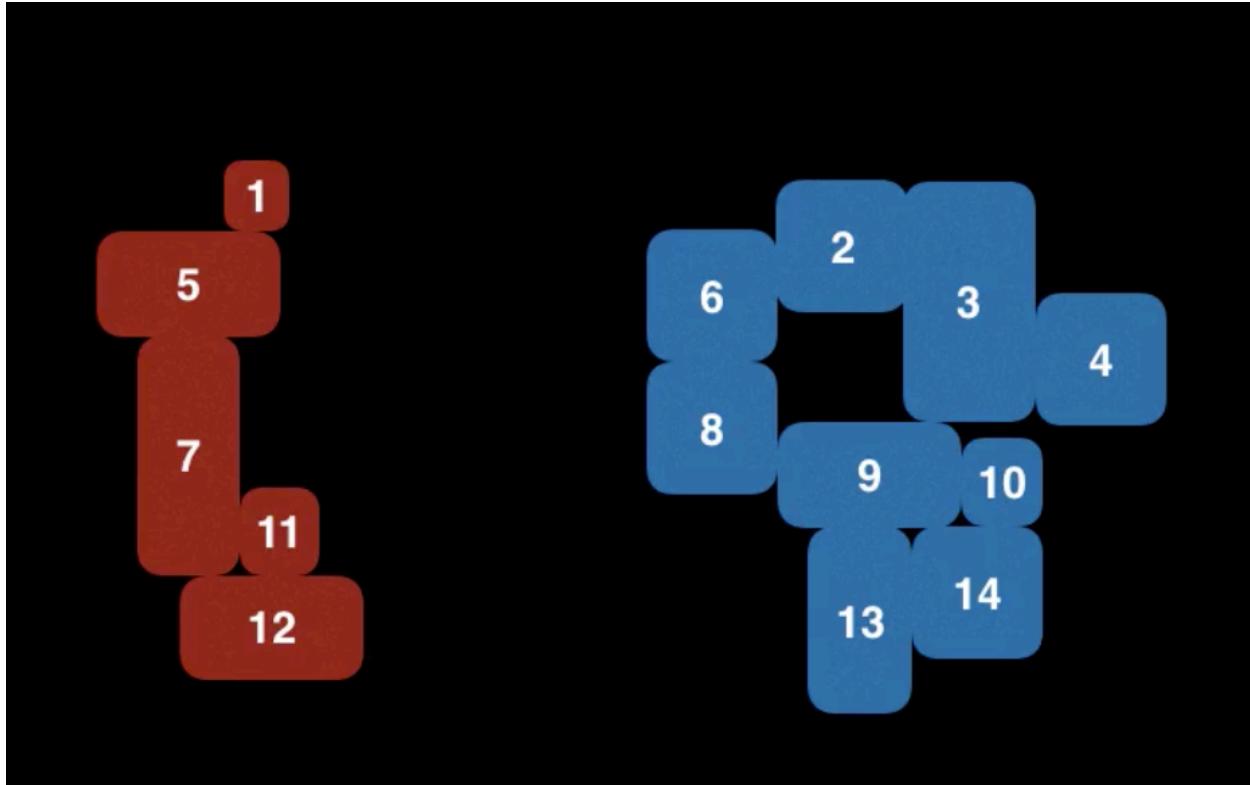
ပုံမှန်ဆို object အစုလိုက်ဆိုလိုရှိရင် parent node တစ်ခုရှိတယ်။ object ကတစ်ခုတည်း ဆိုလိုရှိရင်လည်း သူ့ကိုယ်တိုင်ကိုပဲ parent node အဖြစ် သတ်မှတ်ထားတယ်။ Find က ဘာလုပ်မလဲဆိုတော့ object အချင်းချင်း မပေါင်းခင်(merge) မှာ သူတို့ရဲ့ parent node ကို လိုက်ရှာတယ်။ Union လုပ်မှာက parent node ကို အခြေခံပြီးလုပ်မှာဖြစ်တဲ့ အတွက်ကြောင့်ပါ။

### Union

Find လုပ်ပြီး parent node ကို ရပြီဆိုတာနဲ့ Union လုပ်ပါတယ်။ parent node နှစ်ခုမှာပါတဲ့ objects တွေအားလုံးကိုတစ်ခုတည်း အဖြစ်ပေါင်းလိုက်ပြီး parent node ကိုလဲ တစ်ခုတည်း အဖြစ်သတ်မှတ်လိုက်ပါတယ်။ နှစ်ခုထဲက ဘယ်တစ်ခုကိုဖြစ်ဖြစ်သတ်မှတ်လို့ရပါတယ်။ objects ပိုများများရှိတဲ့ parent node ကိုယူလိုက်တာကတော့ ပိုကောင်းပါတယ်။

## Union Find With Magnet Examples



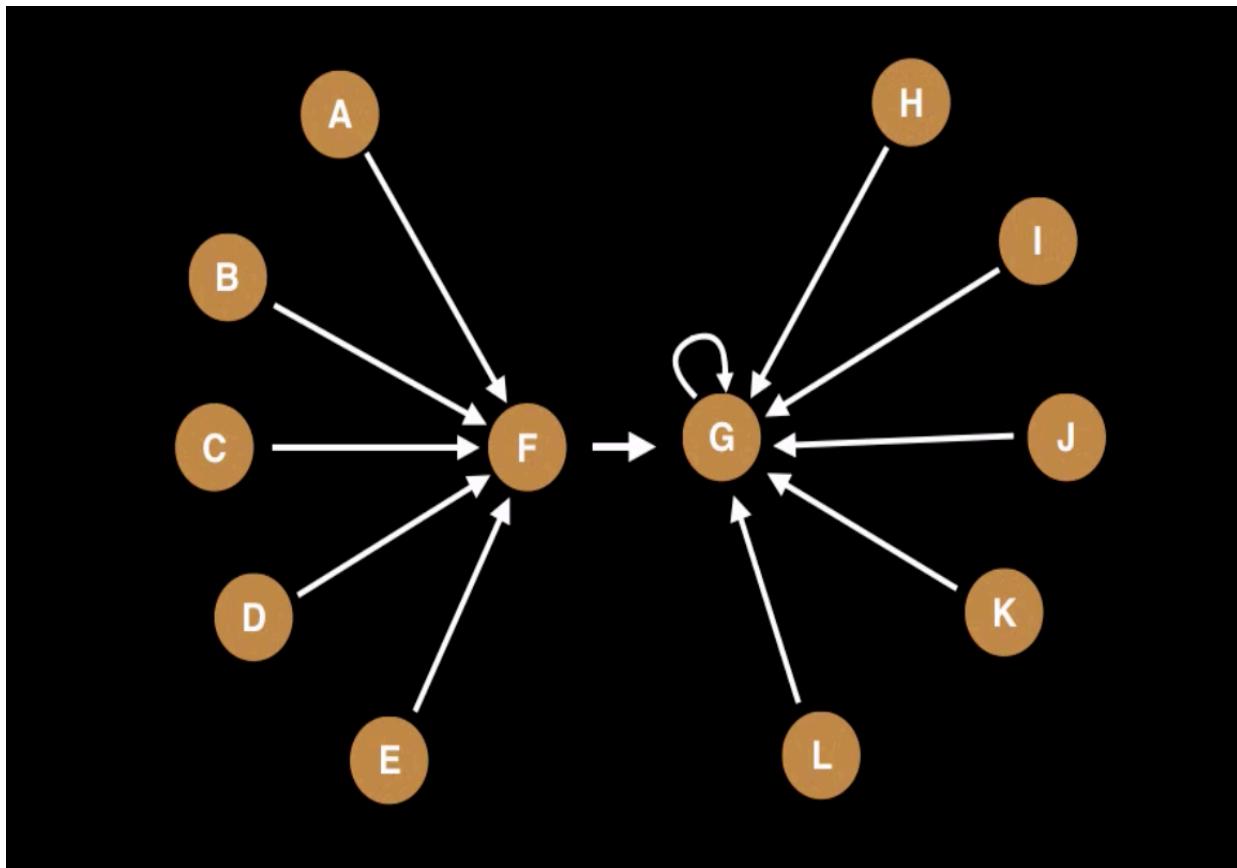


ဒါပေမဲ့ တိုင်ပတ်ကာ တစ်ခုတော့ ရှိတယ်။ အခု လက်ရှိလုပ်နေတဲ့ operation က efficient မဖြစ်ဘူး။ ဘာလို့လဲဆိုတော့ ကျနော်တို့ object နှစ်ခုပေါင်းပြီဆိုပါစို့ A နဲ့ B ပေါင်းတယ်။ A က parent node ဖြစ်သွားတယ်။(A <-B) ပေါ့။ နောက်ထပ် C ဆိုတဲ့ object တစ်ခု ထပ်ပေါင်းမယ်။ ဒီလိုဖြစ်သွားမယ် (A <-B <-C)၊ ဆိုတော့ ကျနော်တို့ C object နဲ့ နောက်ထပ် object တွေ union လုပ်တဲ့အခါ C ရဲ့ parent node ကိုရှာတဲ့ အခါ C ကနေမှုတစ်ဆင့် A အထိသွားရှာရပါတယ်။ ဒါက သုံးခုပဲ ရှိသေးလို့ပါ။ တစ်ကယ်ဆို အများကြီး ရှိနေတဲ့ အချိန်ဆို တစ်ခါတစ်ခါ union လုပ်တော့ မယ်ဆို parent node ကို တစ်ဆင့်ခြင်းဆို သွားရမယ်ဆိုရင်တော့ သိပ်မဟုတ်တော့ပါဘူး။ ဒီလိုဖြစ်လာတဲ့အတွက် path compression ဆိုတဲ့ အရာကို apply လုပ်ထားပါတယ်။

Path compression ဆိုတာက တော့ သူနာမည်အတိုင်းပဲ သွားရမယ့် path ကိုချုပ်လိုက်မှာ။ ဆိုလိုချင်တာက ကျနော်တို့အရင် priority queue article ရေးတုန်းက lookup table နဲ့ node value & position value တွဲပြီးတော့ သိမ်းထားသလိုပဲ။ ဒီမှာလဲ follower node ကို join မလုပ်တော့ဘဲ parent node ကို တိုက်ရှိက်သွား Join လိုက်တယ်။ ဆိုလိုချင်တာက ဒီလို (A <-B <-C) join မယ့်အစား (A <-B), (A <-C) ဆိုပြီး parent node ကိုပဲတိုက်ရှိက် join လိုက်မယ်။ အဲဒါဆိုရင် parent node

ရှာရတော့မယ့်အချိန်မှာ တစ်ဆင့်ချင်းစီ လိုက်ရှာနေစရာမလိုတော့ ဘဲ parent node ကို  
အလွယ်တကူတွေ့ပြီး Union လုပ်နိုင်မှပါ။

## Union Find With Path Compression



ဘယ်နေရာတွေမှာ apply လုပ်လို့ရလဲဆိုရင် connectivity တွေကို unionized လုပ်ထားတဲ့ architecture တွေ algorithm တွေမှာ သုံးလို့ရပါတယ်။ ဥပမာ Kurskal algorithm ဆို node တစ်ခုချင်းဆီကို ascending structure နဲ့ union လိုက်လုပ်ပြီး minimum path ကိုရှာတယ်။  
တစ်ခြား Network connectivity infra တွေမှာလည်း Union Find ကိုသုံးတာတွေရှိပါတယ်။

ကျနော် pseudo code နဲ့ algorithm sample လေးပါထည့်ပေးလိုက်ပါတယ်။

```
function find( x )
    if(x != x.parent)
        x.parent = find(x.parent)
    return x.parent
else
    return x

function union(x,y)
    x = find( x ), y = find( y ) //find the root of each element
    if ( x is equal to y )
        return success
    else
        set x as y's parent
```

## Greedy Algorithm

Greedy algorithm အကြောင်းအရင် မပြောခင်မှာ localized optimum solution နဲ့ globalized optimum solution ကို အရင်ရှင်းလိုက်ပါမယ်။ ကျနော် တို့ program တစ်ခု run တဲ့အခါ အဆင့်တစ်ဆင့်စီ တိုင်းမှာ အကောင်းဆုံး solution ကို ရွှေးသွားတာကို localized optimum solution လိုပေါ်ပြီးတော့ program တစ်ခုလုံးပြီးသွားလို့ နောက်ဆုံးမှာ အကောင်းဆုံး solution ရတာကို globalized solution လိုပေါ်ပါတယ်။ Greedy algorithm ရဲ့ရည်ရွယ်ချက်ကလည်း optimum solution ပေးဖို့အတွက်ပဲ၊ တစ်ဆင့်ခြင်းစီမှာ local optimum solution ကိုရှာပေးရင်းနဲ့ နောက်ဆုံးမှာ global optimum solution ကိုရနိုင်ဖို့အတွက်ပဲ။ ဒါပေမဲ့လည်း သူမှာ drawback တွေရှိနေတဲ့အတွက်ကြောင့် အမြတမ်း global optimum solution တော့မရနိုင်ဘူး။ ဘာလို့လဲဆိုတာကို အောက်မှာဆက်ကြည့်လိုက်ရအောင်။

Greedy က ဘယ်လိုအလုပ်လုပ်လဲဆိုတာ ကျနော် real world example နဲ့ ပြောပြပေးသွားပါမယ်။ ကျနော်တို့မှာ 2 ကျပ်တန် 3 ကျပ်တန် 10 ကျပ်တန် ဆိုပြီး ပိုက်ဆံလေးတွေရှိမယ်။ လိုချင်တာက 15 ကျပ်လိုချင်တယ်။ ဒါလိုရင် greedy က ဘယ်လို run သွားမလဲဆိုတော့ ပထမဦးဆုံး local optimum ဖြစ်တဲ့ အကြီးဆုံး 10 ကိုယူတယ်၊ နောက်တစ်ခေါက်အတွက် 5ကျပ်ဖြည့်ဖို့ကျန်တဲ့အတွက်ကြောင့် 10 ထပ်ယူလို့မရတော့ဘူး၊ အဲ့အတွက် 3 ကိုယူတယ်။ 2 ကျပ်ထပ်ဖြည့်ဖို့ကျန်တယ်။ ဒါ ထပ်ယူလို့မရတော့ဘူး။ 2 ကို ထပ်ယူတယ်။ နောက်ဆုံးမှာ run သွားတဲ့ပုံစံက global optimum လည်း ဖြစ်တယ်။ တစ်ဆင့်ခြင်းဆီမှာလည်း local optimum solution တွေယူသွားနိုင်တယ်။ ဒါက satisfy ဖြစ်တဲ့ condition တစ်ခုပါ။

အကယ်လိုများ ကျနော်တို့ဆီမှာ ၁၀ ကျပ်တန် ရယ် ၇ ကျပ်တန်ရယ် ၁ကျပ်တန်ရယ်ပဲရှိတယ်ဆိုပါစို့၊ greedy ရဲ့ run နေကြပုံစံအတိုင်း ၁၀ကျပ်တန်ကို အရင်ယူမယ်။ ပြီးရင် ၅ကျပ်ထပ်ဖြည့်ဖို့ကျန်တယ်၊ ၇ထပ်လိုက်ရင်လည်းကြီးသွားမှာဖြစ်တဲ့အတွက်ကြောင့် ၁ကျပ်တန်ကို ၅ခါထပ်ဖြည့်ပါတယ်။ ၁၅ကျပ်တော့ ရသွားတယ်၊ ဒါပေမဲ့ သူ ၈ run သွားတဲ့ပုံစံက ( $10+7+5+5+5+5$ ) ဖြစ်သွားတယ်။ process ကို ၆ ခေါက်လောက် run လိုက်ရသလိုဖြစ်သွားတယ်။ အမှန်ဆို  $(7+7+5)$  ဆိုလည်း ၁၅ ရနိုင်ပါတယ်။ ဆိုတော့ greedy run သွားတဲ့ပုံစံက တစ်ဆင့်ခြင်းဆီမှာတော့ local optimum solution ကိုရှာသွားနိုင်ပေမဲ့ global အတွက်ကတော့ အဆင်မပြောဘူးဖြစ်သွားတယ်။

နောက်ထပ် real world example တွေလည်း ရှိသေးတယ်။ စာအရမ်းရည်သွားမှာဆိုးလို့  
ထပ်ထည့်မရေးတော့ပါဘူး။ Optimum solution ကိုရှာတဲ့ Huffman coding algorithm တို့၊  
minimum spanning Tree algorithm တွေဖြစ်တဲ့ prim တို့၊ Kruskal တို့မှာလဲ greedy algorithm  
ကိုသုံးကြပါတယ်။

Greedy algorithm ကို implement လုပ်ဖို့အတွက် program logic  
ကိုကျနော်ထပ်ရှင်းပြပေးပါမယ်။ ရေးရတာလွှယ်တဲ့အတွက်ကြောင့် ကိုယ့်ဘာသာ စမ်းပြီး ရေးကြည့်ဖို့  
suggest လုပ်ချင်လို့ပါ။

ဝင်လာတဲ့ input တွေကို order စီမယ်(optionalပါ)။ ပြီးရင် loop တစ်ခုလုပ်ပြီး destination value  
(ရချင်တဲ့ target value) နဲ့ compare လုပ်ပြီး input ထဲက ရနိုင်တဲ့ အကြီးဆုံး value  
တစ်ခုခြင်းဆီဆွဲထုတ်ပါ ပြီးရင် destination value ကိုတော့ input ထဲက ဆွဲထုတ်ထားတဲ့ value နဲ့  
နှုတ်ထားဖို့လိုပါတယ်။ conditional statement အနေနဲ့ကတော့ loop တိုင်းမှာ destination value  
or (နှုတ်ခံထားရတဲ့ လက်ကျန် destination value) ထက် input ဆွဲထုတ်မယ့် value  
ကကော်သွားလို့မရပါဘူး။ နောက်ဆုံး destination value 0 ဖြစ်တဲ့အချိန် သို့မဟုတ် input ထဲမှာ  
destination value ထက် less than or equal ဖြစ်တဲ့ value မရှိတော့တဲ့အချိန်မှာ program  
ကိုအဆုံးသတ်လို့ရပါပြီ။

## Divide and Conquer Algorithm

Greedy လို algorithmic paradigm တစ်ခုပဲ။ သူက ဘာလုပ်လဲဆိုတော့ problem တစ်ခုရလာပြီ ဆိုရင် အဲ problem ကို sub problem တွေအဖြစ်ခွဲချလိုက်တယ်။ အဲဒါ sub problem တွေကိုလည်း နောက်ထပ် sub problem တွေအဖြစ်ခွဲချလိုက်လို့ရသေးတယ်။ လုံးဝ နောက်ထပ် ထပ်ခွဲထုတ်လို့ မရနိုင်တော့တဲ့ အခြေအနေတစ်ခု အထိ ခွဲချလိုက်တာ တစ်နည်းအားဖြင့် ဆိုရမယ်ဆို problem ကို တစ်ဆင့်ခြင်းဆီ break down လုပ်လိုက်တယ်ပေါ့။ နောက်တစ်ဆင့် အနေနဲ့ ဘာလုပ်လဲဆိုတော့ အဲဒါ ခွဲထားတဲ့ Sub problem လေးတွေတစ်ခုခြင်းဆီကို solution ရှာတယ်။ ရှာပြီး တော့မှ ခုနက ဖြို့ခြားတဲ့ အတိုင်း အဆင့်ဆင့် ပြန်ပေါင်းလိုက်တယ်။ ခွဲထားတုန်းကလည်း recursive approach အနေနဲ့ sub problem တွေ ခွဲထားတယ်၊ ပြန်ပေါင်းတော့လည်း recursively ပဲ အဆင့်ဆင့်ပြန်ပေါင်းတယ်။

အပေါ်မှာပြောထားခဲ့သလိုပဲ divide & conquer က step ၃ ခုနဲ့အလုပ်လုပ်တယ်။

### Divide

Problem တွေကို breaking down အရင်လုပ်တယ်။ recursive approach ကိုသုံးပြီးတော့ problem တစ်ခုကို sub problem တွေအဖြစ်ခွဲချတယ်၊ နောက်ထပ် sub တွေခွဲချလို့မရတော့တဲ့ အထိပေါ့။

### Conquer

Sub problem တွေကို solving လုပ်တဲ့အဆင့်ပေါ့၊ ခွဲချထားတဲ့ problem တစ်ခုခြင်းဆီကို solve လုပ်လိုက်ပါတယ်။

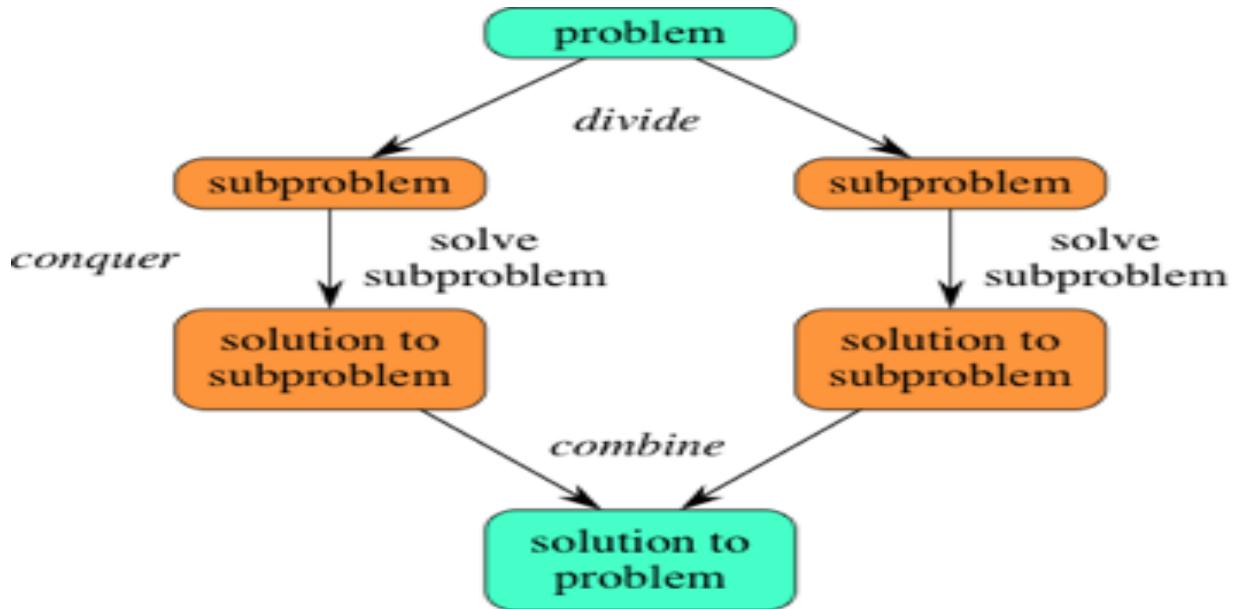
### Combine

Conquer stage မှာ solved လုပ်ထားတဲ့ sub solution တွေကို recursively ပြန် merge လုပ်ပြီးတော့ နှင့် ခွဲချလိုက်တဲ့ မူရင်း problem အတွက် solution ထုတ်ပါတယ်။

ဒီလောက်ဆိုရင်တော့ divide & conquer အကြောင်းကို အကြမ်းဖျင်းအားဖြင့် နားလည်လောက်ပြီ ထင်ပါတယ်။

ဘယ်လိုနေရာတွေမှာ apply လုပ်လဲဆိုတော့ problem ကို divide လုပ်တာတို့၊ recursively run တာတို့ လုပ်တဲ့ sorting algorithms တွေဖြစ်တဲ့ quick sort တို့၊ merge sort တို့ မှာသုံးတယ်။

Binary search algorithm မှတဲ့ apply လုပ်ထားတယ်။ sorting algorithm တွေနဲ့ binary search အကြောင်းလဲ နောက် articles တွေမှာ algorithm code sample တွေနဲ့အတူ ရေးပေးသွားပါမယ်။



## Dynamic Programming

Dynamic programming approach ကလည်း အရှေ့မှာ ရေးထားခဲ့တဲ့ greedy နဲ့ divide & conquer ပုံစံတွေနဲ့ဆင်ပါတယ်။ divide & conquer လိုမျိုး sub problem တွေခဲ့ချုပြုး recursion လုပ်ပြီးပြန်ပေါင်းတယ်။ greedy လိုမျိုး optimum solution ရဖို့အတွက်လည်း လုပ်ထားတယ်။ ဘယ်လို လုပ်ထားလဲဆိုတာ တစ်ချက်လောက်ကြည့်လိုက်ရအောင်။ ဒါ article မဖတ်ခင် divide & conquer နဲ့ greedy ကို အရင်ဖတ်ကြည့်ဖို့ အကြံပေးချင်ပါတယ်။

Dynamic programming မှာ problem တစ်ခု ရှိလာတဲ့အခါမှာ divide & conquer approach ပုံစံမျိုးပဲ sub problem တွေအဖြစ်ခဲ့ချုလိုက်တယ်။ ခွဲပြီးတာနဲ့ အဲဒီ sub problem တွေကို solution လုပ်တယ်၊ sub problem တွေခဲ့တဲ့ အချိန်က နေပြီးတော့ solution formula ထွက်တဲ့အထိ dynamic programming approach က memorized လုပ်ထားတယ်။ နောင်တစ်ချိန် ဒီလို မျိုး sub problem တွေပြီဆို ပြန်ပြီးတော့ သုံးလို့ရအောင်လိုပါ။

ဥပမာ ကျနော်တို့ sub problem တစ်ခုမှာပေါ့ x ကိုရှာတဲ့ solution ကိုရထားတယ်။ နောက်တစ်ခါ x ရဲ့တန်ဖိုးကိုထပ်ရှာရမယ့် sub problem မျိုး ရှိလာခဲ့ရင် ကျနော်တို့မှာ x ရဲ့တန်ဖိုးကို ရှာနိုင်တဲ့ formula ကို memorized လုပ်ထားတဲ့အတွက် နောက်တစ်ခါကိုထပ်ရှင်းစရာ မလိုတော့ဘဲ optimum ဖြစ်တဲ့ solution ကိုရတယ်။ ဥပမာ နောက်တစ်ခု အနေနဲ့ ထပ်ရှင်းအောင် ပြောရမယ်ဆိုရင် 3+4+5 ဆို 12 ရမယ်။ အဲဒီကို 1 ထပ်ပေါင်းမယ်ဆိုပါစို့၊ 13 လိုက်နဲ့ပြောနိုင်တယ်။ ဘာလို့လဲဆိုတော့ ကျနော်တို့ က 1 ကိုပဲထပ်ဆင့် ပေါင်းလိုက်တာ 3+4+5 ပေါင်းတာကို ထပ်မတွက်တော့ဘဲ memorized လုပ်ထားတဲ့ 12 နဲ့ 1 ကိုပဲပေါင်းလိုက်တဲ့အတွက် solution က မြန်မြန်ဆန် ထွက်လာတာပါ။

ဆိုတော့ dynamic programming ကိုဘယ်နေရာတွေမှာ apply လုပ်လို့ရလဲဆိုတော့ problem တွေကို sub structure အဖြစ်ခဲ့ချုလို့ရမယ်၊ ပြီးတော့ sub problem တွေက overlap ဖြစ်လာမယ်ဆိုရင် memorized လုပ်ထားတဲ့ solution ကိုအစားထိုးပြီးတော့ optimize ဖြစ်အောင်လုပ်တယ်။

Divide & conquer မှာတုန်းကတော့ sub problem တွေကို ပြန်ပေါင်းပြီးတော့ over all solution တစ်ခုထုတ်တယ်။ dynamic programming ကတော့ sub problem တစ်ခုခြင်းဆီရဲ့ formula ကို

memorized လုပ်ထားပြီးတော့ နောက်တစ်ဆင့် နောက်တစ်ဆင့် problem တွေအတွက် optimized လုပ်နိုင်ဖို့ကို ပိုပြီး အာရုံစိုက်ထားတယ်။ CPU ရဲ့ computing power ကိုလည်း သက်သာစေတော်ပေါ့။

Greedy မှာတုန်းကတော့ local optimization အတွက် အဆင်ပြုပြီးတော့ dynamic programming ကကျ local အတွက်တင် မဟုတ်ဘဲ overall problem တစ်ခုလုံးအတွက် အဆင်ပြုအောင် address လုပ်ထားတယ်။

ဒီလောက်ဆိုရင်တော့ ဖြင့် dynamic programming အကြောင်းကို theory အရ အကြမ်းဖျင်းနားလည်သွားလောက်ပြီထင်ပါတယ်။

## What is a tree & binary tree?

Data Structure ရဲ့ အပိုင်းမှာ tree ဆိုတာက တော်တော်လေးအရေးပါတယ်။ algorithm တော်တော်များများကို tree တွေနဲ့ represent လုပ်ထားကြတဲ့ အတွက်ကြောင့်ဖြစ်ပါတယ်။ ဒီနေ့မှာ ကျနော် tree ဆိုတာဘာလဲ ဆိုတဲ့ အကြောင်းအရာကို အသုံးများတဲ့ binary tree နဲ့ ဥပမာ ပြုပြီးဆွေးနွေးသွားပါမယ်။

Tree ဆိုတဲ့ သဘောတရားကတော့ ရှင်းတယ်။ Node တစ်ခုနဲ့ တစ်ခု ကို tree structure အတိုင်း တစ်ဆင့်ခြင်းဆီ ချိတ်ဆတ်ထားတယ်၊ ဒါပေမဲ့ ချိတ်ဆက်ထားတဲ့ node တွေက circular ပုံစံ ဖြစ်သွားတယ်ဆို tree လို့ ခေါ်လို့ မရတော့ဘူး။ binary tree အကြောင်းကို ဆက်ကြည့်လိုက်မယ်ဆို tree ဆိုတာ ဘယ်လိုလဲ ဆိုတာ တိတိကျကျမြင်သွားပါလိမ့်မယ်။

Binary tree/binary search tree အတူတူပါပဲ၊ binary tree လို့ ပြောလိုက်ပြီဆိုတာနဲ့ Node တစ်ခုမှာ child node နှစ်ခု လို့ ပြေးမြင်ထားလို့ ရပါတယ်။ binary tree မှာ လည်း သူ့ရဲ့ behavior လေးတွေလဲရှိတယ်။ Node တစ်ခုမှာ အများဆုံး child node နှစ်ခုပဲရှိရမယ်။ left child က parent node ထက် နှည်းရမယ်၊ right child က parent node ထက် ကြီးရမယ်။

Tree structure မှာ လည်း သူ့ရဲ့ အခေါ်အဝေါ်လေးတွေရှိတယ်။ Root - tree တစ်ခုလုံးရဲ့ ထိပ်ဆုံးက Node. Parent - parent node ဆိုတာကတော့ သိတဲ့ အတိုင်း sub tree (child node) တွေရှိ(ပိုင်ဆိုင်)တဲ့ node. Child - node တစ်ခုရဲ့ အောက်မှာ sub tree အနေနဲ့ ရှိနေတဲ့ node. Subtree – node တစ်ခုရဲ့ အောက်မှာ ရှိတဲ့ sub tree structure တစ်ခု(parent ရှိတယ်၊ child ရှိတယ်။). Leaf - child node တွေမရှိတဲ့ node. Tree structure မှာ လည်း အဆင့်ဆင့် သတ်မှတ်ထားတဲ့ level တွေလည်း ရှိတယ်၊ ဥပမာ root node က level 1, root node ရဲ့ အောက်က level 2. Tree structure မှာ search လုပ်တဲ့ အခါတို့ တစ်ခြား operation တွေလုပ်တဲ့ အခါမှာ node တွေတစ်ခုခြင်းဆီ ကို လိုက်ထောက်တာကို လည်း traversing လုပ်တယ်လို့ ခေါ်ပါတယ်။

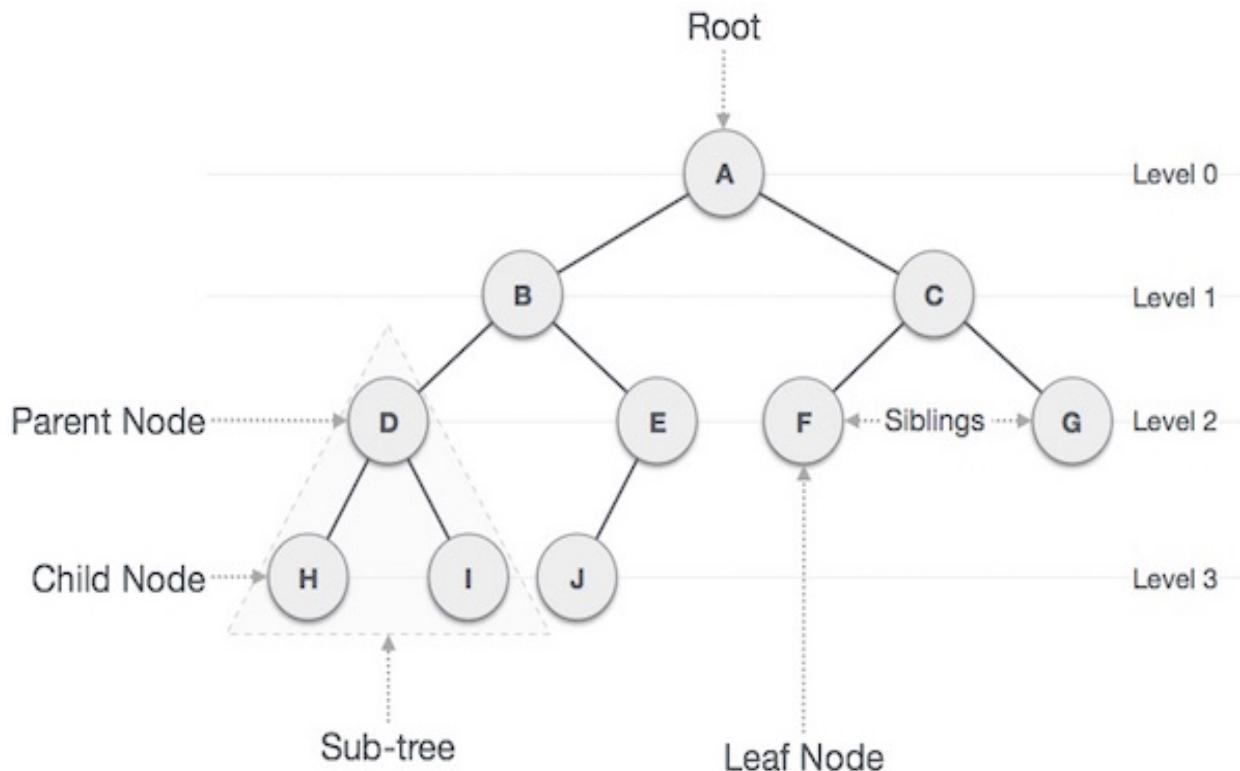
## Binary Tree မှာ Basic Operations တွေကို ကြည့်မယ်ဆိုရင်

- Inserting
- Searching
- Removing ရှိမယ်။

## Node တွေကို Traversal လုပ်တဲ့ နေရာမှာဆိုရင်

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal ဆိုပြီးရှိပါတယ်။

ဒါ လောက်ဆိုရင်တော့ tree ဆိုတာကဘာလဲနဲ့ binary tree အကြောင်းကို Introduction အနေနဲ့  
လုံလောက်ပြီလို့ထင်ပါတယ်။ နောက်နေ့တွေမှာ Basic operation တွေအတွက် article တစ်ခု၊  
traversal လုပ်တဲ့ ပုံစံတွေအတွက် article တစ်ခု စီ သက်သက်ထပ်တင်ပေးသွားပါမယ်။



## Binary tree's operations

ဟိုတစ်နွောကတော့ binary tree အကြောင်းကို ပြောပြပေးပြီးသွားပြီဆိုတော့ အခါ binary tree ရဲ့ operation တွေအကြောင်းကို ပြောပြပေးသွားပါမယ်။ operation တွေကတော့ insert , search , remove ဆိုပြီးတော့ ရှိမယ်၊ တစ်ခုခြင်းဆို ဘယ်လို အလုပ်လုပ်လဲဆိုတာနဲ့အတူ algorithm အတွက် pseudo sample တွေပါ ရေးပေးသွားပါမယ်။

Operation လုပ်တော့မယ်ဆို ကျနော်တို့ အရင်ဆုံး binary tree ရဲ့ property ကို တစ်ချက် remind လုပ်ရပါမယ်။ node တစ်ခုမှာ child နှစ်ခုရှိတယ်၊ left child က အဲဒီ parent node ထက် ယောက်ရမယ်၊ right child က parent node ထက် ကြိုးရမယ်။ ဒါကို မှတ်ထားဖို့လို့မယ်။ operation ဘယ်လိုလုပ်လဲကြည့်ရအောင်။

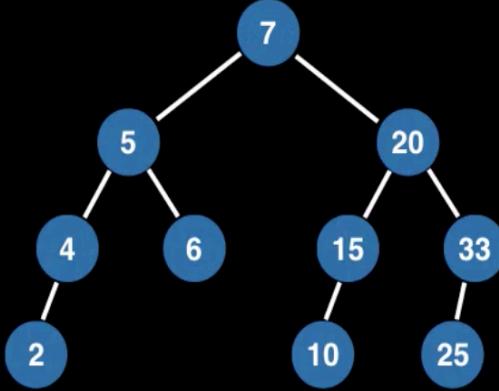
### Insert Operation

Node တစ်ခုစုစုပေါင်းတော့မယ်ဆိုရင် သူ့ရဲ့ထည့်ရမယ့်နေရာကို စဉ်းစားရပါတယ်။ ဘာမှုမရှိသေးတဲ့ tree ကို node တစ်ခုစုစုပေါင်းပြီဆိုရင်တော့ အဲဒီ node က root node ဖြစ်သွားမယ်၊ နောက်ထပ်ထပ်ထည့်ပြီဆို ထည့်တော့မယ့် value က root/parent node ထက် ယောက်လား၊ ကြိုးလား ကြည့်ရမယ်၊ ယောက်လား ဘယ်ဘက်မှာထည့်၊ ကြိုးတယ်ဆို ညာဘက်မှာထည့်ပါမယ်။

# Adding elements to a BST

## Instructions:

```
insert(7)
insert(20)
insert(5)
insert(15)
insert(10)
insert(4)
insert(4)
insert(33)
insert(2)
insert(25)
insert(6)
```

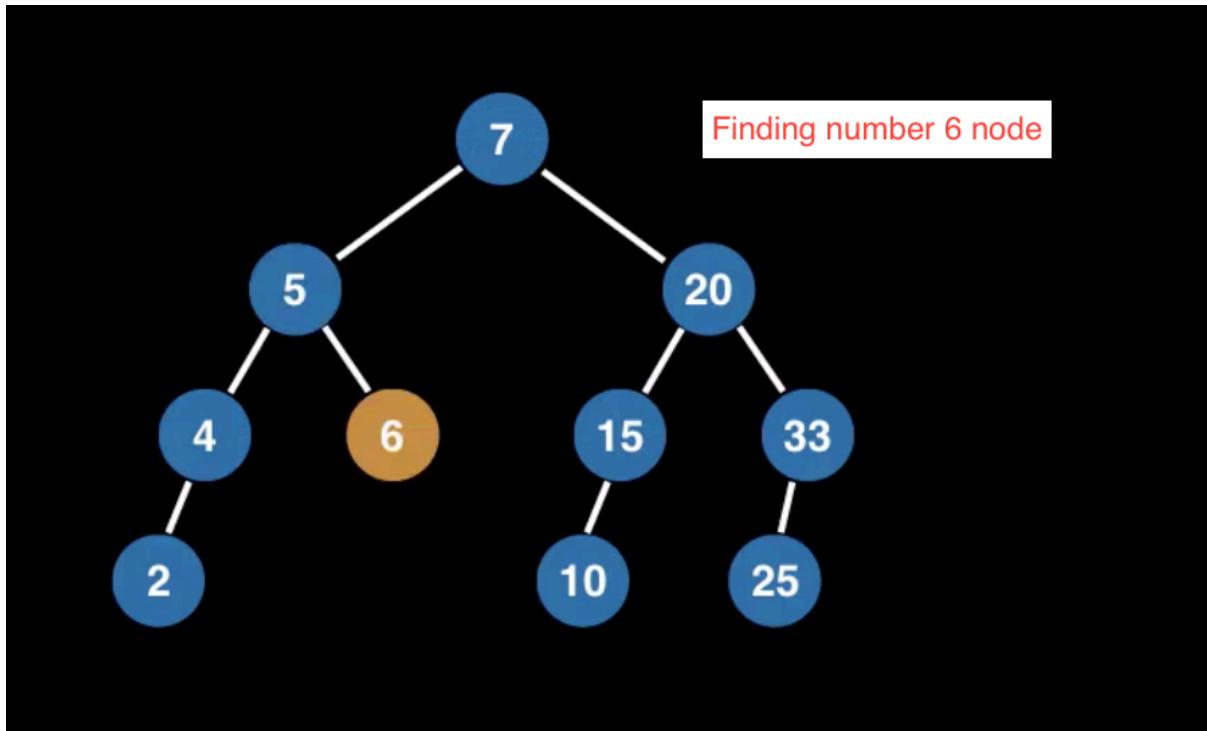


*Image of binarytree*

```
function insert(root , node) //node for new inserting element
if (empty(root))
    return create Root(node); //create root with newly inserted node
if (node <= root.node) // if the inserting node is less than root node
    root.left = create(root.left , node) //insert in left side of the tree
else // if the inserting node is greater than root node
    root.right = create(root.right, node) //insert in right side of the tree
```

## Search Operation

Node တစ်ခုကို search တော့မယ်ဆိုရင် tree ရဲ့ root node ကနေပြီးတော့ စရာတယ်။ ရှာမယ့် value က root node value ထက်ငယ်မယ်ဆို left side ကိုဆင်းပြီးထပ်ရှာတယ်၊ ကြိုးတယ်ဆို right side ကိုဆင်းပြီးထပ်ရှာတယ်။ အဲလိုနဲ့ပဲအဆင့်ဆင့် node value တွေနဲ့ တိုက်ပြီး ကြီးရင် ညာ၊ ငယ်ရင် ဘယ် နဲ့ ရလဒ်တွေ့တဲ့ ထိရှာသွားတယ်။

*Image of binarytree*

```

function search(root, node)
    if (root == null) //failed search
        return null;
    if (node == root.node) //found
        return root;
    if (node < root.node) //if the node is less than root/parent node
        return Search(root.left, node); //continue searching on the left side of tree
    else //if the node is greater than root/parent node
        return Search(root.right, node); //continue searching on the left side of tree
    
```

## Removing operation

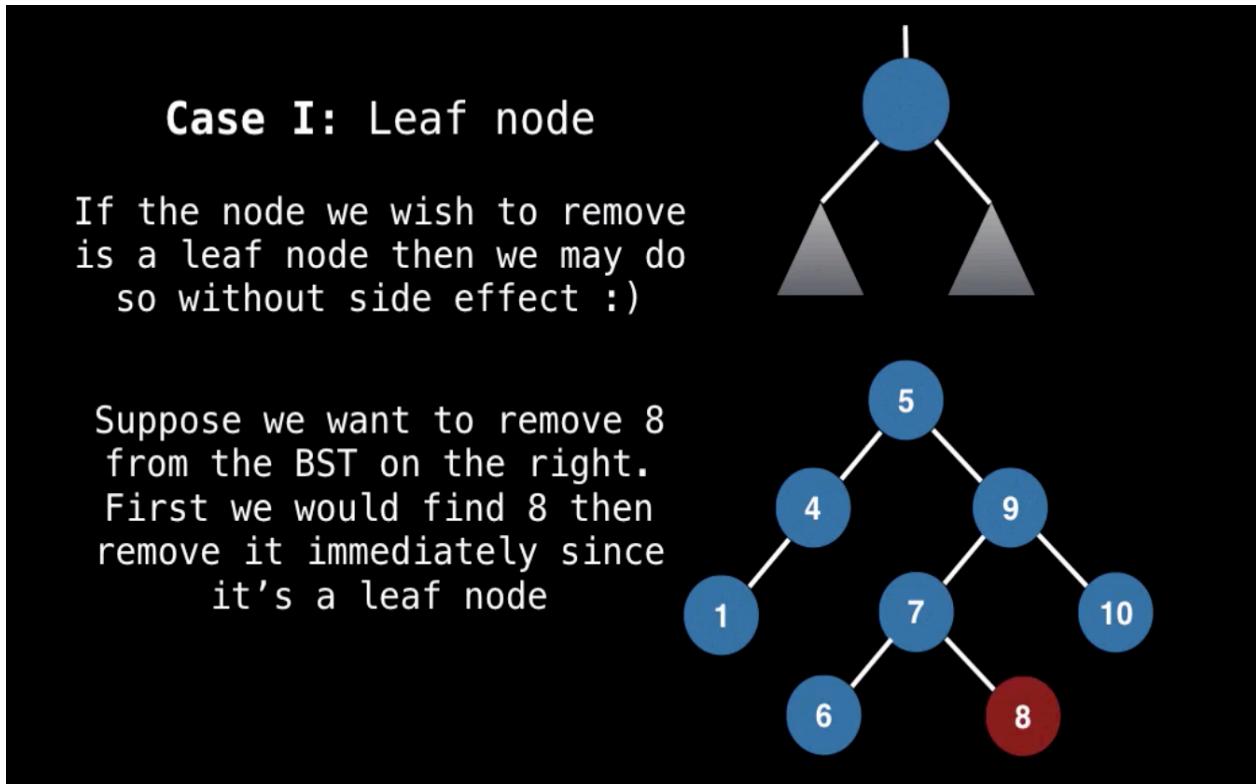
Remove လုပ်တဲ့ နေရာမှာတော့ insert တို့ search တို့လို မရှိရင်းဘူး၊ နည်းနည်း tricky ဖြစ်တယ်။ ဘာလို့လဲဆိုတော့ ဖျက်တဲ့ နေရာမှာ case က ငဲ ခုဖြစ်နိုင်တယ်။

- Case I. Leaf node ကိုဖျက်တဲ့ case (အောက်က child node မရှိတော့တဲ့ node တစ်ခုကိုဖျက်တာ)

- Case II. Child node အနေနဲ့ Left side node ပဲရှိတဲ့ node ကိုဖျက်တဲ့ case
- Case III. Child node အနေနဲ့ right side node ပဲရှိတဲ့ node ကိုဖျက်တဲ့ case
- Case IV. Child node အနေနဲ့ left ငြော ရွောရှိတဲ့ node ကိုဖျက်တဲ့ case

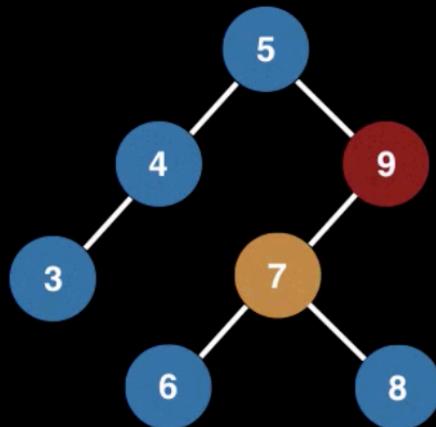
Case II and III ကတေသ့ အတူတူပဲလို့ပြောလို့ရပါတယ်။

Case I ကိုဖျက်တဲ့နေရာမှာတော့ ရှင်းပါတယ်။ သူ့အောက်မှာ တစ်ခြား ဘာ child node မှမရှိတဲ့ အတွက် တန်းဖျက်လိုက်လို့ရပါတယ်။

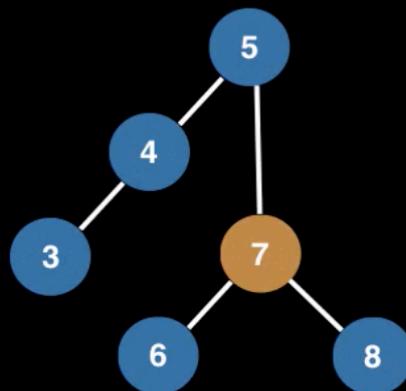


Case II ကိုဖျက်တဲ့နေရာမှာတော့ ဖျက်ရမယ့် node အောက်မှာ left side node ထက်ရှိတယ်ဆိုရင် ဖျက်မယ့် node ကိုဖျက်ပြီး အဲ node အောက်က left side tree ရဲ့parent node နဲ့ အစားထိုးလိုက် ရင်ရပါပြီ။ right side လည်း အဲနည်းအတိုင်းပါပဲ။

Suppose we wish to remove 9,  
then we encounter case II  
with a left subtree

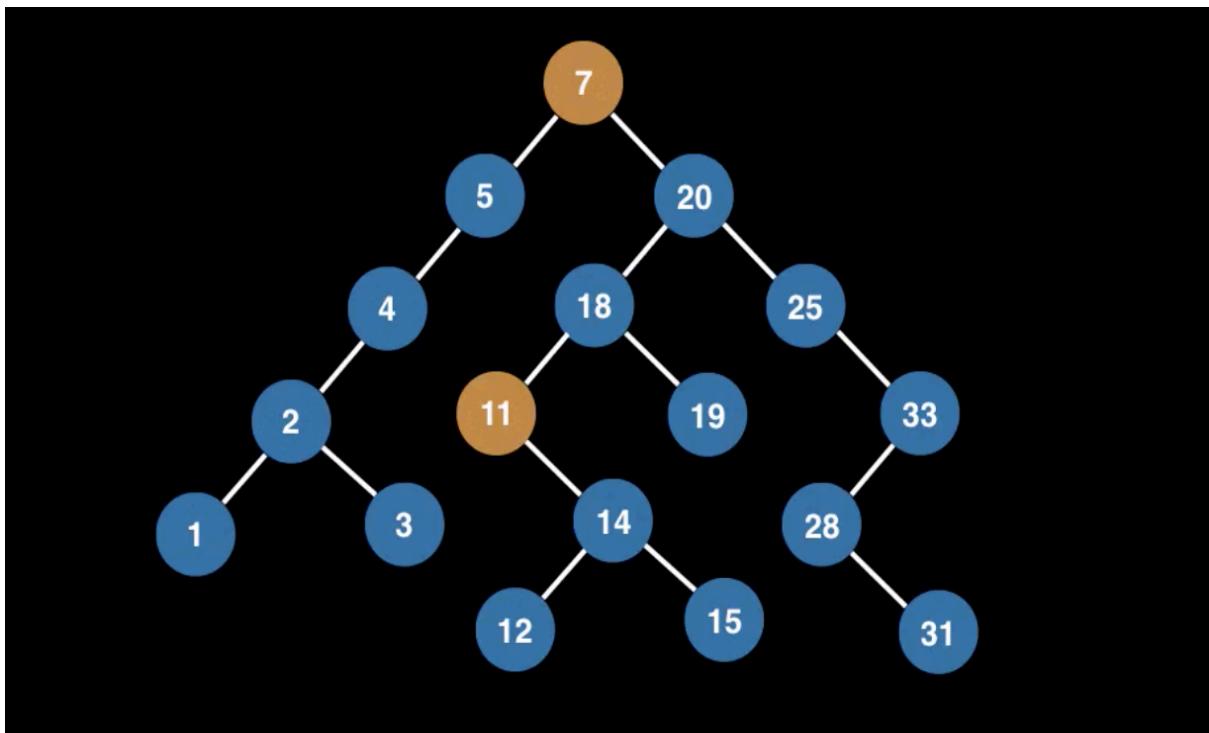


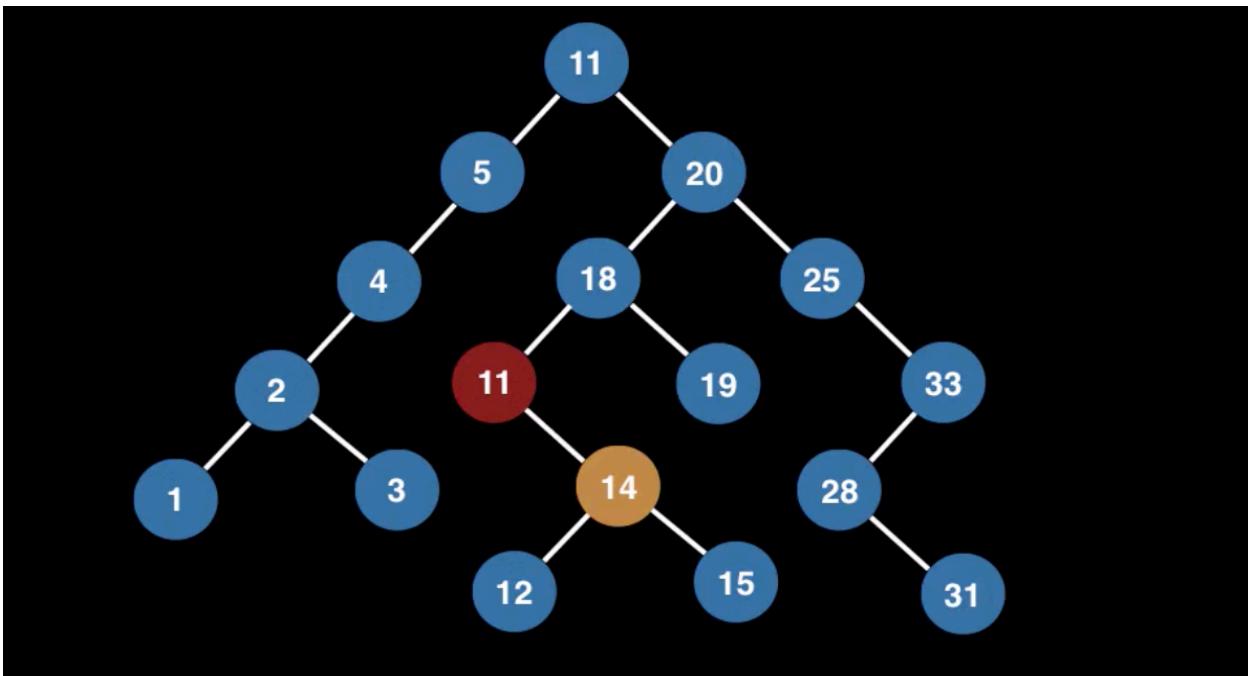
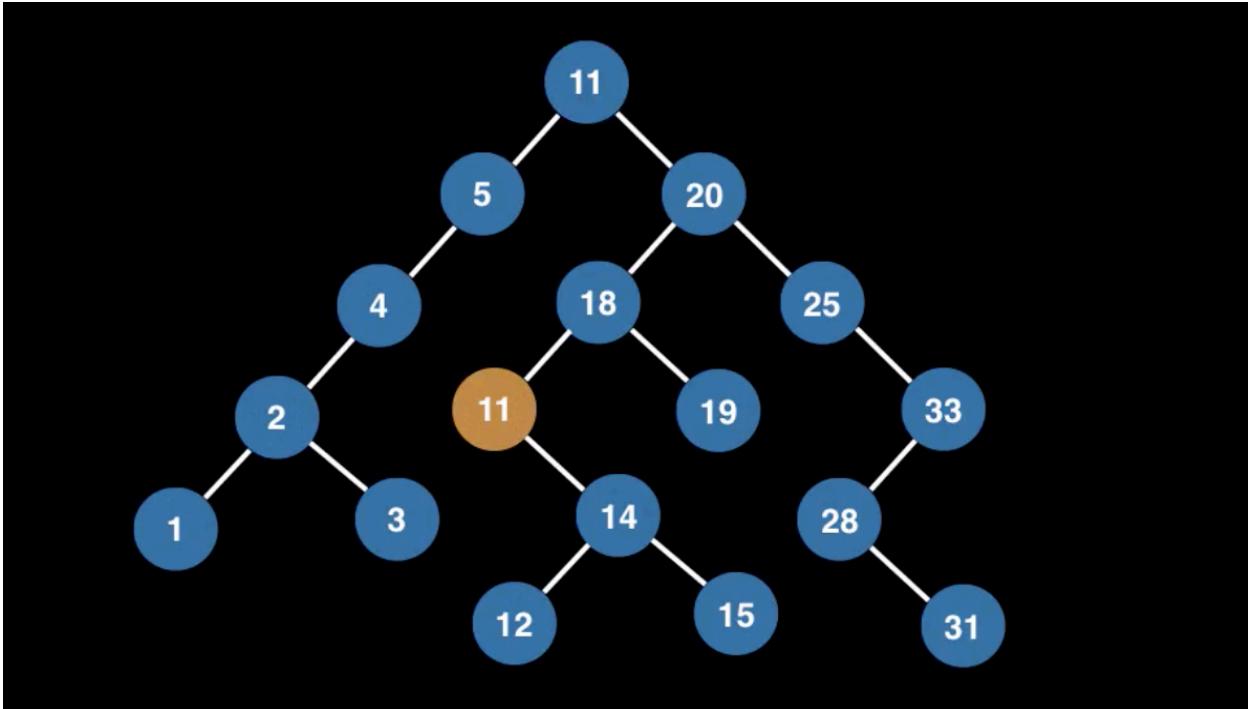
Suppose we wish to remove 9,  
then we encounter case II  
with a left subtree

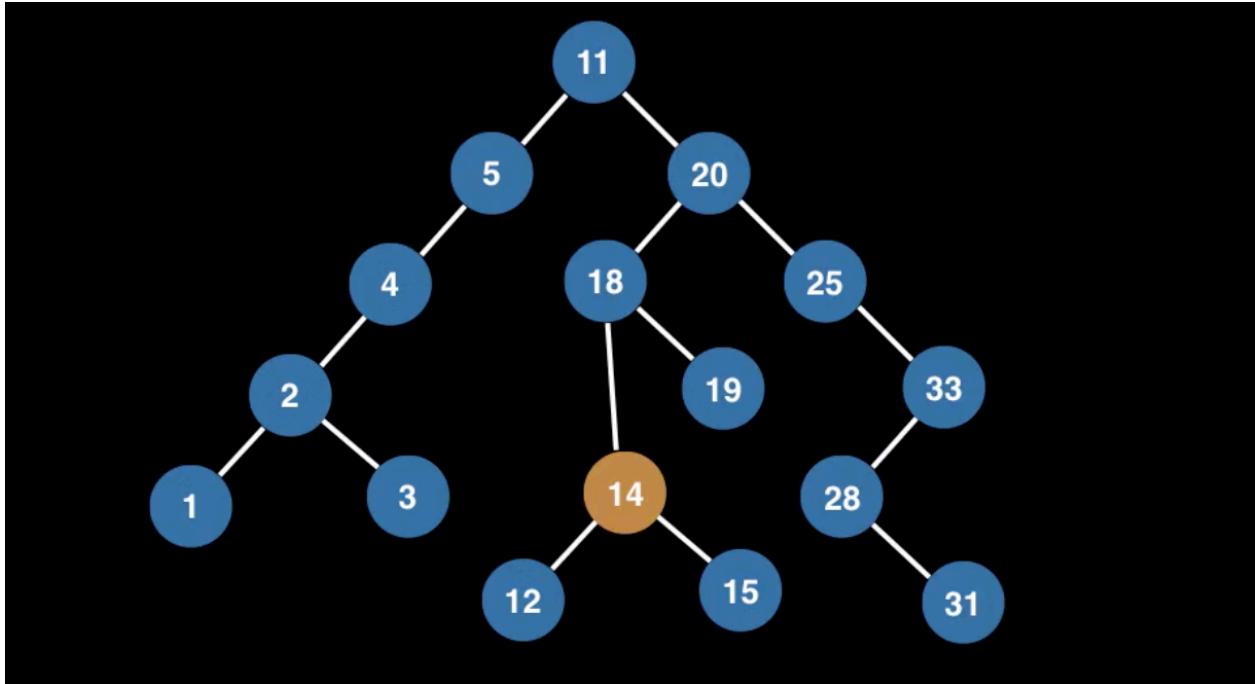


Case IV ကတေသ့ ပိုရှင်ပါတယ်။ ဖျက်မယ့် node တစ်ခု အောက်မှာ left ကော့ right ကော့ ရှိတယ်။ အဲလို့ node ကိုဖျက်တော့မယ်ဆို ပိုပြီးသတိထားရပါတယ်၊ binary tree ရဲ့ property မပျက်အောင်

ဖျက်ရပါတယ်။ ဖျက်တော့မယ်ဆို left side ဘက်ကိုရွှေးမယ်ဆို အကြီးဆုံး value စု right side ကိုရှာမယ်ဆို အငယ်ဆုံး value တို့ကို ရှာပြီးတော့ ဖျက်ရမယ့် node ဆိုကို replace လုပ်ပြီးတော့ ဖျက်ပါမယ်။ left side ကိုရွှေးတယ်ဆို အကြီးဆုံး value ကိုရှာရမှာဖြစ်တဲ့အတွက် tree ရဲ့ညာဘက် အခြမ်းတွေကို ဆင်းပြီးတော့ right sub tree မရှိတော့တဲ့ node အထိဆင်းသွားမယ်ဆို left side ရဲ့ အကြီးဆုံးကိုရှာဖြစ်ပြီးတော့၊ right side ကိုရွှေးမယ်ဆို ဘယ်ဘက်ကိုတောက်လျောက်ဆင်းသွားပြီး left sub tree မရှိတော့တဲ့ node အထိဆို right side ရဲ့အငယ်ဆုံး node ကိုရှာဖြစ်ပါတယ်။ ရလာတဲ့ အငယ်ဆုံး OR အကြီးဆုံး value ကို ဖျက်ရမယ့် node နဲ့ replace လုပ်ပြီး အဲဒီ ခုနက ရှာထားတဲ့ အငယ်ဆုံး OR အကြီးဆုံးရှိတဲ့ node ကိုဖျက်လိုက်ရင်ရပါပြီ။ အဲဖျက်တဲ့ အချိန်မှာတော့ Case I OR case II & III နဲ့ဖျက်သွားလို့ရပါပြီ။







*Image of binarytree*

```
function delete(root, node)
    if (root == null) // failed search
        return null;
    if (node == root.node) // successful search
        return deletecasefour(root);
    if (node < root.node) // node in the left branch
        root.left = Delete(root.left, node);
    else // node > root.node, i.e., node in the right branch
        root.right = Delete(root.right, node);
    return root;

function deletecasefour(root)
    if root has two children
        p = Largest(root.left); // replace root with its immediate predecessor p
        root.node = p.node;
        root.left = Delete(root.left, p)
    return root;
```

```

if root has only left child
    return root.left

if root has only right child
    return root.right

else root has no children
    return null

function Largest(Node root)
    if root has no right child
        return root
    return Largest(root.right)

```

## Binary Tree's traversal

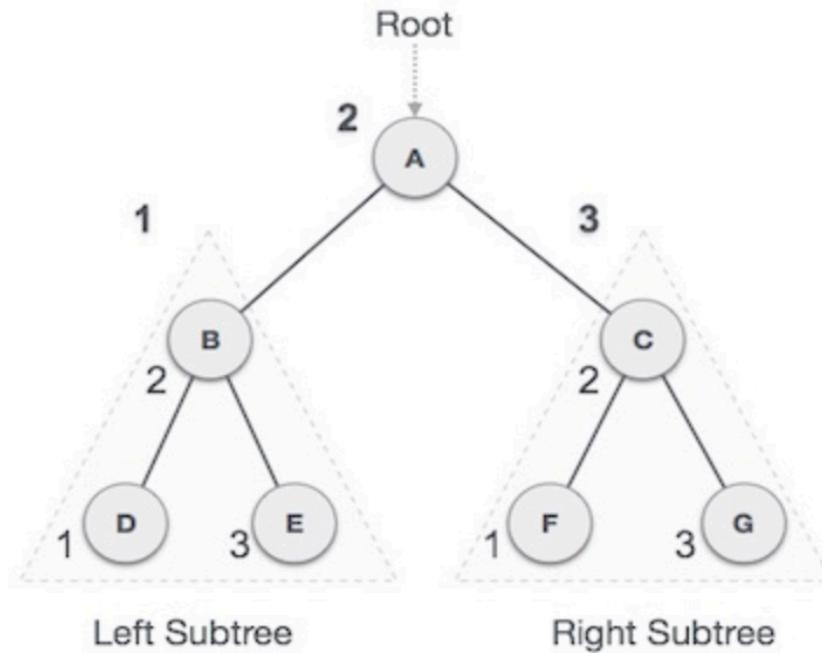
Binary tree မှာ traversal လုပ်တယ်ဆိုတာ tree ထဲမှာရှိတဲ့ node တွေကို visit လုပ်တာကိုဆိုလိုတာပါ။ tree ထဲမှာရှိတဲ့ level 2 က value 7 ရှိတဲ့ node ကိုလှမ်းထောက်ပါဆို ဒီတိုင်း plain ကြီး သွားထောက်လို့မရပါဘူး။ level 0 ဖြစ်တဲ့ root node ကနေစပြီးတော့ တစ်ဆင့်ခြင်းဆီ အဲဒီ node ကိုရောက်ဖို့အထိ visit လုပ်ရပါတယ်။ အဲလို visit လုပ်တာကို traversal လုပ်တယ်လို့ခေါ်ပါတယ်။

Traversal လုပ်တဲ့ နေရာမှာ

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal
- Level order Traversal ဆိုပြီးတော့ ရှုပါတယ်။

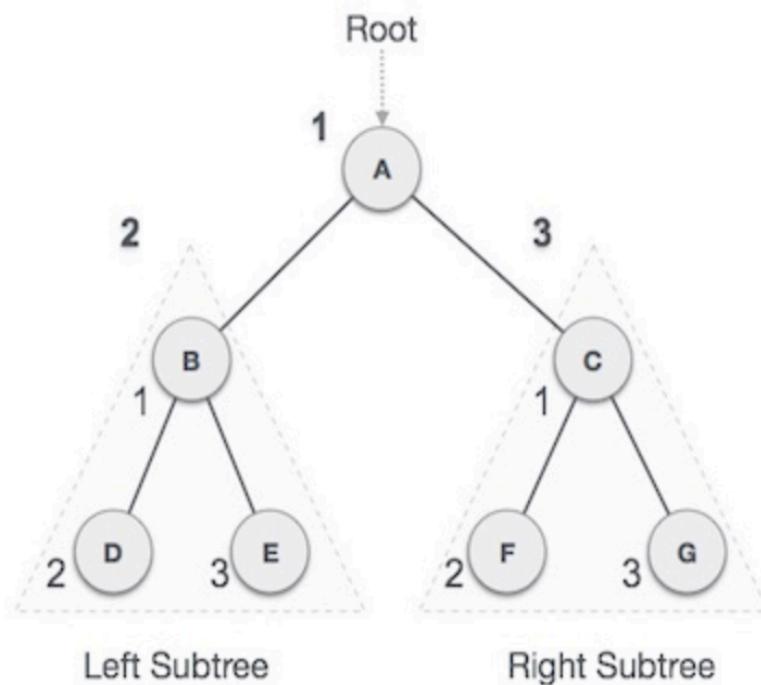
## In-order Traversal

Inorder ကတော့ ascending order နဲ့သွားတယ်၊ ဘာလို့လဲဆိုတော့ သူ traversal လုပ်တာက left sub tree အရင်လုပ်တယ်၊ ပြီးတော့ မှ root ပြီးတော့မှ right sub tree ကိုလုပ်သွားတယ်။ ဆိုတော့ left to right သွားတဲ့ အတွက်ကြောင့် value တွေကလည်း အစဉ်သင့် ascending order နဲ့ sort ပြီးသားဖြစ်သွားပါတယ်။



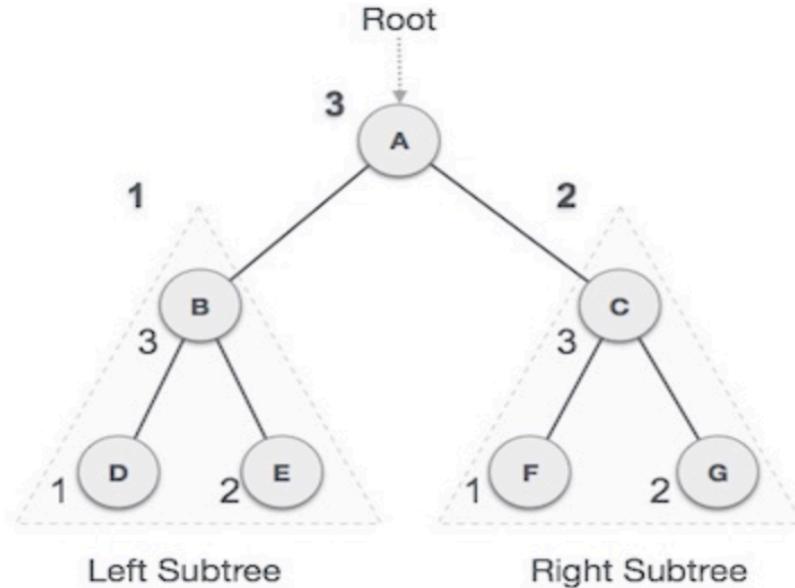
### Pre-order Traversal

Preorder မှတော့ root ကိုအရင်ဆုံး visit လုပ်တယ်၊ ပြီးတော့မှ left sub tree ကို visit လုပ်ပြီး နောက်ဆုံးမှာမှ right sub tree ကို visit လုပ်တယ်။ root->left sub tree->right sub tree



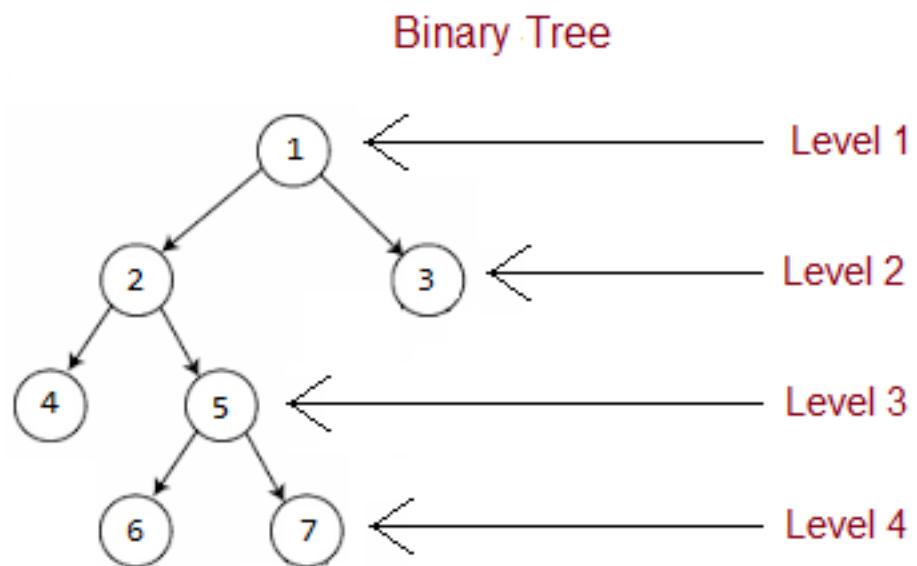
## Post-order Traversal

Postorder မှာတော့ root ကိုနောက်ဆုံးမှ visit လုပ်တယ်။ left sub tree ကို အရင် visit လုပ်တယ်။ ပြီးတော့ right sub tree ။ နောက်ဆုံးမှ root ဆို visit လုပ်ပါတယ်။ left sub tree->right sub tree->root



## Level Order Traversal

Level order ကတော့ tree ရဲ့ level နဲ့ အလိုက် visit လုပ်ပြီး print လုပ်ပါတယ်။ သူအလုပ်လုပ်က အပေါ်ကောင်တွေလောက်မရှိရင်းဘူး၊ queue algorithm ကိုအသုံးပြုပြီးတော့ အလုပ်လုပ်ပါတယ်။ ဘယ်လိုအလုပ်လုပ်လဲဆိုတော့ root node ကနေ စပြီး အလုပ်လုပ်မယ်ဆိုရင် root node ကို အရင် queue ထဲမှာထည့်ထားတယ်။ အဲ့ node ကို queue ထဲက ထုတ်ပြီး print လိုက်ပြီဆိုတာနဲ့ အဲ့ node ရဲ့ left and right child node တွေကို queue ထဲမှာထည့်ပါတယ်။ အဲ့လိုနည်း နဲ့ပဲ iterate လုပ်သွားပြီးတော့ level အလိုက် node တွေကို ရရှိလာပါတယ်။



## Binary Search Algorithm

Binary search ကတေသ့ strong 弗特တဲ့ searching algorithm တစ်ခုဖြစ်ပြီးတော့ divide & conquer algorithm အပေါ်မြို့ပြေားထားပြီးအလုပ်လုပ်ပါတယ်။ Binary search intro နဲ့ time complexity အကြောင်းဖတ်ရန်.

Binary search algorithm အလုပ်လုပ်ဖို့ဆိုရင် data value တွေက sorted form နဲ့ရှိထားရပါတယ်။ ဘာလို့လဲဆိုတော့ သူအလုပ်လုပ်ပုံက data collection ထဲကနေ middle item ကိုလှမ်းထောက်တယ်၊ ရလာတဲ့ key နဲ့ ရှာမယ့် key ကိုတိုက်စစ်ပြီး ငယ်တယ်ဆို ဘယ်ဘက်ကိုရွှေ့ပြီး mid item ထပ်ရှာ၊ ကြိုးတယ်ဆို ညာဘက်ရွှေးပြီး mid item ထပ်ရှာ၊ တူတာတွေပြုဆို ရင်တော့ process ပြီးပြီပေါ့။ မတွေ့သေးသမျှတော့ middle most node ကို ထောက်ထောက်ပြီးတော့ အလုပ်လုပ်သွားတယ်။ ဒါကြောင့်မို့လို့ data collection တွေက တိုက်စစ်ပေးနိုင်ဖို့အတွက် sorted form ဖြစ်နေရမယ်ပြောတာ။

ကျနော်တို့ example တစ်ခုအနေနဲ့ binary search ကိုသံဃားပြီး ရှာကြည့်ကြရအောင်။ Photo attach တွဲပေးထားတာကိုကြည့်ပါ။

Binary Search										
Search 23	0	1	2	3	4	5	6	7	8	9
	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2 <sup>nd</sup> half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 > 56 take 1 <sup>st</sup> half	0	1	2	3	4	L=5	6	M=7	8	H=9
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91

Array 10 ခုပါတဲ့ data collection တစ်ခုမှာ 23 ပါတဲ့ array key ကိုရှာဖြေမယ်ဆိုပါစွဲ။ ပထမဆုံး အဆင့် အနေနဲ့ medium node ကို အရင်ရှာပါတယ်၏ ဒီ formula နဲ့ ရှာလို့ရပါတယ်။  $mid = low + (high - low) / 2$   $mid = 0 + (9 - 0) / 2 = 4$  လိုပဲယူထားလိုက်ပါမယ်။ (integer value of 4.5)

array key 4 ဖြစ်တဲ့ value 16 ရပါတယ်။ ကျနော်တို့ရှာမယ့် 23 က 16ထက်ကြီးတယ်ဆိုတော့ ညာဘက် sub collection ကိုထပ်ရွေးပြီးတော့ medium ထပ်ရှာပါမယ်။ ကြီးတဲ့ ညာဘက် ကိုသွားမယ်ဆို  $low = medium + 1$   $mid = 5 + (9 - 5) / 2 = 7$  ရပါတယ်။

array key 7 ဖြစ်တဲ့ 56 ကိုရပါမယ်။ ကျနော်တို့ ရှာတဲ့ 23 မရသေးပါဘူး။ 23 က 56 ထက်ငယ်တာဖြစ်တဲ့ အတွက် ဘယ်ဘက် sub collection ကိုထပ်ရွေးပြီးတော့ medium ထပ်ရှာပါမယ်။ ငယ်တဲ့ ဘယ်ဘက်ကို သွားမယ်ဆို  $high = medium - 1$   $mid = 5 + (6-5)/2 = 5$  ရပါမယ် (integer value of 5.5)

array key 5 ဖြစ်တဲ့ 23 ကိုရပါပြီ၊ ကျနော်တို့ ရှာချင်တဲ့ array key ကိုရှာတွေ့သွားတာပဲဖြစ်ပါတယ်။

### Pseudo code sample

```
Function search(root, node)
    if (root == null) //failed search
        return null;
    if (node == root.key) //succesfull search
        return root;
    if (node < root.key) //node in left branch
        return Search(root.left, k);
    else //node > root.key, i.e. node in right branch
        return Search(root.right, k);
```

## Hash table (hash function)

Hash table အကြောင်းပြောတော့မယ်ဆိုတော့ hash table ဆိုတာ ဘာလဲကနေစတာပေါ့။ Hash table ဆိုတာ hashing ဆိုတဲ့ technique တစ်ခုကိုသုံးထားပြီးတော့ key နဲ့ value ကို mapping (တဲ့) ထားပေးတဲ့ structure တစ်ခုလို့ ဆိုနိုင်ပါတယ်။ frequency value တွေကို သူရဲ့ associated key တွေနဲ့ တွဲပြီးတော့လည်း သိမ်းပါတယ်။ ကျနော်အရင်က priority queue article ရေးတုန်းက priority queue တွေကို hash table နဲ့ တွဲပြီးအလုပ်လုပ်ပုံကိုလည်းရေးပေးထားပါသေးတယ်။ အောက်က link မှာဝင်ဖတ်ကြည့်လို့ရပါတယ်။ hash table ဘာကြောင့်သုံးရတာလဲ၊ ဘယ်လိုအသုံးဝင်လဲ ဆိုတာတွေကို အဲဒီ article ဖတ်လိုက်ရင်နားလည်သွားပါလိမ့်မယ်။ <http://bit.ly/2Nfd5Wl>

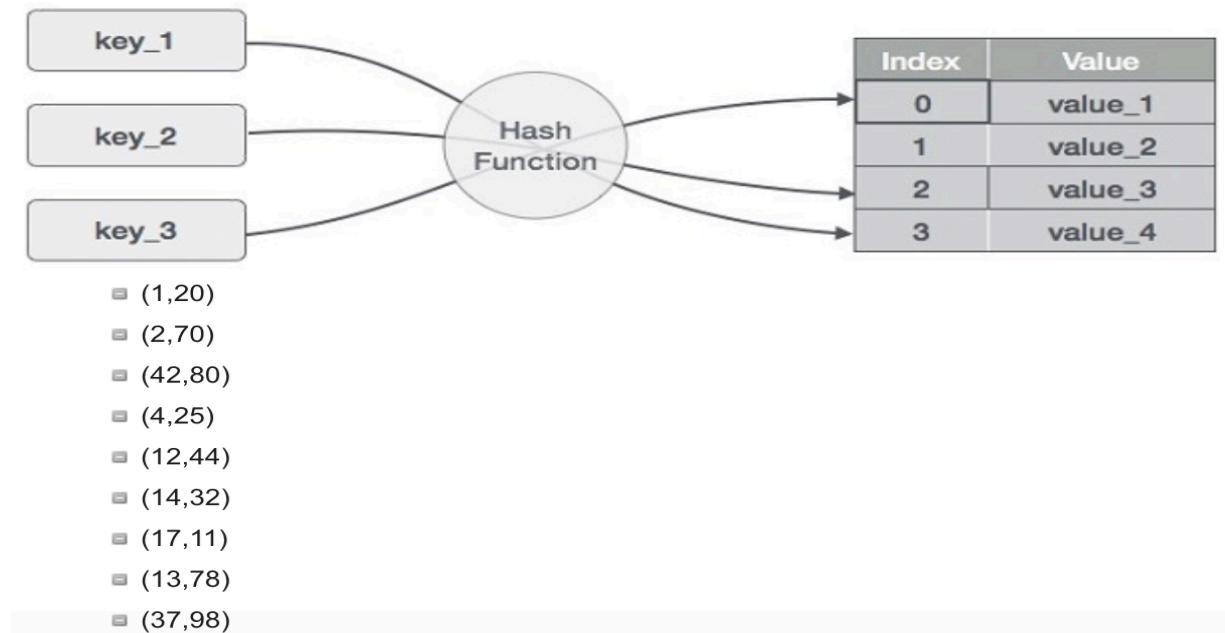
ဒီနေ့တော့ hash table မှာသုံးတဲ့ technique တစ်ခု ဖြစ်တဲ့ hashing အကြောင်းကို ဆွေးနွေးသွားမှာပဲဖြစ်ပါတယ်။

Hash table မှာ key & value map လုပ်မသွားခင်မှာ hashing အရင်လုပ်ရပါတယ်။  
ပြောရမယ်ဆိုရင်တော့ key ကို hash လုပ်တာပေါ့၊ hash လုပ်တဲ့နေရာမှာ modulus နဲ့ hash လုပ်ကြည့်တာပေါ့၊ size 10 ခုရှိတဲ့ table မှာဆို hash ကိုတွက်တဲ့ formula  $F(x) = \text{mod by } 10$   
ဆိုပြီးလုပ်ပြီး တွက်တာပေါ့။ ဥပမာ - key of x ကို hash မယ်ဆိုရင်  $F(x) = (\text{square of } x - 6x + 9)$   
mod 10 ဆိုပြီး ပိုhash လုပ်ကြည့်လို့ရပါတယ်။ x နေရာမှာ integer value  
တွေထည့်ပြီးတွက်ထုတ်ကြည့်မယ်ဆို ၁ ကနေ ၁၀ အတွင်း value တွေတွက်လာလိမ့်ပါမယ်။  
တစ်ကယ်တော့ integer မှုရယ် hash လုပ်လို့ရတာ မဟုတ်ပါဘူး၊ တစ်ခြား data type တွေကိုလည်း  
hash လို့ရပါတယ်၊ ဥပမာ string ဆိုရင် သူရဲ့ ASCII value ထုတ်လိုက်မယ်ဆို integer value  
ပြန်ရပါမယ်၊ hashing function ထဲ ထည့်တွက်လို့ရပါတယ်။

Hash function မှာလည်း သူရဲ့ကိုယ်ပိုင် properties တွေရှိပါတယ်။ - အကယ်လို့  $F(x) \neq F(y)$  ရဲ့ result တွေက တူတယ်ဆိုရင်  $x \neq y$  ရဲ့ တန်ဖိုးတွေက တူကောင်းတူနိုင်ပါတယ်လို့ ဆိုနိုင်ပေမဲ့  $f(x) \neq f(y)$  result ကမတူဘူးဆိုရင်  $x$  and  $y$  ရဲ့ value တွေက လုံးဝတူမှာမဟုတ်ပါဘူး။ (ဒီ property ကဘာအသုံးဝင်လဲဆိုတော့ ထားပါတော့  $x$  and  $y$  က size ကြီးတဲ့ ဖိုင်တွေပဲထားပါတော့ compare လုပ်ဖို့လိုလာပြီဆို  $x$  and  $y$  ကိုတိုက်ရှိက် ထိမယ့်အစား သူတို့ရဲ့ hash value ကို compare လုပ်လိုက်တာ ပိုမြန်ပါတယ်။) - Hash လုပ်လိုက်တဲ့ value က အမြဲတမ်း deterministic ဖြစ်ရပါမယ်။

ဆိုလိုချင်တာက  $f(x)$  ရဲ့ hash value ၂ ထွက်တယ်ဆို အမြတ်မ်း ၂ ပဲထွက်ရပါမယ်၊  
တစ်နည်းအားဖြင့်ဆို constant ဖြစ်ရမယ်လို့လဲဆိုလိုပါတယ်။

ဒါပေမဲ့ တစ်ခုရှိတာက collision ဖြစ်နိုင်တဲ့ case ပါ အပေါ်က ကျနော်ပြထားတဲ့ hash တွက်တဲ့  
function မှာဆို ( $F(x) = (\text{square of } x - 6x + 9)$ ) ,  $x$  ရဲ့ တန်ဖိုးကို 4 နဲ့ 2 နဲ့ ထည့်မယ်ဆို  
ရလာမယ့်အဖြေက 1 ချင်းတူနေပါတယ်။ ဒီလို case မျိုးဆို collision ဖြစ်တယ်လို့ ပြောရမယ်ပေါ့။  
ဒီလိုမျိုး collision တွေဖြစ်လာပြီဆိုရင် ဖြေရှင်းနိုင်ဖို့အတွက်လည်း solution  
တွေထုတ်ထားပါတယ်။ အဲတဲ့ကမှ popular ဖြစ်တာတွေက တော့ separate chaining နဲ့ open  
address ဆိုပြီးရှိပါတယ်။ နောက် article တွေမှာ အဲဒီ solution အကြောင်းတွေရေးပေးသွားပါမယ်။



Sr.No.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

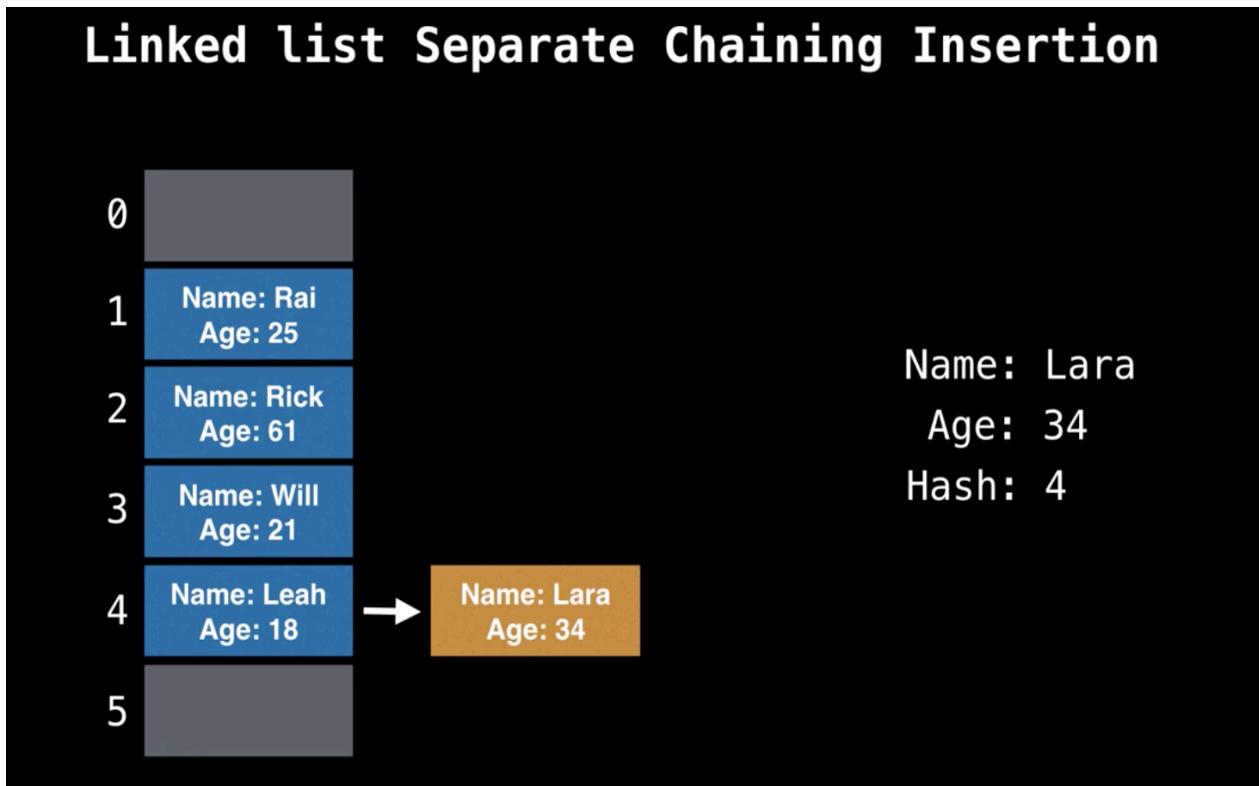
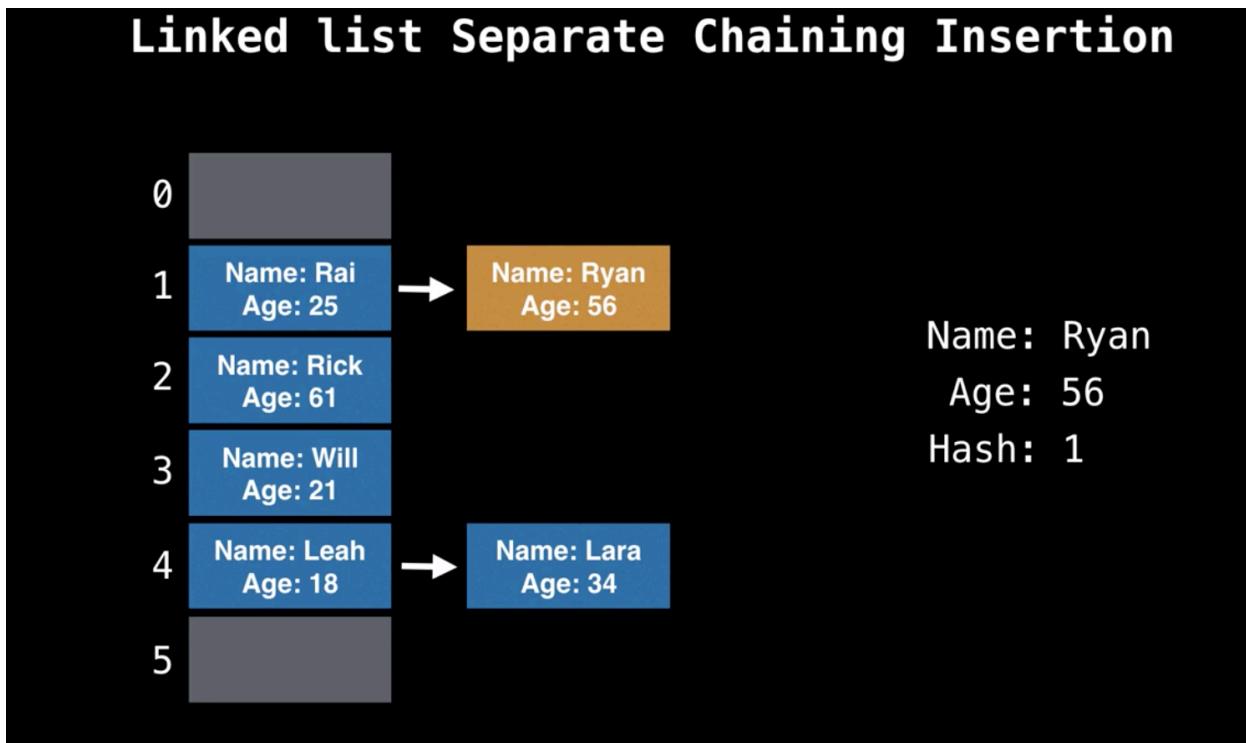
## Hash table (separate chaining)

ပြီးခဲ့တဲ့နောက hash table အကြောင်းကို intro ဝင်ပြီးသွားပြီ။ hash table မှာ issue ဖြစ်နေတဲ့ collision အကြောင်းလည်းရေးခဲ့ပါတယ်။ ဒီနေ့တော့ အဲလို့ collision တွေကို ဖြေရှင်းတဲ့ နည်းလမ်းထဲက popular အဖြစ်ဆုံးနဲ့ နားလည်ရလွယ်တဲ့ method တစ်ခုဖြစ်တဲ့ separate chaining အကြောင်းကို ဆွေးနွေးသွားပါမယ်။ ရှင်းရှင်းပြောရင် separate chaining က linked list algorithm ပေါ်ကို မြှင့်မြှုပ်ပြီးတော့ အလုပ်လုပ်ထားတယ်လို့လဲပြောလို့ရပါတယ်။ ဆိုတော့ ဒီ article ကိုမဖတ်ခင် hash table ရဲ့ introduction နဲ့ linked list အကြောင်းကို ဖတ်ဖို့အကြံပေးချင်ပါတယ်။

Data structure ကိုတစ်ဖြည့်ဖြည့်လေ့လာရင်းနဲ့ သိလာရမှာက data structure တစ်ခုနဲ့ တစ်ခုမှာ မြှုပ်နေတာကိုတွေ့လာရပါလိမ့်မယ်။ ဥပမာအနေနဲ့ပြောရရင် priority queue မှာ hash table ကိုထည့်သုံးသလို အခုရေးမယ့် separate chaining မှာလည်း linked list ကိုသုံးထားတယ်၊ စသည်ဖြင့် တစ်ခြား အများကြီးရှိပါသေးတယ်။

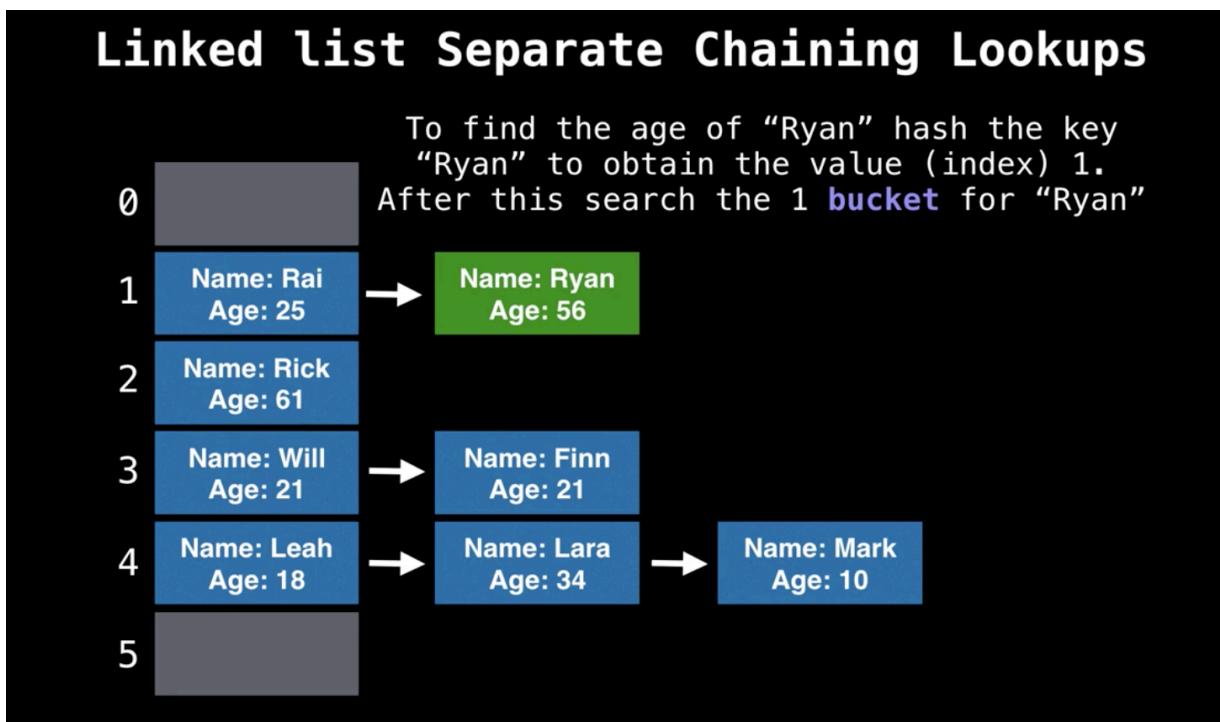
စကားပြီးတွေ့လည်းများပြီမို့ separate chaining စလိုက်ရအောင်၊ hash table မှာဖြစ်လေ့ရှိတဲ့ collision issue ကိုဖြေရှင်းနိုင်မယ့် strategy တွေထဲကတစ်ခုပဲ၊ သူ့ရဲ့ထူးခြားချက်က ဝင်လာသမှု data node အားလုံးကို တစ်ခုမှုဖျက်မပစ်ဘဲနဲ့ collision လည်းမဖြစ်အောင် hash table ရဲ့ data structure လည်းမပျက်အောင် ဖန်တီးပေးထားနိုင်တယ်၊ linked list algorithm ကို အခြေခံပြီး ဖြေရှင်းပေးထားပါတယ်။

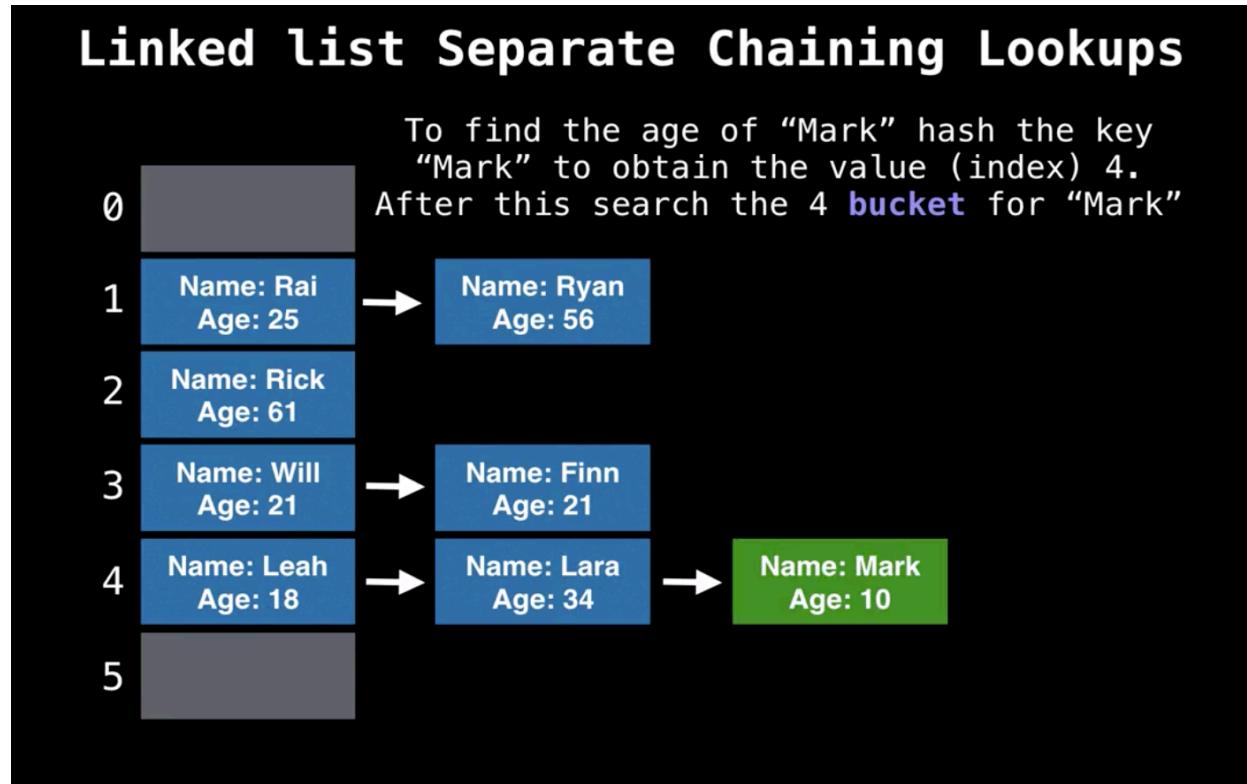
ဖြေရှင်းနည်းကတော့ ရှင်းတယ်။ ကျနော်တို့ ပုံမှန်အတိုင်း hash table ထဲကို hash လုပ်တယ်၊ value တွေထည့်တယ်။ hash လုပ်လိုက်တဲ့ result က table ထဲမှာရှိပြီးသားဆိုရင် collision ဖြစ်တော့ မယ်ပေါ့၊ အဲအချိန်ကျရင်လည်း table ထဲက hash result တူတဲ့ node ရဲ့နောက်မှာပဲ link တွဲပြီးတော့ ထည့်လိုက်တယ်၊ linked list က data insertion လုပ်ပုံနဲ့ အတူတူပါပဲ။ insertion လုပ်တဲ့ case အတွက် ပုံနှစ်ပုံ attach တွဲပေးထားပါတယ်။ ကြည့်ကြည့်ပါ။

**Figure 1****Figure 2**

Insert လုပ်ပြီးသား data node တွေကို ပြန် ဆဲထုတ်မယ် (lookup) လုပ်တော့မယ်ဆိုလဲရှင်းတယ်၊ data ကို hash လုပ်မယ်၊ hash result က 4 လို့ရလာပြီပဲဆိုပါတော့၊ no 4 slot မှာသွားကြည့်မယ်၊ သွားကြည့်တာတော့ ဟုတ်ပါပြီ၊ ကျနော်တို့ လိုချင်တဲ့ node မှန်းဘယ်လိုသိနိုင်မလဲပေါ့၊ hash value ကလည်းတူနေတာဆိုတော့၊ ကြည့်လို့ရပါတယ်၊ ဘာလို့လဲဆိုတော့ insert သွင်းတုန်းက hash value တစ်ခုတည်းသွင်းလိုက်တာ မဟုတ်ပါဘူး၊ တစ်ခြားတန်ဖိုးတွေလဲပါပါတယ်၊ ဥပမာ- လူတစ်ယောက်ရဲ data ဆို name, age, nrc စသည်ဖြင့် ရှိနိုင်တဲ့အတွက် အဲဒီ param တွေနဲ့ပါတိုက်စစ်လို့ရပါတယ်။ ပုံတဲ့ပေးထားပါတယ်။

**Figure 1**



**Figure 2**

Remove လုပ်မယ်ဆိုလဲ lookup လုပ်တဲ့အတိုင်းပဲ သွားပြီးတော့ lookup လုပ်မယ့် option အစား linked list ကို ဖျက်ချလိုက်ရှုပါပဲ။ separate chaining မှာ linked list မှုရယ်မဟုတ်ဘူး၊ တစ်ခြား algorithm တွေနဲ့ အစားထိုးလို့ရတယ်၊ like array, binary tree စသည်ဖြင့် သုံးလို့ရတယ်၊ ဒါပေမဲ့ complexity level ၏ linked list ထက်စာရင် ပို့ရှုပဲတဲ့ အတွက် linked list ကိုပဲ အသုံးပြုကြတာ များပါတယ်။

## Hash table (Open addressing)

Open addressing ဆိတ်ကလည်း hash table မှာ collision ဖြစ်တဲ့အခါကျ ဖြေရှင်းနိုင်တဲ့ solving technique တစ်ခုပဲဖြစ်ပါတယ်။ ရှေ့က article တွေမှာတော့ hash table အကြောင်းနဲ့ collision ဖြစ်ရင်ဖြေရှင်းနိုင်တဲ့ technique တစ်ခုဖြစ်တဲ့ separate chaining အကြောင်းကိုရေးပေးထားပါတယ်။

ကျနော်တို့ hash table ထဲထည့်ဖို့အတွက် key ကို hash လုပ်တယ်။ hash result ထွက်လာပြီဆို အဲဒီ result အတိုင်း table ထဲမှာ position ယူလိုက်တယ်။ hash result တူပြီး ကိုယ်ထည့်မယ့် position မှာ value ရှိပြီးသားဆို collision ဖြစ်ပြီ၊ ဒါကပုံမှန် process ပဲ။ open addressing ဘယ်လိုအလုပ်လုပ်လဲဆိုတော့ အဲဒီ position က taken ဖြစ်နေတယ်ဆိုရင် နောက်ထပ် လွှတ်မယ့် position တစ်ခုကိုဆက်ရှာဖို့ကြိုးစားတယ်။ ဘယ်လိုရှာလဲဆိုတော့ probing sequence formula တစ်ခုထုတ်ပြီးဆက်ရှာတယ်။ ပုံမှန်အတိုင်းဆို ပထမအဆင့်မှာ key ကို hash လုပ်ပြီးရလာတဲ့ result ကို position index အဖြစ်တန်းသတ်မှတ်ပြီးတော့ table ထဲမှာထည့်တယ်၊ probing sequence ကကျ hash result ရယ် မူလkey ရယ်ကို အသုံးပြုပြီးတော့ သူ့၏ formula ထဲမှာ နောက်ထပ် position index အသစ်တစ်ခုရနိုင်ဖို့အတွက်ထွက်ထုတ်ပေးတယ်။

Probing sequence မှာလည်း အမျိုးမျိုးရှိတယ်။ - Linear function နဲ့သုံးမယ်ဆို linear probing - Quadratic function နဲ့သုံးမယ်ဆို quadratic probing - နောက်တစ်နည်းကကျ ပုံမှန်သုံးထားတဲ့ hashing method လိုမျိုးကိုပဲ နောက်ထပ် ထပ်သုံးပြီး position index ကိုထွက်တာ၊ double hashing လိုလဲခေါ်တယ်။ - Random number တစ်ခုကို generate လုပ်ပြီးထွက်ထုတ်တဲ့ method မျိုးကိုလဲသုံးလို့ရပါတယ်။

ဒါပေမဲ့ probing sequence ကိုသုံးတဲ့ အချိန်မှာ issue ရှိနိုင်တဲ့ condition ရှိပါတယ်။ cycles issue လိုလဲခေါ်တယ်။ ဘယ်လို issue ဖြစ်တယ်ဆိုတာ ဥပမာတစ်ခု နဲ့ ယူဉ်ပြီးပြသွားပါမယ်။ probing sequence ကဘယ်လိုအလုပ်လုပ်တယ်ဆိုတဲ့ ပုံစံလည်းပြရင်းနဲ့ပေါ့။

ကျနော်တို့မှာ slot 12 ခန်းရှိတဲ့ structure တစ်ခုရှိတယ်ပဲဆိုပါစို့ (0 to 11)ပေါ့။ slot 0, 4, 8 မှာ data value တွေရှိပြီးသားတွေ။ အိုကော၊ ပုံမှန်အတိုင်း key ကို hash လုပ်ပြီး data ထည့်တာပေါ့။ formula က  $H(key) = key \bmod 12$  နဲ့ ဥပမာထားလိုက်ပါမယ်။ key ကို number 8 အဖြစ်နဲ့

ထည့်လိုက်မယ်ဆို hash result က 8 ထွက်လာမယ်။ data value က ရှိပြီးသားဖြစ်တဲ့ အတွက် probing sequence ကိုသုံးပြီးတော့ နောက်ထပ်နေရာလွတ်တစ်ခုကို ထပ်ရှုပါမယ်။

Probing sequence အတွက်  $P(x) = 4x$  ဆိုပြီး ဥပမာ formula တစ်ခုထားပြီးတော့ position index ထွက်မယ့် formula ကို ဒီလို upgrade လုပ်လိုက်ပါမယ်။ x value ကို 0 ကနေ စပြီးတော့ +1 ပေါင်းသွားပြီး ရှုပါမယ်။

$$\text{Index} = H(k) + P(x) \bmod 12$$

$\text{Index} = 8+0 \bmod 12 = 8$  (number 8 slot မှာ value ရှိတယ်၊ ထပ်ရှာ)  $\text{Index} = 8+4 \bmod 12 = 0$  (number 0 slot မှာ value ရှိတယ်၊ ထပ်ရှာ Note:  $P(x) = 4x$ )  $\text{Index} = 8+8 \bmod 12 = 4$  (number 4 slot မှာ value ရှိတယ်၊ ထပ်ရှာ)  $\text{Index} = 8+12 \bmod 12 = 8$  (number 8 slot မှာ value ရှိတယ်၊ ထပ်ရှာ)

ရလာတဲ့ result ကိုကြည့်မယ်ဆို 8, 0, 4, 8 ဆိုပြီး cycle အတိုင်း loop ပတ်သွားတာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။ ဆိုတော့ probing function မှာ ဒီလို issue တော့ ရှိတယ်၊ သူ့ကိုဖြေရင်းနိုင်ဖို့အတွက် ကတော့ ကိုယ်သုံးမယ့် probing function မှာသတိထားပြီးတော့ ကြိုတင်ထွက်ချက်ခန့်မှန်းထားဖို့လိုအပ်ပါတယ်။ နောက်နေ့တွေမှာ linear , quadratic function နဲ့ double hashing article တွေနဲ့အတူ cycle issue ကိုပါဖြေရင်းနည်းနဲ့ ပြန်လာခဲ့ပါမယ်။

## Hash table - Open addressing (Linear Probing)

ပြီးခဲ့တဲ့နေ့တွေက hash table နဲ့ collision ဖြစ်ရင်ဖြေရှင်းဖို့ technique တွေရေးခဲ့ပါတယ်၊ အဲထဲက မှ open addressing ဆိုတဲ့ technique ကိုသုံးတဲ့နေရာမှာ လိုအပ်မယ့် function တစ်ခုဖြစ်တဲ့ linear probing အကြောင်းကို ဒီနေ့ဆွေးနွေးသွားမှာဖြစ်ပါတယ်။ open addressing မှာ ရှိတဲ့ cycle issue ကိုလဲတစ်ပါတည်း ဆွေးနွေးသွားမှာဖြစ်ပါတယ်။

linear function ကိုသုံးထားလို့ linear probing လို့ခေါ်တယ်။ example .  $p(x) = ax+b$  (a is !=0) . ဒါပေမဲ့ သတိတစ်ခုချုပ်ထားရမှာက cycle issue ကိုပါပဲ၊ cycle issue ရှိနေရင်တော့ ဘယ် probing function မှ မှန်ကန်စွာ အလုပ်လုပ်နိုင်မှာ မဟုတ်ပါဘူး။

အခြောနေတာတွေနားမလည်ဘူးဆိုရင်တော့ အောက်မှာပေးထားတဲ့ article တွေအရင်ဖတ်ဖို့အကြံပေးချင်ပါတယ်။ hash table collision ဖြစ်တဲ့ scenario ၁ ဖြေရှင်းနည်းတွေနဲ့ cycle issue အကြောင်းတွေ နားလည်သွားပါလိမ့်မယ်။

Hash table (hashing)

Hash table (Separate Chaining)

Hash table (Open Addressing)

cycle issue ကိုပြန်trace လိုက်ရမယ်ဆိုရင် cycle issue ဘယ်လိုအခိုန်မှာဖြစ်လဲဆိုတာနဲ့ စလိုက်ရအောင်။ eg. အနေနဲ့  $p(x) = ax+b$  ဆိုတဲ့ formula မှာ a ရဲ့ တန်ဖိုးနဲ့ table တစ်ခုရဲ့ size N ကိုယူပြီးတော့ GCD (greatest common denominator) = (GCD(N,a))

တန်ဖိုးကိုတွက်ထုတ်လိုက်လို 1 နဲ့ညီနေတယ်ဆိုရင် အဲဒီ probing formula မှန်ကန်စွာအလုပ်လုပ်နိုင်မယ်။ 1 နဲ့ မညီရင်တော့ cycle issue တတ်ပါတယ်။ ဒီနေရာမှာ probing အတွက်ရေးနေတာမို့ GCD အတွက် သက်သက်ကြီးရေးမနေတော့ဘူး၊ GCD calculator လို့ရှာလိုက်ပြီးတွက်ကြည့်လို့ရပါတယ်။ အောက်မှာလဲ ပေးထားပါတယ်။

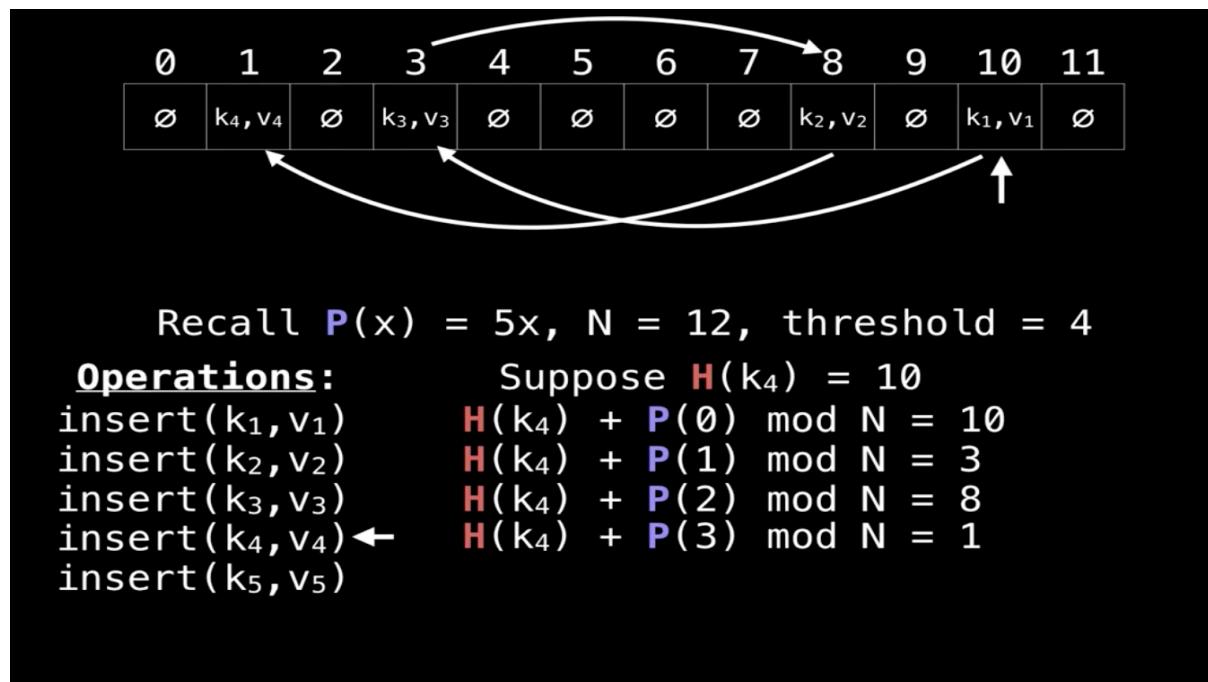
<http://www.alcula.com/calculators/math/gcd/>

ဥပမာ တွေနဲ့ စမ်းတွက်ကြည့်ရအောင်။

Table size = 9 Probing function =  $p(x) = 6x$  Max load factor = 0.667 (6/9) Threshold of the table =  $N * \text{max load factor} = 6$  (table size 9 ခန်းရှိပေမဲ့ 6ခန်းပြည့်သွားရင် table resize ထပ်လုပ်ရပါမယ်။)

GCD တွက်ကြည့်လိုက်မယ်ဆို  $\text{GCD}(N,a) = \text{GCD}(9,6) = 3$  ရပါတယ်။ 1 မဟုတ်တဲ့ အတွက် ဒီ probing function မှာ cycle issue တတ်ပါမယ်။ အောက်မှာကြည့်ရအောင်။

Formula ၂  $p(x) = 6x$  index =  $(h(k)+p(x)) \bmod N$  ဆိုရင်  $h(k)$  ခဲ့တန်ဖိုးကို 5 လို့သတ်မှတ်ပြီးတွက်ကြည့်ရအောင် index =  $(5+0) \bmod 9 = 5$  index =  $(5+6) \bmod 9 = 2$  index =  $(5+12) \bmod 9 = 8$  index =  $(5+18) \bmod 9 = 5$  တွေ့တဲ့အတိုင်း cycle issue တတ်သွားမှာပဲဖြစ်ပါတယ်။ GCD တန်ဖိုးက 1 မဖြစ်နေလိုပါ။ GCD တန်ဖိုးက 1 သာရဲ့လျှင် cycle issue မရှိဘဲ empty slot တွေကိုရှာနိုင်သွားမှာပါ။  $p(x)$  ခဲ့ဆ တန်ဖိုး ( $a$  တန်ဖိုး) သာ 1 တို့ 4 တို့ဖြစ်ခဲ့မယ်ဆိုရင် GCD တန်ဖိုး 1 ရတဲ့အတွက် cycle issue မရှိနိုင်တော့ပါဘူး။



ဒီနေရာမှာနောက်တစ်ခုမှတ်ထားစရေးရှိတာက table resizing ပါ၊ အပေါ်မှာရှိတဲ့ ကျနော်တို့ရဲ့ table Threshold က 6 ပါ။ ပြည့်သွားမယ်ဆိုရင် တော့ resize ထပ်လုပ်ဖို့လိုပါတယ်။ double resizing

လုပ်လိုက်တယ်ဆိုပါစ္စေး resizing လုပ်ပြီးပြီဆို key and value ကို formula (table size N ကပြောင်းသွားပါပြီ) အတိုင်း ပြန်တွက်ပြီးပြန်ထည့်ပေးဖို့လိုပါတယ်။

နောက် article မှာ open addressing မှာသံဃားတဲ့ quadratic function နဲ့အတူပြန်လာခဲ့ပါမယ်။

## Hash table - Open addressing (Quadratic Probing)

Hash table အကြောင်းရေးနေတဲ့ series ထဲ က open addressing မှာသုံးတဲ့ probing function တစ်ခုဖြစ်ပါတယ်။ အရင် article တွေ ဖတ်ပြီးမှ ဒီ article ဖတ်တာ ပိုအဆင်ပြေပါမယ်။

Hash table (hashing)

Hash table (Separate Chaining)

Hash table (Open Addressing)

Hash table (Open Addressing - Linear probing)

ဒီ article ကတော့ နည်းနည်းတို့သွားမယ်။ ဘာလို့လဲဆိုတော့ data collision ဖြစ်တာတို့ open addressing သုံးတဲ့ နေရာမှာ cycle issue ဖြစ်တာတွေအတွက် အရှေ့က article တွေမှာ လုံလုံလောက်လောက်ရှင်းပြပြီးသွားပြီဖြစ်ပါတယ်။

Quadratic probing ကတော့ quadratic formula ကိုသုံးပြီး probing လုပ်တယ်။ eg. (a\*square of x + bx +c). a != 0 ဖြစ်ရမယ်။ အရင် article တွေမှာ ပြောခဲ့သလိုပဲ probing function တွေမှာ cycle issue ရှိတတ်သလိုပဲ အခုပြာမယ့် quadratic probing မှာလည်း ဒီ issue ရှိပါတယ်။ ကျွန်ုတဲ့ process တွေက linear probing နဲ့အတူတူပဲ။ မတူတဲ့ အချက်က linear probing မှာ cycle issue ကိုရှင်းနိုင်ဖို့အတွက် GCD တန်ဖိုးရှာပြီး ထိန်းသွားတယ်။ quadratic မှာတော့ ရှင်းနိုင်မယ့် formula တွေအများကြီးရှိတယ်။ အဲထဲကမှ သုံးမျိုးလောက်ကို sharing လုပ်ပေးလိုက်ပါမယ်။ udemy course တစ်ခုကနေ မြို့ပြမ်းယူထားပါတယ်။

1.  $P(x) = \text{square of } x$  , table size N number က prime ဖြစ်ပြီးတော့ 3 ထက်ကိုးရမယ်၊ max load factor ကဲလဲ less than or equal by  $1/2$  ဖြစ်ရမယ်။
2.  $P(x) = (\text{square of } x + x)/2$  , ပြီးတော့ table size N က power of 2 ဖြစ်ပါမယ်။ ဥပမာ 2 power 2 = 4 , 2 power 3 = 8 စသည်ဖြင့်ပေါ့။
3.  $P(x) = (-1 \text{ power of } x) * \text{square of } x$ , ပြီးတော့ table size N က prime number ဖြစ်ပြီးတော့  $3 \bmod 4$  နဲ့ identical ဖြစ်ရမယ်။ ဥပမာ size N က 23 (prime number) .  $3 \bmod$

4 ဆို 3 ရမယ်။ ထိနည်းလည်းကောင်းပဲ  $23 \bmod 4$  ဆိုရင်လည်း 3 ရမယ်။  
ဒီလိုမျိုးဖြစ်ရမယ်လိုအပိုလိုတာပါ။

Number 1 , 2 နည်းကတော့ တွက်ရတာ ပိုလွယ်မယ်လိုထင်ပါတယ်။ အပေါ်က ရေးသားတဲ့ formula တွေအတိုင်း probing ကိုတွက်မယ်ဆိုရင် cycle issue မဖြစ်နိုင်တော့ဘဲနဲ့ completely fill up လုပ်နိုင်သွားမှာပါ။

0	1	2	3	4	5	6	7
$k_3, v_3$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$k_2, v_2$	$k_1, v_1$	$\emptyset$

$$\text{Recall } P(x) = (x^2 + x)/2, N = 8, \text{ threshold} = 3$$

### Operations:

insert( $k_1, v_1$ )  
insert( $k_2, v_2$ )  
**insert( $k_3, v_3$ )**  
insert( $k_4, v_4$ )  
insert( $k_3, v_5$ )  
insert( $k_6, v_6$ )  
insert( $k_7, v_7$ )

Suppose  $H(k_3) = 5$

$$\begin{aligned}
 H(k_3) &+ P(0) \bmod N \\
 5 &+ 0 \bmod 8 = 5 \\
 H(k_3) &+ P(1) \bmod N \\
 5 &+ 1 \bmod 8 = 6 \\
 H(k_3) &+ P(2) \bmod N \\
 5 &+ 3 \bmod 8 = 0
 \end{aligned}$$

Table ရဲ့ threshold ပြည့်သွားလို့ table resizing လုပ်တဲ့ ပုံစံလည်း linear probing မှာလုပ်တာနဲ့  
အတူတူပါပဲ။ cycle issue ကိုကိုင်တွယ်ပုံပဲကွာသွားမှာပါ။

နောက် article မှာ double hashing အကြောင်းရေးပေးသွားပါမယ်။ ဒါဆိုရင်တော့ hash table  
အကြောင်း လည်းတော်တော်စုံသွားပြီး လုံလောက်ပြီလိုထင်ပါတယ်။

## Hash table - Open addressing (Double Hashing)

Hash table အကြောင်းရေးနေတဲ့ series ထဲ က open addressing မှာသုံးတဲ့ probing function တစ်ခုဖြစ်ပါတယ်။ အရင် article တွေ ဖတ်ပြီးမှ ဒီ article ဖတ်တာ ပိုအဆင်ပြေပါမယ်။

Hash table (hashing)

Hash table (Separate Chaining)

Hash table (Open Addressing)

Hash table – Open addressing (Linear Probing)

Hash table – Open addressing (Quadratic Probing)

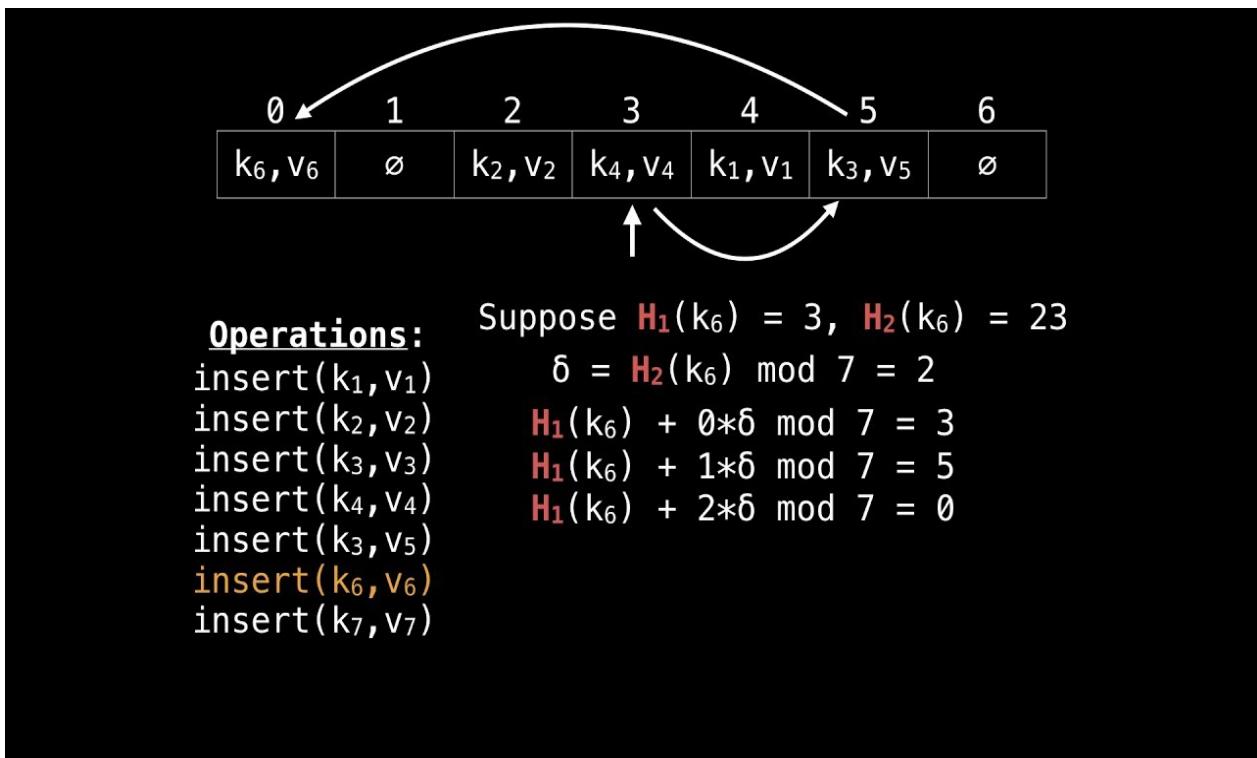
double hashing လိုအပိုပေမဲ့ အရှေ့က article တွေလိုမျိုးဖြစ်တဲ့ probing function တစ်ခုပဲ ဖြစ်ပါတယ်။ သူကတော့ probing formula ထုတ်တဲ့ အချိန်မှာ hashing နောက်တစ်ခု ထပ်ထည့်ပြီး probing လုပ်လိုက်သလိုပေါ့။ ဥပမာ  $P(key,value) = value * H(key) . H(key)$ ဆိုတာ ကတော့ ကျနော်တို့အခုပြောတဲ့ secondary function ပဲဖြစ်ပါတယ်။ Hash table နဲ့သူ့ပဲပတ်သတ်တဲ့ အကျိုးအကြောင်းတွေကိုတော့ ရှေ့က article တွေမှာ တော်တော်များများပြောပြီးသွားပြုဆိုတော့ ဒီ double hashing article မှာ cycle issue ဖြစ်တဲ့အချိန် ဘယ်လိုကိုင်တွေယ်မလဲ နဲ့ အခုသုံးတဲ့ double hasing ကိုဘယ်လိုလုပ်လဲဆိုတာပါပဲ။ ကျန်တာတွေကတော့ ရှေ့က probing function တွေလုပ်ထုံးလုပ်နည်းနဲ့ အတူတူပါပဲ

Cycle issue ကိုဖြေရှင်းနည်းက linear probing မှာ ရှင်းတာနဲ့ဆင်ပါတယ်။ GCD အားကိုးနဲ့ပါပဲ။ အရင်ဆုံး delta value တစ်ခုထုတ်တယ်။ Delta = 2ndhashfunction mod N , ဒီနေရာမှာ table size N က prime number ဖြစ်နေရပါမယ်။ delta ကို 0 ထွက်လို့မရပါဘူး။ 0 ထွက်ရင် equation ထဲ ထည့်သုံးလိုက်ရင် cycle issue ထဲရောက်မှာကျိန်းသေပါတယ်။ ဆိုတော့ delta value က greater than or equal to 1 and less than N ဖြစ်ရပါတယ်။ ဆိုတော့ နောက်တစ်ဆင့် အနေနဲ့ GCD တန်ဖိုးကိုတွက်တော့မယ်ဆို GCD(delta,N) နဲ့တွက်ပါမယ်။ table size N က prime number

ဖြစ်နေတဲ့အတွက် ရလဒ်က 1 ထွက်ပါမယ်။ GCD တန်ဖိုး 1 ရမယ်ဆိုရင် cycle issue ကိုဖြေရှင်းပြီးသားဖြစ်သွားပါလိမ့်မယ်။

Hashing ကိုပြောရမယ်ဆိုရင် double hash လုပ်တဲ့နေရာမှာ လုပ်ရမယ့် value တွေက fundamental blocks တွေပဲဖြစ်ပါတယ်။ ဥပမာ integer, strings, real number စသည်ဖြင့်ပေါ့။ fundamental blocks တွေအတွက် hash function သုံးတော့မယ်ဆိုရင် universal hash functions တွေထဲက လုမ်းခွွဲသုံးပါတယ်။ အောက်မှာ wiki link ပါပါတယ်။

[https://en.wikipedia.org/wiki/Universal\\_hashing](https://en.wikipedia.org/wiki/Universal_hashing)



resizing လုပ်တဲ့ technique ကလည်း ကျွန်တဲ့ probing function တွေနဲ့အတူတူပါပဲ။ ဒါဆိုရင်တော့ hash table ခဲ့ open addressing မှာ popular ဖြစ်တဲ့ probing function သုံးခုလုံးပြီးသွားပြီ။ လက်ရှိမှာတော့ hash table ဝဲကို key value တွေ insert လုပ်တဲ့ case နဲ့ပဲပြောထားတာဆိုတော့ နောက် article မှာ removing case ကိုရေးပေးသွားပါမယ်။

## Hash table – Removing elements (open addressing)

Hash table အကြောင်းရေးနေတဲ့ series ထဲကနေ key တွေ remove လုပ်တဲ့ article ပဲဖြစ်ပါတယ်။ open addressing နဲ့ ဥပမာပြုပြီးပြောသွားပါမယ်။ အရင် article တွေဖော်ပြီးမှ ဒါ article ဖတ်တာ ပိုအဆင်ပြုပါမယ်။

ကျနော်တို့ hash table မှာ key value တွေထည့်တော့မယ်ဆိုရင် hash လုပ်ပြီးထည့်တယ်။ collision တွေဖြစ်တဲ့အခါ separate chaining တို့ open addressing တို့ technique တွေသုံးပြီးဖြေရှင်းကြပါတယ်။ ထည့်တဲ့အခိုန်မှာလဲ hash လုပ်ပြီးထည့်တယ်ဆို remove လုပ်တဲ့ အခိုန်မှာလည်း hash လုပ်ပြီးပဲremove လုပ်ပါတယ်။ ဆိုလိုချင်တာက hash table ထဲမှာထည့်ဖို့အတွက် key ကို hash လုပ်ပြီး position index ရှာတယ်။ remove လုပ်တဲ့ အခိုန်မှာလဲ ဖျက်ရမယ့် position index ကို hash လုပ်ပြီးပဲရှာပါတယ်။ ဒါပေမဲ့လည်း remove လုပ်တာကဒီလောက်ကြီး မရိုးရှင်းပါဘူး။ အောက်မှာ scenario လေးတစ်ခုနဲ့ဆွဲးနေးသွားပါမယ်။

# Hash table (HT) Removing elements open addressing

A quick guide to removing key-value pairs  
in a hash table via open addressing

ကျနော်တို့မှာ table size 5 ခုပါတဲ့ အရာတစ်ခုရှိတယ်ဆိုပါစို့။ အခန်း 1(key1, val1) , 2(key2, val2) , 3(key3, val3) မှာ key&value တွေရှိတယ်။ key2 ကိုစမ်းဖျက်ကြည့်ရမယ်ဆို အရင်ဆုံး key2 ကို

hash လုပ်ပြီးတော့ position index ကိုရှာပါမယ်။ အကယ်လို့ result က 2 ရတယ်ဆိုရင်တော့ အထဲကို ကြည့်ပြီး key2 သေချာတယ်ဆို ဖျက်မယ်။ အကယ်လို့ 2 မဟုတ်တဲ့ တစ်ခြားတစ်ခုရဲ့မယ်ဆိုရင် အထဲကိုစစ်ကြည့်မယ်၊ key2 မဟုတ်ဘူးဆို မဖျက်ဘဲနဲ့ ဆက်ပြီးတော့ probing လုပ်ပြီးတော့ key2 ကိုရှာပြီးဖျက်ပါမယ်။ ဒါက key2 ကိုရပြီ၊ ဖျက်လိုက်ပြီရင် အဲ slot index က null ဖြစ်သွားပါတယ်။

Null ဖြစ်သွားရင်ဘာဖြစ်လဲဆိုတော့ Hash table မှာ searching လုပ်တဲ့အချိန်မှာ issue ဖြစ်ပါတယ်။ ဆိုပါတော့ ကျနော်တို့ က key3 ကိုရှာချင်တယ်။ probing လုပ်ပြီးတော့ တစ်ဆင့်ချင်းဆီသွားတယ်။ index 1 ရတယ်၊ စစ်တယ်၊ key1 တွေ့တယ်၊ probing ဆက်လုပ်တယ်၊ index2 မှာ null ကြီးဖြစ်နေတယ်။ hashtable ရဲ့သဘောတရားက အဲလို့ null ကို အရင်တွေ့သွားပြီဆို သူ့တို့ရှာမယ်။ key3 က မရှိတော့ဘူးလို့ယူဆလိုက်ပါတယ်။ ဆိုတော့ remove လုပ်တဲ့ case မှာ remove ပြီးတိုင်း null မထားဘဲနဲ့ unique marker တစ်ခုထားလိုက်ပါတယ်။

အဲဒီ unique marker ကို tombstone လို့ခေါ်ပါတယ်။ remove လုပ်ပြီးတိုင်း အဲဒီ slot index မှာ tombstone လေးတွေထားခဲ့တယ်။ ဒါဆိုရင် searching လုပ်တဲ့အချိန်မှာ tombstone တွေတွေရင် skip လုပ်ပြီး probing ဆက်လုပ် ဆက်ရှာ၊ ဒါမျိုးလုပ်သွားလို့ရပါတယ်။ tombstone နဲ့ပတ်သက်ပြီး မှတ်စရာ J မျိုးရှိပါတယ်။

ဒါကတော့ table resizing လုပ်လိုက်တဲ့အချိန်ကျ tombstone တွေအကုန်ပျောက်ပါတယ်။ null တွေပြန်ဖြစ်တယ်ပေါ့။ resize လုပ်ပြီးသွားရင် key&value အကုန် hash လုပ်ပြီးပြန်ထည့်တဲ့အတွက် tombstone တွေမလိုတော့တဲ့အတွက်ကြောင့်ဖြစ်ပါတယ်။ နောက်တစ်ချက်က tombstone တွေကို key&value တွေနဲ့ အစားထိုးလို့ရပါတယ်။ ဥပမာပြောရရင် ကျနော်တို့ key တစ်ခုကို lookup (search) လုပ်တော့မယ်ဆိုပါစို့။ probing လုပ်တယ်၊ စစ်ချင်းမှာတင် tombstone တွေ့တယ်၊ ရှာတာမတွေ့သေးတဲ့ အတွက် probing ထပ်လုပ်ပြီး ထပ်ရှာတယ်။ ငံ ခေါက်မြောက်မှာ ကိုယ်ရှာတဲ့ key တွေ့ပြီးပဲဆိုပါစို့၊ lookup လုပ်တဲ့ process အပြီးမှာ ခုနတုန်းက ရှာတဲ့ process မှာ ကိုယ်ရှာတဲ့ key မတွေ့သေးခိုန်မှာ တွေ့လိုက်တဲ့ tombstone ရှိတဲ့ slot နေရာမှာ သွားအစားထိုးထားလို့ရပါတယ်။ ဒါဆိုရင်နောင်တစ်ချိန် lookup ပြန်လုပ်တဲ့နေရာမှာ ငံ ကြိမ်ရှာစရာမလိုတော့ဘဲ first try မှာတင်ရှာတွေ့သွားမှာပဲဖြစ်ပါတယ်။

ဒါဆိုရင်တော့ hash table နဲ့ပတ်သက်ပြီး article တွေစုံသွားပြီလို့ထင်ပါတယ်။

## Fenwick tree (Binary Indexed Tree)

Fenwick tree အကြောင်းစပြောတော့မယ်ဆိုရင်တော့ fenwick ကိုဘာကြောင့်သုံးတာလဲ fenwick ကဘာလဲဆိုတာနဲ့ စဆွဲးနွေးကြရအောင်။

ဥပမာ တစ်ခုနဲ့ပြောရမယ်ဆိုရင် ကျနော်တို့မှာ array 10 ခန်းရှိတဲ့ data တစ်ခုရှိတယ်ဆိုပါစွာ။  
လိုချင်တာက range query ပုံစံမျိုးလိုချင်တာ။ (eg. Sum of array index 1 to 7, array အခန်း ၁ ကနေပြီးတော့ ၇ အထိပေါင်းခြင်းရလဒ်).  
ပေါင်းလို့ရလားဆိုတော့ ရတယ် လေ။ array ၁ ခန်းခြင်းဆို  
လိုက်ထောက်ပြီးတော့ ပေါင်းရုပဲ။ array 1 + array 2 + array 3 etc. ပုံစံမျိုးပေါ့။ ဒါပေမဲ့  
တစ်ခုခြင်းဆိုလိုက်ထောက်နေရတဲ့အတွက် linear time ကြာတယ်။ ဆိုလိုချင်တာက range  
သာကြိုးလာမယ်ဆိုရင် performance ကလည်း ထပ်တူ နေးလာလိမ့်မယ်။

အဲလို တစ်ခန်းခြင်းဆိုလိုက်ထောက်မယ့်အစား prefix sum ဆိုတဲ့ array structure  
တစ်ခုဆောက်လိုက်မယ်။ array အခန်း ၁၁ ခန်းနဲ့ပေါ့။ အဲဒီ ၁၁ ခန်းမှာ array index တွေကို  
တစ်ခန်းခြင်းဆိုလိုက်ထောက်ပြီး ပေါင်းပြီးသား result တွေကို store လုပ်ထားမယ်။ sum range  
query လုပ်တော့မယ်ဆို ဥပမာ index 1 to 7 ရဲ့ ပေါင်းလဒ်ကိုလိုချင်ပြီဆို prefix sum လုပ်ထားတဲ့  
array ဆိုကနေ value of index 7 – value of index 1 ဆို လိုချင်တဲ့ အဖြောက်ရသွားမှာဖြစ်ပါတယ်။  
array တစ်ခန်းခြင်းဆိုထောက်ပြီးလဲ ပေါင်းစရာမလိုတော့ဘူးပေါ့။ attach တွဲထားတဲ့ example  
ကြည့်နိုင်ပါတယ်။

0	1	2	3	4	5	6	7	8	9		
A =	5	-3	6	1	0	-4	11	6	2	7	
P =	0	5	2	8	9	9	5	16	22	24	31

Sum of A from [2,7) = P[7] - P[2] = 16 - 2 = 14

ဒါဆို ကျနော်တို့မှာ လက်ရှိ normal array နဲ့ prefix sum လုပ်ထားတဲ့ array ဆိုပြီး နှစ်ခုရှိတယ်။ အဲထဲကမှ normal array ထဲက value တစ်ခုကို change ချင်တယ်ဆိုပါစို့၊ change လုပ်မယ်ဆိုရင် prefix sum လုပ်ထားတဲ့ array ပျက်သွားပြီတော့ အစအဆုံး sum value တွေကို ပြန်တွက်ပေးဖို့လိုလာပါမယ်။ ဒီနေရာမှာ fenwick tree ဆိုပြီး ဝင်လာတာပါ။ range query တွေလဲဆွဲလိုရမယ်။ key ကိုလည်း update လုပ်လိုရမယ်၊ linear time မကြာဘဲနဲ့ performance ကောင်းကောင်းနဲ့ပေါ့။

ဒါ article မှာတော့ fenwick ကိုသုံးပြီးတော့ range query တွက်နည်းရေးပေးသွားပါမယ်။ fenwick နဲ့ range query လုပ်တော့မယ်ဆိုတော့ least significant bit (LSB)

အကြောင်းလေးပါထည့်ပြောသွားလိုက်ပါမယ်။ LSB ဆိုတာ binary number series မှာရှိတဲ့ lowest bit, ညာဘက်ကနေစပြီး count လုပ်ပါတယ်။ ဥပမာ 10001 ဆိုရင် ညာဘက်ကနေစ count ပြီး LSB က 1 နေရာမှာရှိပါတယ်။ 10010 ဆိုရင် 2 နေရာမှာရှိပါတယ်။ 10100 ဆိုရင် 3 နေရာ၊ 11000 ဆိုရင် 4၊ 10000 ဆိုရင် 5။ ဒီလောက်ဆိုရင် LSB သိပြီလိုထင်ပါတယ်။

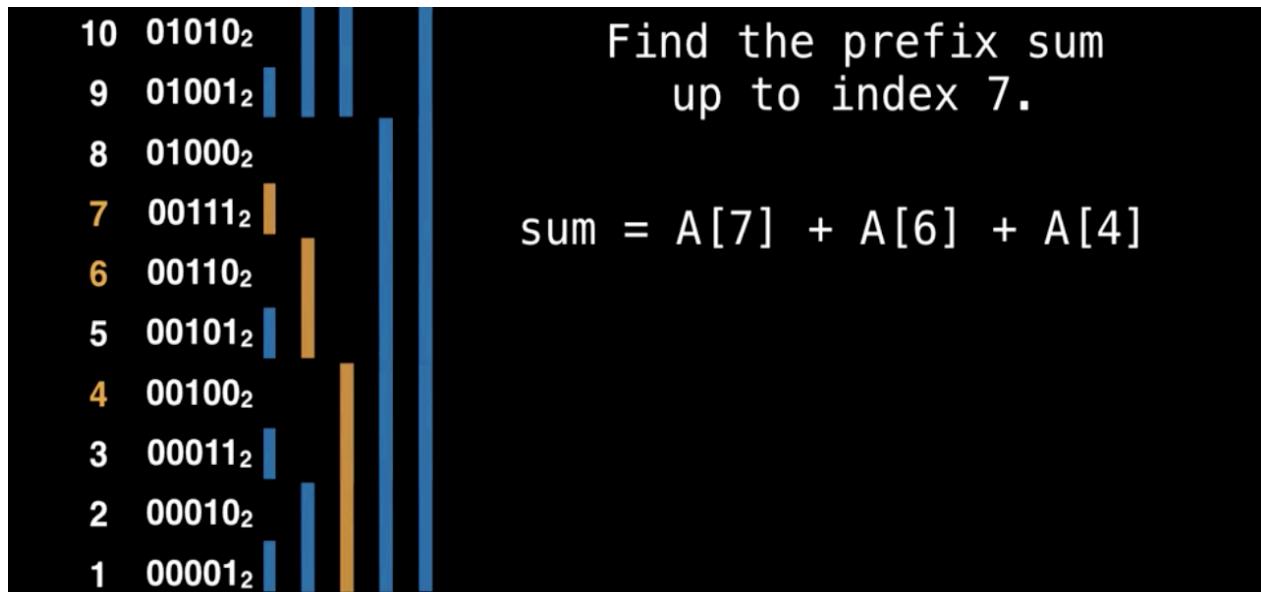
အပေါ်ကပြောထားတဲ့အတိုင်း ကျနော်တို့မှာ array 10 ခန်းပါတဲ့ data တစ်ခုရှိတယ်ဆိုပါစို့။ index 1 to 10 ကို binary နဲ့ အတူ range value တွေစမ်းတွက်ကြည့်ရအောင်။

Index 10 = 1010, LSB = 2, ပြီးရင် 2 to the power of LSB - 1(fenwick မှာ base အတွက် 2 ကိုသုံးပါတယ်). ဆိုတော့ 2 to the power of LSB-1 = 2 ရပါတယ်။

Index 9 ဆိုရင် = 1001, LSB = 1, 2 to the power of 1-1 = 2 power 0 = 1 ရပါတယ်။

အခုလို့ဥပမာပေးပြီး တွက်ထုတ်ထားတဲ့ range value တွေကို range query ဆွဲတဲ့ နေရာမှာ ဘယ်လိုသုံးလဲဆိုတာကြည့်ရအောင်။ Index 7 ခဲ့ prefix sum value ကိုလိုချင်တယ်ဆိုပါစို့။

Sum = Index[7] + Index[6] + Index[4] ရပါတယ်။ ဘာလို့လဲဆိုတော့ Index 7 ခဲ့ range value က 1 ပါ။ ဆိုတော့ အောက်တစ်ထစ်ဆင်းပါတယ်။ Index[6] ရပါတယ်၊ သူ့ခဲ့ range value က 2 ရပါတယ်။ အောက်နှစ်ထစ်ဆင်းပါတယ်၊ index[4] ရပါတယ်။ index[4] ခဲ့ range value 4 ရတဲ့အတွက် 0 အထိဆင်းသွားပြီး end ဖြစ်သွားပါတယ်။ attach တွဲထားပေးတဲ့ example ကြည့်လို့ရပါတယ်။



နောက်တစ်ခုအနေနဲ့  $\text{index}[4]$  ရဲ့ prefix sum ကိုတွက်ကြည့်ရအောင်။  $\text{index}[4]$  ရဲ့ prefix sum က  $\text{index}[4]$  အတိုင်းပါပဲ။ တစ်ချက်ထဲနဲ့ ပြီးသွားပါတယ်။ efficient ဖြစ်တယ်ဆိုတာ ဒါမျိုးကိုပြောတာပါ။

ဒါလိုရင် ကျနော်တို့ လိုချင်တဲ့ range query sum value ကိုဘယ်လိုတွက်မလဲရှင်းပါတယ်။ ဥပမာ  $\text{index } 3 \text{ to } 8 \text{ sum value}$  ကိုလိုချင်တယ်ဆိုပါစို့။  $\text{index } 8$  ရဲ့ prefix sum value ရှာမယ်။  $\text{index } 3$  ရဲ့ prefix sum value ရှာမယ်။ ပြီးရင် 8 ထဲက 3 ကိုနှုတ်လိုက်ရင် လိုချင်တဲ့ 3 to 8 range sum value ကိုရပါပြီ။

article ကိုတော့ အတတ်နိုင်ဆုံး နားလည်ရလွယ်အောင်ရေးထားပေးပါတယ်။ နားမလည်လို့ ဆွေးနွေးချင်တာရှိရင် လာပြောလို့ရပါတယ်။ နောက်နေ့ မှာ point update လုပ်တဲ့ အကြောင်း ရေးပေးသွားပါမယ်။

## Fenwick tree (Point Update)

အရှေ့က ရေးခဲ့တဲ့ topic မှာ fenwick tree ရဲ့ intro နဲ့ sum range query

တွက်နည်းကိုပြောပြပေးခဲ့ပါတယ်။ အခု article မှာတော့ မူလရှိပြီးသား array မှာ array value ကိုပြောင်းချင်ရင် fenwick မှာဘယ်လိုပုံလုပ်လဲဆိုတာကိုပြောပြပေးသွားပါမယ်။

ဒီနေရာမှာပြောစရာရှိတာက range query ပဲတွက်တွက် point update ပဲလုပ်လုပ် range value တန်ဖိုးကိုအရင်တွက်ဖို့လိုပါတယ်။ တွက်နည်းကိုလည်း အရင် article မှာရေးပေးခဲ့ပါတယ်။ revision ပြန်တဲ့အနေနဲ့ထပ်ရေးလိုက်ပါတယ်။

array 10 ခန်းပါတဲ့ data တစ်ခုရှိတယ်ဆိုပါစို့။ index 1 to 10 ကို binary နဲ့ အတူ range value တွေစမ်းတွက်ကြည့်ရအောင်။

Index 10 = 1010 , LSB = 2, ပြီးရင် 2 to the power of LSB - 1(fenwick မှာ base အတွက် 2 ကိုသုံးပါတယ်). ဆိုတော့ 2 to the power of LSB-1 = 2 ရပါတယ်။

Index 9 ဆိုရင် - 1001 , LSB = 1, 2 to the power of 1-1 = 2 power 0 = 1 ရပါတယ်။

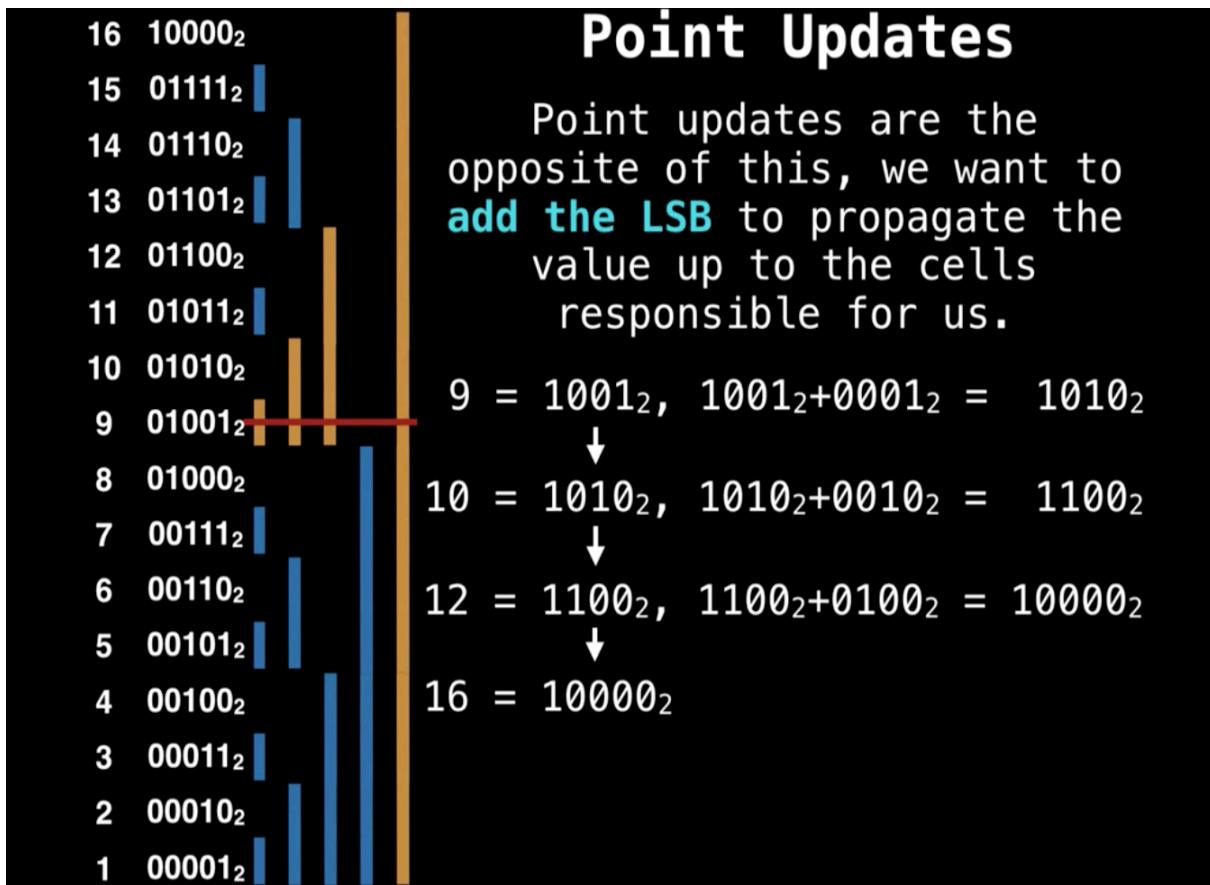
အဲဒီ range value တွေကိုသုံးပြီးတော့ prefix sum ကိုတွက်ပါတယ်။ index 7 ရဲ့prefix sum ကိုလိုချင်တယ်ဆို

Prefix sum = Index[7] + Index[6] + Index[4] ရပါတယ်။ ဘာလိုလဲဆိုတော့ Index 7 ရဲ့range value ၂ ပါ။ ဆိုတော့ အောက်တစ်ထစ်ဆင်းပါတယ်။ Index[6] ရပါတယ်၊ သူရဲ့range value ၂ ရပါတယ်။ အောက်နှစ်ထစ်ဆင်းပါတယ်၊ index[4] ရပါတယ်။ index[4] ရဲ့range value 4 ရတဲ့အတွက် 0 အထိဆင်းသွားပြီး end ဖြစ်သွားပါတယ်။ အရင် article ကစာဖြစ်ပါတယ်။

အခုမှာပြောချင်တာကိုရောက်ပါပြီ။ array တစ်ခုရဲ့value ကို change ချင်ရင်ကျဲ့ prefix sum တွက်တာနဲ့ပြောင်းပြန် အချိုးကျပါတယ်။ prefix sum မှာက range value ကိုတွက်ပြီး prefix sum အတွက် တစ်ဆင့်ခြင်းဆီး အောက်ဆင်းသွားပါတယ်။ point update လုပ်တဲ့နေရာမှာကျဲ့ range value အတိုင်း အပေါ်တတ်သွားရပါမယ်။ ဥပမာ တစ်ခုနဲ့ကြည့်ကြည့်ရအောင်။

Array 16 ခန်းပါတဲ့ data structure တစ်ခုနဲ့ ပြောကြည့်ပါမယ်။ ကျဖော်တို့ က index 9 မှာရှိတဲ့ value ကိုပြောင်းချင်ပါတယ်။ ဒါဆိုရင် range value တွေတွေက်ပြီး တစ်ဆင့်ခြင်းဆို အပေါ်တတ်သွားပါမယ်။ attachment image ပါကြည့်လို့ရပါတယ်

Index 9 ရဲ့ range value 1 ရပါတယ် = index 10 Index 10 ရဲ့ range value 2 ရပါတယ် = index 12 Index 12 ရဲ့ range value 4 ရပါတယ် = index 16 Index 16 မှာ end ဖြစ်သွားပါပြီ။ ဆိုတော့ index 9 မှာရှိတဲ့ value တစ်ခုကို ပြောင်းမယ်ဆို 9 to 16 အထိ လိုက်ပြောင်းနေစရာမလိုပါဘူး။ index 9,10,12,16 ကိုပြောင်းလိုက်ရင်အဆင်ပြေားပါပြီ။



## Fenwick Tree Construction

Fenwick tree ရဲ့ range query တွက်နည်းရော point update လုပ်နည်းရော ပြောပြီးပြဆိုတော့  
နောက်ဆုံးအနေနဲ့ အရေးကြီးဆုံး fenwick tree ဘယ်လိုအောက်လဲဆိုတာ ရေးပေးသွားပါမယ်။  
range query တို့ point update တို့မလုပ်ခင်မှာ fenwick tree က အရင်အောက်ထားရမှာ  
ဖြစ်ပါတယ်။ fenwick tree ကို tree structure အတိုင်းလဲ ဆောက်သွားလို့ရတယ်။ ဒါ article မှာတော့  
ဖတ်ရလွယ်အောင်နဲ့ နားလည်ရလွယ်အောင် linear structure နဲ့ ဆောက်သွားပြပါမယ်။

Attach တွဲပေးထားတဲ့ ပုံက ကျေနော် နမူနာ ဆွဲပြထားတဲ့ fenwick tree ဖြစ်ပါတယ်။  
ဘယ်လိုတွက်ပြသွားလဲဆိုတာ ရှင်းပြသွားပါမယ်။ စာနဲ့ ပုံနဲ့ တွဲပြီး ဖတ်ကြည့်လို့ရတာပေါ့။

Index	Binary	Input	Fenwick value
10	1010	8	7
9	1001	-1	-1
8	1000	2	(21-2+4) = 23
7	0111	4	4
6	0110	1	-2
5	0101	-3	-3
4	0100	5	19
3	0011	6	6
2	0010	-2	8
1	0001	10	10

တွက်တဲ့နေရာမှာ range value တွေကိုတွက်ပြီးတော့ fenwick value တွေထုတ်သွားပါမယ်။ range  
value တွက်နည်းတွေကို ရှေ့က article တွေမှာပြန်ကြည့်နိုင်ပါတယ်။ စတွက်ကြည့်ရအောင်။

Range value of index 1 = 1, အပေါ်1 ဆင့်တတ်ပြီးပေါင်း =  $10 + (-2) = 8$ , index 2 = 8 Range value of index 2 = 2, အပေါ်2 ဆင့်တတ်ပြီးပေါင်း =  $8 + 5 = 13$ , index 4 = 13 Range value of index 3 = 1, အပေါ်1 ဆင့်တတ်ပြီးပေါင်း =  $6 + 13 = 19$ , index 4 = 19 Range value of index 4 = 4, အပေါ်4 ဆင့်တတ်ပြီးပေါင်း =  $19 + 2 = 21$ , index 8 = 21 Range value of index 5 = 1, အပေါ်1 ဆင့်တတ်ပြီးပေါင်း =  $-3 + 1 = -2$ , index 6 = -2 Range value of index 6 = 2, အပေါ်2 ဆင့်တတ်ပြီးပေါင်း =  $-2 + 21 = 19$ , index 8 = 19 Range value of index 7 = 1, အပေါ်1 ဆင့်တတ်ပြီးပေါင်း =  $4 + 19 = 23$ , index 8 = 23 Range value of index 8 = 8, အပေါ်8 ဆင့်တတ်လိုမရတော့တဲ့ အတွက် ignore Range value of index 9 = 1, အပေါ်1 ဆင့်တတ်ပြီးပေါင်း =  $-1 + 8 = 7$ , index 10 = 7

Fenwick tree တွက်ပြီးပြီဆိုတော့ range query ဆွဲတာတို့ point update လုပ်တာတို့ စလုပ်လို့ ရပါပြီ။ မှန်မမှန် proof လုပ်တဲ့အနေနဲ့ sum query တစ်ခုလောက်ဆွဲကြည့်ပါမယ်။

Index 7 ရဲ့ prefix sum ကိုတွက်ကြည့်ရအောင်။ ပုံမှန်အတိုင်းတွက်မယ်ဆိုရင်  $10 - 2 + 6 + 5 - 3 + 1 + 4 = 21$  ရပါမယ်

Fenwick နဲ့တွက်ကြည့်မယ်ဆို index 7 ရဲ့ prefix sum ကိုလိုချင်ရင်. Prefix sum of index 7 =  $\text{index}(7) + \text{index}(6) + \text{index}(4)$ . ps(နာမလည်ရင် ရှေ့က article တွေပြန်ဖတ်ပါ။) Prefix sum of index 7 =  $4 - 2 + 19 = 21$  ရပါမယ်။

ကိုယ့်ဘာသာ input data ကို တစ်ခြား ဂဏန်းတွေ စမ်းထည့်ကြည့်ပြီးလဲတွက်ကြည့်လို့ရပါတယ်။

## AVL Tree

AVL tree ဆိုတာကတော့ binary tree ကို modify ထပ်လုပ်ထားတဲ့ balanced binary tree ကိုအခြေခံထားတဲ့ algorithm တစ်ခုပဲဖြစ်ပါတယ်။ Binary tree ရှိခဲ့သားနဲ့ balanced binary ဆိုပြီး ဘာလို့ထပ်လာတာလဲမေးစရာရှိပါတယ်။ ရှင်းပါတယ် binary tree ကမပြည့်စုံသေးလို့ပါ။ ဘာလို့လဲဆိုတော့ binary tree ရဲ worst case တွေမှာ performance က linear time အထိ ပြန်နိမ့်ကျသွားလို့ပါ။ ဥပမာ 1, 2, 3, 4, 5, 6, 7 ကို binary tree နဲ့ စီထည့် လိုက်မယ်ဆို tree က right side ကိုပဲ တစ်ဖတ်သတ်ချည်းစီသွားလိုက်မယ် ဘာလို့ right side ကိုဆင်းသွားလဲဆိုတာမသိရင် binary tree articles တွေပြန်ဖတ်ဖို့လိုပါတယ်။ အောက်က link တွေမှာဖတ်ပါ။

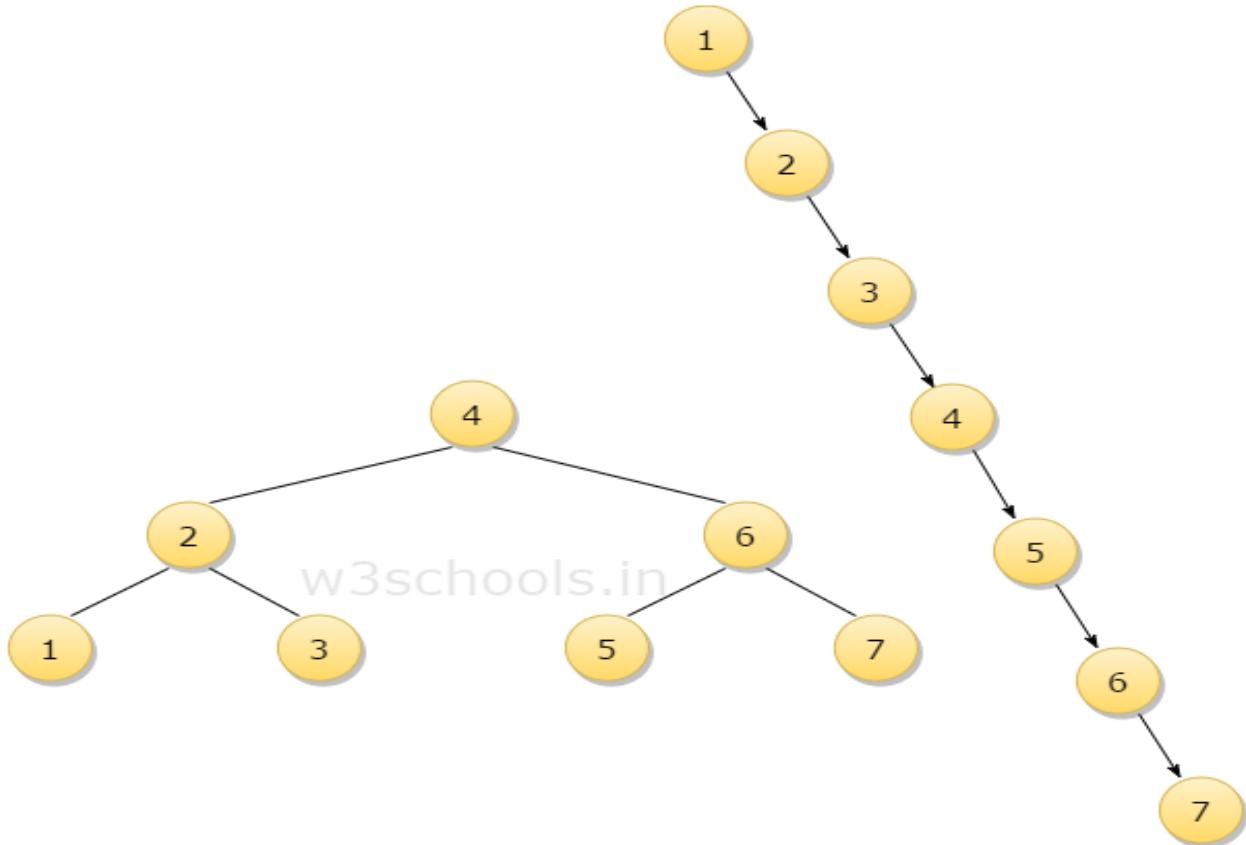
Binary tree

Binary tree's operation

Binary tree's traversal

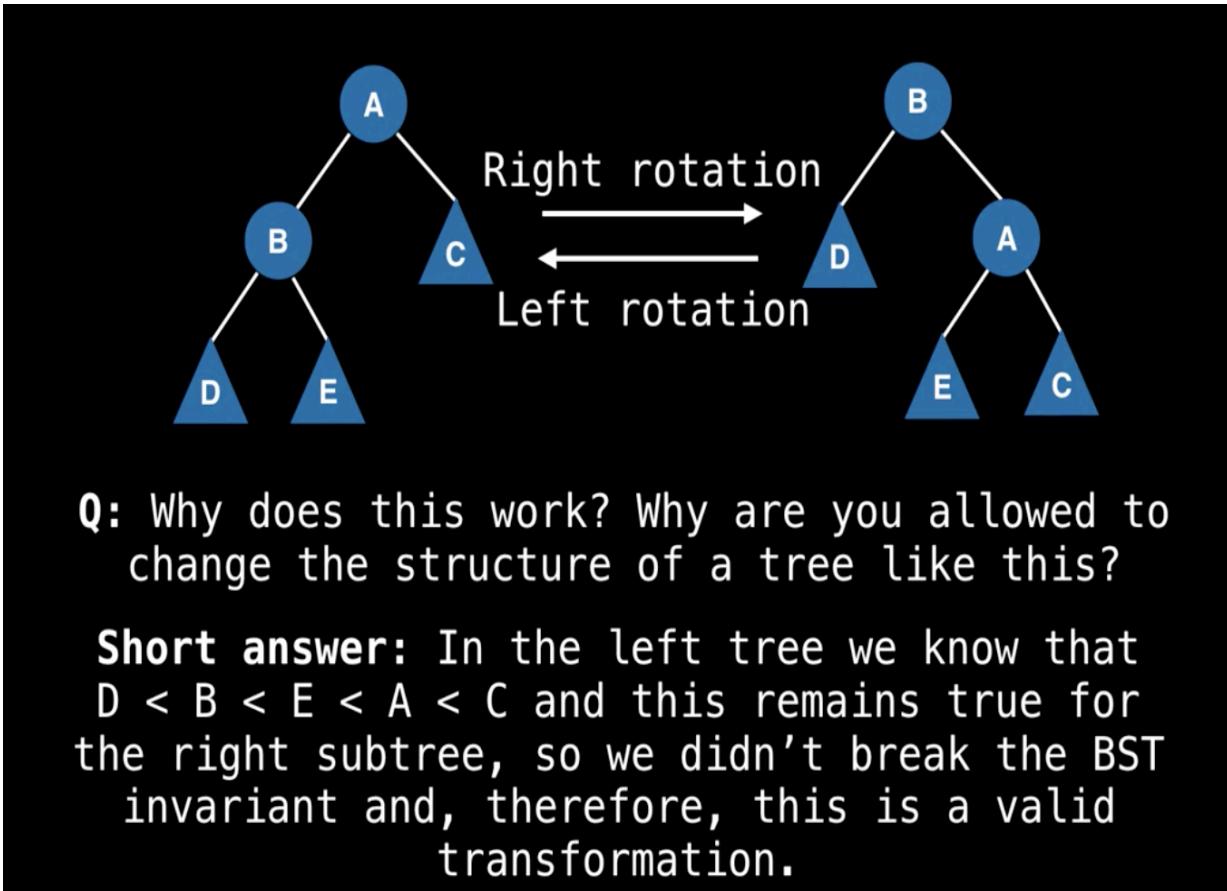
Binary search

စကားပြန်ဆက်ရမယ်ဆို ဆိုလိုချင်တာက node တစ်ခု insert ထည့်တော့မယ်ဆို process က 7 ခါလုပ်ရပါမယ်၊ ဆိုတော့ linear အတိုင်းပဲပြန်ကြောသွားပါလိမ့်မယ်။ အဲအတွက်ကြောင့် မို့လို့ binary tree တွေကို balanced ထပ်လုပ်ဖို့လိုအပ်လာတာပါ။ balanced လုပ်ထားတဲ့ tree ဆိုရင် linear time မကြာဘဲ နဲ့ process ကိုလျော့ချုပ်ပြီး performance အတွက်ပိုကောင်းပါတယ်။ attachment မှာ sample ကြည့်နိုင်ပါတယ်။

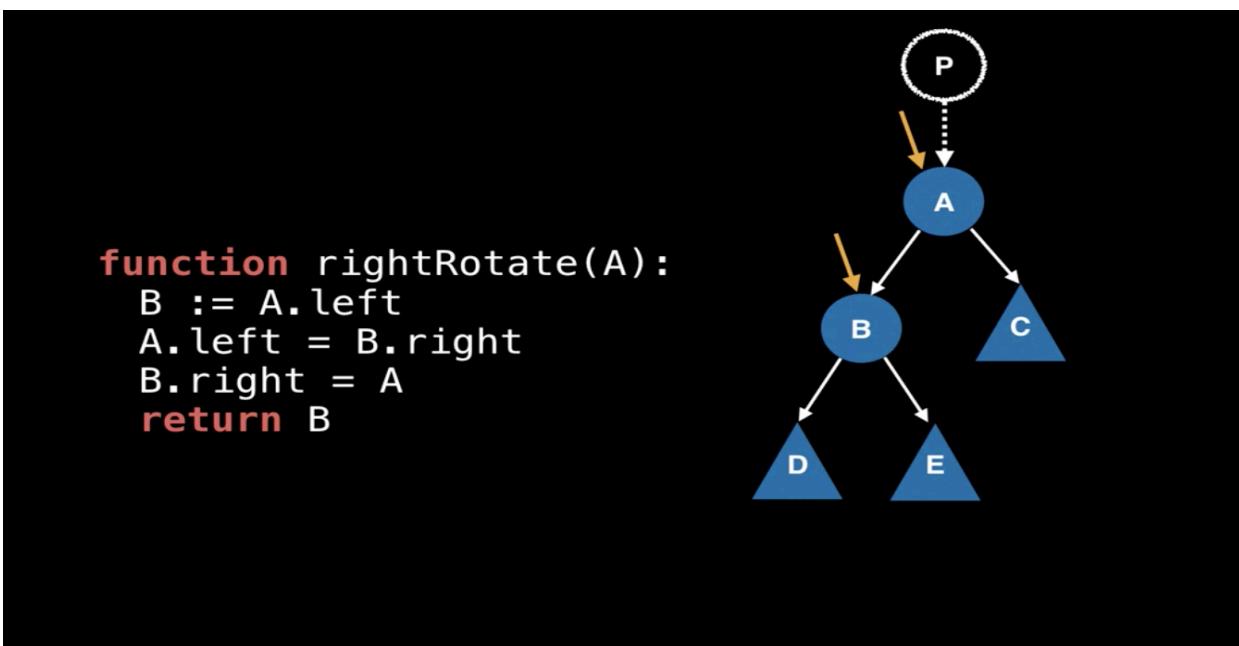


Balanced binary tree မှာ အခရာကျတာက binary tree ကို rules (ဘယ်ကငယ်ညာကြီး) ကိုလိုက်နာနိုင်ပါ။ နောက်တစ်ချက် က tree rotation ပါပဲ။ Tree rotation ဘယ်လိုလုပ်လဲဆိုတာ အောက်က attachment example ပုံလေးတွေမှာ တစ်ဆင့်ခြင်းဆီကြည့်လိုပါတယ်။

## Tree Rotation Sample



## Tree Rotation Example 1



## Tree Rotation Example 2

```
function rightRotate(A):
    B := A.left
    A.left = B.right
    B.right = A
    return B
```

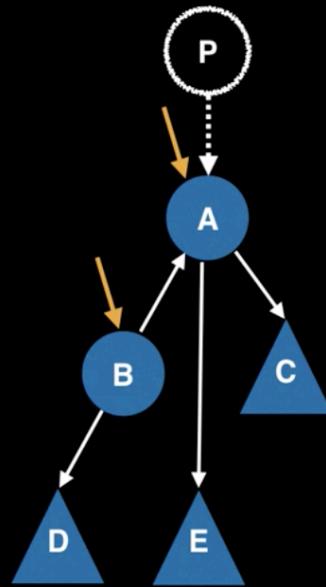
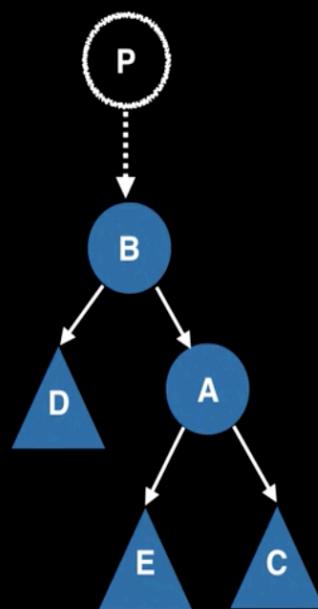


Image of avl

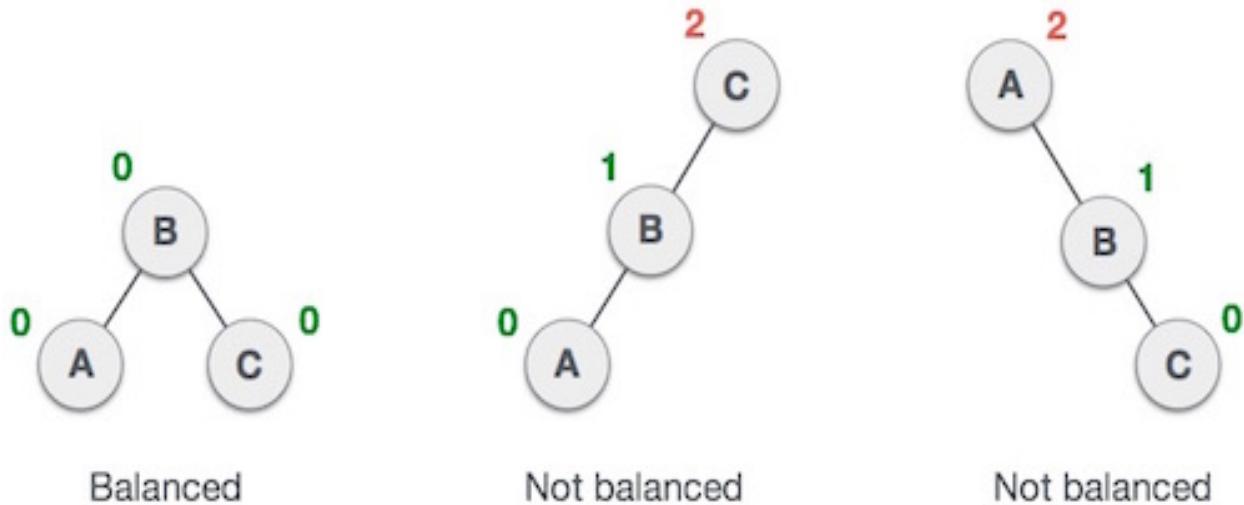
## Tree Rotation Example 3

```
function rightRotate(A):
    B := A.left
    A.left = B.right
    B.right = A
    return B
```



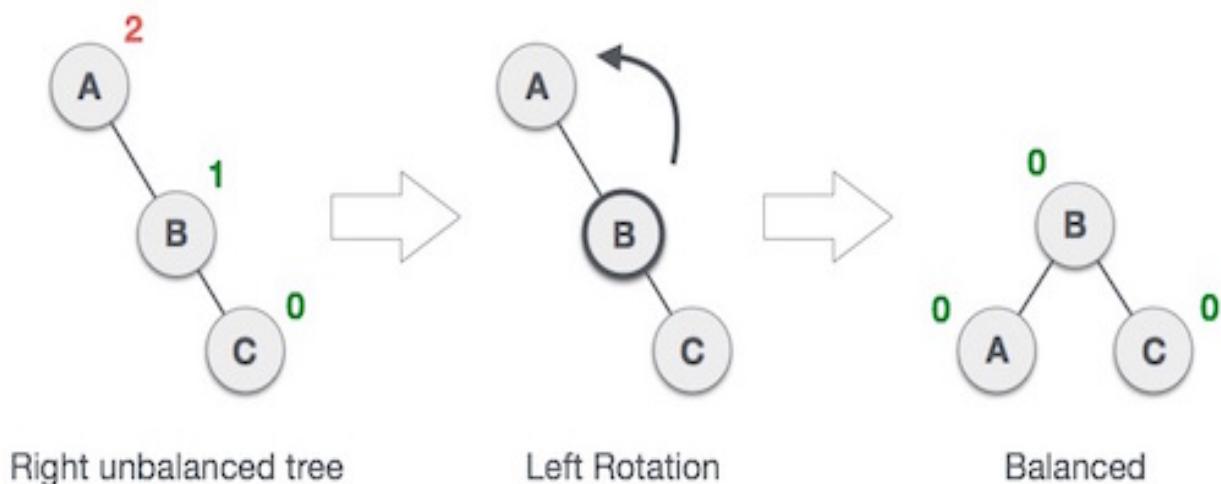
AVL tree ဆိုတာလဲ balancing လုပ်ထားတဲ့ tree ပါပဲ။ AVL မှာတော့ tree ရဲ့ height ကိုယူပြီးတော့ balance ဖြစ်အောင်လုပ်ပါတယ်။ height တွေကိုအခြေခံပြီး balance factor တွက်ပါတယ်။

$\text{BalanceFactor} = \text{height(left-subtree)} - \text{height(right-subtree)}$ . Sub tree တစ်ခုမှာ balance factor ကို 1 ထက်ကြီးလို့မရပါဘူး။ ဆိုလိုချင်တာက difference 1 ပဲလက်ခံပါတယ်။ ဥပမာ -1, 1, 0 ဖြစ်မှသာ balance ဖြစ်ပါမယ်။ နမူနာပုံ attach တဲ့ပေးလိုက်ပါတယ်။

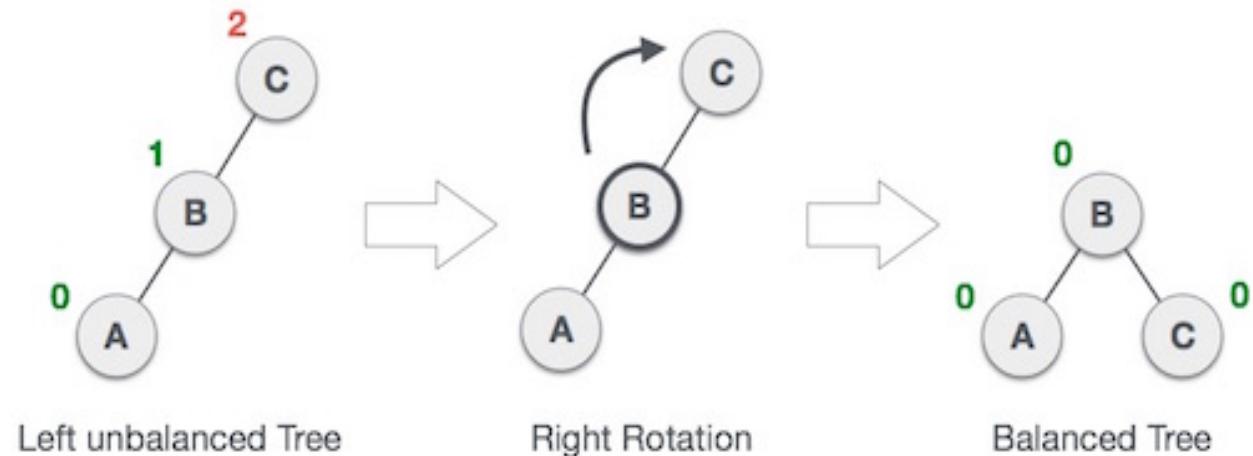


Rotation 4 မျိုး ရှိပါတယ်။ စာနဲ့ရှင်းပြတာထက်စာရင် ပုံလေးတွေနဲ့ ပို့နားလည်လွယ်မယ်ထင်ရတဲ့ အတွက် tutorial point က ပုံလေးတွေယူပြီး reference လုပ်လိုက်ပါတယ်။

## Left Rotation



## Right Rotation



## Left Right Rotation

State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes <b>C</b> an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of <b>C</b>. This makes <b>A</b>, the left subtree of <b>B</b>.</p>
	<p>Node <b>C</b> is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making <b>B</b> the new root node of this subtree. <b>C</b> now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

## Right Left Rotation

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes <b>A</b>, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along <b>C</b> node, making <b>C</b> the right subtree of its own left subtree <b>B</b>. Now, <b>B</b> becomes the right subtree of <b>A</b>.</p>
	<p>Node <b>A</b> is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making <b>B</b> the new root node of the subtree. <b>A</b> becomes the left subtree of its right subtree <b>B</b>.</p>
	<p>The tree is now balanced.</p>

Insert လုပ်တာ remove လုပ်တာတွေကတော့ binary tree အတိုင်းပါပဲ။ ပိုသွားတာက insert လုပ်တဲ့ အချိန်မှာ binary tree အတိုင်း insert လုပ်ပြီးရင် sub tree ရဲ့ balance factor ကိုပြန်စစ်ရတာပါပဲ။ balance factor မမှန်ရင် rotation လုပ်ပြီး balance ညှိရမှာမို့လိုပါ။

## References

ရေးထားတဲ့စာတွေအတွက် reference ယူထားတာက udemy ၏ courses တွေပါမယ်၊ tutorialpoints , geeksforgeeks အစရိတ်တဲ့ website တွေကနေ ယူထားပါတယ်။

photos တော်တော်များများကတော့ william fiset ရဲ့udemy course ကနေယူထားပါတယ်။

photos credit to udemy(william fiest), tutorialpoints, geeksforgeeks, w3school.in.

အခု series PDF လေးသည် ကျနော်ကိုယ်တိုင်လေ့လာရင်းနဲ့ သိလာတာတွေ ရေးထားတာ  
ဖြစ်တဲ့အတွက် လိုအပ်တာလေးတွေပါကောင်းပါနေနိုင်သေးပါတယ်။

Thanks for reading.