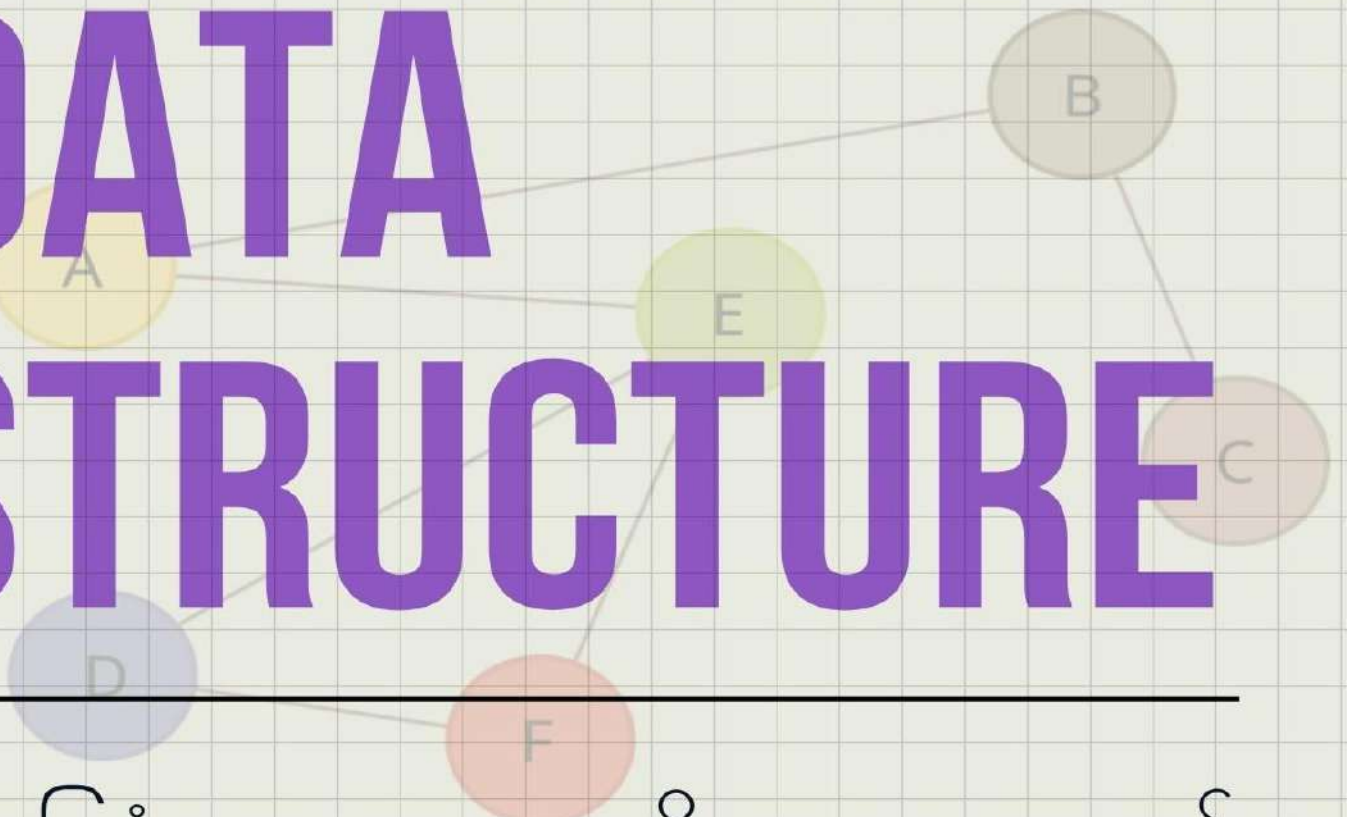
EXCLUSIVE EDITION

CODE with Thura

လွယ်ကူလေ့လာ

DATA STRUCTURE



အခြေခံမှ စလေ့လာလိုသူများအတွက်

CODE WITH THURA

လွယ်ကူလေ့လာ

Data Structure

အခြေခံမှ စ,လိုသူများအတွက် လက်စွဲ

Thura (Code with Thura)

လွယ်ကူလေ့လာ Data Structure

Translated Edition of “Absolute Beginner’s Guide to Algorithms” (Part 1)

By Kirupa Chinnathambi

Translated and Adapted by: Thura (Code with Thura)

Year: 2025

မာတိကာ

ဆက်သွယ်လိုပါက thura.00011@gmail.com သို့လည်းကောင်း၊ 'Code with Thura' ၏ တရားဝင် ဝက်ဘ်ဆိုဒ် စာမျက်နှာဖြစ်သည့် https://www.codewiththura.com/contact သို့လည်းကောင်း ဝင်ရောက် ဆက်သွယ် မေးမြန်းနိုင်ပါသည်။

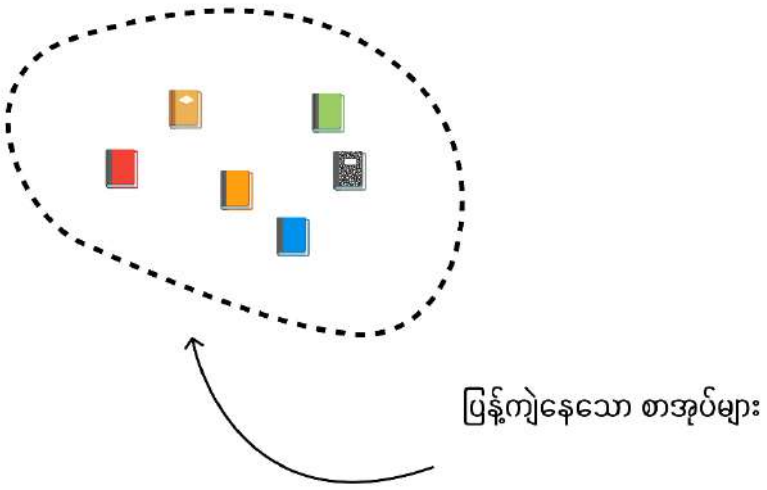

CODE
with Thura

အခန်း (၁) - Data Structure

Data Structure မိတ်ဆက်

Programming ဆိုတာ Data အချက်အလက်တွေကို လိုအပ်သလို ပြင်ဆင်ပြောင်းလဲနိုင်ဖို့ ကွန်ပျူတာကို ညွှန်ကြားပေးရတဲ့ လုပ်ငန်းစဉ်တစ်ခုလို့ အကြမ်းဖျင်း မှတ်ယူလို့ရပါတယ်။ အဲဒီလို လိုအပ်ချက်ပေါ်မူတည်ပြီး Data တွေကို အသက်သာဆုံးနဲ့ ထိထိရောက်ရောက် ပြင်ဆင်ပြောင်းလဲ သိမ်းဆည်းပေးနိုင်ဖို့ Data Structure သဘောတရားတွေကို အသုံးပြုရပါတယ်။

Data Structure ရဲ့ သဘောတရားကို နားလည်ဖို့အတွက် နမူနာတစ်ခုကို စဉ်းစားကြည့်ပါမယ်။ (ပုံ ၁-၁)။ ဆိုပါစို့၊ ကျွန်တော်တို့ဆီမှာ စာအုပ်တွေ အများကြီးရှိမယ်။ ဝတ္ထုတို၊ ဝတ္ထုရှည်၊ ဘာသာပြန်၊ ပုံနှိပ်စာအုပ် စသည်ဖြင့် ပါဝင်ပါမယ်။ လက်ရှိမှာ စာအုပ်တွေက နေရာအနှံ့ ပြန့်ကျဲနေမယ်၊ အဲဒီအတွက် စာအုပ်တွေကို တစ်နေရာတည်းမှာ သိမ်းဆည်းထားနိုင်ဖို့ စီစဉ်ပါမယ်။

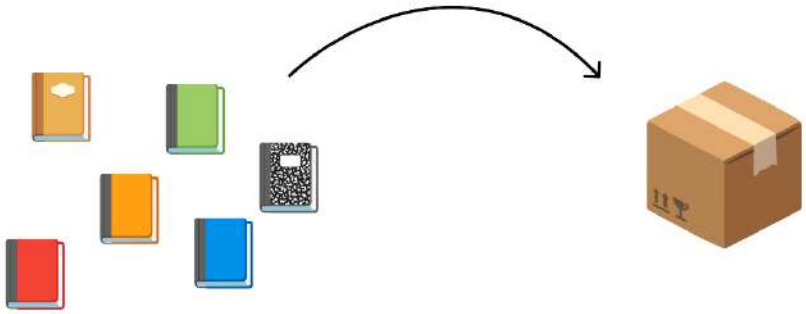


ပုံ ၁-၁၊ စာအုပ်တွေကို တစ်နေရာတည်းမှာ သိမ်းဆည်းပါမယ်

ပထမဆုံးအနေနဲ့ သေတ္တာတစ်လုံးကို အသုံးပြုပါမယ်။ စာအုပ်တွေအားလုံးကို သေတ္တာထဲမှာ စုပြုံပြီး ပစ်ထည့်၊ သိမ်းဆည်းထားလိုက်ပါမယ် (ပုံ ၁-၂ ကိုကြည့်ပါ)။ လတ်တလောမှာ အဆင်ပြေသွားပေမယ့် တချိန်ချိန်မှာ သေတ္တာထဲက စာအုပ်တစ်အုပ်ကို ပြန်ရှာလိုတဲ့အခါ အခက်တွေ့ပါမယ်။ ကျွန်တော်တို့ လိုချင်တဲ့စာအုပ်ကို မွှေနှောက်ပြီး ရှာဖွေရတော့မှာဖြစ်ပါတယ်။ တကယ်လို့ အဲဒီစာအုပ်ဟာ သေတ္တာရဲ့

အောက်ဆုံးမှာ ရောက်နေရင်တော့ ပိုပြီးခက်ခက်ခဲခဲ ရှာဖွေရတော့မှာပါ။ ကျွန်တော်တို့အတွက် အချိန်ပိုကုန်စေပါတယ်။ ဒီနည်းလမ်းဟာ သိပ်တော့အဆင်မပြေလှပါဘူး။

ဒီတော့ စာအုပ်တွေကို သိမ်းဆည်းဖို့အတွက် တခြားနည်းလမ်းတစ်ခုကို စဉ်းစားကြည့်ပါမယ်။



ပုံ ၁-၂။ သေတ္တာထဲမှာ စုပြုံပြီး ထည့်လိုက်ပါမယ်

စာအုပ်တွေကို စာအုပ်စင်လေးပေါ်မှာ စာအုပ်အမျိုးအစားအလိုက် အကန့်ခွဲပြီး သိမ်းဆည်းပါမယ်။ ကနဦး စစချင်းမှာတော့ စာအုပ်တွေကို အမျိုးအစားအလိုက် တစ်အုပ်ချင်း နေရာချရတာဖြစ်လို့ အချိန်ကုန်၊ လူပင်ပန်းစေနိုင်ပါတယ်။ ဒါပေမယ့် နောင်တစ်ချိန်မှာ လိုချင်တဲ့ စာအုပ်ကို စာအုပ်အမျိုးအစားအလိုက် အလွယ်တကူ ပြန်ရှာလို့ရသွားစေမှာ ဖြစ်ပါတယ်။ သေတ္တာထဲမှာ သိမ်းဆည်းသလိုမျိုး မတွေ့မချင်း မွှေနှောက်ရှာဖွေဖို့ မလိုအပ်တော့ပါဘူး။ ကျွန်တော်တို့အတွက် အချိန်ကုန်သက်သာစေမှာဖြစ်ပါတယ်။

အပေါ်မှာဖော်ပြခဲ့တဲ့ နည်းလမ်း ၂ ခုကို နှိုင်းယှဉ်သုံးသပ်ကြည့်ပါမယ်။

<u>သေတ္တာထဲမှာ စုပုံသိမ်းဆည်းခြင်း</u>

၁။ စာအုပ်အသစ်တွေကို ထပ်ပြီးပေါင်းထည့်ဖို့လိုအပ်လာတဲ့အခါ အင်မတန် လွယ်ကူပါတယ်။ သေတ္တာထဲမှာ ဒီတိုင်း ပစ်ထည့်လိုက်ရုံပါပဲ။ ထွေထွေထူးထူး လုပ်ဆောင်စရာမရှိပါဘူး။

၂။ စာအုပ်တွေကို ပြန်ရှာလိုတဲ့အခါ အချိန်တစ်ခုပေးရပါမယ်။ အပေါ်ဆုံးမှာ ရောက်နေတဲ့ စာအုပ်တွေဆိုရင် လွယ်လွယ်ကူကူ ရှာတွေ့နိုင်ပေမယ့် အောက်ဆုံးမှာ ရောက်နေတဲ့ စာအုပ်တွေဆိုရင်တော့ ပြန်ရှာတဲ့အခါ အချိန်ကုန်စေနိုင်ပါတယ်။

၃။ မလိုအပ်တော့တဲ့ စာအုပ်တွေကို ဖယ်ထုတ်ချင်တဲ့အခါမှာလည်း အချိန်ပေးရပါမယ်။ စာအုပ်တွေကို ပြန်ရှာသလိုပါပဲ၊ အပေါ်ဆုံးမှာရှိနေတဲ့ စာအုပ်တွေဆိုရင် လွယ်လွယ်ကူကူ ဖယ်ထုတ်နိုင်ပေမယ့် အောက်ဆုံးမှာ ရောက်နေတဲ့ စာအုပ်တွေဆိုရင်တော့ ပြန်ထုတ်ဖို့အတွက် လုံလောက်တဲ့အချိန်တစ်ခု ပေးရမှာဖြစ်ပါတယ်။

<u>စင်ပေါ်မှာ အမျိုးအစားလိုက် သိမ်းဆည်းခြင်း</u>

၁။ စာအုပ်အသစ်တွေကို ထပ်ပြီးပေါင်းထည့်ဖို့ လိုအပ်လာတဲ့အခါ အချိန်တစ်ခုပေးရပါမယ်။ စာအုပ်အမျိုးအစားအလိုက် သိမ်းဆည်းထားတာဖြစ်လို့ အမျိုးအစားအလိုက် သက်ဆိုင်ရာနေရာမှာ သိမ်းဆည်းပေးရမှာ ဖြစ်ပါတယ်။ အချိန်တစ်ခုကြာနိုင်ပါတယ်။

၂။ စာအုပ်တွေကို ပြန်ရှာလိုတဲ့အခါ လွယ်ကူ မြန်ဆန်ပါမယ်။ စာအုပ်အမျိုးအစားအလိုက် သက်ဆိုင်ရာ နေရာမှာ ရှာဖွေလိုက်ရုံပါပဲ။

၃။ မလိုအပ်တော့တဲ့ စာအုပ်တွေကို ဖယ်ထုတ်ချင်တဲ့အခါမှာလည်း လွယ်ကူမြန်ဆန်ပါမယ်။ စာအုပ်တွေကို ပြန်ရှာသလိုမျိုး စာအုပ်အမျိုးအစားအလိုက် သက်ဆိုင်ရာနေရာကနေ ရှာပြီး ဖယ်ထုတ်လိုက်ရုံပါပဲ။

ကျွန်တော်တို့ အပေါ်မှာ ဆွေးနွေးခဲ့တဲ့အတိုင်းပါပဲ၊ စာအုပ်တွေကို သိမ်းဆည်းတဲ့ နည်းလမ်း (၂) မျိုးလုံးမှာ သူ့ရဲ့ ကောင်းကျိုး၊ ဆိုးကျိုးတွေ ရှိပါတယ်။ ဘယ်နည်းလမ်းကို ရွေးချယ်မလဲဆိုတာက ကိုယ်ဘယ်လို အသုံးပြုမလဲဆိုတဲ့ အပေါ်မှာ မူတည်ပါတယ်။ ဥပမာ၊ စာအုပ်တွေကို မကြာခဏ ပြန်မရှာဘူးဆိုရင် သေတ္တာထဲ ထည့်သိမ်းတာက အသင့်တော်ဆုံးပါ။ ဒါပေမဲ့ မကြာခဏ ပြန်ထုတ်ဖတ်ဖို့ လိုအပ်မယ်ဆိုရင်တော့ စာအုပ်စင်ပေါ်မှာ တင်ထားတာက ပိုပြီး အဆင်ပြေပါတယ်။ ဒီလိုပဲ၊ ကိုယ့်ရဲ့ အခြေအနေနဲ့ လိုအပ်ချက်ပေါ် မူတည်ပြီး သင့်တော်တဲ့ နည်းလမ်းကို ရွေးချယ်အသုံးပြုရမှာ ဖြစ်ပါတယ်။

ဒီအချက်က ကွန်ပျူတာ ပရိုဂရမ်တွေ ရေးသားတဲ့အခါမှာလည်း အတူတူပါပဲ။ ပရိုဂရမ်ရဲ့ လိုအပ်ချက်ပေါ် မူတည်ပြီး မှန်ကန်တဲ့ Data Structure တွေကို ရွေးချယ်နိုင်ဖို့က အလွန်အရေးကြီးပါတယ်။ စာအုပ်သေတ္တာနဲ့ စာအုပ်စင် ဥပမာအတိုင်းပါပဲ၊ Data Structure တိုင်းမှာလည်း သူတို့ရဲ့ ကောင်းတဲ့အချက်တွေနဲ့ မကောင်းတဲ့အချက်တွေ ရှိကြပါတယ်။ ဘယ်အချိန်၊ ဘယ်နေရာမှာ ဘယ် Data Structure ကို အသုံးပြုသင့်တယ်ဆိုတာကို ဆုံးဖြတ်နိုင်ဖို့က Developer တစ်ယောက်မှာ မရှိမဖြစ် ရှိထားရမယ့် စွမ်းရည်တစ်ခုပဲ ဖြစ်ပါတယ်။

အသုံးများတဲ့ Data Structure တွေကို ဖော်ပြပေးပါမယ်။ Array, Linked List, Stack, Queue, Tree, Binary Tree, Binary Search Tree, Heap, Hash-Table (Hash-map သို့ Dictionary), Trie တို့ဖြစ်ပါတယ်။

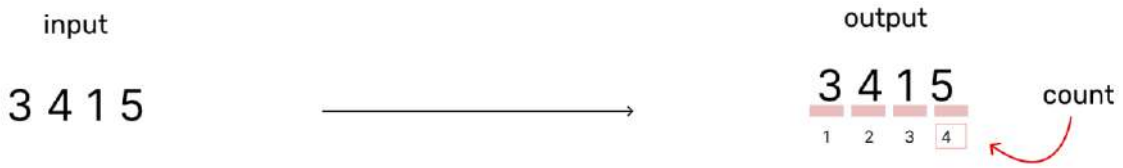
နောက်လာမယ့် အခန်းတွေမှာ ဒီ Data Structure တစ်ခုချင်းစီကို ပိုပြီး အသေးစိတ် လေ့လာသွားပါမယ်။ ဘယ်လို အခြေအနေမျိုးတွေမှာ ဘယ် Data Structure ကို သုံးသင့်တယ်၊ ဘယ်လို အခြေအနေမှာ အကောင်းဆုံးနဲ့ အသင့်တော်ဆုံးဖြစ်မလဲ စတာတွေကိုလည်း အသေးစိတ် ဆက်လက်ဆွေးနွေးသွားမှာ ဖြစ်ပါတယ်။

အခန်း (၂) - Big O Complexity

ကုဒ်တွေရဲ့ စွမ်းဆောင်ရည်ကို သုံးသပ်တဲ့အခါ ထည့်သွင်းစဉ်းစားရမယ့် အချက် ၂ ချက်ရှိပါတယ်။ Time Complexity (အချိန် ကန့်သတ်ဘောင်) နဲ့ Space Complexity (နေရာ ကန့်သတ်ဘောင်) ပါ။ လွယ်လွယ်ပြောရရင် Time Complexity ဆိုတာ ကုဒ်ကိုတွက်ချက်ဖို့ အချိန် ဘယ်လောက်ယူရသလဲကို ရည်ညွှန်းတာဖြစ်ပြီး၊ Space Complexity ကတော့ Memory ထပ်ဆောင်း ဘယ်လောက်များများ လိုအပ်သလဲဆိုတာကို ရည်ညွှန်းပါတယ်။ တိတိကျကျပြောရရင်တော့ Time Complexity ဆိုတာ ထည့်ပေးလိုက်တဲ့ Data အရွယ်အစားပေါ်မူတည်ပြီး Algorithm ရဲ့ တွက်ချက်ဖို့ လိုအပ်မယ့် အချိန်ပမာဏဖြစ်ပါတယ်။ အလားတူစွာပဲ၊ Space Complexity ဆိုတာ ထည့်ပေးလိုက်တဲ့ Data အရွယ်အစားပေါ်မူတည်ပြီး Algorithm ရဲ့ တွက်ချက်ဖို့ လိုအပ်မယ့် Memory ပမာဏဖြစ်ပါတယ်။

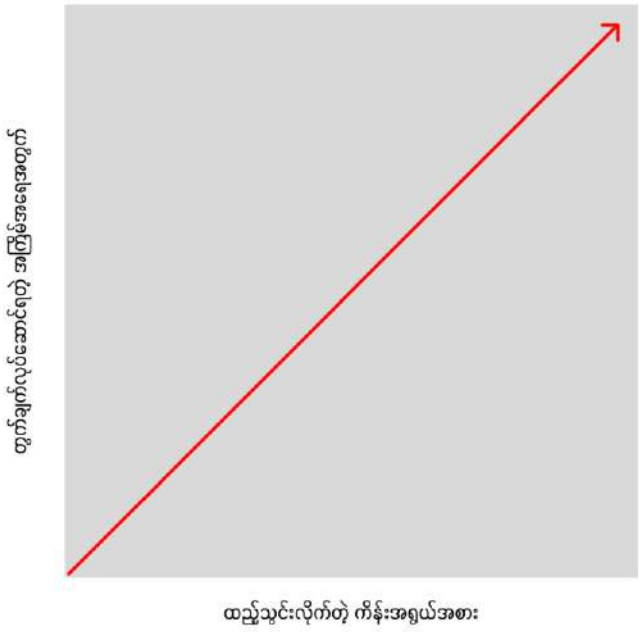
ကျွန်တော်တို့ဟာ ရေးသားထားတဲ့ကုဒ်တွေကို တတ်နိုင်သမျှ Memory ပမာဏ အနည်းဆုံးနဲ့မြန်မြန်ဆန်ဆန် အလုပ်လုပ်ဖို့သာ လိုချင်ကြတာဖြစ်ပါတယ်။ ဒါပေမယ့် လက်တွေ့မှာ ကျွန်တော်တို့ရဲ့ကုဒ်တွေဟာ ထည့်ပေးလိုက်ရတဲ့ Input Size တနည်းအားဖြင့် Data ပမာဏ၊ သွင်ပြင်လက္ခဏာပေါ်မူတည်ပြီး အလုပ်လုပ်ဆောင်နိုင်စွမ်းကွာခြားသွားပါတယ်။ Input ပမာဏတစ်စုံအတွက် ကုဒ်ရဲ့ Performance (စွမ်းဆောင်ရည်) ကို ကိုယ်ပိုင်နာရီတစ်လုံးစီနဲ့ သီးခြား တိုင်းတာလို့ရနိုင်ပေမဲ့၊ တကယ်တမ်း Input ပမာဏအားလုံးအတွက် ယေဘုယျကျကျ ဆုံးဖြတ်ပေးနိုင်မယ့် တိုင်းတာမှုမျိုး လိုအပ်ပါတယ်။ ဒီနေရာမှာ Big-O ပါဝင်လာရပါပြီ။

နမူနာကြည့်ရအောင်ပါ။ ကျွန်တော်တို့ရေးသားထားတဲ့ကုဒ်၊ တနည်းအားဖြင့် Function တစ်ခု သို့မဟုတ် Algorithm တစ်ခုက ကိန်းတစ်ခုကို Input အဖြစ် ရယူပြီး Digit (ဂဏန်း) ဘယ်နှစ်လုံးပါဝင်သလဲ တွက်ထုတ်ပေး နိုင်မယ်ဆိုပါစို့။ အကယ်၍ အဲဒီထဲကို 3415 ဆိုတဲ့ကိန်းကို ထည့်ပေးလိုက်ရင် ဂဏန်းအရေအတွက်က 4 လုံး ရရှိမှာပါ။ (ပုံ ၂-၁ ကို ကြည့်ပါ။)



ပုံ ၂-၁၊ ကိန်းမှာ ဂဏန်း (Digit) ဘယ်နှစ်လုံးပါဝင်သလဲ တွက်ထုတ်ခြင်း

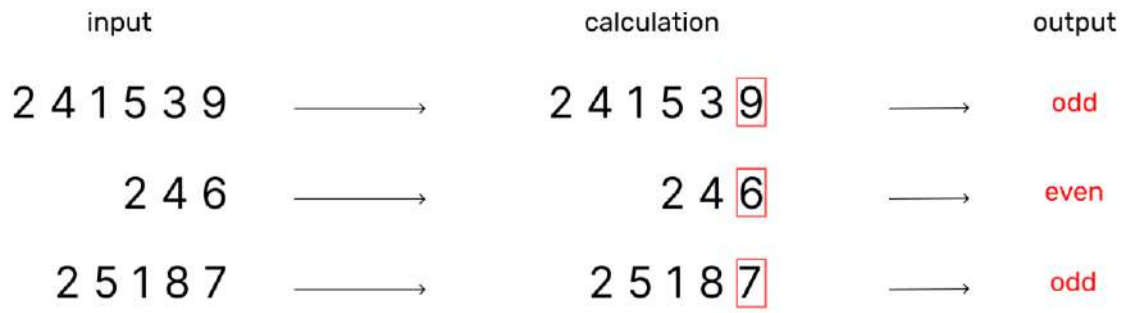
ထည့်သွင်းလိုက်တဲ့ကိန်းက 241,539 ဖြစ်ခဲ့ရင် ဂဏန်းအရေအတွက်ဟာ 6 လုံး ရရှိမှာပါ။ ဒီသဘောတရားကို လေ့လာလိုက်ရင် တွက်ချက်ပေးရတဲ့ အကြိမ်အရေအတွက်ဟာ ထည့်သွင်းလိုက်တဲ့ ကိန်းပမာဏနဲ့ တိုက်ရိုက်သက်ဆိုင်နေတာကို တွေ့ရမှာဖြစ်ပါတယ်။ ကိန်းပမာဏ အရွယ်အစားပေါ်လိုက်ပြီး ဂဏန်းအရေအတွက်ကို တွက်ထုတ်ရတာဖြစ်ပါတယ်။ တနည်းအားဖြင့် ကိန်းတန်ဖိုး ကြီးလာလေ၊ တွက်ချက်လုပ်ဆောင်ရတဲ့ အကြိမ်ရေ ပိုများလာလေဖြစ်ပါတယ်။ ထည့်သွင်းလိုက်တဲ့ ကိန်းအရွယ်အစားနဲ့ တွက်ချက်လုပ်ဆောင်ရတဲ့ အကြိမ်အရေအတွက်ကို ဂရပ်တစ်ခုအဖြစ် ရေးဆွဲလိုက်ရင် Linear Growth (မျဉ်းဖြောင့်အတိုင်း ၄၅ ဒီဂရီထောင်တတ်သွားတဲ့ပုံစံ) ကိုမြင်တွေ့ရမှာဖြစ်ပါတယ်။ (ပုံ ၂-၂ ကို ကြည့်ပါ။)



ပုံ ၂-၂။ Linear Growth မှာ တွက်ချက်ရတဲ့အကြိမ်အရေတွက်ဟာ ထည့်သွင်းလိုက်တဲ့ကိန်းပမာဏနဲ့ တိုက်ရိုက်သက်ဆိုင်ပါတယ်

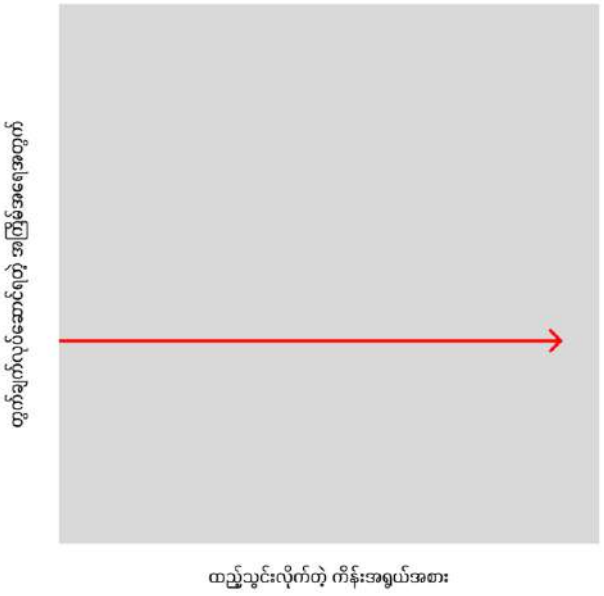
နောက်ထပ် နမူနာကုဒ်တစ်ခုကို ထပ်စဉ်းစားပါမယ်။ ကျွန်တော်တို့ရဲ့ကုဒ်က ကိန်းတစ်ခုကို Input အဖြစ် ရယူပြီး စုံကိန်း၊ မကိန်း ခွဲခြားပေးနိုင်တယ်ဆိုပါစို့။ ကိန်းတစ်ခုဟာ စုံကိန်းဟုတ်မဟုတ်၊ မကိန်းဟုတ်မဟုတ် ခွဲခြားနိုင်ဖို့ လုပ်ရမယ့်အလုပ်က အဲဒီကိန်းရဲ့ Last Digit (နောက်ဆုံး ဂဏန်း) ကို ကြည့်ပြီး တွက်ချက်မှုအနည်းငယ် ပြုလုပ်လိုက်ရုံပါပဲ။ (ပုံ ၂-၃ ကိုကြည့်ပါ။)

ဒီနေရာမှာဆိုရင် ထည့်ပေးလိုက်တဲ့ကိန်းက ဘယ်လောက်ကြီးတယ်ဆိုတာ အရေးမပါတော့ပါဘူး။ ကျွန်တော်တို့လုပ်ရတဲ့ အလုပ်က ကိန်းရဲ့ Last Digit လေးကိုပဲ စစ်ဆေးပြီး တွက်ချက်ဆုံးဖြတ်လို့ရနိုင်ပါတယ်။ တွက်ချက်မှုဟာ ကိန်းရဲ့အရွယ်အစားပေါ် မမူတည်တော့ပါဘူး။



ပုံ ၂-၃၊ ကိန်းကို စုံကိန်း၊ မကိန်း ခွဲခြား တွက်ထုတ်ခြင်း

ဆိုတော့ ဒီတွက်ချက်မှုဟာ မပြောင်းလဲတဲ့တွက်ချက်မှုအကြိမ်အရေတွက် လိုအပ်တယ်လို့ မှတ်ယူနိုင်ပါတယ်။ ဂရုပ်တစ်ခုအဖြစ် ရေးဆွဲလိုက်ရင် အလျားလိုက် ရေပြင်ညီ မျဉ်းဖြောင့်တစ်ခုကို မြင်တွေ့ရမှာဖြစ်ပါတယ်။ (ပုံ ၂-၄)။



ပုံ ၂-၄၊ တွက်ချက်ရတဲ့အကြိမ်အရေတွက်ဟာ တသမတ်တည်း မပြောင်းမလဲဖြစ်နေပါတယ်

Big-O သင်္ကေတ မိတ်ဆက်

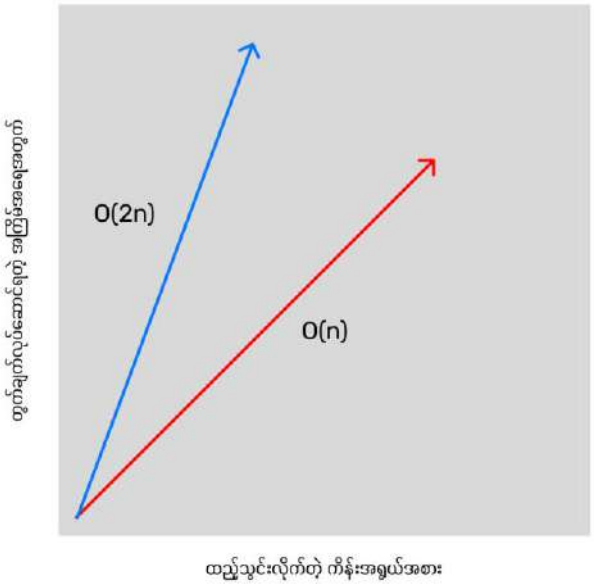
Big-O သင်္ကေတဆိုတာ မတူညီတဲ့ Input Size တွေအတွက် Upper Bound သို့ Worst-Case Scenario လို့ခေါ်တဲ့ အဆိုးဆုံးအခြေအနေမှာ ကုဒ်တွေဟာ ဘယ်လောက်အထိ တွက်ချက်နေရာယူရသလဲဆိုတာ ဖော်ပြပေးနိုင်တဲ့ သချိ်္ဂာနည်းဆိုင်ရာ ဖော်ပြချက် ဖြစ်ပါတယ်။ Big-O သင်္ကေတဟာ Algorithm ရဲ့ Performance ကို သက်ရောက်နိုင်မယ့် တကယ်အရေးပါတဲ့ Factor (အကြောင်းတရား) တွေပေါ်မှာပဲ တွက်ချက်ထားတာဖြစ်ပါတယ်။ Big-O သင်္ကေတကို စလေ့လာချင်းမှာ နားလည်ဖို့ ခက်ခဲတယ်လို့ ထင်ကောင်းထင်နိုင်ပါတယ်။ အတွေ့များ၊ အသုံးများလာတဲ့အခါ ရင်းနှီးကျွမ်းဝင်လာပါလိမ့်မယ်။

အပေါ်မှာ ပြောခဲ့တဲ့ Linear Growth အခြေအနေကို ဖော်ပြမယ့် Big O သင်္ကေတဟာ $O(n)$ ဖြစ်ပါတယ်။ Constant Growth ကို ဖော်ပြမယ့် သင်္ကေတက $O(1)$ ဖြစ်ပါတယ်။ Big-O ရဲ့ O ကို “Order of” လို့ခေါ်ပါတယ်။ Algorithm ရဲ့ Growth Rate ကို ရည်ညွှန်းပါတယ်။ တွက်ချက်ဖို့ ဘယ်လောက် အချိန်ကြာနိုင်သလဲ၊ Memory ဘယ်လောက်ယူသလဲ စသည်ဖြင့်ပါ။ n က တနည်းအားဖြင့် Argument ပါ၊ အဆိုးဆုံးအခြေအနေမှာ တွက်ချက်လုပ်ဆောင်ရမယ့် အကြိမ်အရေအတွက်ကို ကိုယ်စားပြုပါတယ်။

ဥပမာ၊ Big-O သင်္ကေတ $O(n)$ ဖြစ်တယ်ဆိုရင် ဆိုလိုတာက ကျွန်တော်တို့ကုဒ်ရဲ့ Time သို့မဟုတ် Space လိုအပ်ချက်ဟာ Input အရွယ်အစားပေါ်မူတည်ပြီး Linear Growth (မျဉ်းဖြောင့်အတိုင်း ထောင်တက်သွားတယ်)လို့ ဆိုလိုတာဖြစ်ပါတယ်။ Input Size သာ နှစ်ဆတိုးသွားခဲ့ရင် Time သို့ Space လိုအပ်ချက်ဟာလည်း နှစ်ဆတိုးသွားမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလိုပဲ Big-O သင်္ကေတက $O(n^2)$ ဖြစ်တယ်ဆိုရင် Algorithm ရဲ့ Time သို့မဟုတ် Space လိုအပ်ချက်ဟာ Input Size ပေါ်မူတည်ပြီး နှစ်ထပ်ကိန်းတန်ဖိုးတိုးတိုးလာမှာဖြစ်ပါတယ်။ Input Size နှစ်ဆတိုးသွားခဲ့ရင် Time သို့ Space လိုအပ်ချက်ဟာ လေးဆတိုးသွားမှာ ဖြစ်ပါတယ်။ တကယ်တော့ နှစ်ထပ်ကိန်းတန်ဖိုး တိုးသွားတာဟာ အဆိုးဝါးဆုံး အခြေအနေမဟုတ်သေးပါဘူး။ ပိုပြီး ဆိုးဝါးတဲ့ အခြေအနေတွေ ရှိပါသေးတယ်။

အပေါ်မှာ ပြောခဲ့သလို Big-O သင်္ကေတဟာ Algorithm ရဲ့ Performance ကို သက်ရောက်နိုင်မယ့် တကယ်တမ်း အရေးပါတဲ့ Factor တွေပေါ်မှာပဲ တွက်ချက်ထားတာဖြစ်ပါတယ်။ ဥပမာအနေနဲ့ Linear Case ကို ကြည့်မယ်ဆိုရင် n အတွက် ထပ်ပေါင်းတန်ဖိုးက $O(2n)$ ဖြစ်နေသည်ဖြစ်စေ၊ $O(n+5)$ ဖြစ်နေသည်ဖြစ်စေ၊ ဒါမှမဟုတ် $O(4n-n/2)$ ဖြစ်နေသည်ဖြစ်စေ အရေးမကြီးပါဘူး။ တကယ်တမ်း $O(n)$ အနေနဲ့ပဲ အရိုးရှင်းဆုံးယူဆသွားမှာပါ။ နမူနာ ဂရပ် ပုံ ၂-၅ မှာ ကြည့်ပါ။ $O(2n)$ က $O(n)$ ထက် အလုပ်ပိုလုပ်ရမယ်လို့ ထင်ရပေမဲ့ Input Size အရမ်းကြီးမားလာတဲ့အခါ ဒီခြားနာမှုဟာ ထည့်သွင်းစဉ်းစားဖို့ မလိုသလောက်၊ တနည်းအားဖြင့် အရေးမပါသလောက် ဖြစ်သွားမှာဖြစ်ပါတယ်။ ကျွန်တော်တို့ ကြားဖူးနေကြ စကားတစ်ခုရှိပါတယ်။

“အသေးအမွှားကိစ္စတွေကို ခေါင်းစားမနေနဲ့၊ အရေးပါတဲ့ အရာတွေကိုပဲ အာရုံစိုက်ပါ” ဆိုတဲ့အတိုင်း Big-O ကလဲ အလားတူ ပြုမူတာပဲ ဖြစ်ပါတယ်။



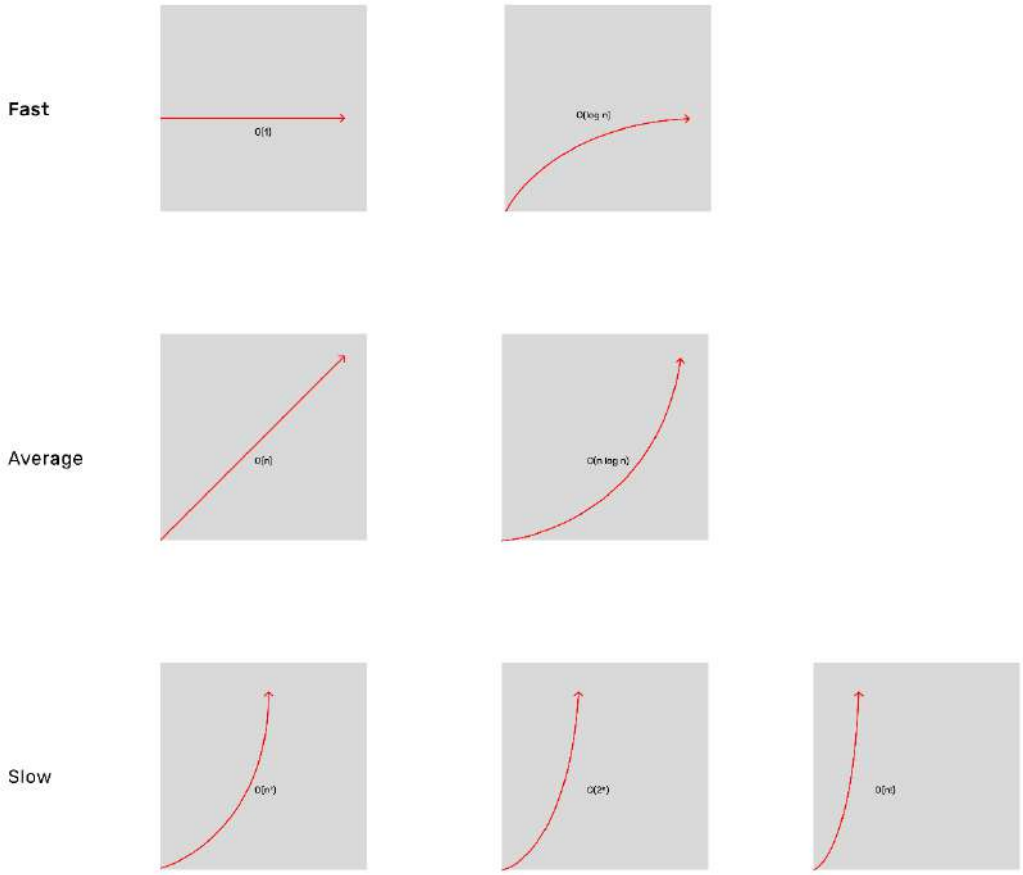
ပုံ ၂-၅၊ *O(2n) vs O(n)*

Big-O သင်္ကေတတွေနဲ့ သူတို့ရဲ့ အဓိပ္ပာယ်ကို ဖော်ပြပေးပါမယ်။

O(1) – Constant Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့မဟုတ် Space ဟာ ကိန်းသေပမာဏအဖြစ် မပြောင်းမလဲရှိနေမှာကိုဆိုလိုတာဖြစ်ပါတယ်။ ဆိုလိုတာက Input Size ဘယ်လောက်ပဲကြီးလာပါစေ တွက်ချက်ရတဲ့ Complexity က မပြောင်းလဲဘဲ တစ်သမတ်တည်း ရှိနေမှာဖြစ်ပါတယ်။ ဥပမာ၊ Array ရဲ့ Element တစ်ခုကို Index ထောက်ပြီး ခေါ်ယူတဲ့အခြေအနေမျိုးတွေ၊ အပေါ်မှာဖော်ပြခဲ့တဲ့ စုံကိန်း၊ မကိန်း တွက်ချက်တဲ့အခြေအနေတွေဖြစ်ပါတယ်။

O(log n) – Logarithmic Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့မဟုတ် Space ဟာ သေးငယ်တဲ့ပမာဏလောက်သာ ပြောင်းလဲသွားမှာကိုဆိုလိုတာဖြစ်ပါတယ်။ ဆိုလိုတာက Input Size နှစ်ဆတိုးသွားရင် တွက်ချက်ရတဲ့အကြိမ်အရေအတွက် Complexity က အနည်းငယ်လောက်သာ တိုးလာမှာဖြစ်ပါတယ်။ ဥပမာ၊ Binary Search လိုမျိုးဖြစ်ပါတယ်။

O(n) – Linear Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့မဟုတ် Space ဟာ Linear Growth မျဉ်းဖြောင့်အတိုင်း ထောင်တတ်ပြောင်းလဲသွားမှာကို ဆိုလိုတာဖြစ်ပါတယ်။ ဆိုလိုတာက Input Size နှစ်ဆတိုးသွားရင် တွက်ချက်ရတဲ့အကြိမ်အရေအတွက် Complexity ကလည်း နှစ်ဆတိုက်ရိုက်တိုးလာမှာဖြစ်ပါတယ်။ ဥပမာ၊ Array တစ်ခုပေါ်မှာ Iterating လုပ်တဲ့အခါမျိုးတွေဖြစ်ပါတယ်။



ပုံ ၂-၆၊ Big O Notation Graph များ

O(n log n) - Linearithmic Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့ Space ဟာ Linear Growth Rate ထက် အနည်းငယ် ပိုမြန်ပြီး ပြောင်းလဲသွားမှာကို ဆိုလိုတာဖြစ်ပါတယ်။ Merge-sort တို့၊ Quick-sort တို့လို Sorting Algorithm တွေမှာ တွေ့ရတတ်ပါတယ်။

$O(n^2)$ - Quadratic Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့ Space ဟာ နှစ်ထပ်ကိန်းတန်ဖိုး တိုးတိုးလာမှာကို ဆိုလိုတာဖြစ်ပါတယ်။ Bubble-sort တို့၊ Selection Sort တို့လို Algorithm တွေမှာ တွေ့ရတတ်ပါတယ်။

$O(2^n)$ - Exponential Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့ Space ဟာ ထပ်ကိန်းတန်ဖိုး တိုးတိုးလာမှာကို ဆိုလိုတာဖြစ်ပါတယ်။ ဒီအခြေအနေဟာ ပုံမှန်အားဖြင့် အဆိုးဝါးဆုံး Inefficient မဖြစ်တဲ့အခြေအနေအဖြစ် မှတ်ယူနိုင်ပါတယ်။

$O(n!)$ - Factorial Complexity: ဒီသင်္ကေတဟာ Input Size ပေါ်မူတည်ပြီး Time သို့ Space ဟာ Factorial တန်ဖိုး တိုးတိုးလာမှာကို ဆိုလိုတာဖြစ်ပါတယ်။ Input Size နှစ်ဆတိုးသွားခဲ့ရင် Time သို့ Space လိုအပ်ချက်ဟာ ကြောက်ခမန်းလိလိ တိုးလာမှာ ဖြစ်ပါတယ်။ ဒါဟာ အဆိုးဝါးဆုံးအခြေအနေလို့ မှတ်ယူလို့ရပါတယ်။

အခန်း (၃) – Array များ

ဒေတာဖွဲ့စည်းပုံတွေကို လေ့လာတဲ့အခါမှာ အခြေခံအကျဆုံးဖြစ်တဲ့ Array ကို စတင်လေ့လာကြရပါမယ်။ Array တွေဟာ တခြားဒေတာဖွဲ့စည်းပုံတွေမှာ ထည့်သွင်း အသုံးပြုကြတဲ့အထိ အရေးပါတဲ့ အခန်းကဏ္ဍမှာ ရှိပါတယ်။ အခန်း ၁ မှာ တင်ပြခဲ့တဲ့ စာအုပ်သေတ္တာ ဥပမာကို ပြန်စဉ်းစားကြည့်ပါ။ Array ရဲ့ ဖွဲ့စည်းပုံကို အဲဒီသေတ္တာထဲက အခန်းငယ်လေးတွေအဖြစ် မြင်ယောင်ကြည့်လို့ရပါတယ်။

ဒီအခန်းမှာ Array ဆိုတာဘာလဲ၊ ဘာကြောင့် ဒီလောက်လူသုံးများရတာလဲ၊ ဘယ်လို အခြေအနေတွေမှာ သုံးသင့်တာလဲ စသည်ဖြင့် အသေးစိတ် လေ့လာသွားပါမယ်။

Array ဆိုတာဘာလဲ

စာရွက်တစ်ရွက်ပေါ်မှာ “ဈေးဝယ်စာရင်း” တစ်ခုကို ရေးကြည့်ပါ။ ဝယ်ယူရမယ့် ပစ္စည်းတွေကို သုညကနေ စပြီး အစဉ်လိုက် နံပါတ်တပ်ထားတဲ့ စာရင်းပုံစံမျိုးပါ။ (ပုံ ၃-၁)။

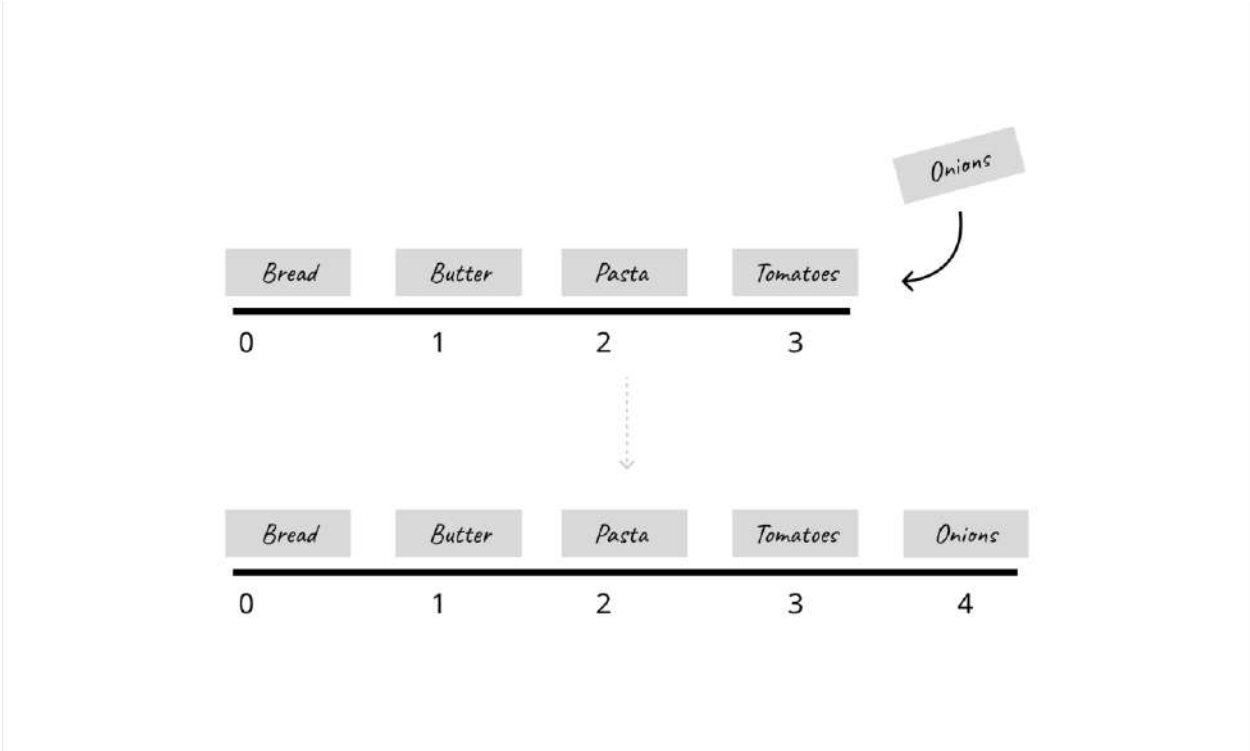
1. Bread
2. Butter
3. Pasta
4. Tomatoes

ပုံ ၃-၁၊ ဈေးဝယ်စာရင်း

ဒီဈေးဝယ်စာရင်းကို ဒစ်ဂျစ်တယ်ပုံစံနဲ့ ဖော်ပြဖို့ဆိုရင် Array တွေကို အသုံးပြုရပါမယ်။ Array ဆိုတာ ဒေတာတွေကို နံပါတ်စဉ်အလိုက် သိမ်းဆည်းဖို့ တည်ဆောက်ထားတဲ့ ဒေတာဖွဲ့စည်းပုံတစ်ခုပါ။ နမူနာ ဈေးဝယ်စာရင်းကို Array အဖြစ် ပြောင်းလဲလိုက်ရင် ပုံ ၃-၂ မှာ ပြထားတဲ့အတိုင်း ဖြစ်သွားပါလိမ့်မယ်။

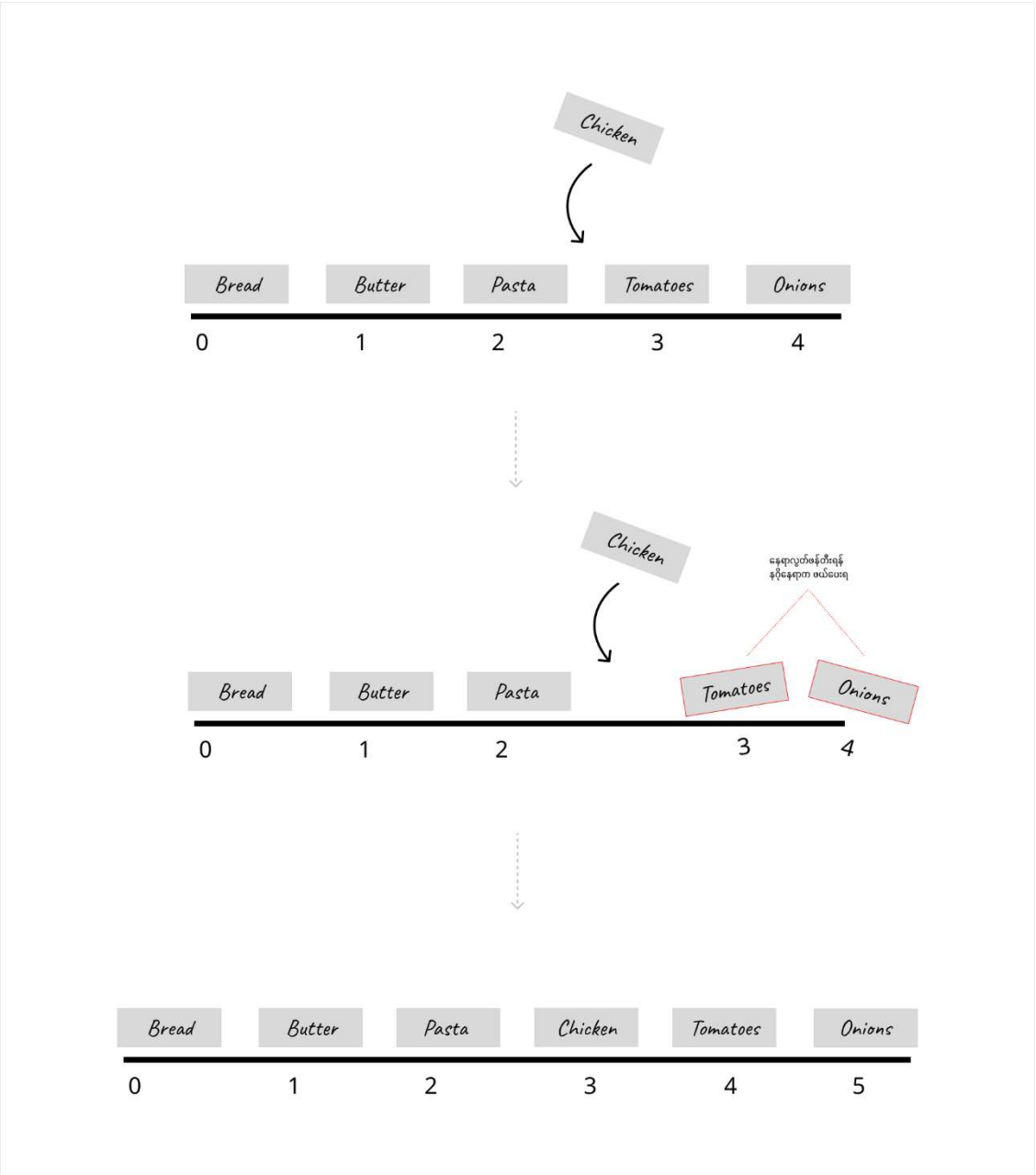
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Item အသစ်ကို Array ရဲ့ နောက်ဆုံးအခန်းမှာ ထည့်သွင်းရတဲ့ လုပ်ငန်းစဉ်က အလွယ်ကူဆုံးနဲ့ အရိုးရှင်းဆုံး ဖြစ်ပါတယ်။ လွယ်လွယ်ပြောရရင် ထည့်သွင်းလိုက်တဲ့ Item အသစ်ကို သက်ဆိုင်ရာ အခန်းနံပါတ် တနည်းအားဖြင့် Index တန်ဖိုး သတ်မှတ်ပေးလိုက်ရုံပါပဲ။ တခြား ဘာမှ ထွေထွေထူးထူး လုပ်ဆောင်စရာ မလိုပါဘူး။ လုပ်အားပမာဏနည်းပြီး လွယ်ကူရိုးရှင်းတဲ့ လုပ်ငန်းစဉ်ဖြစ်ပါတယ်။ (ပုံ ၃-၃ ကို ကြည့်ပါ။)



ပုံ ၃-၃၊ Array ထဲကို Item အသစ်ထည့်သွင်းခြင်း

Array ရဲ့ အလယ်ခန်း (သို့မဟုတ်) ပထမအခန်းမှာ Item အသစ်တစ်ခု ထည့်သွင်းမယ်ဆိုရင်တော့ ထင်သလောက် မလွယ်ကူတော့ပါဘူး။ ပထမဆုံးလုပ်ရမယ့် အလုပ်က နေရာလွတ် ဖန်တီးပေးရမှာပါ။ (ပုံ ၃-၄ ကိုကြည့်ပါ။)



ပုံ ၃-၄၊ Item အသစ်အတွက် နေရာလွတ်ဖန်တီးခြင်း

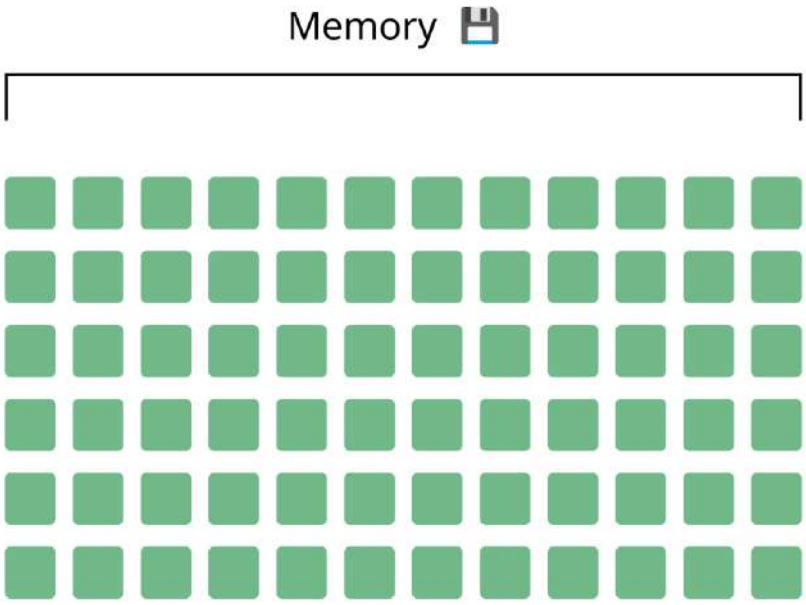
နေရာလွတ်ဖန်တီးတယ်ဆိုတာ Item အသစ်ကို နေရာချပေးဖို့အတွက် Array ထဲမှာ ရှိနေပြီးသား Item တစ်ခုချင်းစီကို နေရာပြောင်းရွှေ့ပေးရတာပါ။ နေရာပြောင်းရွှေ့ရတဲ့အတွက် သက်ဆိုင်ရာ Index တန်ဖိုးတွေကိုလည်း ပြန်တွက်ချက်ပေးရပါမယ်။ ရွှေ့ရတဲ့ Item တွေ များလေလေ၊ လုပ်ဆောင်မယ့် အချိန်နဲ့ စွမ်းအား ပိုမိုသုံးစွဲရလေလေ ဖြစ်ပါတယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Array နဲ့ Memory အကြောင်း

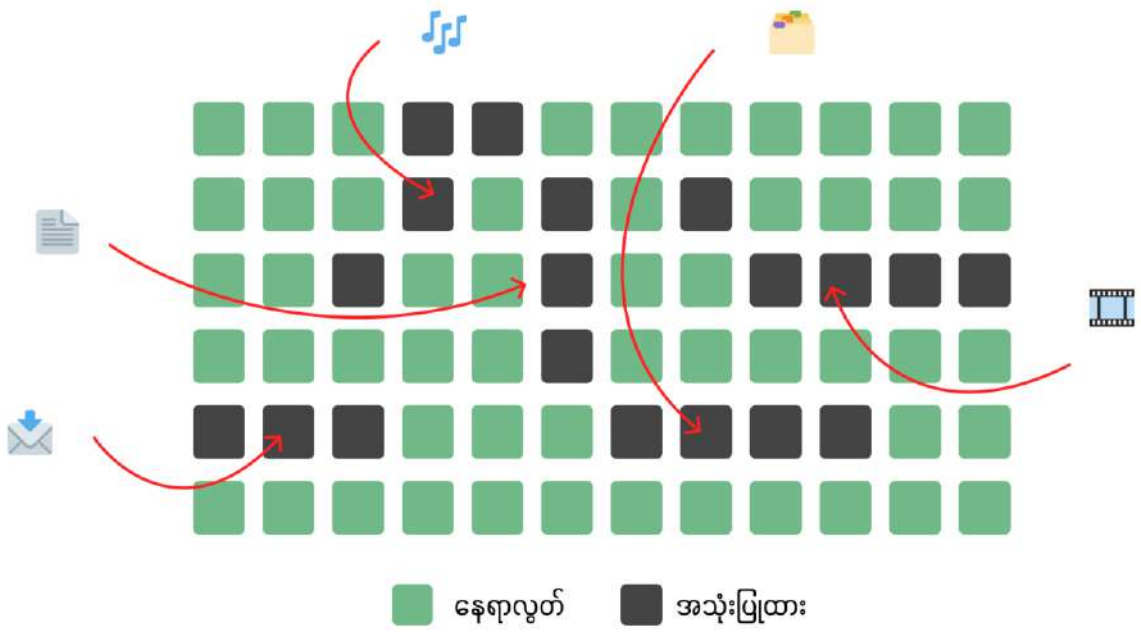
JavaScript လိုမျိုး High-Level Programming Language တွေကို အသုံးပြုတဲ့အခါ Memory အပိုင်းကို ကျွန်တော်တို့ ကိုယ်တိုင် စီမံခန့်ခွဲဖို့ မလိုအပ်ပါဘူး။ System က အလိုအလျောက် လုပ်ဆောင်ပေးသွားပါတယ်။ ဒီအပိုင်းမှာ Array တွေကို အတွင်းကျကျ လေ့လာနိုင်ဖို့အတွက် ကွန်ပျူတာရဲ့နောက်ကွယ်က Memory လုပ်ငန်းစဉ်တချို့ကို လေ့လာသွားပါမယ်။

Memory ကို နားလည်လွယ်အောင် ပြောရရင်၊ Data အချက်အလက်တွေကို သိမ်းဆည်းထားနိုင်တဲ့ နေရာအပိုင်းအစလေးတွေလို့ မြင်ယောင်ကြည့်လို့ရပါတယ်။ (ပုံ ၃-၁၀ ကိုကြည့်ပါ။)



ပုံ ၃-၁၀၊ ကွန်ပျူတာရဲ့ Memory နေရာများ

တကယ်တမ်း လက်တွေ့မှာတော့ ကျွန်တော်တို့ရဲ့ Memory ဟာ အခုလိုမျိုး ရှင်းလင်းနေမှာ မဟုတ်ပါဘူး။ ကွန်ပျူတာက တခြားအလုပ်တွေ အများကြီးကို တစ်ချိန်တည်း လုပ်ဆောင်နေရတာကြောင့် နေရာယူထားတဲ့ အပိုင်းတွေလည်း အများအပြား ရှိနေမှာပါ။ (ပုံ ၃-၁၁ လိုမျိုးပါ။)



ပုံ ၃-၁၁၊ အသုံးပြုထားသော Memory နေရာများ

Memory ထဲကို Item အသစ်တွေ ထည့်မယ်ဆိုရင်၊ လွတ်နေတဲ့ နေရာတွေထဲမှာပဲ ထည့်ရပါတယ်။ ဒီအချက်ဟာ Array တွေနဲ့ ပြန်ဆက်စပ်ပါတယ်။ Array တစ်ခု ဖန်တီးတဲ့အခါ ကျွန်တော်တို့ အရင်ဆုံး လုပ်ရတာက Memory ထဲမှာ Allocate (နေရာသတ်မှတ်) ပေးရတာပါ။ ဘာကြောင့်လဲဆိုတော့ Array တွေရဲ့ သဘောအရ Item တွေကို Memory ထဲက ကပ်လျက် တစ်ဆက်တည်း ရှိနေတဲ့ နေရာတွေမှာပဲ သိမ်းဆည်းရတာကြောင့်ဖြစ်ပါတယ်။

Array တစ်ခုကို စဖန်တီးတဲ့အခါ လိုအပ်မယ့် အရွယ်ပမာဏတစ်ခုကို Memory ထဲမှာ ကြိုတင်သတ်မှတ်ပေးရပါတယ်။ ဥပမာ၊ ကျွန်တော်တို့ Array တစ်ခု ဖန်တီးတယ်ဆိုပါစို့။ လတ်တလော ကျွန်တော်တို့ Array က အလွတ်ဖြစ်နေပေမယ့်လည်း၊ လိုအပ်မယ့် အရွယ်အစား ပမာဏတစ်ခုကို Memory ထဲမှာ ကြိုတင်ရယူထားတာမျိုးပါ။ (ပုံ ၃-၁၂ ကိုကြည့်ပါ။)

ကျွန်တော်တို့ရဲ့ ဥပမာမှာ Memory နေရာ ၇ ခု သတ်မှတ်ထားပါတယ်။ Array ထဲကို Item တွေ ထပ်ထည့်လာတာနဲ့အမျှ သတ်မှတ်ထားတဲ့ Memory နေရာတွေက ဖြည်းဖြည်းချင်း ပြည့်လာမှာပါ။ (ပုံ ၃-၁၃ လိုမျိုးပါ။)

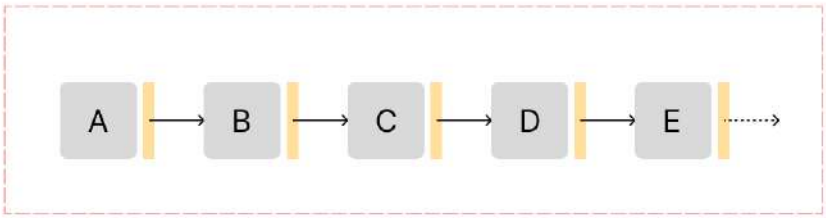
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၄) - Linked List များ

မကြာခဏ ပြောင်းလဲနေနိုင်တဲ့ ဒေတာအချက်အလက်တွေ ကိုင်တွယ်ဖြေရှင်းတဲ့အခါ Linked List တွေကို အသုံးပြုနိုင်ပါတယ်။ အခြား Data Structure တွေနဲ့ မတူတာက Linked List မှာ အချက်အလက်တစ်ခုစီကို Node လို့ခေါ်တဲ့ အရာလေးတွေနဲ့ ချိတ်ဆက်ထားတာပါ။ တစ်နည်းအားဖြင့် ရထားတွဲတွေလို တစ်ခုနဲ့တစ်ခု ဆက်စပ်နေတယ်လို့ ပြောလို့ရပါတယ်။ ဒီအခန်းမှာ Linked List တွေရဲ့ အခြေခံဖွဲ့စည်းပုံ၊ ယေဘုယျလက္ခဏာတွေကို အသေးစိတ်လေ့လာသွားပါမယ်။

Linked List ကို စတင် ထိတွေ့ခြင်း

Linked List တွေကို Array တွေနည်းတူ ဒေတာအစုအဝေးတွေ သိမ်းဆည်းဖို့အတွက် အသုံးပြုနိုင်ပါတယ်။ ပုံ ၄-၁ မှာ ဆိုရင် A ကနေ E အထိ စကားလုံးတွေကို သိမ်းဆည်းထားတဲ့ Linked List



ပုံ ၄-၁၊ Linked List

တစ်ခုရဲ့ ဥပမာကို တွေ့ရမှာပါ။

Linked List တွေက တစ်ခုနဲ့တစ်ခု ချိတ်ဆက်ထားတဲ့ Node တွေကို အခြေခံပြီး အလုပ်လုပ်ပါတယ်။

Node တစ်ခုစီမှာ အပိုင်း ၂ ပိုင်းပါဝင်ပါတယ်။

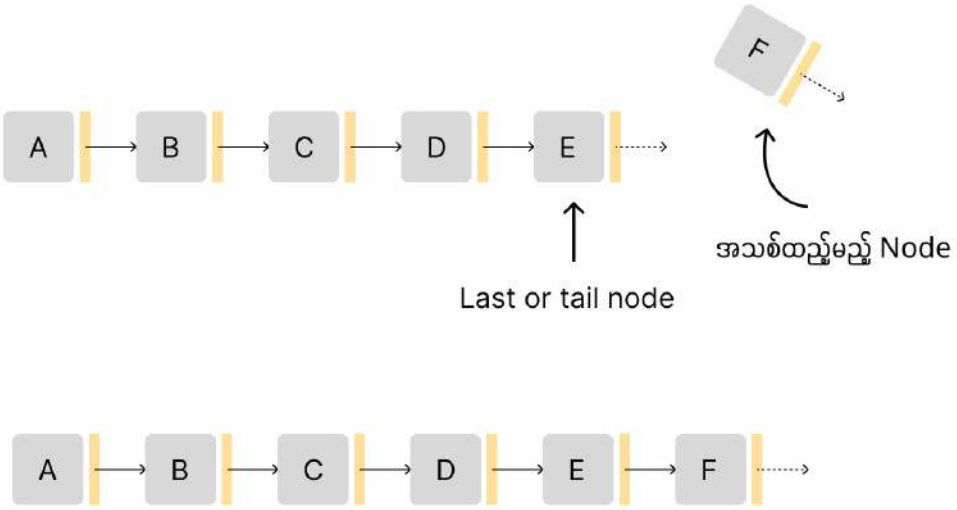
- သိမ်းဆည်းထားတဲ့ ဒေတာ၊
- နောက် Node တစ်ခုကို ညွှန်ပြတဲ့ Next Pointer (တနည်းအားဖြင့် Reference) တို့ဖြစ်ပါတယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Node များ ထည့်သွင်းခြင်း

Linked List ထဲကို Node တွေ ဘယ်လိုမျိုးထည့်သွင်းရလဲဆိုတာကို ဆက်လေ့လာပါမယ်။ Node တွေထည့်သွင်းတယ်ဆိုတာထက်၊ Node အသစ်တစ်ခု ဖန်တီးပြီး Next Pointer တချို့ကို ပြင်ဆင်ရတာမျိုးပါ။ ဥပမာကို ကြည့်ပါမယ်။

F Node ကို အဆုံးမှာထည့်ချင်တယ်ဆိုပါစို့။ ကျွန်တော်တို့လုပ်ရမှာက နောက်ဆုံး E Node ရဲ့ Next Pointer ကို ထည့်မယ့် F Node ဆီညွှန်ပြအောင် Update လုပ်ပေးဖို့ပါပဲ။ (ပုံ ၄-၄)။



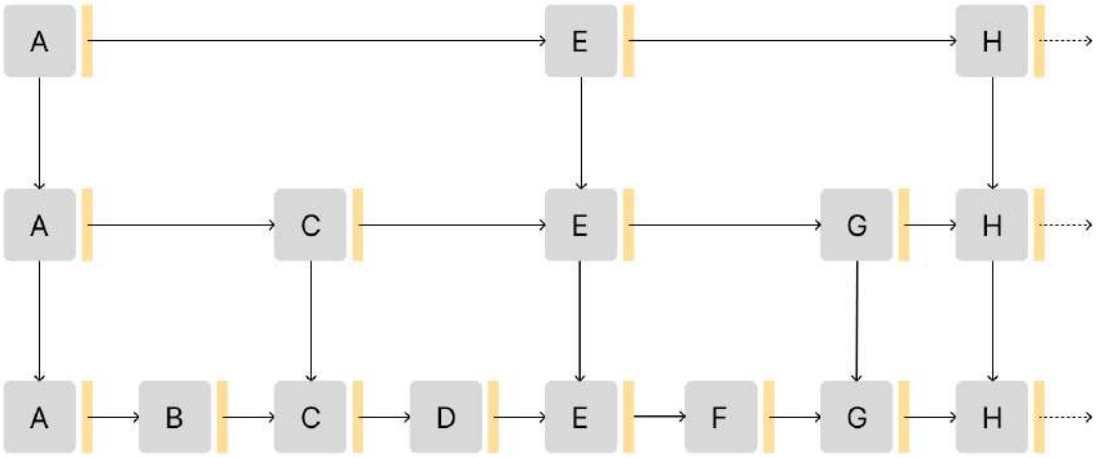
ပုံ ၄-၄၊ အဆုံးမှာ Node အသစ်ထည့်ခြင်း

ကျွန်တော်တို့ ဘယ်နေရာမှာ Node အသစ်ကို ထည့်ထည့်၊ လုပ်ဆောင်ပုံကတော့ အတူတူပါပဲ။ C နဲ့ D Node တွေကြားမှာ Q Node အသစ်ကို ထည့်ချင်တယ်ဆိုပါစို့။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Skip List

Skip list တွေက Linked List တွေထက် ပိုမြန်ပါတယ်။ Skip List ဆိုတာ Linked List မှာ “Skip” Link တွေ ထပ်ပေါင်း ပါဝင်လာတာမျိုးပါ။ ဒီ Skip Links တွေက Shortcut တွေလိုမျိုး အလုပ်လုပ်ပြီး List ထဲက ဒေတာအချက်အလက်တွေဆီ ပိုမြန်မြန် ရောက်အောင်၊ ရှာနိုင်အောင် လုပ်ဆောင်ပေးပါတယ်။ (ပုံ ၄-၁၀)။



ပုံ ၄-၁၀၊ Skip List

Skip List ရဲ့ Level တစ်ခုစီတိုင်းက တချို့ Element တွေအတွက် ပိုပြီး ထိထိရောက်ရောက် ရှာနိုင်စေပါတယ်။ ရှာနေတဲ့ ဒေတာအချက်အလက်ပေါ် မူတည်ပြီး၊ List ထဲမှာ အလျားလိုက် ဘိမှမဟုတ် အပေါ်အောက် ဖြတ်သန်းသွားလာနိုင်တဲ့အတွက် ရှာဖွေရတာ ပိုမြန်စေတာပါ။ Skip List တွေကို ပမာဏကြီးမားတဲ့ Dataset တွေအတွက် မကြာမဏ ရှာဖွေမှုလုပ်ဖို့ လိုအပ်တဲ့အခါ အသုံးများပါတယ်။

Linked List အတွက် နမူနာကုဒ်ကို အောက်မှာဖော်ပြထားပါတယ်။

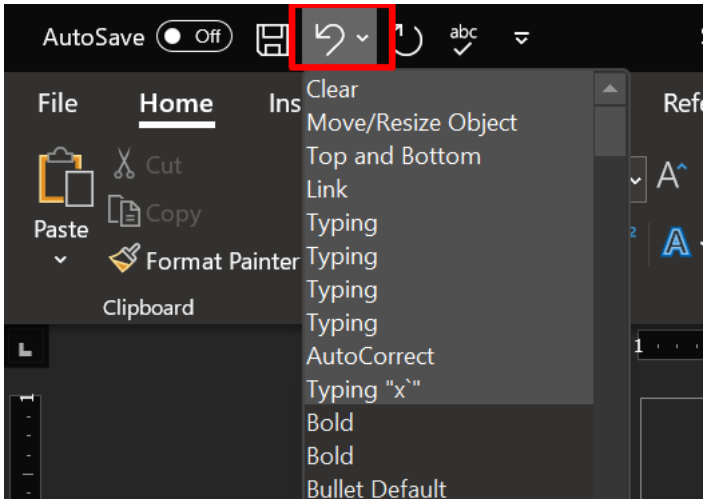
```
class LinkedListNode {
  constructor(data, next = null) {
    this.data = data;
    this.next = next;
  }
}

class LinkedList {
  constructor() {
```

```
    this.head = null;
    this.tail = null;
    this.size = 0;
  }

  addFirst(data) {
    const newNode = new LinkedListNode(data, this.head);
    this.head = newNode;
    if (!this.tail) {
      this.tail = newNode;
    }
    this.size++;
  }

  addLast(data) {
    const newNode = new LinkedListNode(data);
    if (!this.head) {
      this.head = newNode;
      this.tail = newNode;
    } else {
      this.tail.next = newNode;
      this.tail = newNode;
    }
    this.size++;
  }

  addBefore(beforeData, data) {
    const newNode = new LinkedListNode(data);
    if (this.size === 0) {
      this.head = newNode;
      this.size++;
      return;
    }
    if (this.head.data === beforeData) {
      newNode.next = this.head;
      this.head = newNode;
      this.size++;
      return;
    }
    let current = this.head.next;
    let prev = this.head;
    while (current) {
      if (current.data === beforeData) {
        newNode.next = current;
        prev.next = newNode;
```

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၅) - Stack များ

ကွန်ပျူတာစာရိုက်နေစဉ်မှာ တစ်ခုခုကို Undo (ပြန်ဖျက်) သို့မဟုတ် Redo (ပြန်လုပ်ခြင်း) လုပ်ဖူးပါသလား။ (ပုံ ၅-၁)။ ဒါမှမဟုတ် Browser မှာ ရှေ့နောက် အပြန်အလှန် သွားလာကြည့်ရှုဖူးပါသလား။



ပုံ(၅-၁)။ Undo (ပြန်ဖျက်) သို့မဟုတ် Redo (ပြန်လုပ်ခြင်း)

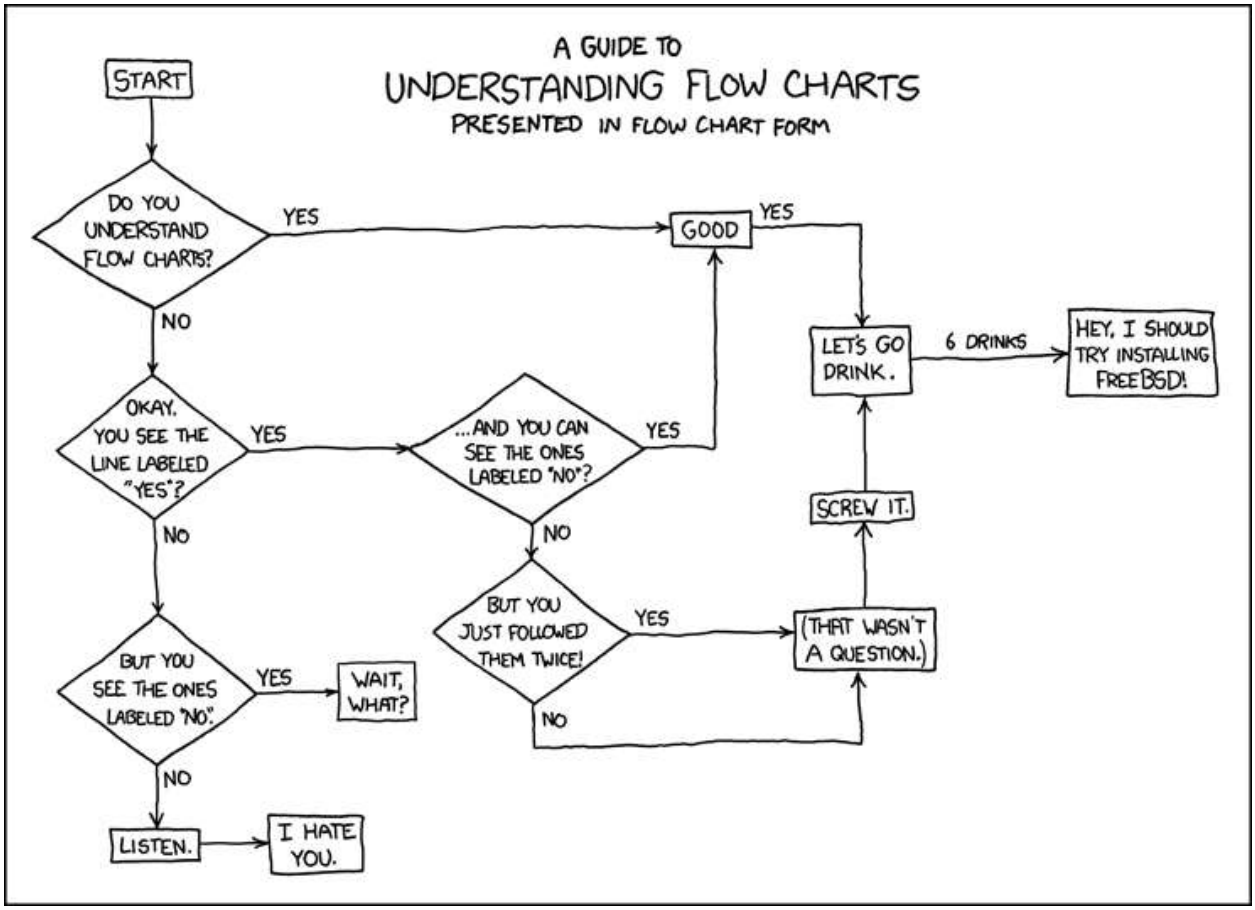
ကျွန်တော်တို့ ရေးသားလိုက်တဲ့အရာတွေကို အဆင့်ဆင့် မှတ်သားထားနိုင်ဖို့ ကွန်ပျူတာက ဘယ်လို လုပ်ဆောင်ရသလဲဆိုတာ စဉ်းစားကြည့်ဖူးပါသလား။

အထက်ပါမေးခွန်းတွေထဲက တစ်ခုခုကို 'ဟုတ်ကဲ့' လို့ဖြေခဲ့မယ်ဆိုရင်၊ မိတ်ဆွေဟာ ဒီသင်ခန်းစာရဲ့ အဓိကဇောတ်ကောင် ဖြစ်တဲ့ Stack (စတက်ခ်) ဒေတာဖွဲ့စည်းပုံကို တွေ့ကြုံခဲ့ဖူးပါပြီ။ ဒီအခန်းမှာ Stack အကြောင်းကို လေ့လာကြပါမယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၇) – Tree များ

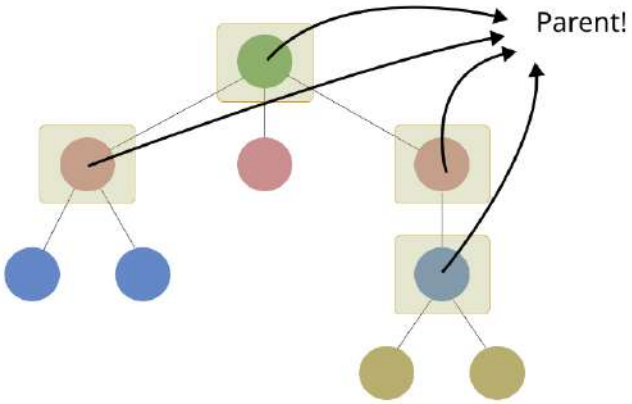
ကျွန်တော်တို့ ပတ်ဝန်းကျင်ကို ကြည့်လိုက်မယ်ဆိုရင်၊ ကျွန်တော်တို့ ကိုင်တွယ်နေရတဲ့ ဒေတာအချက်အလက် အများစုဟာ Hierarchical (အဆင့်ဆင့်ရှိတဲ့ ပုံစံမျိုးတွေ) အတိုင်းဖြစ်နေတာကို တွေ့ရပါလိမ့်မယ်။ တနည်းအားဖြင့် မိဘနဲ့ သားသမီးလိုမျိုး အဆင့်အဆင့်ဆက်နွယ်မှုပုံစံကို ဆိုလိုတာပါ။ သာဓကအနေနဲ့ဆိုရင် Family Tree (မိသားစုဇယား) တွေ၊ Organizational Chart (အဖွဲ့အစည်းဖွဲ့စည်းပုံဇယား) တွေ ၊ Flow Chart / Diagram (လုပ်ငန်းစဉ်အဆင့်ဆင့်ကို ပြတဲ့ ပုံ/ဇယား) တွေ စသည်ဖြင့် တွေ့ရှိရမှာပါ။ (ပုံ ၇-၁ ကိုကြည့်ပါ။)



ပုံ ၇-၁၊ နာမည်ကြီး Webcomic ဆိုက်ဒ်တခုဖြစ်တဲ့ xkcd ရဲ့ နမူနာရုပ်ပြောင် Flow Chart တခုဖြစ်ပါတယ်။

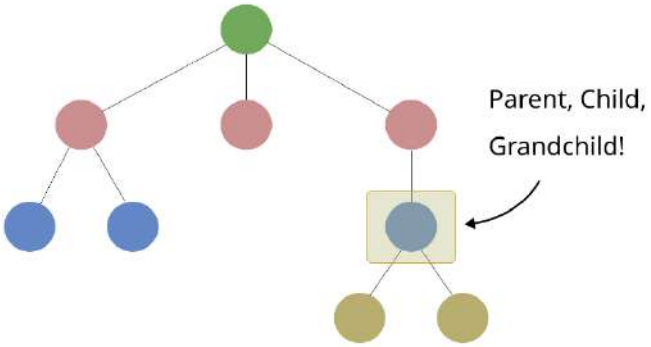
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Child Node တွေ ရှိလာတဲ့အခါမှာ Parent Node တွေလည်း ရှိလာပါတယ်။ Parent Node ဆိုတာ Child Node တွေ အပေါ်မှာရှိနေတဲ့ ဘယ် Node မဆို ဖြစ်နိုင်ပါတယ်။ (ပုံ ၇-၅)။



ပုံ ၇-၅၊ Parent Node များ

ဒီနေရာမှာ တစ်ခု ပြောချင်တာက Parent ဒါမှမဟုတ် Children ဆိုတဲ့ အဓိပ္ပာယ်ဟာ ကျွန်တော်တို့ Tree ရဲ့ ဘယ်အစိတ်အပိုင်းကို ကြည့်နေလဲဆိုတဲ့အပေါ်မှာ မူတည်ပြီး ပြောင်းလဲနိုင်ပါတယ်။ Node တစ်ခုဟာ တစ်ချိန်တည်းမှာ Child လည်း ဖြစ်နိုင်သလို၊ Parent, Grandparent, Grandchild စသည်ဖြင့် အမျိုးမျိုး ဖြစ်နိုင်ပါတယ်။ (ပုံ ၇-၆ ကိုကြည့်ပါ။) ဒါပေမဲ့ သာမန်အားဖြင့်တော့ Node တစ်ခုကို Child ဒါမှမဟုတ် Parent လို့ပဲ ရည်ညွှန်းလေ့ရှိပါတယ်။ ဒီထက်ပိုပြီး အဆင့်တွေကို ထည့်ပြောတာက ရှုပ်ထွေးမှုကို ဖြစ်စေနို



ပုံ ၇-၆၊ Node တစ်ခုမှာ မတူညီတဲ့ အခေါ်အဝေါ်တွေ ရှိနိုင်ပါတယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

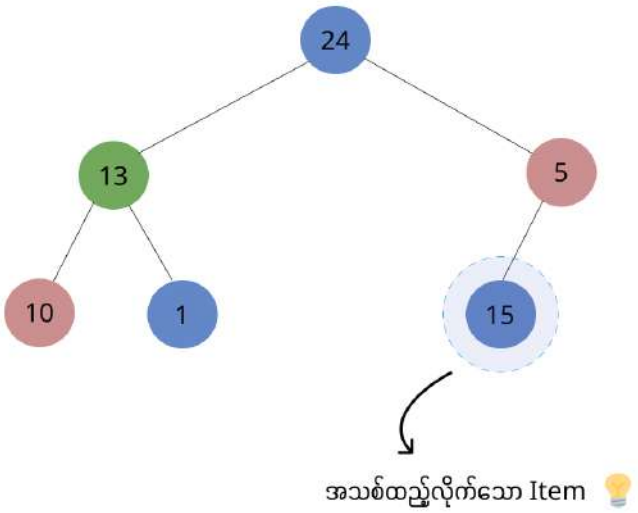
Element တစ်ခု ထည့်သွင်းခြင်း

Heap ထဲကို Element တစ်ခု ထည့်ဖို့ဆိုရင်

- Time complexity: O(log n) နဲ့
- Space complexity: O(1) ပါ။ - n ဆိုတာ Heap ထဲက Element အရေအတွက်ပါ။

Heap ထဲကို Item တစ်ခု ထည့်ဖို့အတွက် ပထမဆုံးအဆင့်အနေနဲ့ Item အသစ်ကို Heap ရဲ့ အဆုံးမှာ ထည့်ရပါမယ်၊ ပြီးရင် ဒုတိယအဆင့်အနေနဲ့ Heap ရဲ့ တည်ဆောက်ပုံဂုဏ်သတ္တိ ပြန်ရတဲ့အထိ၊ Root Node အသစ်ကို Bubbling Up လုပ်ငန်းစဉ် လုပ်ဆောင်ပေးရပါမယ်။ (နမူနာကုဒ်မှာဆိုရင် #bubbleUp ပါ။)

Item အသစ်ကို Heap ရဲ့ အဆုံးမှာ ထည့်တဲ့ ပထမအဆင့်က Constant Time O(1) ပဲ ကြာပါတယ်။ ဥပမာ၊ နမူနာပုံ (၁၀-၂၇) မှာ Node 15 ကို ထည့်သွင်းတဲ့ပုံစံမျိုးပါ။ ဘာကြောင့်လဲဆိုတော့ ကျွန်တော်တို့ရဲ့ Heap က Array ကို အသုံးပြုပြီး Implement လုပ်ထားတဲ့အတွက်ပါ။ တိတိကျကျပြောရရင် Array ရဲ့ နောက်ဆုံးအခန်းမှာ Item တွေ ထည့်သွင်းတဲ့ လုပ်ငန်းစဉ်က အင်မတန်မြန်ဆန်တဲ့အတွက်ပါ၊ ဒါဟာ Array ရဲ့ အားသာချက်တစ်ခုပါပဲ။



ပုံ ၁၀-၂၇၊ ရှေ့ပိုင်းမှာ ဖော်ပြခဲ့သလို၊ Item တစ်ခုကို ထည့်သွင်းခြင်း

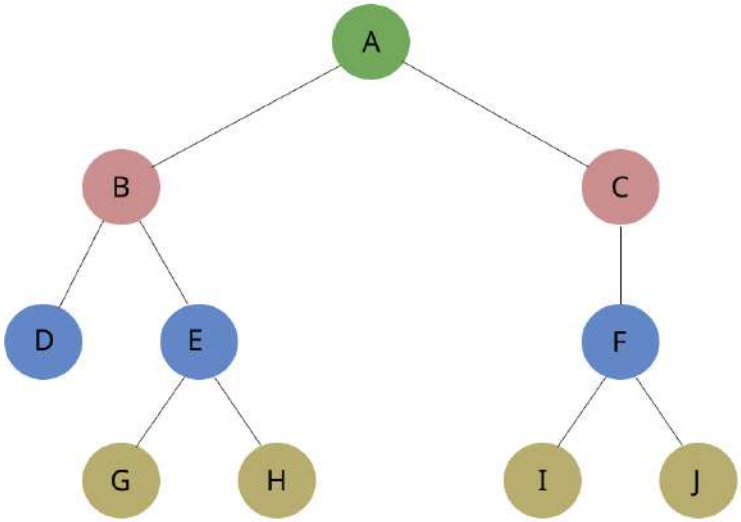
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၈) - Binary Tree များ

ဒီအခန်းမှာတော့ Binary Tree လို့ခေါ်တဲ့ Tree Data Structure အမျိုးအစားတစ်ခုအကြောင်းကို လေ့လာသွားမှာ ဖြစ်ပါတယ်။ အရင်တစ်ခေါက်က Tree ရဲ့ အခြေခံသဘောတရားတွေကို လေ့လာခဲ့ပြီးပြီဆိုတော့ အခုတစ်ခါ ပိုပြီးနက်နက်ရှိုင်းရှိုင်း လေ့လာကြည့်ရအောင်ပါ။

Binary Tree ဆိုတာ ဘာလဲ

ပုံမှန် Tree တွေလိုပဲ Binary Tree ကလည်း ဒေတာအချက်အလက်တွေကို Hierarchical ပုံစံ အဆင့်ဆင့် သိမ်းဆည်းနိုင်တဲ့ Data Structure တစ်မျိုးပါပဲ။ ပုံ ၈-၁ မှာ Binary Tree ရဲ့ ပုံစံကို တွေ့မြင်နိုင်ပါတယ်။



ပုံ ၈-၁၊ နမူနာ Binary Tree တစ်ခု

Binary Tree တွေမှာ သာမန် Tree တွေနဲ့မတူတဲ့ အဓိကစည်းမျဉ်းသတ်မှတ်ချက် Rule (၃) ခု ရှိပါတယ်။

၁။ Node တစ်ခုမှာ အများဆုံး Children နှစ်ခုပဲ ရှိနိုင်ပါတယ်။ (သုည၊ တစ်ခု သို့မဟုတ် နှစ်ခု ဖြစ်နိုင်ပါတယ်)

၂။ Tree တစ်ခုလုံးမှာ Root Node တစ်ခုတည်းပဲ ရှိရပါမယ်။

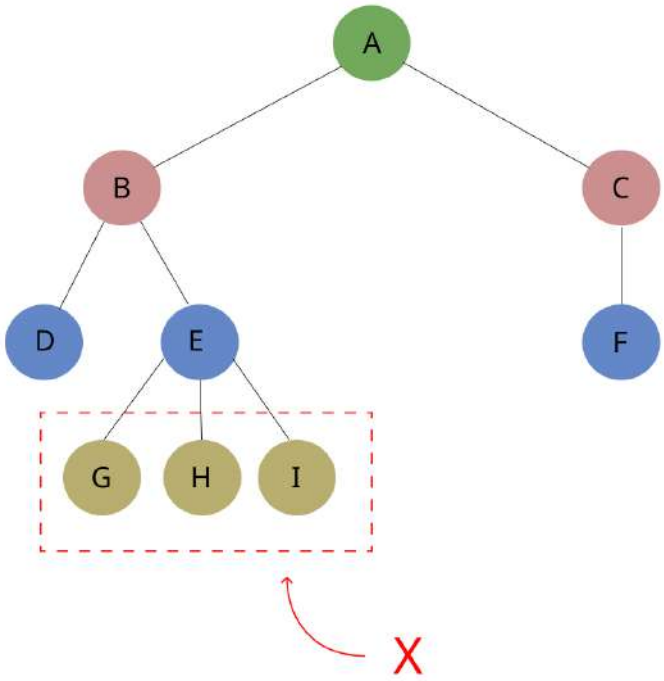
၃။ Root Node ကနေ Node တစ်ခုကို ရောက်ဖို့ လမ်းကြောင်း (Path) တစ်ခုတည်းသာ ရှိရပါမယ်။

ဒီ Rule တွေကို သေချာနားလည်ထားဖို့ အရေးကြီးပါတယ်။ ဘာကြောင့်လဲဆိုတော့ ဒါတွေက Binary Tree ဘယ်လိုအလုပ်လုပ်သလဲဆိုတာကို ရှင်းပြပေးနိုင်သလို၊ နောက်ပိုင်းမှာလေ့လာရမယ့် Binary Search Tree လိုမျိုး တခြား Tree အမျိုးအစားတွေကိုလည်း နားလည်ဖို့ အထောက်အကူ ဖြစ်စေတဲ့အတွက်ကြောင့် ဖြစ်ပါတယ်။

စည်းမျဉ်း (Rule) များအကြောင်း အသေးစိတ်

စည်းမျဉ်း (၁): Node တစ်ခုမှာ အများဆုံး Children နှစ်ခုပဲ ရှိနိုင်ပါတယ်

Binary Tree ထဲက Node တိုင်းမှာ Child Node အရေအတွက်ဟာ သုည၊ တစ်ခု သို့မဟုတ် နှစ်ခုပဲ ရှိနိုင်ပါတယ်။ အကယ်၍ Node တစ်ခုမှာ Child Node နှစ်ခုထက် ပိုလာခဲ့ရင်တော့ ဒါဟာ Binary Tree ရဲ့ စည်းမျဉ်းကို ချိုးဖောက်တာ ဖြစ်ပါတယ်။ (ပုံ ၈-၂ ကိုကြည့်ပါ)။



ပုံ ၈-၂။ နှစ်ခုထက်ပိုတဲ့ Child Node တွေကို ခွင့်မပြုပါဘူး

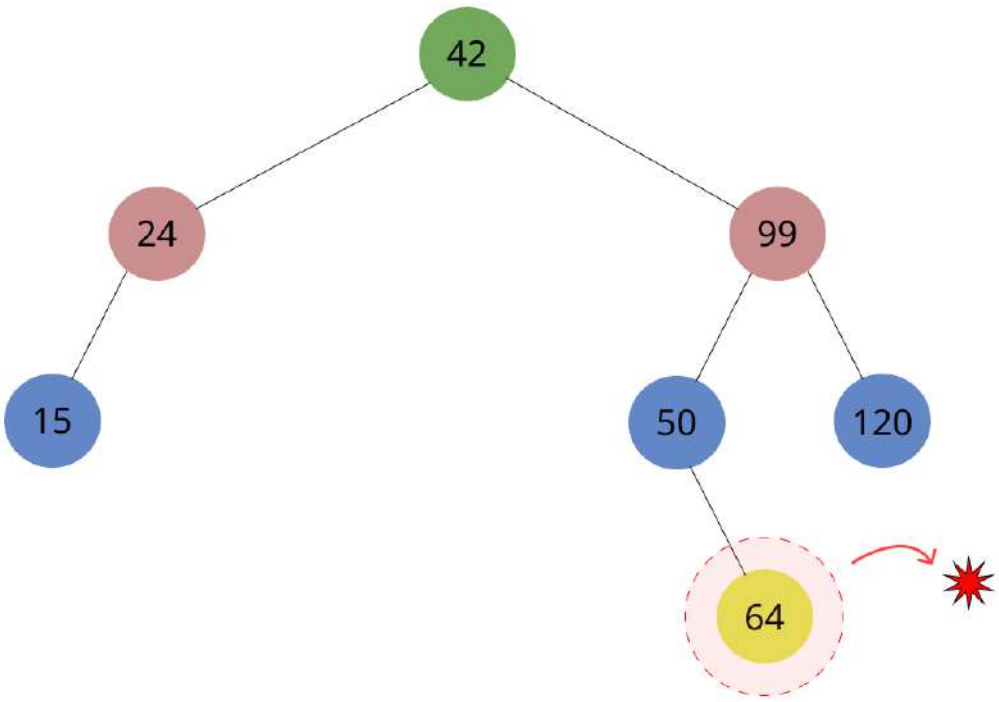
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Node များ ဖယ်ရှားခြင်း

Node တွေ ထည့်ရတဲ့ အချိန်တွေ ရှိသလို၊ Node တွေ ဖယ်ရှားရမယ့် အချိန်တွေလည်း ရှိပါတယ်။ Binary Search Tree ကနေ Node တွေကို ဖယ်ရှားတာက Node တွေ ထည့်သွင်းတာထက် အနည်းငယ် ပိုရှုပ်ထွေးပါတယ်။ ဘာကြောင့်လဲဆိုတော့ ဘယ် Node ကို ဖယ်ရှားနေလဲဆိုတဲ့အပေါ် မူတည်ပြီး လုပ်ဆောင်ပုံက ကွဲပြားလို့ပါ။

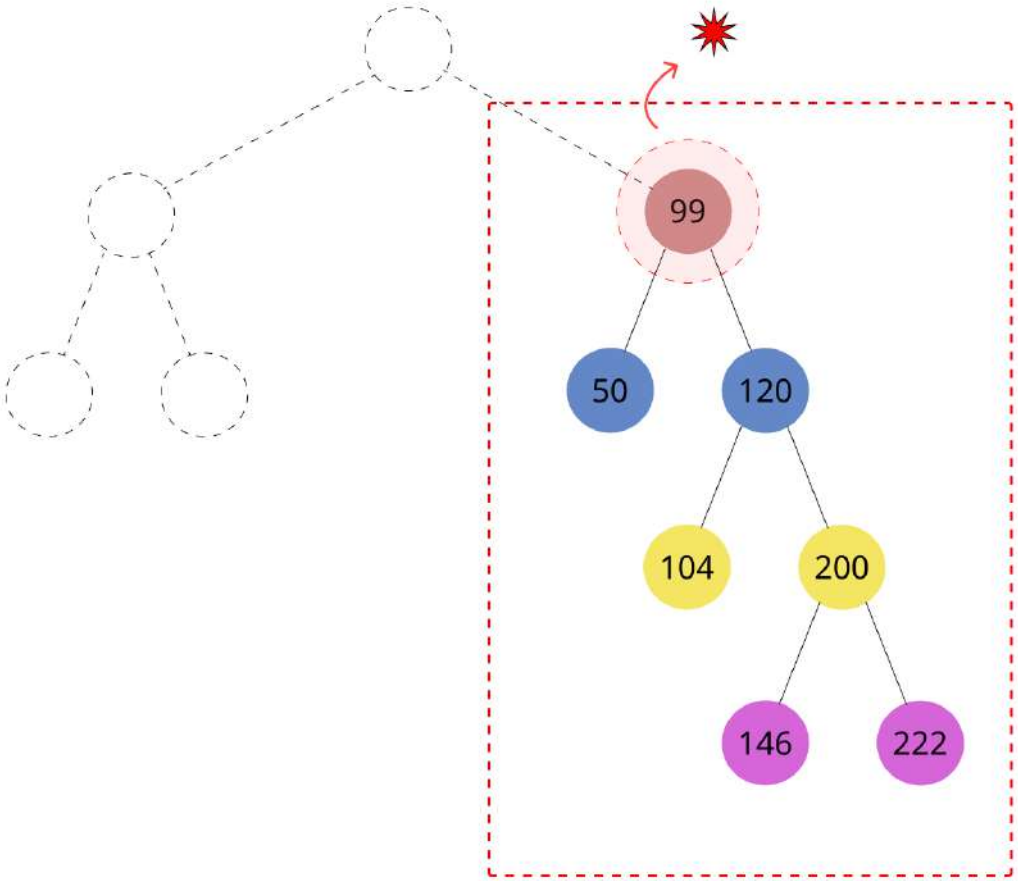
Leaf Node ကို ဖယ်ရှားခြင်း

ကျွန်တော်တို့ ဖယ်ရှားချင်တဲ့ Node က Leaf Node ဖြစ်နေမယ်ဆိုရင် ဒီလုပ်ငန်းစဉ်ဟာ ရိုရှင်းပါတယ်။ အစောပိုင်းက နမူနာ Binary Search Tree ကိုပဲ ဆက်သုံးပါမယ်။ တန်ဖိုး 64 ပါတဲ့ Leaf Node ကို ဖယ်ရှားမယ်ဆိုပါစို့ (ပုံ ၉-၁၁)။



ပုံ ၉-၁၁၊ တန်ဖိုး 64 ပါတဲ့ Leaf Node ကို ဖယ်ရှားပါမယ်

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။



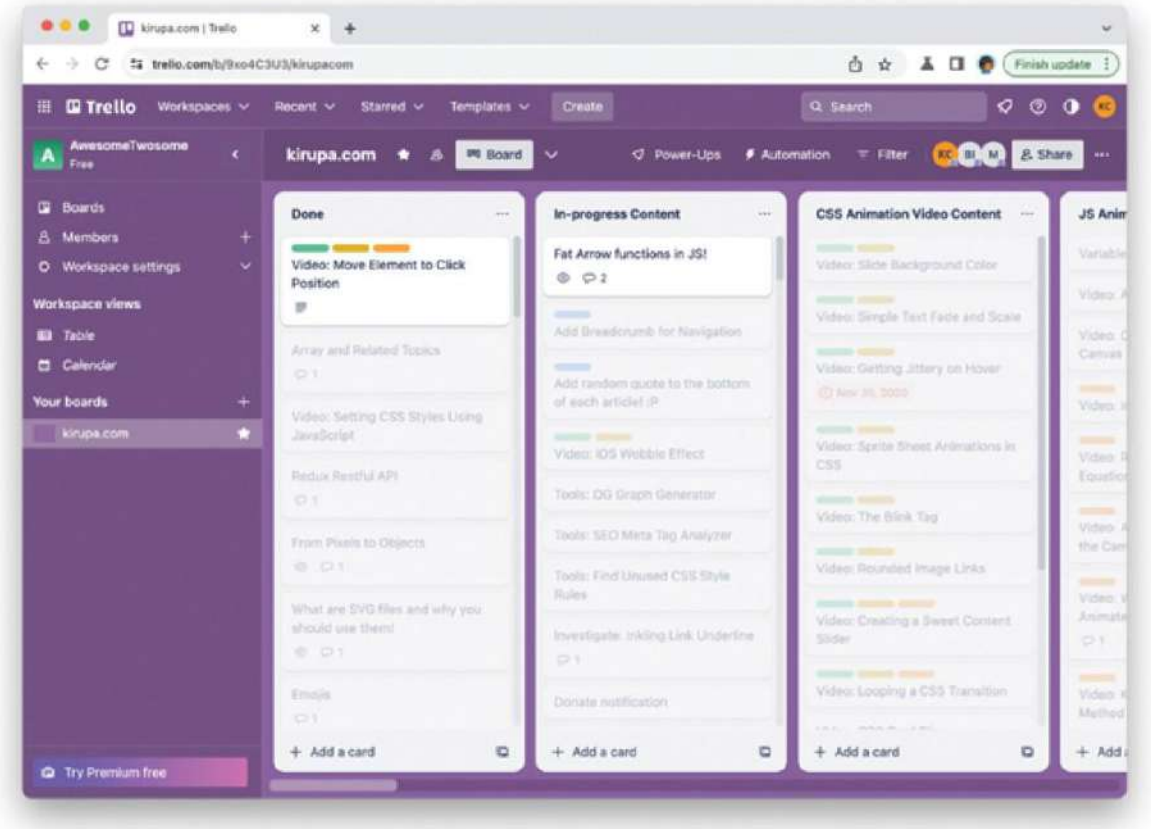
ပုံ ၉-၁၈၊ Node 99 ကို ဖယ်ရှားမယ့် အခြေအနေအတွက် ညာဘက်အကိုင်းအခက် (Right Subtree)

အဲဒီ Subtree ထဲမှာ ဘယ် Node က 99 ပြီးရင် အနီးစပ်ဆုံး အကြီးဆုံးတန်ဖိုး (Next Highest Value) ဖြစ်သလဲ၊ (တနည်းအားဖြင့် အဲဒီ Subtree ထဲမှာ တန်ဖိုးအငယ်ဆုံးက ဘယ် Node ဖြစ်မလဲ) ဆိုတာ ရှာရမှာဖြစ်ပါတယ်။။။ လက်ရှိအခြေအနေမှာဆိုရင်တော့ Node 104 က အဖြေဖြစ်ပါတယ်။ ဒီတော့ ကျွန်တော်တို့ လုပ်ရမှာက Node 99 ကို ဖယ်ရှားပြီး သူ့နေရာမှာ Node 104 ကို အစားထိုးထည့်လိုက်ရုံပါပဲ (ပုံ ၉-၁၉)။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၁၀) – Heap များ

ကျွန်တော်တွေမှာ စိတ်ကူးအကြံဉာဏ်တွေ အများကြီးရှိတတ်ပေမယ့်၊ အဲဒီအကြံဉာဏ်တွေကို လက်တွေ့လုပ်ဆောင်ကြတဲ့အခါ ဘယ်ဟာကို အရင်လုပ်ရမလဲဆိုတာ ဝေခွဲမရဖြစ်တတ်ကြပါတယ်။ ဒီလိုအခက်အခဲကို ဖြေရှင်းဖို့အတွက် Task Management App တွေ၊ Project Management Tool တွေကို အသုံးပြုလေ့ရှိကြပါတယ်။ (ပုံ ၁၀-၁ မှာ ပြထားပါတယ်)



ပုံ ၁၀-၁၊ နမူနာ Task Board Application တစ်ခု

ဒီလိုမျိုး Tool တွေ၊ App တွေ အများကြီးရှိပါတယ်။ ယေဘုယျအားဖြင့်တော့ အောက်မှာဖော်ပြထားတဲ့ လုပ်ငန်းစဉ်တွေ ပါဝင်ကြပါတယ်။

- ကျွန်တော်တို့ လုပ်မယ့်တဲ့အလုပ်တွေကို စာရင်းပြုစုနိုင်ခြင်း
- အရေးကြီးတဲ့အလုပ်တွေကို ဦးစားပေး သတ်မှတ်နိုင်ခြင်း

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

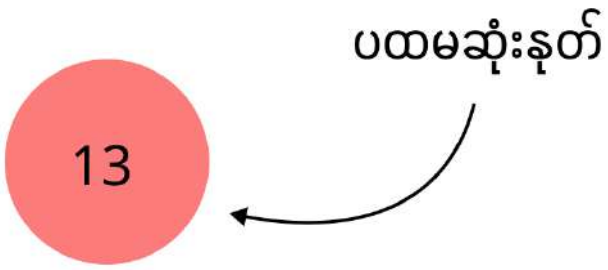
ဒီအချက်ဟာ Performance အကြောင်း ပြောတဲ့အခါမှာ အလေးပေးရမယ့် အချက်တစ်ခုပါ။ ဘာကြောင့်လဲဆိုရင် Balanced Binary Tree တွေမှာ Tree ရဲ့ အမြင့် (Height) က $O(\log n)$ ဖြစ်တဲ့အတွက် Data Operation အတော်များများအတွက် အကောင်းဆုံးအခြေအနေပါပဲ။

Heap Operation များ

Heap ရဲ့ ရည်ရွယ်ချက်က Priority အမြင့်ဆုံး ဒေတာကို အမြန်ဆုံး ဖယ်ရှားနိုင်ဖို့ ဖြစ်ပါတယ်။ ဒီလိုလုပ်နိုင်ဖို့အတွက် ကျွန်တော်တို့ဟာ Heap ထဲကို ဒေတာတွေ ထည့်သွင်းပေးဖို့လည်း လိုအပ်ပါတယ်။ ဒါကြောင့် Heap ရဲ့ အဓိက လုပ်ငန်းစဉ် Operation နှစ်ခုဖြစ်တဲ့ ဒေတာတွေထည့်သွင်းခြင်းနဲ့ ဖယ်ရှားခြင်းအကြောင်းတွေကို အသေးစိတ် လေ့လာသွားပါမယ်။

Node တစ်ခု ထည့်သွင်းခြင်း

Node တစ်ခု ထည့်သွင်းခြင်းကနေ စတင်လေ့လာပါမယ်။ ဘာမှမထည့်သွင်းရသေးတဲ့ ဗလာ အခြေအနေကနေ ကျွန်တော်တို့ရဲ့ Heap ကို စတင် တည်ဆောက်ပါမယ်။ ပထမဆုံး ထည့်သွင်းမယ့် ဒေတာကတော့ 13 ပါ။ (ပုံ ၁၀-၅ မှာ ပြထားပါတယ်။)



ပုံ ၁၀-၅၊ ပထမဆုံး ထည့်သွင်းသော Node

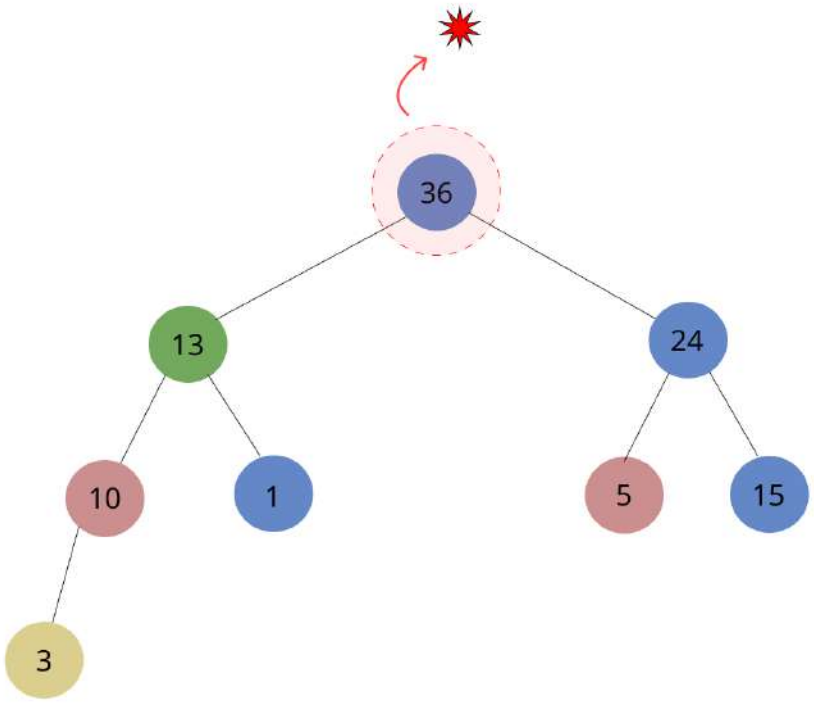
ဒါဟာ ကျွန်တော်တို့ရဲ့ ပထမဆုံး Node ဖြစ်တဲ့အတွက် အပေါ်ဆုံးမှာ Root Node အဖြစ် ရောက်ရှိသွားပါတယ်။ အင်မတန် ရိုးရှင်းတဲ့ အခြေအနေပါ။ ဒါပေမဲ့ နောက်ထပ်ထည့်မယ့် ဒေတာတွေ အတွက်ကတော့ အောက်မှာ ဖော်ပြထားတဲ့စည်းမျဉ်းတွေကို အတိအကျ လိုက်နာရပါမယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အပေါ်ဆုံး Root Node ကို ဖယ်ရှားခြင်း

နောက်ထပ်လေ့လာမယ့် လုပ်ငန်းစဉ်ကတော့ Root Node ကို ဘယ်လို ဖယ်ရှားမလဲဆိုတာပါ။ တနည်းအားဖြင့် Heap ထဲက အကြီးဆုံးနဲ့ အရေးအကြီးဆုံး တန်ဖိုးကို ဘယ်လို ဖယ်ရှားရမလဲဆိုတာပါပဲ။ အပေါ်ဆုံးမှာရှိတဲ့ Root Node ကို ဖယ်ရှားတာဟာ ကျွန်တော်တို့ ရှေ့မှာလေ့လာခဲ့တဲ့ Heap ထဲကို Node တွေ ထည့်သွင်းတဲ့ လုပ်ငန်းစဉ်မျိုးနဲ့ လုံးဝကွဲပြားပါတယ်။ စိတ်ဝင်စားစရာကောင်းတဲ့ အပြုအမူတချို့ လည်းရှိနေပါတယ်။ ဒီတော့ စလိုက်ကြရအောင်ပါ၊ အပေါ်က ကျွန်တော်တို့ သုံးခဲ့တဲ့ Heap ဥပမာနဲ့ပဲ ဆက်သွားပါမယ်။

ကျွန်တော်တို့ လုပ်ချင်တာကတော့ တန်ဖိုး 36 ရှိတဲ့ အပေါ်ဆုံး Node ကို ဖယ်ရှားဖို့ပါ။ (ပုံ ၁၀-၁၆ မှာ ပြထားပါတယ်။)



ပုံ ၁၀-၁၆၊ Root Node ကို ဖယ်ရှားခြင်း

Heap ထဲကနေ Root Node ကို ဖယ်ရှားတဲ့အခါ Heap ရဲ့ ဂုဏ်သတ္တိကို ဆက်လက် ထိန်းသိမ်းထားနိုင်ဖို့ လိုအပ်ပါတယ်။ ဆိုလိုတာက Root Node အသစ်ဖြစ်လာမယ့် တန်ဖိုးဟာ Heap ထဲမှာ အကြီးဆုံးတန်ဖိုးအဖြစ် ဆက်ရှိနေရမှာပါ။

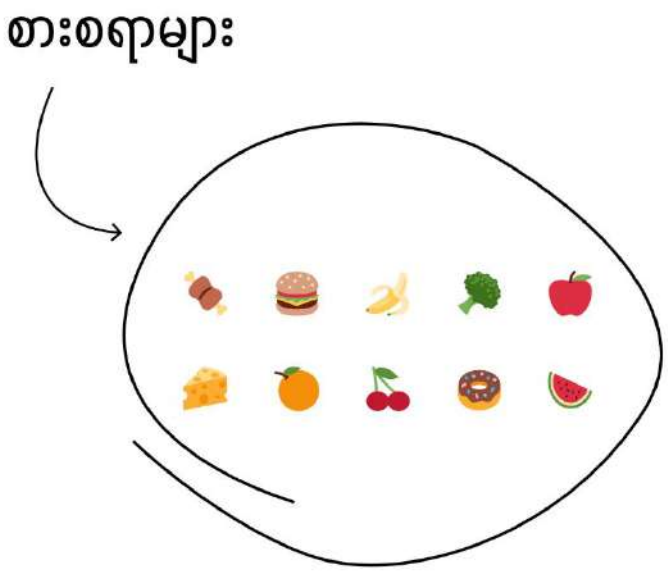
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၁၁) - Hashtable (Hashmap သို့မဟုတ် Dictionary)

Data Structure အမျိုးအစားတွေကြားထဲမှာ Hashtable ဟာ မတူကွဲပြားတဲ့ သဘောသဘာဝရှိနေပါတယ်။ ဘာကြောင့်လဲဆိုရင် Hashtable တွေဟာ ဒေတာတန်ဖိုးတွေကို အလျင်မြန်ဆုံး သိမ်းဆည်းနိုင်ပြီး၊ ပြန်လည်ရယူဖို့လည်း အလျင်မြန်ဆုံးလုပ်ဆောင်နိုင်လို့ပါ။ ဒီလို လျင်မြန်မှုတွေကြောင့် Hashtable တွေကို Cache System တွေ ဒါမှမဟုတ် တချို့သော Data Structure တွေနဲ့ Algorithm တွေရဲ့ အခြေခံဖွဲ့စည်းပုံတွေအဖြစ် အသုံးပြုကြတာဖြစ်ပါတယ်။ ဒီအခန်းမှာ Hashtable တွေအကြောင်းနဲ့၊ သူတို့ရဲ့ အလုပ်လုပ်ပုံတွေကို အသေးစိတ် လေ့လာသွားပါမယ်။

နမူနာစက်ရုပ်လေး - Hashtable ကို နားလည်ဖို့ ဥပမာ

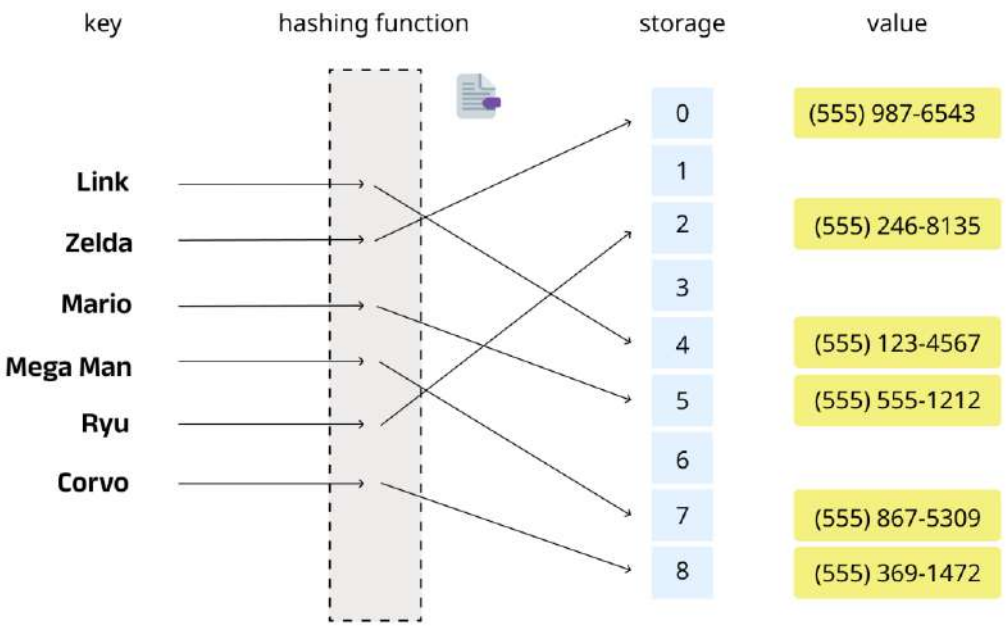
Hashtable ဘယ်လိုအလုပ်လုပ်လဲဆိုတာကို ရှင်းပြဖို့အတွက် ဥပမာတစ်ခုကို အသုံးပြုသွားပါမယ်။ ဆိုပါစို့၊ ကျွန်တော်တို့ဟာ မတူညီတဲ့ စားစရာတွေကို သိမ်းဆည်းဖို့ စီစဉ်နေတယ်လို့ စဉ်းစားကြည့်ပါ။ (ပုံ ၁၁-၁ ကိုကြည့်ပါ။)



ပုံ ၁၁-၁၊ စားစရာတွေ အကြောင်း ပြောရအောင်ပါ။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

ဒီအချက်အလက်တွေကို Hashtable ထဲမှာ ထည့်လိုက်တာနဲ့ ပုံ ၁၁-၁၅ ထဲမှာ ပြထားသလို ပုံစံကို တွေ့ရမှာပါ။ Key တစ်ခုချင်းစီကို Hashing Function ဆီ ပို့ပြီး သိမ်းဆည်းရမယ့်နေရာ (Storage Location) ကို ရွေးချယ်ပါတယ်။ အဲဒီနောက် ရရှိလာတဲ့နေရာမှာ Value ကို ထည့်သွင်းသွားတာဖြစ်ပါတယ်။



ပုံ ၁၁-၁၅၊ Hashing Function ရဲ့ အလုပ်လုပ်ပုံ

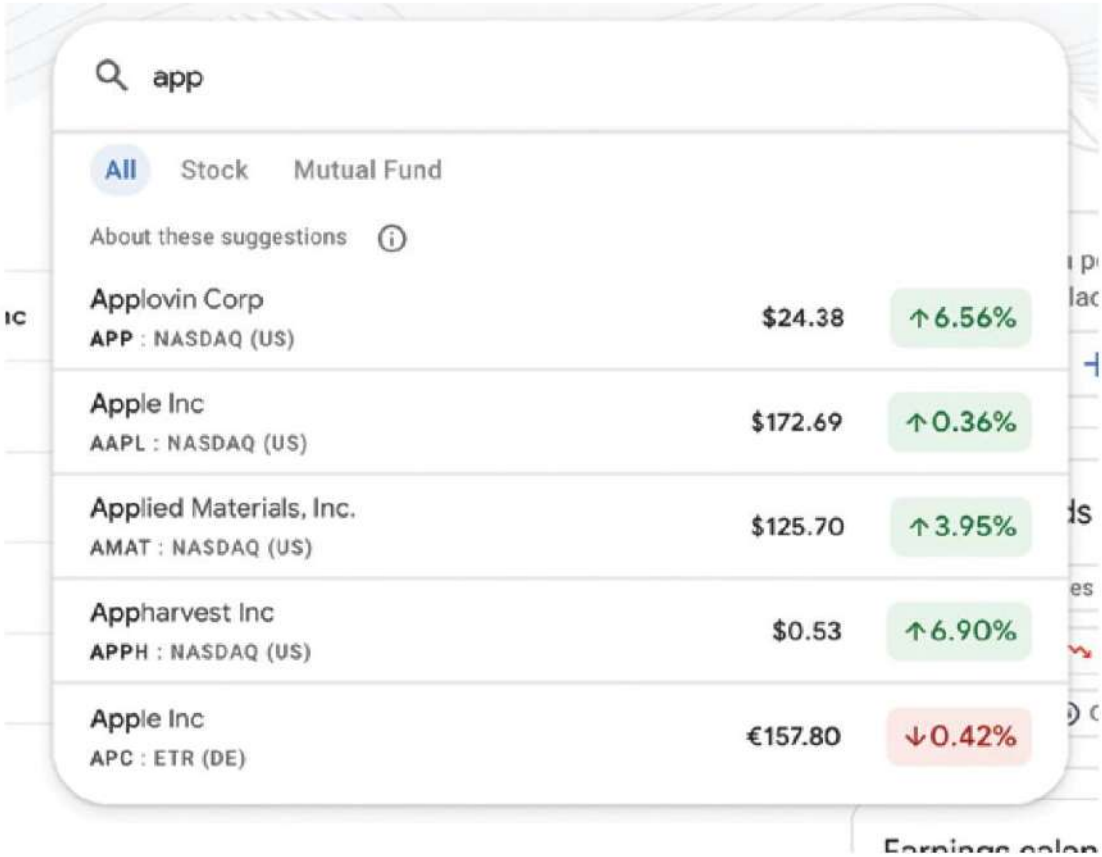
Hashtable ထဲက Item တွေကို ဖတ်ခြင်း

Hashtable ထဲက တန်ဖိုးတစ်ခုခုကို ဖတ်ချင်တယ်ဆိုပါစို့။ ဥပမာ၊ Mega Man ရဲ့ ဖုန်းနံပါတ်ကို လိုချင်တယ်၊ ဒါဆိုရင် ကျွန်တော်တို့လုပ်ရမှာက 'Mega Man' ဆိုတဲ့ Key ကို ထည့်ပေးလိုက်ဖို့ပါပဲ။ Hashing Function က အဆိုပါ Key ကို အသုံးပြုပြီး သိမ်းထားတဲ့နေရာကို တွက်ချက်ပါမယ်၊ အဲဒီနောက် ကျွန်တော်တို့ လိုချင်တဲ့ဖုန်းနံပါတ်ကို အမြန်ဆုံး ပြန်ထုတ်ယူပေးမှာဖြစ်ပါတယ်။ (ပုံ ၁၁-၁၆)။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၁၂) - Trie (သို့မဟုတ် Prefix Tree)

ကျွန်တော်တို့ အင်တာနက်စာမျက်နှာတစ်ခုပေါ် ရောက်နေတယ်ဆိုပါစို့။ စာရိုက်ထည့်ရမယ့် နေရာ (Input Field) မှာ စာစရိုက်လိုက်တဲ့အခါ၊ ကျွန်တော်တို့ ရိုက်ထည့်လိုက်တဲ့ စာလုံးအနည်းငယ်ကို အခြေခံပြီး ခန့်မှန်းရလဒ်တွေကို မြင်တွေ့ရတတ်ပါတယ်။ (ပုံ ၁၂-၁ ကိုကြည့်ပါ။)



ပုံ ၁၂-၁၊ Auto Complete နမူနာ

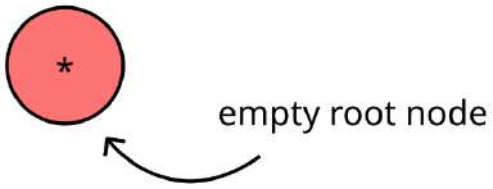
ကျွန်တော်တို့ ဆက်ရိုက်လေလေ၊ ရလဒ်တွေက ပိုပြီးတိကျလာပြီး ကျွန်တော်တို့ ရိုက်ချင်တဲ့ စကားလုံး ဒါမှမဟုတ် စကားစုတစ်ခုလုံးကို တိတိကျကျ ခန့်မှန်းပေးနိုင်တဲ့အထိ ဖြစ်လာပါတယ်။ ဒီလို အလိုအလျောက် ဖြည့်စွက်ပေးတဲ့ (Autocompletion) လုပ်ငန်းစဉ်မျိုးဟာ ဒီနေ့ခေတ်မှာ အထူးအဆန်းမဟုတ်တော့ပါဘူး။ ကျွန်တော်တို့ရဲ့ User Unterface (UI) တွေ အားလုံးနီးပါးမှာ ဒီလိုလုပ်ငန်းစဉ်တွေ ပါဝင်နေပါတယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်

စကားလုံးများ ထည့်သွင်းခြင်း

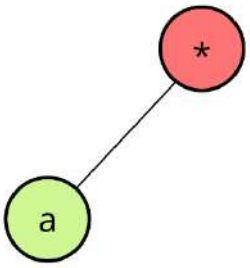
ကျွန်တော်တို့ “apple” ဆိုတဲ့ စကားလုံးကို Trie ထဲမှာ သိမ်းဆည်းချင်တယ်ဆိုပါစို့။ ပထမဆုံးလုပ်ရမှာက ဒီစကားလုံးကို စာလုံးတစ်လုံးချင်းစီ (a, p, p, l, e) အဖြစ် ခွဲထုတ်လိုက်ရပါမယ်။ ပြီးရင် Trie Tree Structure ကို စတင်တည်ဆောက်ရပါမယ်။

Trie ရဲ့ အစမှာတော့ ဘာမှမပါသေးတဲ့ Root Node တစ်ခုပဲ ရှိပါတယ်။ (ပုံ ၁၂-၃ ကိုကြည့်ပါ။)



ပုံ ၁၂-၃၊ အစမှာတော့ ဘာမှမပါသေးတဲ့ Root Node တစ်ခုပဲ ရှိပါတယ်

နောက်တစ်ဆင့်အနေနဲ့ သိမ်းဆည်းချင်တဲ့ စကားလုံး (apple) ထဲက ပထမဆုံးစာလုံး (a) ကိုယူပြီး Root Node ရဲ့ Child အဖြစ် ထည့်သွင်းပါမယ်။ (ပုံ ၁၂-၄ ကိုကြည့်ပါ။)



ပုံ ၁၂-၄၊ အသစ်ထည့်လိုက်တဲ့ Child Node

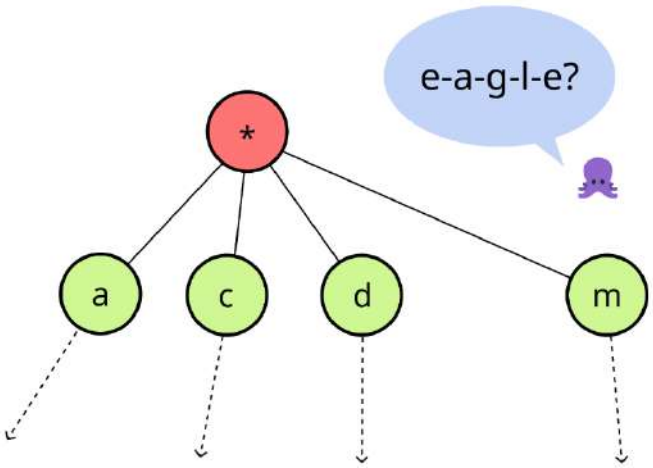
နောက်စာလုံး “p” အတွက်လည်း အလားတူပုံစံအတိုင်းပဲ “a” Node ရဲ့ Child အဖြစ် ထည့်သွင်းပါတယ်။ (ပုံ ၁၂-၅ ကိုကြည့်ပါ။)

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

စကားလုံးများ ရှာဖွေခြင်း

အင်္ဂလိပ်အဘိဓာန်ထဲက စကားလုံးအားလုံးကို Trie ထဲ ထည့်ထားတယ်လို့ စဉ်းစားကြည့်ပါ။ အကယ်၍ စကားလုံးတစ်ခုခုကို အဘိဓာန်ထဲ ပါဝင်တဲ့ စကားလုံးဟုတ်မဟုတ် စစ်ဆေးကြည့်မယ်ဆိုရင် ဒီစကားလုံးကို ရှာဖွေဖို့ ကျွန်တော်တို့ရဲ့ Trie ကို ဘယ်လို ထိထိရောက်ရောက် အသုံးချနိုင်မလဲဆိုတာ လေ့လာကြည့်ကြပါမယ်။

ရှေ့မှာဖော်ပြခဲ့တဲ့ နမူနာ Trie ထဲမှာ “eagle” ဆိုတဲ့ စကားလုံးရှိမရှိ ရှာဖွေတယ်ဆိုပါစို့။ ကျွန်တော်တို့ လုပ်ရမှာက စာလုံးတစ်လုံးချင်းစီ (e, a, g, l, e) ခွဲထုတ်လိုက်ပြီး ပထမဆုံးစာလုံးက Root Node ရဲ့ Child အဖြစ် ရှိမရှိ စစ်ဆေးရပါမယ်။ (ပုံ ၁၂-၁၃ ကိုကြည့်ပါ။)



ပုံ ၁၂-၁၃၊ ရှာဖွေမှုတစ်ခု စတင်ခြင်း

ကျွန်တော်တို့မှာ ရှိတဲ့ Root Node ရဲ့ Child တွေက a, c, d, နဲ့ m ပါ။ “e” ဆိုတဲ့ စာလုံး မရှိတဲ့အတွက် ဒီနေရာမှာပဲ ရှာဖွေမှုကို ရပ်လိုက်လို့ရပါတယ်။ ပထမဆုံးစာလုံး မရှိဘူးဆိုရင်၊ နောက်ဆက်တွဲ စာလုံးတွေလည်း သေချာပေါက် ရှိမှာမဟုတ်တဲ့အတွက်ကြောင့်ပါ။

နောက်ထပ် ရှာဖွေမှုတစ်ခုအနေနဲ့ “monk” ဆိုတဲ့ စကားလုံးကို Trie ထဲမှာ ရှိမရှိ စစ်ဆေးပါမယ်။ လုပ်ငန်းစဉ်အဆင့်ဆင့်က အတူတူပါပဲ။ ကျွန်တော်တို့ ရှာနေတဲ့ စကားလုံးရဲ့ ပထမဆုံးစာလုံး “m” က Trie မှာ Root Node ရဲ့ Child အဖြစ် ရှိမရှိ စစ်ဆေးပါတယ်။ ဒီတစ်ခေါက်တော့ ရှိနေပါတယ်။ (ပုံ ၁၂-၁၄ ကိုကြည့်ပါ။)

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူနိုင်ပါသည်

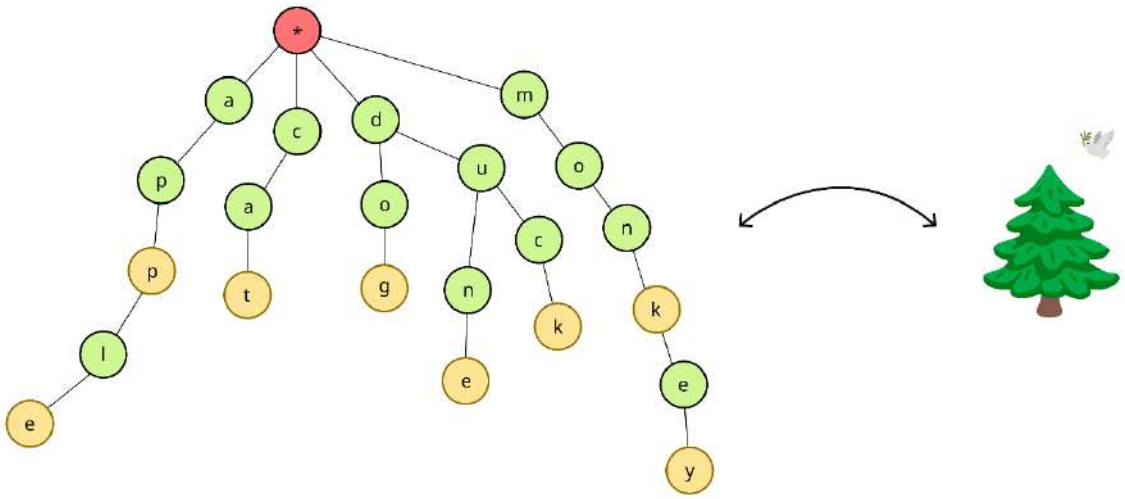
ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

Trie အကြောင်း ပိုမို လေ့လာခြင်း

ရှေ့အပိုင်းတွေမှာ Trie တွေရဲ့ အခြေခံတွေကို လေ့လာပြီးပြီဆိုတော့၊ ဒီတစ်ခါမှာတော့ ပိုမိုနက်ရှိုင်းတဲ့ အကြောင်းအရာတွေကို ဆက်လက်လေ့လာကြည့်ရအောင်ပါ။

Trie ဆိုတာက String တွေ ဒါမှမဟုတ် Character Sequence တွေကို ထိရောက်စွာ သိမ်းဆည်းဖို့နဲ့ ရှာဖွေဖို့ အထူးသင့်တော်တဲ့ Tree-based Data Structure တစ်မျိုးပါ။ အဓိကအားသာချက်က String တွေကို ထည့်သွင်းတာ (Insert)၊ ပယ်ဖျက်တာ (Delete) နဲ့ ရှာဖွေတာ (Search) စတဲ့ လုပ်ငန်းတွေမှာ အလွန်ထိရောက်မှုရှိတာ ဖြစ်ပါတယ်။

ပထမဆုံး၊ Trie ဆိုတာ Tree-based Data Structure တစ်ခုဆိုတာကို မှတ်သားထားရပါမယ်။ (ပုံ ၁၂-၂၀ ကိုကြည့်ပါ။)



ပုံ ၁၂-၂၀၊ Trie ဆိုတာ Tree-based Data Structure တစ်ခုပါ

Trie ရဲ့ ဖွဲ့စည်းပုံကို ကြည့်မယ်ဆိုရင် Node တစ်ခုချင်းစီမှာ Data အစိတ်အပိုင်းလေးတွေ ပါဝင်ပါတယ်။ ကျွန်တော်တို့ ဥပမာမှာဆိုရင် အဲဒီအစိတ်အပိုင်းလေးတွေက စာလုံးတွေဖြစ်ပြီး အဲဒီ စာလုံးလေးတွေပေါင်းစပ်ပြီး စကားလုံးတွေ ဖြစ်လာပါတယ်။

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။

အခန်း (၁၃) – Graph များ

ဒီအခန်းမှာ Graph data structure အကြောင်းကို လေ့လာသွားပါမယ်။ Graph Data Structure ကို အသုံးပြုတဲ့နေရာတွေ အလွန်များသလို၊ သူ့မှာ ကောင်းကျိုးတွေလည်း အများအပြားပါတယ်။ ဒါကြောင့် Graph Theory (ဂရပ်ဖ် သီအိုရီ) လို့ခေါ်တဲ့ သီးခြားဘာသာရပ်တစ်ခုတောင် ရှိနေပါသေးတယ်။ အထူးသဖြင့် ဒီဘာသာရပ်ကို မဟာတန်းတွေ၊ ဂုဏ်ထူးဆောင်တန်းတွေနဲ့ PhD တန်းတွေမှာ သုတေသနလုပ်ဖို့ လေ့လာကြရပါတယ်။ Graph ပေါ်မှာ အခြေခံပြီး ရေးသားထုတ်ဝေထားတဲ့ စာတမ်းတွေ၊ စာအုပ်တွေလည်း အများကြီးရှိပါတယ်။ တချို့ အနုပညာရှင်တွေနဲ့ အဆိုတော်တွေတောင် Graph အကြောင်းကို ရုပ်ရှင်တွေ၊ သီချင်းတွေ ဖန်တီးထားတာမျိုး ရှိကောင်းရှိနေနိုင်ပါတယ်။

ဆိုတော့ ဒီနေရာမှာ ပြောချင်တဲ့ အဓိကအချက်ကတော့ Graph တွေနဲ့ ပတ်သက်ပြီး လေ့လာစရာတွေ အများကြီးရှိတယ်ဆိုတာပါပဲ။ ဒီစာအုပ်ထဲမှာတော့ အကုန်လုံးကိုတော့ လွှမ်းခြုံနိုင်မှာ မဟုတ်ပါဘူး။ ဒါပေမဲ့ Programming လောကမှာ အသုံးများတဲ့ အဓိကအကြောင်းအရာတွေကိုတော့ လွှမ်းခြုံလေ့လာသွားပါမယ်။

Graph ဆိုတာဘာလဲ

Graph တွေဟာ အချက်အလက်တွေကို စုစည်းပြီး စီစဉ်ဖော်ပြဖို့နဲ့၊ တစ်ခုနဲ့တစ်ခု ဘယ်လို ချိတ်ဆက်နေလဲဆိုတာကို နားလည်ဖို့ နည်းလမ်းတစ်ခုပါ။ **ချိတ်ဆက်မှု** ဆိုတဲ့စကားလုံးက အရေးကြီးပါတယ်။ Graph တွေက အချက်အလက်တွေကြားက ဆက်စပ်မှုတွေကို ရှာဖွေပြီး ခွဲခြမ်းစိတ်ဖြာဖို့ ကူညီပေးပါတယ်။ ဥပမာတစ်ခုနဲ့ စကြရအောင်ပါ။ Jerry ဆိုတဲ့ ပျော်ပျော်နေတတ်တဲ့ လူတစ်ဦးရှိတယ် ဆိုပါစို့။ (ပုံ ၁၃-၁)။

ပုံ ၁၃-၁၊ ဂရပ်ဖ်ထဲကို ပထမဆုံးထည့်မယ်အရာ

ဤစာမျက်နှာကို ရည်ရွယ်ချက်ရှိရှိ လွတ်ထားခြင်းဖြစ်သည်။ အပြည့်အစုံဖတ်ရှုရန် ဝယ်ယူရရှိနိုင်ပါသည်။