Fundamentos de PPD
Concorrência e Canais

Exemplos para entendimento e discussão

Fernando Luís Dotti – PUCRS

Exercício valendo nota

1) Notas de trabalhos: 1-CH, 2-MC, 3-P@D
2) Este exercício é parte da nota 1-CH
3) Vale a execução da tarefa

Motivação

- Ensinar/aprender concorrência é um desafio
  - diversas publicações neste sentido
- Assunto não trivial
  - raciocínio temporal; abstração;
- Falta de apoio em geral:
  - ex.: visualização de execuções concorrentes
    - não há ferramenta razoável
- Questões locais:
  - horário da disciplina não favorece.

Motivação

- Experimento didático
  - elementos básicos:
    - estudante ativo ... fisicamente ativo também
    - provocar surgimento de dúvidas sobre conceitos básicos
    - provocar discussão/cooperação visando entendimento
  - questão:
    o uso de encenações pelos estudantes ajuda na compreensão dos conceitos de concorrência?

Detalhamento

- distribuição de casos a grupos
  - grupos de tamanho conforme problema
- discutir em grupo e com professor para compreensão:
  - do problema e do programa concorrente
  - problema e programa fornecidos
- elaborar
  - como explanar aos colegas
  - encenação do funcionamento
- explanar e encenar para a turma dia 21.08
  - nota é pela realização
- avaliação do aprendizado

Exercício – casos a grupos

1) 3 estudantes
2) 6 estudantes
3) 5 estudantes
4) 10 estudantes (ou mais)
5) 10 estudantes (ou mais)
6) 10 estudantes (ou mais)
7) 14 estudantes

Exercício – compreensão em grupo

- diálogo entre colegas
- perguntas ao professor
- execução do programa fornecido
- etc!!!

Elaboração de explanação

- como explicar aos colegas ?
  - Definir o problema em português
  - Em direção à estrutura da solução
    - Fatorar o problema em partes concorrentes
    - Explicitar processos e forma de comunicação (agora canais)
  - Explanar código

Elaboração de encenação

- *explicar execução do programa com uma encenação*
  - *adotar uma metáfora : programa -> encenação*
  - *deixar clara esta metáfora*
- *SUGESTÕES (ou exemplo):*
  - *cada estudante é um processo (main é um processo)*
  - *representar demais elementos importantes:*
    - *canais (classes ? entre alunos)*
    - *conteúdo lido/escrito nos canais (folhas escritas?)*
  - *representar todo ciclo de vida*
    - *main inicia, cria processos, canais, estruturas, ... processos executam, acabam, finaliza...*
  - *criar um processo ... chamar um colega, dar os parâmetros*
  - *criar um canal ... alocar uma classe, uma area do quadro ?*

Exercício 1

Exercício 1

- Somar um vetor *concorrentemente.*
  - Realizar partes de um trabalho concorrentemente permite paralelizar em uma plataforma com diversas CPUs.
- Note o uso das palavras:
  - *concorrência* como coexistência de computações e de
  - *paralelismo* como a "instanciação de concorrência" quando se tem mais de uma CPU em uso simultaneamente.

A cor no código é respectiva à cor do comentário abaixo:

- func sum é instanciada duas vezes no programa.
- Suas variáveis de escopo local são individuais.
- A espera da main acontece na leitura do canal “c”
- É comum lançar go rotinas passando canais de retorno por parâmetro para que a computação possa ocorrer concorrentemente e o resultado recebido posteriormente via este canal.

Tente generalizar este programa para diversos tamanhos de vetor inicial e diferentes números de processos sum

```
package main

import "fmt"

func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        s := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // receive from c

        fmt.Println(x, y, x+y)
}
```

Saída: –5 17 12

Exercício 1

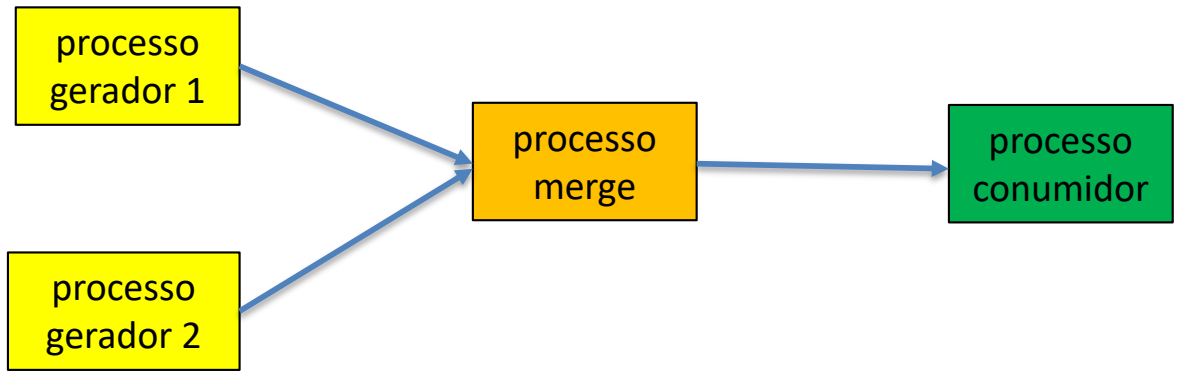
- Explicar
- Encenar
  - use vetor de dez elementos
  - note que o tamanho do canal é zero

Exercício 2

Exercício 2

- Criar grafos de processamento de streams.
  Cada vez mais, dados são gerados continuamente e devem passar por estágios de processamento.
  O uso de canais facilita a modelagem deste tipo de situação.
  - Identifique como isso pode ser feito, veja como processos podem ser usados de forma modular e seus canais formam a interface com os outros.

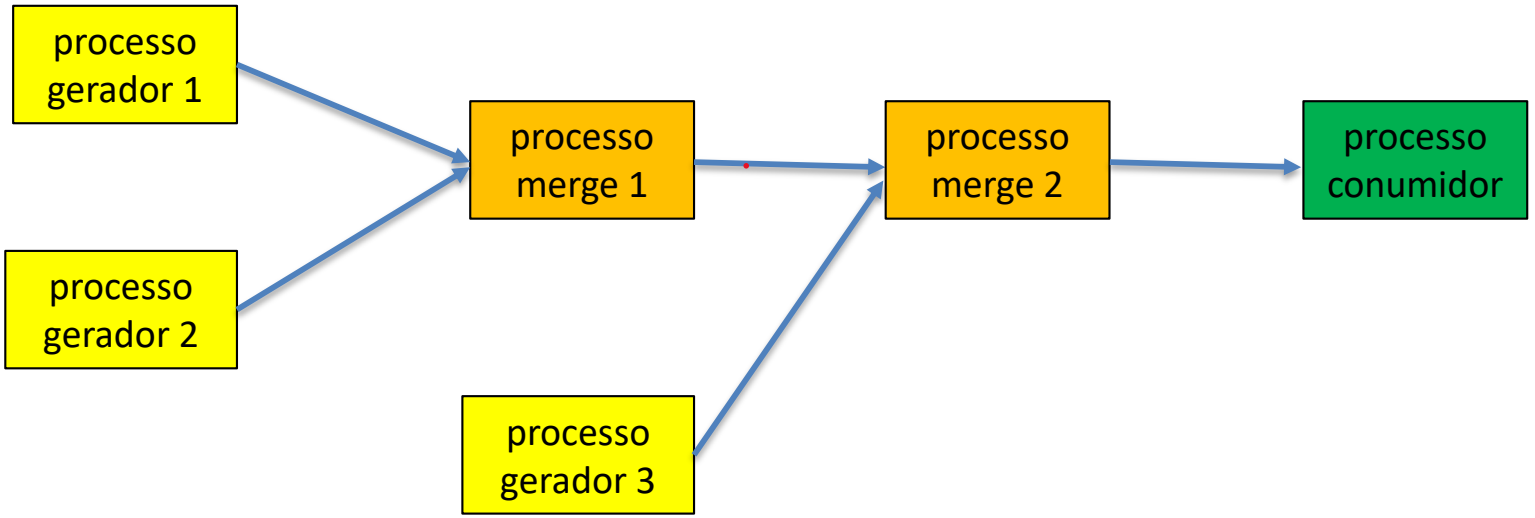
2.(a) Desenvolva um algoritmo que faz o merge de duas sequências de dados (inteiros).
Ele recebe em dois canais de entrada, faz merge e escreve em um canal de saída. O merge é apenas a junção, sem critério, transforma duas correntes de itens em uma.



```go
package main
import (
    "math/rand"
    "time"
)
const N = 200

func geraValores(cOut chan int) {
    for i := 0; i < N; i++ {
        numero := rand.Int31n(2000) + 1
        cOut <- int(numero)
    }
}

func merger(rec1 chan int, rec2 chan int, cOut
chan int) {
    for {
        select {
        case dado := <-rec1:
            println("Gerador 1, dado:", dado)
            cOut <- dado
        case dado := <-rec2:
            println("Gerador 2, dado:", dado)
            cOut <- dado
        }
    }
}
```

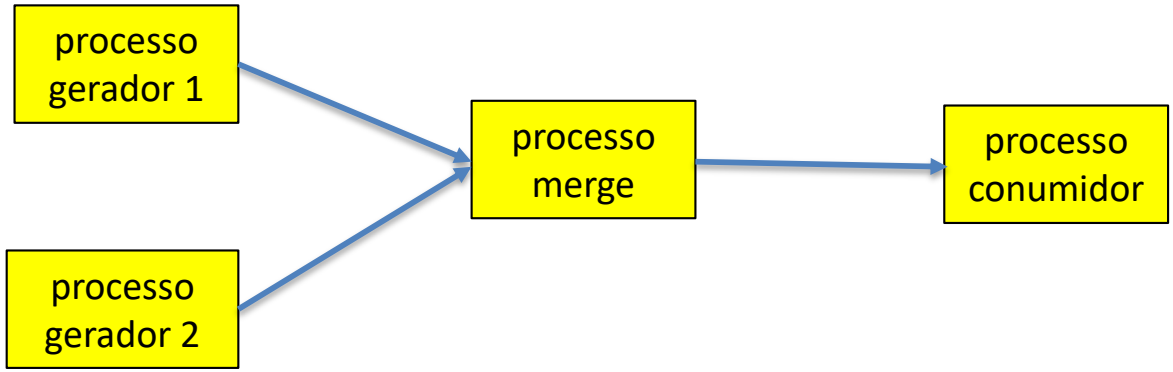
```go
func consumer(rec chan int) {
    nDados := 0
    for {
        if nDados == 2*N {
         fin <- true
         break
           }
        dado := <-rec
        nDados++
        println("Chegou dado", dado)
    }
}

var fin chan bool
func main() {
    rand.Seed(time.Now().UnixNano())
    geradorUm := make(chan int)
    geradorDois := make(chan int)
    geradorSend := make(chan int)
    go consumer(geradorSend)
    go merger(geradorUm, geradorDois,
              geradorSend)
    go geraValores(geradorUm)
    go geraValores(geradorDois)
    fin = make(chan bool)
    <-fin
}
```

2.(b) Monte outra topologia com estes tipos de processos. Ex.:



```
func main() {

    geradorUm := make(chan int)
    geradorDois := make(chan int)
    mergerIntermediarioReceive := make(chan int)
    geradorTres := make(chan int)
    consumerReceive := make(chan int)

    go consumer(consumerReceive)
    go merger(geradorUm, geradorDois, mergerIntermediarioReceive)
    go merger(geradorTres, mergerIntermediarioReceive, consumerReceive)
    go geraValores(geradorUm)
    go geraValores(geradorDois)
    go geraValores(geradorTres)

fin = make(chan bool)
<-fin
}
```

2.(c) Que implicação pode ter a mudança de tamanho nos canais ?



Exercício 2

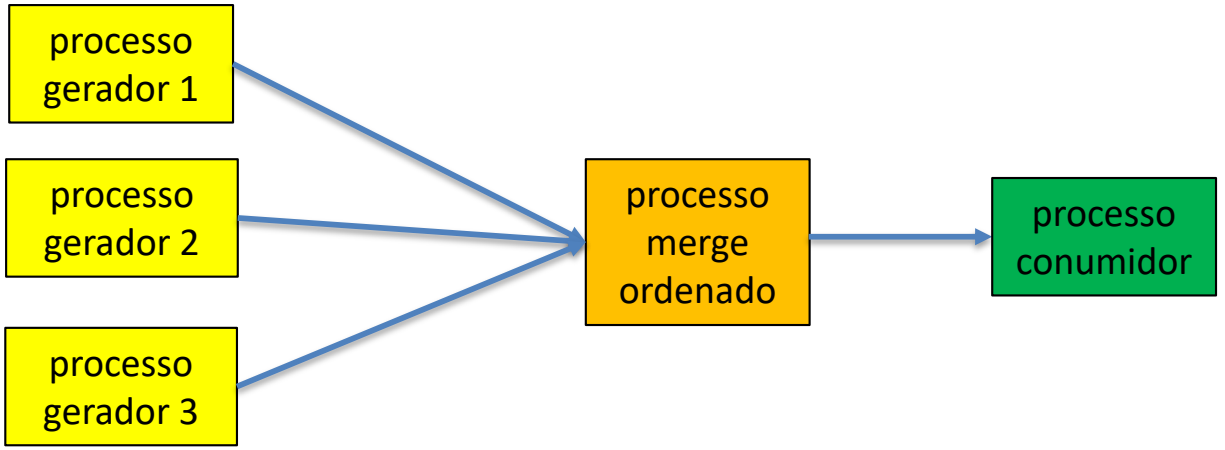
- Explicar e encenar casos a, b, c
  - note que os canais tem tamanho 0
  - passa inteiros

Exercício 3

- Múltiplos ordenados

3. Desenvolva um algoritmo para gerar como saída todos os múltiplos de 2, 3 e 5, em ordem crescente.

Use 4 processos: 3 deles multiplicam pelo fator (2, 3 ou 5) e um deles faz o merge ordenado.



```go
package main

import (
    "fmt"
    "time"
)

func main() {
    var entrada1 chan int = make(chan int)
    var entrada2 chan int = make(chan int)
    var entrada3 chan int = make(chan int)
    var saida chan int = make(chan int)

    go gerador(entrada1, 2)
    go gerador(entrada2, 3)
    go gerador(entrada3, 5)
    go merge(entrada1, entrada2, entrada3, saida)
    go consumer(saida, 1)
    fin := make(chan struct{})
    <-fin
}

func gerador(entrada chan int, multiplicador int) {
    for i := 0; true; i++ {
        time.Sleep(500 * time.Millisecond)
        entrada <- i * multiplicador
    }
}

func consumer(saida chan int, t int) {
    for {
        x := <-saida
        fmt.Print(x, " ")
    }
}
```

```go
func merge(entrada1, entrada2, entrada3, out chan int) {
    v1 := <-entrada1
    v2 := <-entrada2
    v3 := <-entrada3
    for {
        min := v1 // acha o menor
        if v2 < min {
            min = v2
        }
        if v3 < min {
            min = v3
        }
        out <- min
        // le proximo valor da serie do menor
        if min == v1 {
            v1 = <-entrada1
        }
        if min == v2 {
            v2 = <-entrada2
        }
        if min == v3 {
            v3 = <-entrada3
        }
    }
}
```

Exercício 4

Exercício 4

- Cálculo concorrente de primos.
  O cálculo se um número é primo pode ser computacionalmente custoso. Suponha que temos um conjunto de valores e, para cada um deles, devemos avaliar se é primo.
  Fazer isto sequencialmente é demorado. Se o computador tem vários núcleos, por exemplo, isto pode ser acelerado.
- Como isto pode ser feito ?

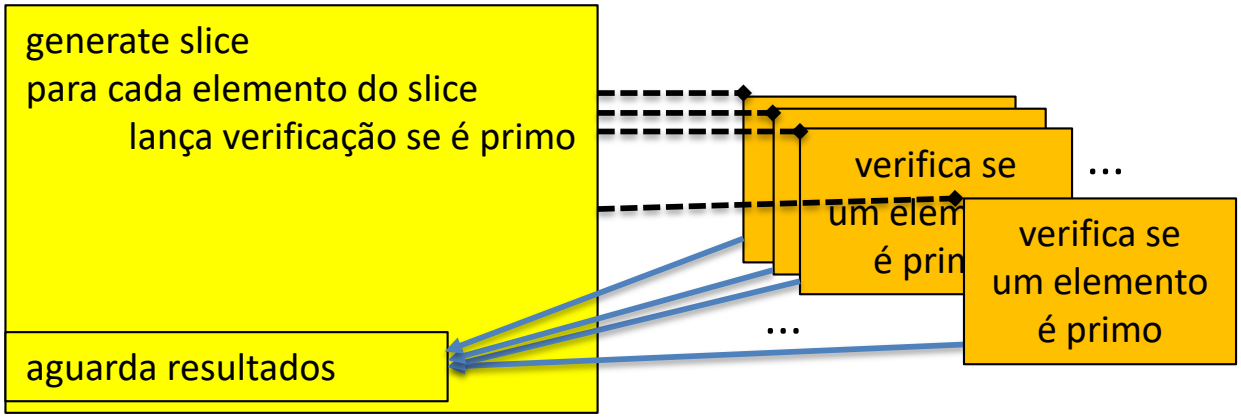
Desenvolva um algoritmo que, dado um conjunto de valores inteiros com N elementos, calcula quantos valores primos existem no conjunto.
A versão sequencial é dada.
Faça uma versão concorrente.

contador de nros primos. versão sequencial.

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
)

const N = 2000

func main() {
    fmt.Println("------ DIFFERENT prime count
IMPLEMENTATIONS -------")
    slice := generateSlice(N)
    p := contaPrimosSeq(slice)
    fmt.Println(" ------ n primos : ", p)
}
```

```go
// Generates a slice of size, size filled with random
numbers
func generateSlice(size int) []int {
    slice := make([]int, size, size)
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < size; i++ {
        slice[i] = rand.Intn(999999999) // -
rand.Intn(9999999999999)
    }
    return slice
}

func contaPrimosSeq(s []int) int {
    result := 0
    for i := 0; i < N; i++ {
        if isPrime(s[i]) {
            fmt.Println(" ------ primos : ", s[i])
            result++
        }
    }
    return result
}

// Is p prime?
func isPrime(p int) bool {
    if p%2 == 0 {
        return false
    }
    for i := 3; i*i <= p; i += 2 {
        if p%i == 0 {
            return false
        }
    }
    return true
}
```

Solução concorrente ?



```go
package main
// ... const e imports identicos aa sequencial
func main() { ...
} // identico a versao sequencial

func generateSlice(size int) []int { ...
} // identico a versao sequencial

func contaPrimos(s []int) int {
    primo := make(chan int) //canal que recebe o index dos numeros primos
    notPrimo := make(chan struct{})
    result := 0

    //inicia N funcoes concorrentes
    for i := 0; i < N; i++ {
        go isPrime(s[i], i, primo, notPrimo)
    }
    // le os retornos, seja de primo ou notPrimo
    for i := 0; i < N; i++ {
        select {
        case i := <-primo:
            fmt.Println(" ------primos: ", s[i])
            result++
        case <-notPrimo:
        }
    }
    return result
}
```

```go
func isPrime(p int, ident int, out chan int,
             notPrimo chan struct{}) {
    //aborta a funcao se nao for primo
    if p%2 == 0 {
        notPrimo <- struct{}{}
        return
    }
    for i := 3; i*i <= p; i += 2 {
        if p%i == 0 {
            notPrimo <- struct{}{}
            return
        }
    }
    // adiciona o index ao canal de primos
    out <- ident
}
```

Exercício 4

- Explicar
- Encenar
  - usar algo para claramente representar os canais (mesa?)
  - notar o uso concorrente destes canais
  - encenar de forma a mostrar o paralelismo possivel (varios processos calculando!)

Exercício 5

Exercício 5

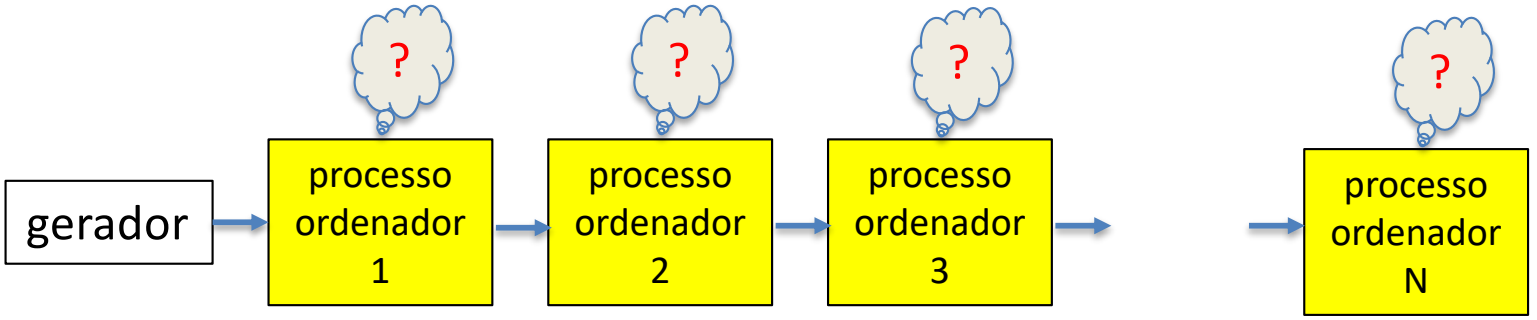
- Pipe Sort
  - o "insertSort" insere valores na posição correta de um conjunto ordenado de valores
  - pipeSort é uma versão concorrente do insert sort. Os valores, ao serem inseridos, ficam ordenados com relação aos já existentes.
    Mas como fazer isto de forma concorrente ?

Pipe sort

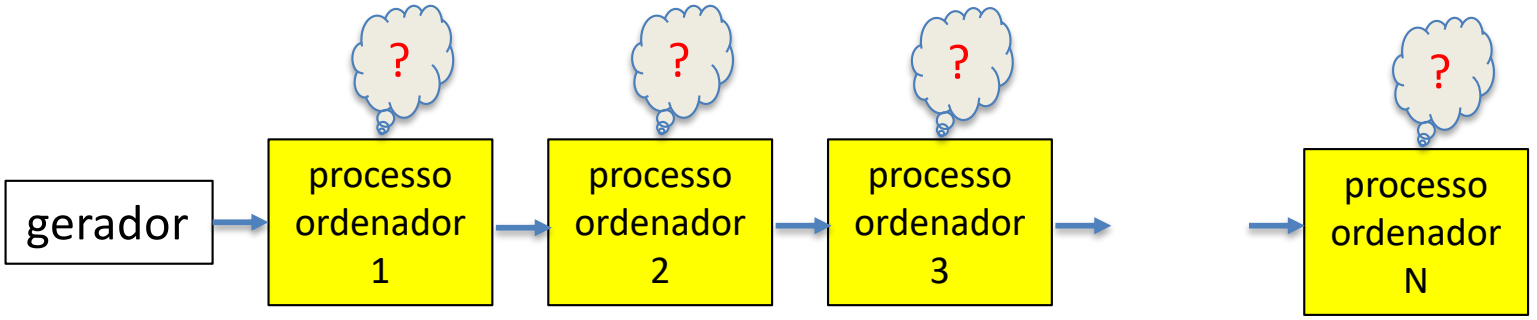
Suponha que um processo gera N valores aleatórios e passa estes valores para um pool de N processos pré-criados. A função do pool de processos é ordenar os valores.
Uma solução possível é:

- forme uma topologia conectando os N processos em linha. Cada processo recebe de um canal de entrada e escreve em um canal de saída.
- cada processo mantém um valor
- cada vez que o processo recebe um valor na entrada, ele
  armazena o valor se for o primeiro recebido
  compara com o valor armazenado, passa o mais alto, mantém o outro armazenado
- assim, todos valores aa direita do processo serão maiores que o que ele mantém. Ao final da inserção de N valores no pool, eles estão ordenados conforme a ordem dos processos do pool.

“Pipe Sort”


?
?
?
?
gerador
processo ordenador 1
processo ordenador 2
processo ordenador 3
processo ordenador N

“Pipe Sort”



(para refletir ... onde está a concorrência?)

```go
func main() {
    fmt.Println("------ Pipe Sort -------")
    var result chan int = make(chan int)
    var canais [N + 1]chan int
    for i := 0; i <= N; i++ { // aloca canais
        canais[i] = make(chan int)
    }
    // Monta pipeline com N processos concorrentes.
    for i := 0; i < N; i++ {
        go cellSorter(i, canais[i], canais[i+1], result, MAX)
    }
    // gera valores aleatorios para o pipeline
    fmt.Println(" ------ input -------")
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < N; i++ {
        valor := rand.Intn(MAX) - rand.Intn(MAX)
        canais[0] <- valor // manda valor para a primeira cellSorter
        fmt.Println(" in ", i, " ", valor)
    }
    canais[0] <- MAX + 1 // depois de mandar N valores, insere sinal de final

    fmt.Println(" ------ resultado -------")
    for i := 0; i < N; i++ {
        fmt.Println(" result ", i, " ", <-result)
    }
    <-canais[N] // le sinal de fim do ultimo processo
}

// ---------------------------------------------------------------------
// cellSorter

func cellSorter(i int, in chan int, out chan int, result chan int, max int) {
    var myVal int
    var undef bool = true
    for {
        n := <-in // rotina reage a uma entrada, altera estado e gera saida
        if n == max+1 { // sinal de final de stream de numeros
            result <- myVal // devolve valor guardado
            out <- n // passa a diante sinal de fim
            break // para
        }
        if undef { // se primeiro valor
            myVal = n // guarda
            undef = false
        } else if n >= myVal { // se valor maior ou igual a este passa adiante senao fica
            out <- n
        } else {
            out <- myVal
            myVal = n
        }
    }
}
```

Exercício 5

- Explicar
- Encenar
  - mostrar claramente a disposição de processos (colegas) e canais (mesas?)
  - encenar de forma a mostrar o comportamento paralelo dos processos

Exercício 6

Merge sort

Enquanto pipe sort é um exemplo de estrutura estática de processos, aqui temos criação dinâmica de processos.
O mergeSort é recursivo. Ele divide o vetor de elementos a ordenar sucessivamente.

Ao chegar nas folhas, ordena cada uma. No retorno da recursão acontece o merge.

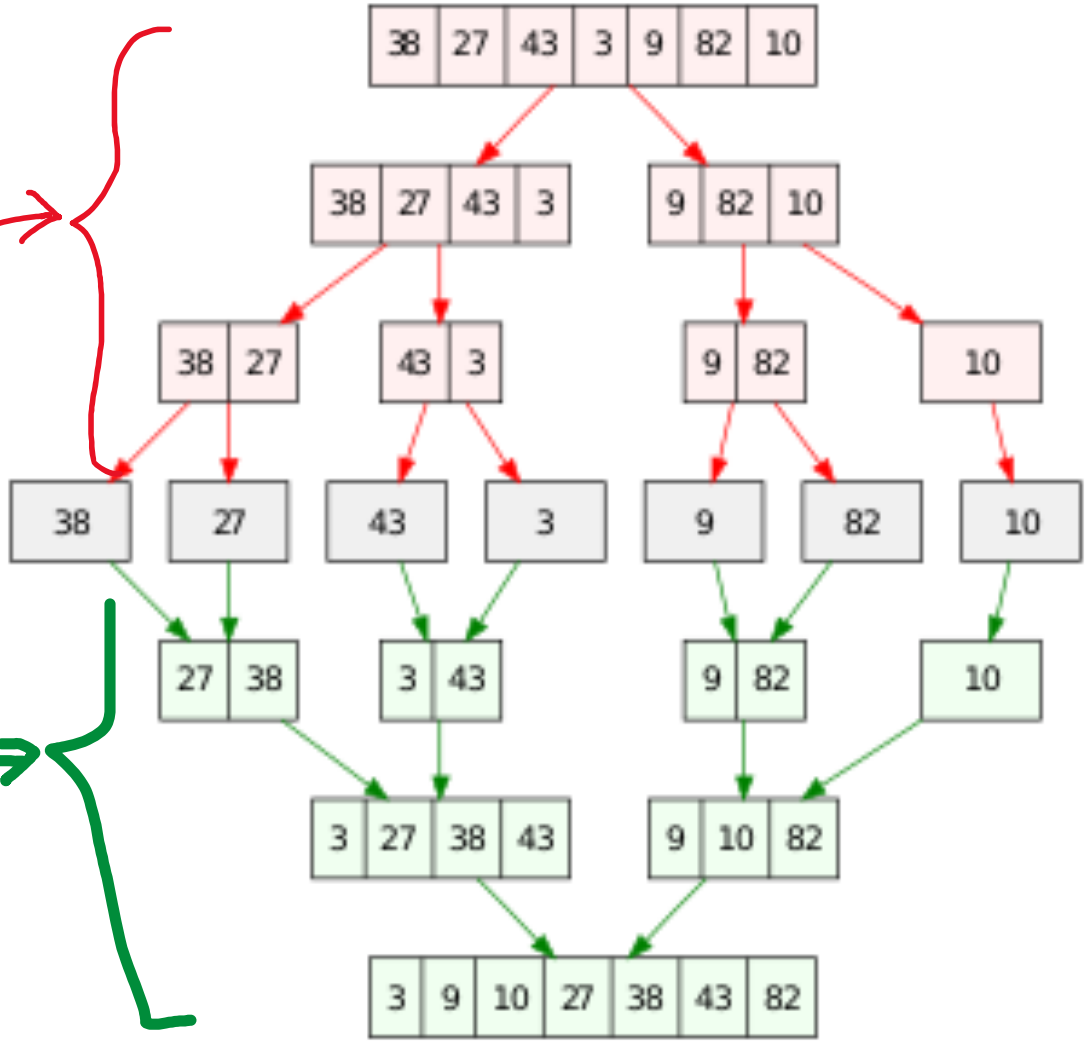


figura de wikipedia

Merge Sort

```
func mergeSortGo(s []int) []int {
    if len(s) > 1 {
        middle := len(s) / 2
        return merge(mergeSortGo(s[:middle]),
                     mergeSortGo(s[middle:]) )
    }
    return s
}


// VERSAO SEQUENCIAL
```

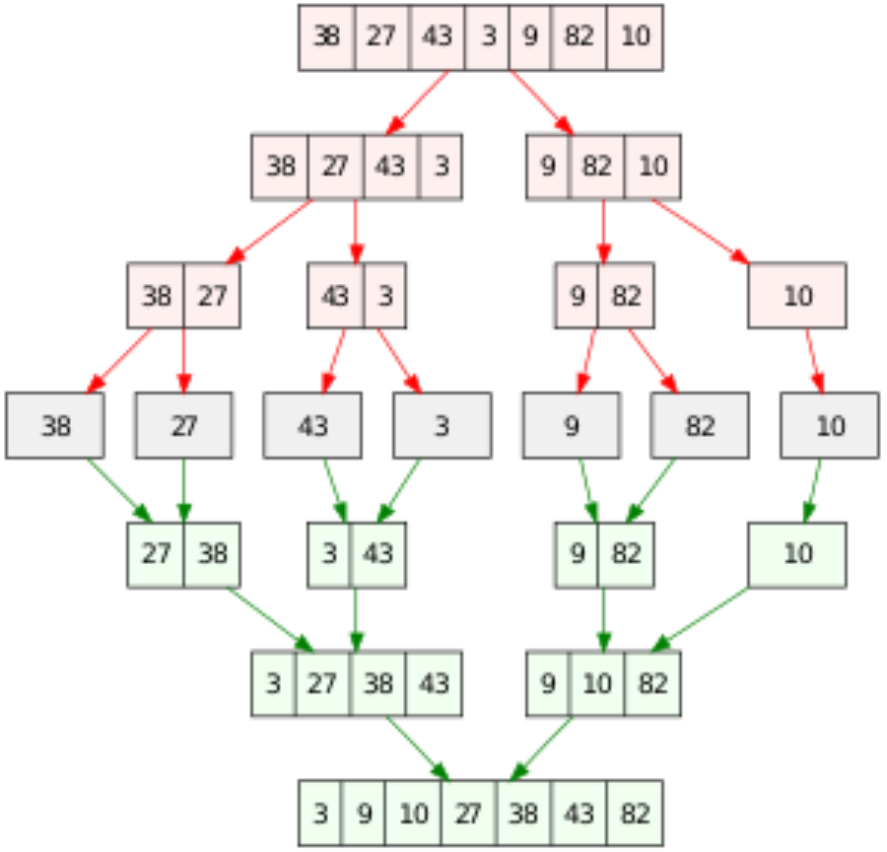


figura de wikipedia

Merge Sort

como usar concorrência ?

```
func mergeSortGo(s []int) []int {
    if len(s) > 1 {
        middle := len(s) / 2
        return merge(mergeSortGo(s[:middle]),
                     mergeSortGo(s[middle:]) )
    }
    return s
}
```

Merge Sort Sequencial

```go
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

func main() {
    slice := generateSlice(20)
    mergeSortGo(slice)
}

// Generates a slice of size,
// size filled with random numbers
func generateSlice(size int) []int {
    slice := make([]int, size, size)
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < size; i++ {
        slice[i] = rand.Intn(999) -
                   rand.Intn(999)
    }
    return slice
}
```

```go
// ------------------------------------
// mergeSortGo usa facilidades de slices

func mergeSortGo(s []int) []int {
    if len(s) > 1 {
        middle := len(s) / 2
        return merge(mergeSortGo(s[:middle]),
                     mergeSortGo(s[middle:]))
    }
    return s
}
```

Merge Sort Concorrente

```go
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

func main() {
    slice := generateSlice(20)
    mergeSortGoPar(slice)
}

// Generates a slice of size,
// size filled with random numbers
func generateSlice(size int) []int {
    slice := make([]int, size, size)
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < size; i++ {
        slice[i] = rand.Intn(999) -
                   rand.Intn(999)
    }
    return slice
}
```

```go
func mergeSortGoPar(s []int) []int {

    if len(s) > 1 {
        middle := len(s) / 2

        var s1 []int
        var s2 []int
        c := make(chan struct{}, 2)

        go func() {
            s1 = mergeSortGoPar(s[middle:])
            c <- struct{}{}
        }()

        go func() {
            s2 = mergeSortGoPar(s[:middle])
            c <- struct{}{}
        }()

        <-c
        <-c

        return merge(s1, s2)

    }
    return s
}
```

Merge Sort Concorrente

Observações:

- criação dinâmica de
  - processos
  - canais

Exercício 6

- Explicar
- Encenar
  - criacao dinamica de processos
    E
  - criacao dinamica de canais

  - encenar de forma a mostrar o comportamento paralelo dos processos
  - deixar claro quais processos compartilham quais canais

Exercício 7

Caminhamento em árvore

Muitas vezes pode ser importante realizar buscas e cômputos concorrentes sobre estruturas. Por exemplo, vimos a soma dos valores de um vetor.

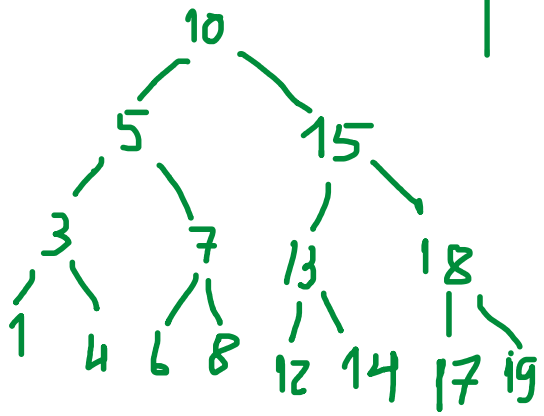
Imagine uma árvore binária de valores inteiros. E você tem operações que devem percorrer a mesma. Voce quer fazer isto de forma concorrente.

- Como estruturar isto ?

Caminhamento em árvore

- Como estruturar isto ?
  - trocamos chamada recursiva de função por criação de processo
  - trocamos retorno da funçao por canal de retorno
  - assim, as subárvores da esquerda e da direita podem ser percorridas concorrentemente

```go
package main
import (
    "fmt"
)
type Nodo struct {
    v int
    e *Nodo
    d *Nodo
}
func caminhaERD(r *Nodo) {
    if r != nil {
        caminhaERD(r.e)
        fmt.Print(r.v, ", ")
        caminhaERD(r.d)
    }
}
// -------- SOMA ----------
// soma sequencial recursiva
func soma(r *Nodo) int {
    if r != nil {
        //fmt.Print(r.v, ", ")
        return r.v + soma(r.e) + soma(r.d)
    }
    return 0
}
```

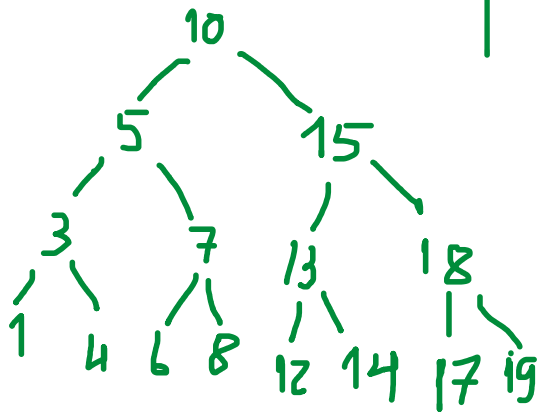



```go
func main() {
// ---------  vamos criar uma arvore
    root := &Nodo{v: 10,
        e: &Nodo{v: 5,
            e: &Nodo{v: 3,
                e: &Nodo{v: 1, e: nil, d: nil},
                d: &Nodo{v: 4, e: nil, d: nil}},
            d: &Nodo{v: 7,
                e: &Nodo{v: 6, e: nil, d: nil},
                d: &Nodo{v: 8, e: nil, d: nil}}},
        d: &Nodo{v: 15,
            e: &Nodo{v: 13,
                e: &Nodo{v: 12, e: nil, d: nil},
                d: &Nodo{v: 14, e: nil, d: nil}},
            d: &Nodo{v: 18,
                e: &Nodo{v: 17, e: nil, d: nil},
                d: &Nodo{v: 19, e: nil, d: nil}}}}
    fmt.Println()
    fmt.Print("Valores na árvore: ")
    caminhaERD(root)
    fmt.Println()
    fmt.Println("Soma: ", soma(root))
    fmt.Println("SomaConc: ", somaConc(root))
}
```

?

Como fazer o caminhamento da soma serconcorrente ?

```go
package main
import (
    "fmt"
)
type Nodo struct {
    v int
    e *Nodo
    d *Nodo
}
func caminhaERD(r *Nodo) {
    if r != nil {
        caminhaERD(r.e)
        fmt.Print(r.v, ", ")
        caminhaERD(r.d)
    }
}
// -------- SOMA ----------
// soma sequencial recursiva
func soma(r *Nodo) int {
    if r != nil {
        //fmt.Print(r.v, ", ")
        return r.v + soma(r.e) + soma(r.d)
    }
    return 0
}
```

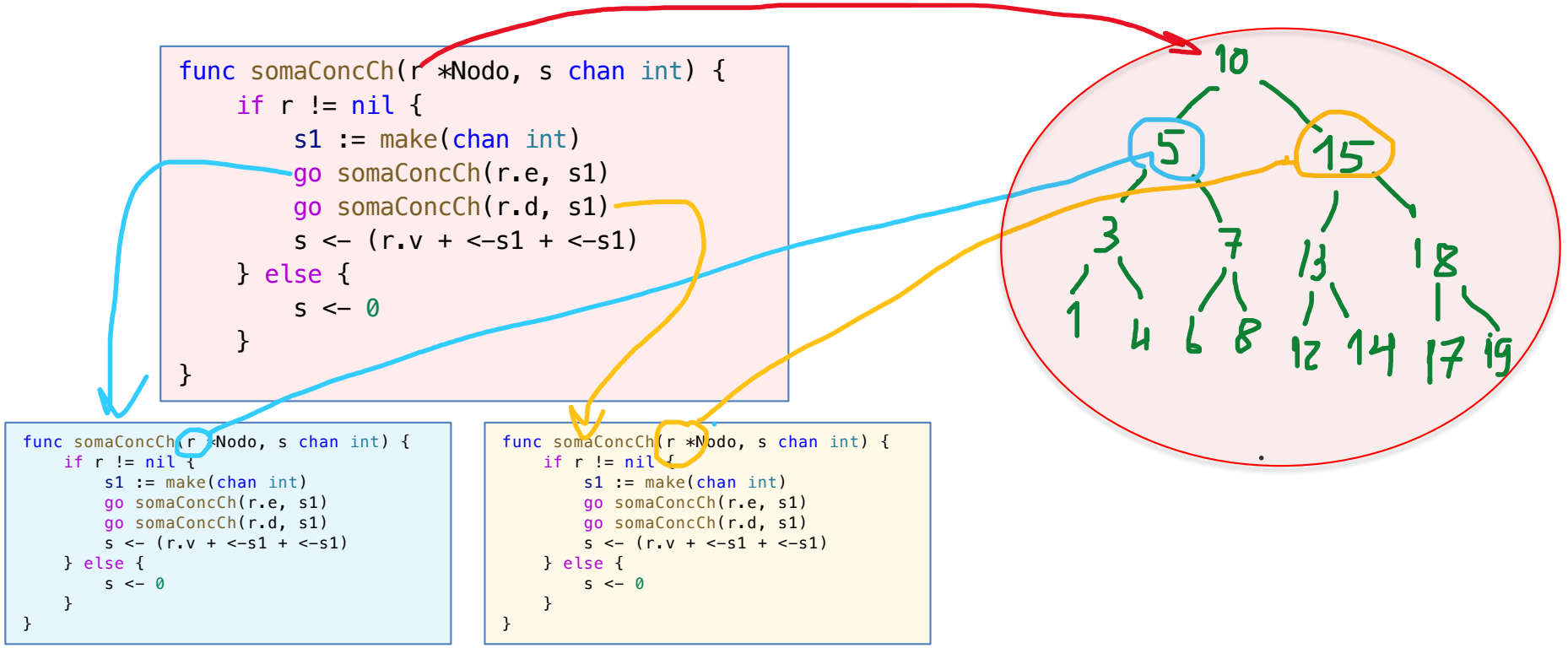



```go
func main() {
// ---------  vamos criar uma arvore
    root := &Nodo{v: 10,
        e: &Nodo{v: 5,
            e: &Nodo{v: 3,
                e: &Nodo{v: 1, e: nil, d: nil},
                d: &Nodo{v: 4, e: nil, d: nil}},
            d: &Nodo{v: 7,
                e: &Nodo{v: 6, e: nil, d: nil},
                d: &Nodo{v: 8, e: nil, d: nil}}},
        d: &Nodo{v: 15,
            e: &Nodo{v: 13,
                e: &Nodo{v: 12, e: nil, d: nil},
                d: &Nodo{v: 14, e: nil, d: nil}},
            d: &Nodo{v: 18,
                e: &Nodo{v: 17, e: nil, d: nil},
                d: &Nodo{v: 19, e: nil, d: nil}}}}
    fmt.Println()
    fmt.Print("Valores na árvore: ")
    caminhaERD(root)
    fmt.Println()
    fmt.Println("Soma: ", soma(root))
    fmt.Println("SomaConc: ", somaConc(root))
}
```

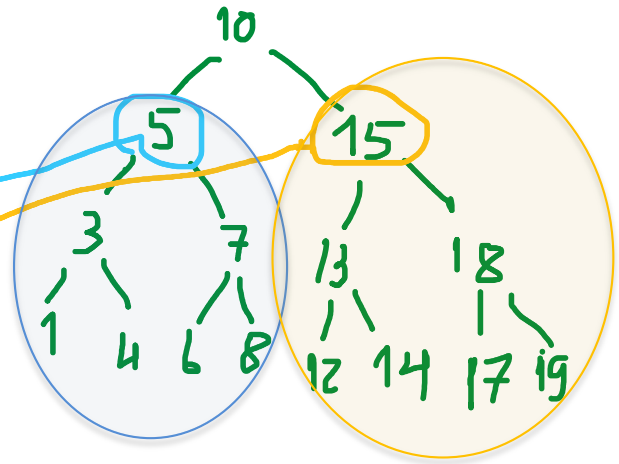
?

```go
// Funcao "wraper" tem mesma interface
//  da sequenciala - retorna valor.
// Internamente dispara recursao com
//  somaConcCh usando canais
func somaConc(r *Nodo) int {
    s := make(chan int)
    go somaConcCh(r, s)
    return <-s
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```


func somaConcCh(r *Nodo, s chan int) {
if r != nil {
s1 := make(chan int)
go somaConcCh(r.e, s1)
go somaConcCh(r.d, s1)
s <- (r.v + <-s1 + <-s1)
} else {
s <- 0
}
}
10
5
15
3
7
13
18
1
4
6
8
12
14
17
19

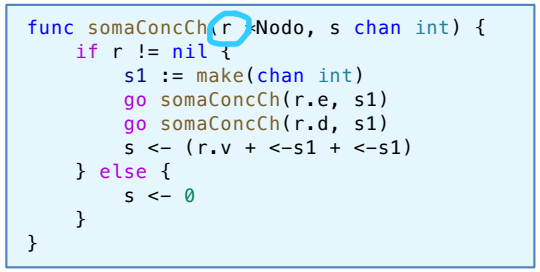
```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

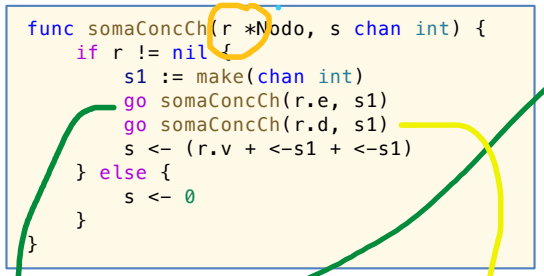


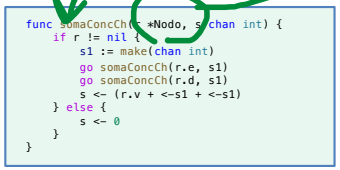

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

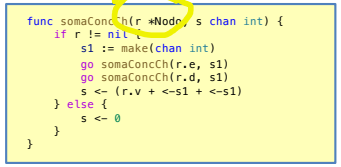
```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

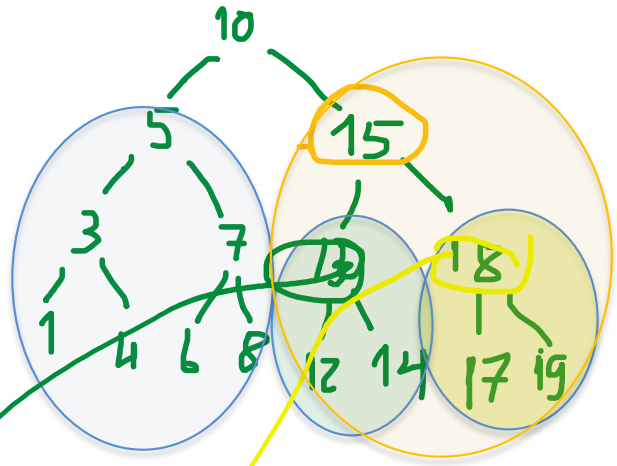
```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```



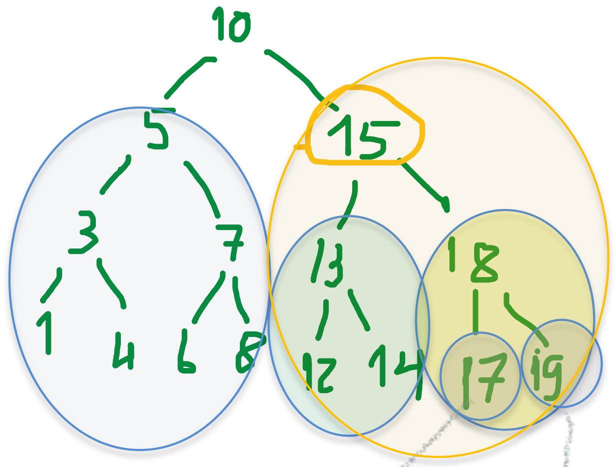








```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```




```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

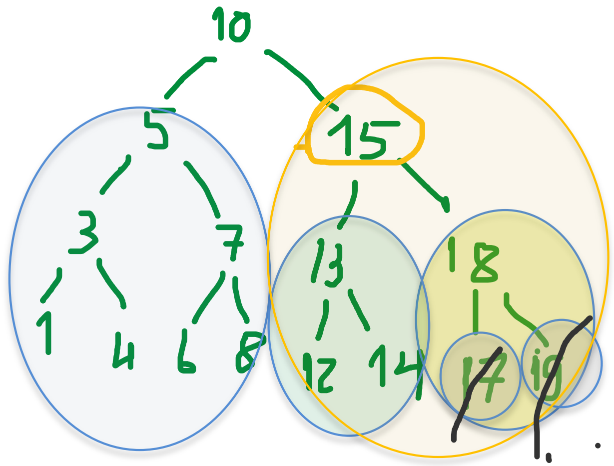
```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
	if r != nil {
		s1 := make(chan int)
		go somaConcCh(r.e, s1)
		go somaConcCh(r.d, s1)
		s <- (r.v + <-s1 + <-s1)
	} else {
		s <- 0
	}
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```




```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

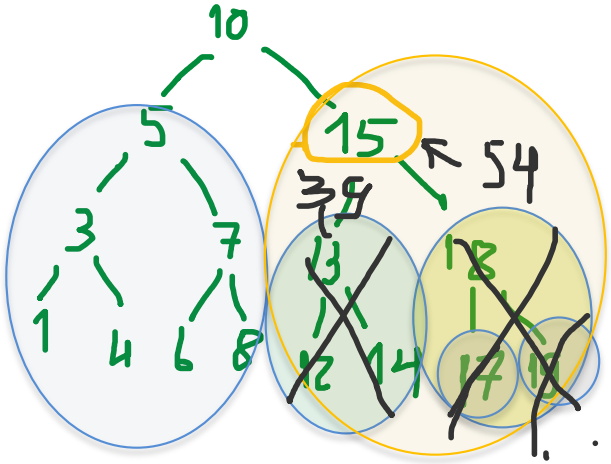
```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```


17 + 19


```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```




```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

39 54

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

13 + 12 + 14

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

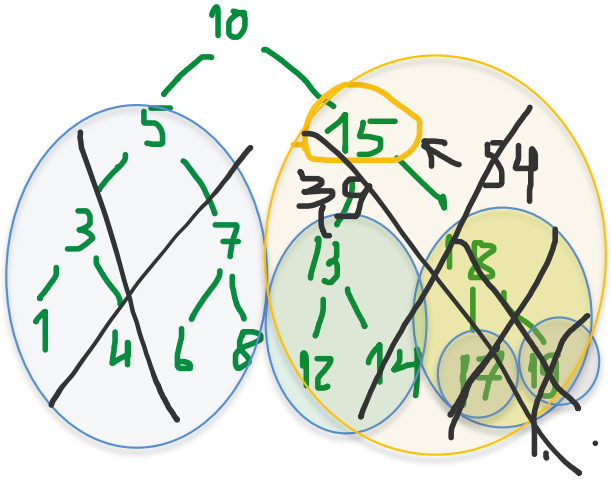
18 + 17 + 19

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

10 33 108




```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

5 7 21

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

15 39 54

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

13 + 12 + 14

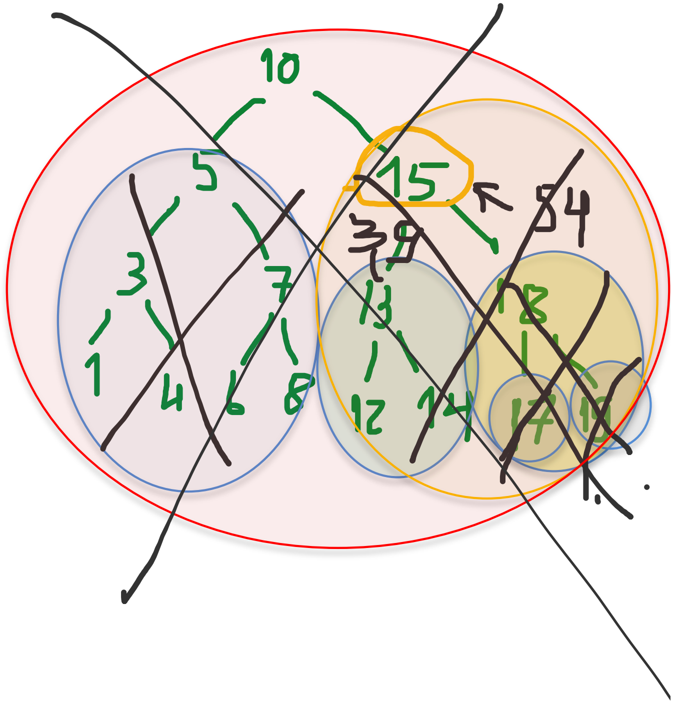
```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

18 + 17 + 19

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```




```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

```go
func somaConcCh(r *Nodo, s chan int) {
    if r != nil {
        s1 := make(chan int)
        go somaConcCh(r.e, s1)
        go somaConcCh(r.d, s1)
        s <- (r.v + <-s1 + <-s1)
    } else {
        s <- 0
    }
}
```

Exercício 7

- Explicar
- Encenar
  - note que os varios processos terão *referencias a nodos da árvore*
    a partir de onde computam
  - represente esta árvore de forma que os vários processos possam referenciar (no quadro ?)
  - os processos (colegas) tem canais entre si. estes canais devem ficar claros

Caminhamento em árvore

- como voce faria uma pesquisa concorrente para retornar todos valores impares da árvore ?

Checkpoint

Aqui espera-se que você saiba:

Lançar processos concorrentes e esperar sua finalização

Sincronizar/comunicar processos com canais sincronizantes e bufferizados

Fazer escolhas não determinísticas de sincronização, para tratamento de diferentes eventos

Que quanto mais sincronização, menor é o nível de concorrência

Usar Go para construir programas concorrentes simples