1. Considere uma árvore de pesquisa binária que armazena números inteiros. A árvore é representada usando nós que possuem um atributo **value** (que armazena o valor inteiro), um atributo **left** para a subárvore esquerda e um atributo **right** para a subárvore direita. Suponha que a árvore esteja inicialmente vazia e que várias threads possam acessar a árvore simultaneamente. Escreva pseudocódigo para implementar as operações de inserção e pesquisa da árvore de maneira segura para thread usando semáforos ou variáveis condicionais. Você pode usar o modelo abaixo, como referência para sua implementação.

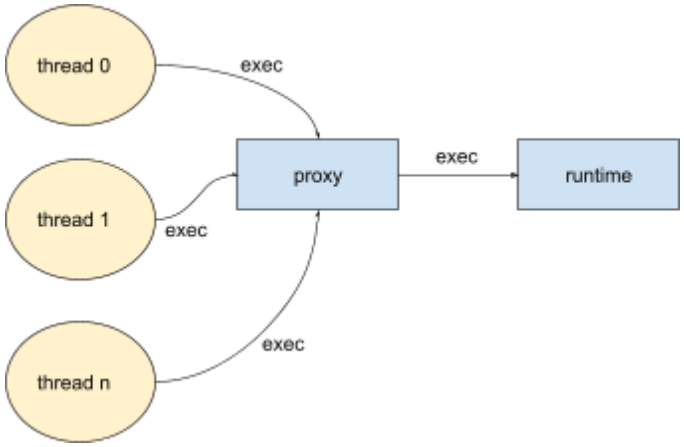
```
class BinarySearchTree()

    void func insert(int valueToInsert)
    boolean func search(int   valueToSearch)

    class Node(int value)
        this.value = value
        this.left = None
        this.right = None
```

Possível implementação:

```
class BinarySearchTree
    Semaphore mutex = 1

    void insert(int valueToInsert)
        Node newNode = Node(valueToInsert)
            mutex.wait() // Aguarda permissão para acessar a árvore
        if root is None
            root = newNode
        else
            insertNode(root, newNode)
        mutex.signal() // Libera o acesso à árvore

    void insertNode(Node currentNode, Node newNode)
        if newNode.value < currentNode.value
            if currentNode.left is None
```

```
                    currentNode.left = newNode
                else
                    insertNode(currentNode.left, newNode)
            else if newNode.value > currentNode.value
                if currentNode.right is None
                    currentNode.right = newNode
                else
                        insertNode(currentNode.right,
newNode)

        boolean search(int valueToSearch)
                 mutex.wait() // Aguarda permissão para
acessar a árvore
                     boolean result = searchNode(root,
valueToSearch)
            mutex.signal() // Libera o acesso à árvore
            return result

             boolean searchNode(Node currentNode, int
valueToSearch)
            if currentNode is None
                return false
            else if valueToSearch == currentNode.value
                return true
            else if valueToSearch < currentNode.value
                     return searchNode(currentNode.left,
valueToSearch)
            else
                     return searchNode(currentNode.right,
valueToSearch)

    class Node
        int value
        Node left
        Node right

        constructor Node(value)
            this.value = value
            this.left = None
            this.right = None
```

2. Muitos sistemas implementam uma arquitetura que considera um componente proxy que intercepta chamadas para um componente que é responsável por processar requisição (vamos chamar este último de runtime), tal como no esquema abaixo.



Nesse esquema, threads são criadas (p.ex thread 0, 1 .. n) e estas threads executam a função **exec** do código do proxy. O **proxy** tem somente uma função: controlar o repasse de execuções para a **runtime**.

Sua implementação deve considerar no mínimo as seguintes funções. Você **NÃO**
deve implementar as funções destacadas em negrito.

Proxy	Runtime
void exec(Request req)	**void recover()**
	void exec(Request req)
boolean isAvailable()	
boolean runtimeIsHealthy()	

Par Atente para os seguintes requisitos:

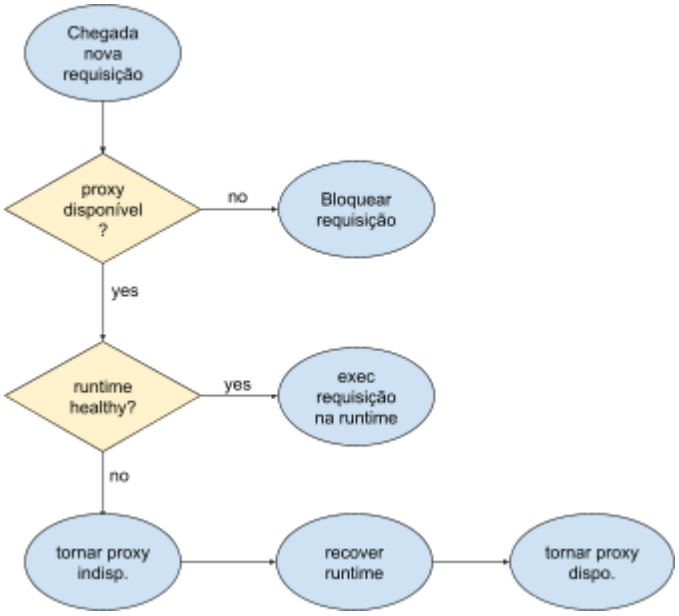
- Threads são criadas externamente ao seu código. Essas threads chamam a função **request** do **proxy;**
- Requests que chegam no **proxy,** devem ser repassados para a **runtime** somente se o proxy estiver

disponível (função **isAvailable()**) e se o runtime estiver **healthy**;

- Se o proxy detectar que **runtime** não está **healthy**, o **proxy** mudar para seu status para **indisponível**. O valor de sua função isAvailable deve ser falso. Também, o **proxy** deve chamar a função **recover** da **runtime**. Ao fim da execução desta função, o **runtime** estará **healthy** novamente e o **proxy** volta para o estado disponível;

- Só uma thread por vez deve executar **exec** na runtime;
- Caso uma thread não possa executar **exec** na **runtime** esse fluxo não deve ser descartado. A thread deve bloquear até poder prosseguir.

Abaixo, uma versão visual de parte destes requisitos. Não necessariamente, você precisa implementar o código nesta sequência.



Você pode criar funções auxiliares para a implementação da abstração Proxy. O controle de concorrência pode ser feito tanto com semáforos quanto por variáveis condicionais. É importante inicializar esses tipos corretamente, p.ex no construtor de suas entidades ou na função **main**, se for o caso.

Caso queira que o proxy mantenha também suas threads, use a seguinte função para criar as threads.

int new_thread(function f, arg)

onde **f** é o nome da função que a thread recém criada, arg é um argumento de qualquer tipo que pode ser passado para função **f**. Além disso, a função **new_thread** retorna um inteiro que identifica a thread recém criada.

Possível implementação:

```
class Proxy
    boolean available = true
    Semaphore mutex = 1
    Condition condition

    constructor(runtime)
        this.runtime = runtime
        this.condition = new Condition()

    method exec(Request req)
        mutex.acquire()
        while not available
            condition.wait()
        mutex.release()

        if not runtime.isHealthy()
            mutex.acquire()
            available = false
            mutex.release()
            runtime.recover()
            mutex.acquire()
            available = true
            condition.signal()
            mutex.release()

        runtime.exec(req)

    method isAvailable()
        mutex.acquire()
        boolean result = available
        mutex.release()
        return result
```

3. Implementação concorrente da LinkedList

   Considere a implementação de uma LinkedList que armazena números inteiros. Deseja-se criar uma versão concorrente dessa LinkedList, onde as operações de inserção, busca e remoção podem ser realizadas de forma segura por várias threads simultaneamente. Para isso, você deve utilizar semáforos para controlar o acesso às operações da LinkedList.

   Sua implementação deve garantir que múltiplas threads possam realizar operações de inserção, busca e remoção de forma segura, evitando problemas como condição de corrida e inconsistências na estrutura da LinkedList.

   Você pode utilizar semáforos ou outras estruturas de controle de concorrência, como mutexes, para garantir a exclusão mútua e sincronização adequada entre as threads.

   Apresente sua solução em pseudocódigo, incluindo a inicialização dos semáforos ou outras estruturas necessárias para controlar a concorrência nas operações da LinkedList.

   Certifique-se de considerar os casos em que uma thread pode ser bloqueada ao aguardar acesso à LinkedList e como a estrutura da LinkedList deve ser manipulada de forma consistente durante as operações concorrentes.

   Possível implementação:

```
class ConcurrentLinkedList
    Node head
    Semaphore insertSemaphore = 1
    Semaphore searchSemaphore = 1
    Semaphore removeSemaphore = 1

    class Node
```

```
    int value
    Node next

void insert(int value)
    insertSemaphore.wait()
    Node newNode = new Node(value)
    newNode.next = head
    head = newNode
    insertSemaphore.signal()

boolean search(int value)
    searchSemaphore.wait()
    Node current = head
    while current != null
        if current.value == value
            searchSemaphore.signal()
            return true
        current = current.next
    searchSemaphore.signal()
    return false

boolean remove(int value)
    removeSemaphore.wait()
    if head == null
        removeSemaphore.signal()
        return false

    if head.value == value
        head = head.next
        removeSemaphore.signal()
        return true

    Node prev = head
    Node current = head.next
    while current != null
        if current.value == value
            prev.next = current.next
            removeSemaphore.signal()
            return true
        prev = current
        current = current.next
```

```
            removeSemaphore.signal()
            return false
```

4. Implementação concorrente da Hashtable

   Considere a implementação de uma Hashtable que armazena pares de chave-valor, onde as chaves são strings e os valores são inteiros. Deseja-se criar uma versão concorrente dessa Hashtable, onde as operações de inserção, busca e remoção podem ser realizadas de forma segura por várias threads simultaneamente. Para isso, você deve utilizar semáforos para controlar o acesso às operações da Hashtable.

   Sua implementação deve garantir que múltiplas threads possam realizar operações de inserção, busca e remoção de forma segura, evitando problemas como condição de corrida e inconsistências na estrutura da Hashtable.

   Você pode utilizar semáforos ou outras estruturas de controle de concorrência, como mutexes, para garantir a exclusão mútua e sincronização adequada entre as threads.

   Apresente sua solução em pseudocódigo, incluindo a inicialização dos semáforos ou outras estruturas necessárias para controlar a concorrência nas operações da Hashtable.

   Certifique-se de considerar os casos em que uma thread pode ser bloqueada ao aguardar acesso à Hashtable e como a estrutura da Hashtable deve ser manipulada de forma consistente durante as operações concorrentes.

   Possível implementação:

```
class ConcurrentHashtable
    Hashtable hashtable
    Semaphore insertSemaphore = 1
    Semaphore searchSemaphore = 1
    Semaphore removeSemaphore = 1
```

```
void insert(string key, int value)
    insertSemaphore.wait()
    hashtable.insert(key, value)
    insertSemaphore.signal()

int search(string key)
    searchSemaphore.wait()
    int value = hashtable.search(key)
    searchSemaphore.signal()
    return value

boolean remove(string key)
    removeSemaphore.wait()
    boolean removed = hashtable.remove(key)
    removeSemaphore.signal()
    return removed
```