برنامه نویسی multi\_ Threading در زبان Rust را با ذکر مثال ساده توضیح دهید :

Use std::thread; // برای کار با نخ‌ها

Use std::sync::{Arc, Mutex}; // برای اشتراک‌گذاری ایمن داده‌ها

Fn main() {

Let counter = Arc::new(Mutex::new(0)); // ایجاد یک شمارنده با Mutex و Arc

Let mut handles = vec![]; // برای ذخیره دستگیره‌های نخ‌ها

For \_ in 0..10 {

Let counter = Arc::clone(&counter); // کلون کردن Arc برای استفاده در هر نخ

Let handle = thread::spawn(move || {

Let mut num = counter.lock().unwrap(); // قفل کردن Mutex برای دسترسی ایمن به شمارنده

\*num += 1; // افزایش شمارنده

});

Handles.push(handle); // ذخیره کردن دستگیره نخ

}

For handle in handles {

Handle.join().unwrap(); // منتظر تمام نخ‌ها می‌مانیم

}

Println!("Final count: {}", \*counter.lock().unwrap()); // نمایش نتیجه نهایی

}

برنامه نویسی parallel programming در زبان Rust را با ذکر مثال توضیح دهید

Use std::thread;

Fn main() {

Let numbers = vec![1, 2, 3, 4, 5]; // آرایه‌ای از اعداد

Let mut handles = vec![]; // برای ذخیره دستگیره‌های نخ‌ها

For &num in &numbers {

// ایجاد نخ برای هر عدد

Let handle = thread::spawn(move || {

// محاسبه مربع عدد و چاپ آن

Let square = num \* num;

Println!("Square of {} is {}", num, square);

});

Handles.push(handle); // ذخیره کردن دستگیره نخ

}

For handle in handles {

Handle.join().unwrap(); // منتظر تمام نخ‌ها می‌مانیم

}

}

[dependencies]

Rayon = "1.7"

Use rayon::prelude::\*;

Fn main() {

Let numbers = vec![1, 2, 3, 4, 5]; // آرایه‌ای از اعداد

Let squares: Vec<i32> = numbers

.par\_iter() // تبدیل به ایتراور موازی

.map(|&num| num \* num) // محاسبه مربع

.collect(); // جمع آوری نتایج

For (num, square) in numbers.iter().zip(squares.iter()) {

Println!("Square of {} is {}", num, square);

}

}

کتابخانه های machine Learning در زبان Rust را نام ببرید و یک مثال ساده بزنید

[dependencies]

Ndarray = "0.15.4"

Rustlearn = "0.8.1"

Use ndarray::Array2;

Use rustlearn::neighbors::k\_neighbors\_classifier::KneighborsClassifier;

Use rustlearn::prelude::\*;

Fn main() {

// داده‌های تمرینی: سن، قد

Let x\_train: Array2<f32> = Array2::from\_shape\_vec((6, 2), vec![

1.0, 2.0,

1.5, 1.5,

2.0, 1.0,

3.0, 3.0,

3.5, 2.0,

4.0, 4.0,

]).unwrap();

// برچسب‌های داده‌های تمرینی: 0 برای کلاس اول و 1 برای کلاس دوم

Let y\_train = Array2::from\_shape\_vec((6, 1), vec![

0,

0,

0,

1,

1,

1,

]).unwrap();

// ایجاد مدل K-NN با k=3

Let mut knn = KneighborsClassifier::fit(&x\_train, &y\_train).unwrap();

Knn.set\_n\_neighbors(3);

Let x\_test: Array2<f32> = Array2::from\_shape\_vec((2, 2), vec![

2.5, 2.5, // داده‌ای برای پیش‌بینی

4.5, 4.5,

]).unwrap();

// پیش‌بینی با مدل

Let y\_pred = knn.predict(&x\_test).unwrap();

// نمایش نتایج

Println!("{:?}", y\_pred);

}

Lazy Loading چیست با ذکر مثال در زبان Rust توضیح دهید :

Lazy Loading یک الگوی طراحی است که در آن بارگذاری داده‌ها یا منابع تنها در زمانی انجام می‌شود که به آن‌ها نیاز است. این الگو به خصوص برای بهینه‌سازی مصرف حافظه و زمان بارگذاری استفاده می‌شود. در الگوی بارگذاری تنبل، مقدار یا شیء (object) فقط در زمان واقعی نیاز به آن بارگذاری یا محاسبه می‌شود، در نتیجه به جلوگیری از بارگذاری غیر ضروری داده‌ها کمک می‌کند

Use std::fs;

Use std::path::Path;

Struct LazyLoader {

File\_path: String,

Content: Option<String>,

}

Impl LazyLoader {

Fn new(file\_path: &str) -> Self {

LazyLoader {

File\_path: String::from(file\_path),

Content: None,

}

}

Fn load(&mut self) -> &String {

If self.content.is\_none() {

// بارگذاری محتوای فایل تنها در صورت نیاز

Let content = fs::read\_to\_string(&self.file\_path).expect("Unable to read file");

Self.content = Some(content);

}

Self.content.as\_ref().unwrap() // برگشتن به محتوای بارگذاری شده

}

}

Fn main() {

Let mut loader = LazyLoader::new("example.txt"); // فرض کنید، این فایل وجود دارد

// محتوای فایل تنها زمانی بارگذاری می‌شود که به آن نیاز داشته باشیم

Println!("Content of file: {}", loader.load());

}

ساختمان داده Binary SearchTree را در زبان Rust پیاده سازی نمایید :

#[derive(Debug)]

Struct TreeNode {

Value: i32,

Left: Option<Box<TreeNode>>,

Right: Option<Box<TreeNode>>,

}

Impl TreeNode {

Fn new(value: i32) -> TreeNode {

TreeNode {

Value,

Left: None,

Right: None,

}

}

}

Struct BinarySearchTree {

Root: Option<Box<TreeNode>>,

}

Impl BinarySearchTree {

Fn new() -> BinarySearchTree {

BinarySearchTree { root: None }

}

Fn insert(&mut self, value: i32) {

Self.root = Self::insert\_node(self.root.take(), value);

}

Fn insert\_node(node: Option<Box<TreeNode>>, value: i32) -> Option<Box<TreeNode>> {

If let Some(mut n) = node {

If value < n.value {

n.left = Self::insert\_node(n.left, value);

} else {

n.right = Self::insert\_node(n.right, value);

}

Some(n)

} else {

Some(Box::new(TreeNode::new(value)))

}

}

Fn search(&self, value: i32) -> bool {

Self::search\_node(&self.root, value)

}

Fn search\_node(node: &Option<Box<TreeNode>>, value: i32) -> bool {

If let Some(n) = node {

If value == n.value {

Return true;

} else if value < n.value {

Return Self::search\_node(&n.left, value);

} else {

Return Self::search\_node(&n.right, value);

}

}

False

}

Fn inorder\_traversal(&self) -> Vec<i32> {

Let mut result = Vec::new();

Self::inorder\_node(&self.root, &mut result);

Result

}

Fn inorder\_node(node: &Option<Box<TreeNode>>, result: &mut Vec<i32>) {

If let Some(n) = node {

Self::inorder\_node(&n.left, result);

Result.push(n.value);

Self::inorder\_node(&n.right, result);

}

}

}

Fn main() {

Let mut bst = BinarySearchTree::new();

Bst.insert(5);

Bst.insert(3);

Bst.insert(7);

Bst.insert(2);

Bst.insert(4);

Bst.insert(6);

Bst.insert(8);

Println!("In-order Traversal: {:?}", bst.inorder\_traversal());

Println!("Searching for 4: {}", bst.search(4));

Println!("Searching for 10: {}", bst.search(10));

}

ساختمان داده AVL Tree در زبان Rust پیاده سا'ی کنید :

#[derive(Debug)]

Struct TreeNode {

Value: i32,

Height: i32,

Left: Option<Box<TreeNode>>,

Right: Option<Box<TreeNode>>,

}

Impl TreeNode {

Fn new(value: i32) -> TreeNode {

TreeNode {

Value,

Height: 1, // ارتفاع اولیه گره برابر با 1 است

Left: None,

Right: None,

}

}

}

Struct AVLTree {

Root: Option<Box<TreeNode>>,

}

Impl AVLTree {

Fn new() -> AVLTree {

AVLTree { root: None }

}

Fn insert(&mut self, value: i32) {

Self.root = Self::insert\_node(self.root.take(), value);

}

Fn insert\_node(node: Option<Box<TreeNode>>, value: i32) -> Option<Box<TreeNode>> {

// درج گره جدید در درخت

Let mut node = match node {

Some(n) => n,

None => return Some(Box::new(TreeNode::new(value))),

};

If value < node.value {

Node.left = Self::insert\_node(node.left, value);

} else if value > node.value {

Node.right = Self::insert\_node(node.right, value);

} else {

Return Some(node); // مقدار تکراری؛ درج نکردن

}

// بروزرسانی ارتفاع گره

Node.height = 1 + std::cmp::max(Self::get\_height(&node.left), Self::get\_height(&node.right));

// بالانس کردن درخت

Self::balance\_node(node)

}

Fn get\_height(node: &Option<Box<TreeNode>>) -> i32 {

Match node {

Some(n) => n.height,

None => 0,

}

}

Fn get\_balance(node: &Option<Box<TreeNode>>) -> i32 {

Match node {

Some(n) => Self::get\_height(&n.left) – Self::get\_height(&n.right),

None => 0,

}

}

Fn rotate\_right(y: Box<TreeNode>) -> Box<TreeNode> {

Let x = y.left.expect("Left child should exist for right rotation");

Let t2 = x.right;

// انجام چرخش

Let mut new\_y = \*y;

New\_y.left = t2;

New\_y.height = 1 + std::cmp::max(Self::get\_height(&new\_y.left), Self::get\_height(&new\_y.right)));

Let mut new\_x = \*x;

New\_x.right = Some(Box::new(new\_y));

New\_x.height = 1 + std::cmp::max(Self::get\_height(&new\_x.left), Self::get\_height(&new\_x.right)));

Box::new(new\_x)

}

Fn rotate\_left(x: Box<TreeNode>) -> Box<TreeNode> {

Let y = x.right.expect("Right child should exist for left rotation");

Let t2 = y.left;

// انجام چرخش

Let mut new\_x = \*x;

New\_x.right = t2;

New\_x.height = 1 + std::cmp::max(Self::get\_height(&new\_x.left), Self::get\_height(&new\_x.right)));

Let mut new\_y = \*y;

New\_y.left = Some(Box::new(new\_x));

New\_y.height = 1 + std::cmp::max(Self::get\_height(&new\_y.left), Self::get\_height(&new\_y.right)));

Box::new(new\_y)

}

Fn balance\_node(node: Box<TreeNode>) -> Option<Box<TreeNode>> {

Let balance = Self::get\_balance(&Some(node));

If balance > 1 {

If Self::get\_balance(&node.left) < 0 {

// چرخش چپ راست

Let left = node.left.unwrap();

Let new\_left = Self::rotate\_left(left);

Return Some(Self::rotate\_right(Box::new(node)));

}

// چرخش راست

Return Some(Self::rotate\_right(Box::new(node)));

}

If balance < -1 {

If Self::get\_balance(&node.right) > 0 {

// چرخش راست چپ

Let right = node.right.unwrap();

Let new\_right = Self::rotate\_right(right);

Return Some(Self::rotate\_left(Box::new(node)));

}

// چرخش چپ

Return Some(Self::rotate\_left(Box::new(node)));

}

Some(Box::new(node))

}

Fn inorder\_traversal(&self) -> Vec<i32> {

Let mut result = Vec::new();

Self::inorder\_node(&self.root, &mut result);

Result

}

Fn inorder\_node(node: &Option<Box<TreeNode>>, result: &mut Vec<i32>) {

If let Some(n) = node {

Self::inorder\_node(&n.left, result)

;

Result.push(n.value);

Self::inorder\_node(&n.right, result);

}

}

}

Fn main() {

Let mut avl\_tree = AVLTree::new();

Let values = vec![10, 20, 30, 40, 50, 25];

For value in values {

Avl\_tree.insert(value);

}

Println!("In-order Traversal: {:?}", avl\_tree.inorder\_traversal());

}

ساختمان داده max heap\_Tree را در زبان Rust پیاده سازی نمایید :

#[derive(Debug)]

Struct MaxHeap {

Data: Vec<i32>,

}

Impl MaxHeap {

Fn new() -> MaxHeap {

MaxHeap { data: Vec::new() }

}

Fn insert(&mut self, value: i32) {

Self.data.push(value);

Self.bubble\_up(self.data.len() – 1);

}

Fn bubble\_up(&mut self, index: usize) {

Let mut current\_index = index;

While current\_index > 0 {

Let parent\_index = (current\_index – 1) / 2;

If self.data[current\_index] <= self.data[parent\_index] {

Break;

}

Self.data.swap(current\_index, parent\_index);

Current\_index = parent\_index;

}

}

Fn extract\_max(&mut self) -> Option<i32> {

If self.data.is\_empty() {

Return None;

}

Let max = self.data[0];

Let last = self.data.pop().unwrap();

If !self.data.is\_empty() {

Self.data[0] = last;

Self.bubble\_down(0);

}

Some(max)

}

Fn bubble\_down(&mut self, index: usize) {

Let mut current\_index = index;

Let length = self.data.len();

Loop {

Let left\_child\_index = 2 \* current\_index + 1;

Let right\_child\_index = 2 \* current\_index + 2;

Let mut largest\_index = current\_index;

If left\_child\_index < length && self.data[left\_child\_index] > self.data[largest\_index] {

Largest\_index = left\_child\_index;

}

If right\_child\_index < length && self.data[right\_child\_index] > self.data[largest\_index] {

Largest\_index = right\_child\_index;

}

If largest\_index == current\_index {

Break;

}

Self.data.swap(current\_index, largest\_index);

Current\_index = largest\_index;

}

}

Fn build\_heap(arr: &[i32]) -> MaxHeap {

Let mut heap = MaxHeap { data: arr.to\_vec() };

Let length = heap.data.len();

For i in (0..length / 2).rev() {

Heap.bubble\_down(i);

}

Heap

}

}

Fn main() {

Let mut max\_heap = MaxHeap::new();

Let values = vec![3, 1, 6, 5, 2, 4];

For value in values {

Max\_heap.insert(value);

}

Println!("Max Heap: {:?}", max\_heap.data);

// Extract the maximum value

If let Some(max) = max\_heap.extract\_max() {

Println!("Extracted Max: {}", max);

}

Println!("Max Heap after extraction: {:?}", max\_heap.data);

// Building a max heap from an array

Let arr = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1];

Let max\_heap\_from\_arr = MaxHeap::build\_heap(&arr);

Println!("Max Heap from array: {:?}", max\_heap\_from\_arr.data);

}app.pyrequest.name