Bing

Я попробую написать реализацию B-дерева на golang. Вот что у меня получилось:

```
// B-дерево - это сбалансированное дерево поиска, в котором каждый узел может иметь более двух потомков.
// Каждый узел состоит из массива ключей и массива указателей на дочерние узлы.
// Ключи в каждом узле отсортированы по возрастанию, а указатели расположены так, что
// все ключи в поддереве, на которое указывает i-й указатель, меньше i-го ключа и больше (i-1)-го ключа.
// Количество ключей в каждом узле ограничено параметром t, называемым минимальной степенью дерева.
// Каждый узел, кроме корня, должен содержать не менее t-1 ключей и не более 2t-1 ключей.
// Корень может содержать от 0 до 2t-1 ключей. Если корень содержит 0 ключей, то дерево пустое.
// Каждый листовой узел должен иметь одинаковую глубину, называемую высотой дерева.

package main

import (
	"fmt"
)

// Node - структура для представления узла B-дерева
type Node struct {
	keys     []int  // массив ключей
	children []*Node // массив указателей на дочерние узлы
	leaf     bool   // флаг, является ли узел листовым
	n        int    // количество ключей в узле
}

// BTree - структура для представления B-дерева
type BTree struct {
	root *Node // корень дерева
	t    int   // минимальная степень дерева
}

// NewNode - функция для создания нового узла B-дерева
func NewNode(t int, leaf bool) *Node {
	node := &Node{}
	node.keys = make([]int, 2*t-1)   // выделяем память под максимальное количество ключей
	node.children = make([]*Node, 2*t) // выделяем память под максимальное количество указателей
	node.leaf = leaf                 // устанавливаем флаг листового узла
	node.n = 0                       // изначально узел не содержит ключей
	return node
}

// NewBTree - функция для создания нового B-дерева
func NewBTree(t int) *BTree {
	tree := &BTree{}
	tree.root = nil // изначально дерево пустое
	tree.t = t      // устанавливаем минимальную степень дерева
	return tree
}

// Search - функция для поиска ключа в B-дереве
// Возвращает пару (узел, индекс), если ключ найден, или (nil, -1), если нет
func (tree *BTree) Search(key int) (*Node, int) {
	return tree.search(tree.root, key) // начинаем поиск с корня дерева
}

// search - вспомогательная функция для рекурсивного поиска ключа в поддереве с корнем x
func (tree *BTree) search(x *Node, key int) (*Node, int) {
	if x == nil { // если поддерево пустое, то ключ не найден
		return nil, -1
	}
	i := 0
	for i < x.n && key > x.keys[i] { // ищем индекс первого ключа в узле x, который не меньше искомого
		i++
	}
	if i < x.n && key == x.keys[i] { // если мы нашли совпадение, то возвращаем пару (узел, индекс)
		return x, i
	}
	if x.leaf { // если мы достигли листового узла и не нашли совпадения, то ключ не найден
		return nil, -1
	}
	return tree.search(x.children[i], key) // иначе переходим к поиску в подходящем дочернем поддереве
}

// Insert - функция для вставки ключа в B-дерево
func (tree *BTree) Insert(key int) {
	r := tree.root          // запоминаем корень дерева
	if r == nil {           // если дерево пустое, то создаем новый корень с одним ключом
		r = NewNode(tree.t, true)
		r.keys[0] = key
		r.n = 1
		tree.root = r // обновляем корень дерева
		return        // завершаем вставку
	}
	if r.n == 2*tree.t-1 { // если корень переполнен (содержит максимальное количество ключей), то разбиваем его на два поддерева и создаем новый корень над
		s := NewNode(tree.t, false)   // создаем новый корень s с одним указателем на старый корень r
		s.children[0] = r             // делаем r первым ребенком s
		tree.root = s                 // обновляем корень дерева на s
		tree.splitChild(s, 0)         // разбиваем r на два поддерева по середине и перемещаем средний ключ из r в s
		tree.insertNonFull(s, key)    // вставляем ключ в подходящее поддерево s
	} else {                          // если корень не переполнен, то просто вставляем ключ в подходящее поддерево r
		tree.insertNonFull(r, key)
	}
}

// splitChild - функция для разбиения переполненного дочернего узла y родительского узла x
func (tree *BTree) splitChild(x *Node, i int) {
	y := x.children[i]              // запоминаем переполненный дочерний узел y
	z := NewNode(tree.t, y.leaf)    // создаем новый узел z той же степени и типа что и y
	z.n = tree.t - 1                // заполняем z правой половиной ключей из y
	for j := 0; j < tree.t-1; j++ {
		z.keys[j] = y.keys[j+tree.t] // копируем правые t-1 ключей из y в z
	}
	if !y.leaf {                    // если y не листовой узел, то копируем правые t указателей из y в z
		for j := 0; j < tree.t; j++ {
			z.children[j] = y.children[j+tree.t]
```

```
                }
        }
        y.n = tree.t - 1                // обновляем количество ключей в y на левую половину
        for j := x.n; j >= i+1; j-- {   // сдвигаем указатели в x на одну позицию вправо для освобождения места для z
                x.children[j+1] = x.children[j]
        }
        x.children[i+1] = z             // делаем z (i+1)-м ребенком x
        for j := x.n - 1; j >= i; j-- { // сдвигаем ключи в x на одну позицию вправо для освобождения места для среднего ключа из y
                x.keys[j+1] = x.keys[j]
        }
        x.keys[i] = y.keys[tree.t-1]    // перемещаем средний ключ из y в x
        x.n++                           // увеличиваем количество ключей в x на один
}

// insertNonFull - функция для вставки ключ
```