Sprawozdanie №2

Złożone struktury danych

Andrei Staravoitau, nr. albumu: 150218, grupa: I6, rok: I, sem: 2

25.04.2021

BST jest binarnym drzewem poszukiwań, w którym każdy węzeł spełnia reguły:

- Jeśli węzeł posiada lewe poddrzewo, to wszystkie węzły w tym poddrzewie mają wartość niewiększą od wartości danego węzła.
- Jeśli węzeł posiada prawe poddrzewo, to wszystkie węzły w tym poddrzewie są niemniejsze od wartości danego węzła.

Mocne i słabe strony:

- W BST elementy są zawsze posortowane
- W BST wyszukiwanie elementów jest bardzo szybkie
- Dodawanie elementu do listy jednokierunkowej jest szybsze

Złożoność operacji BST:

- Dodanie elementu - Najgorszy przypadek: O(n); Śr: O(log n)
- Wyszukiwanie elementu - Najgorszy przypadek: O(n); Śr: O(log n)
- Usunięcie elementu - Najgorszy przypadek: O(n); Śr: O(log n)
- Sprawdzenie liczby elementów - Najgorszy oraz śr przypadek: O(n)

BST:

```
#include <stdio.h>
#include <stdlib.h>

#define REL(a,b) (a)>(b) ? (1):(0)

typedef struct bstree {
    int key;
    struct bstree* left;
    struct bstree* right;
    struct bstree* parent;
}bstree;

bstree* CreateLeaf(int a, bstree* p) {
    bstree* t = malloc(sizeof(bstree));

    if (!t) {
        printf("Brak pamieci !!!\n");
        return NULL;
    }

    t->key = a;
    t->left = NULL;
    t->right = NULL;
    t->parent = p;

    return t;
}

void AddLeaf(int a, bstree** tr, bstree* parent) {
    if (*tr) {
        if (REL((*tr)->key, a))
            AddLeaf(a, &(*tr)->left, *tr);
        else
            AddLeaf(a, &(*tr)->right, *tr);
    }
    else
        *tr = CreateLeaf(a, parent);
}

bstree* DeleteTree(bstree* tr) {
    if (tr->left)
        DeleteTree(tr->left);
    if (tr->right)
        DeleteTree(tr->right);

    free(tr);

    return NULL;
}

void PrintTreeInorder(const bstree* tr) {
    if (tr) {
        PrintTreeInorder(tr->left);
        printf("%d ", tr->key);
        PrintTreeInorder(tr->right);
    }
}
```

```
void PrintTreePreorder(const bstree* tr) {
    if (tr) {
        printf("%d ", tr->key);
        PrintTreePreorder(tr->left);
        PrintTreePreorder(tr->right);
    }
}

void PrintTreePostorder(const bstree* tr) {
    if (tr) {
        PrintTreePostorder(tr->left);
        PrintTreePostorder(tr->right);
        printf("%d ", tr->key);
    }
}

int Find(bstree* tr, int val) {
    int node = 0;

    while (tr) {
        if (val < tr->key)
        {
            tr = tr->left;
            node++;
            if (val == tr->key)
                return node;
        }
        else
        {
            tr = tr->right;
            node++;
            if (val == tr->key)
                return node;
        }
    }
}


int Level(bstree* tr, int* tab, int length) {
    int max = 0, node = 0;


    for (int i = 0; i < length; i++)
    {
        node = Find(tr, tab[i]);
        if (node > max)
            max = node;
    }

    return max;
}


int main(void)
{
    int liczba[] = { 15 ,5 ,16 ,20 ,3 ,12 ,18 ,23 ,10 ,13 ,6 ,7 };
    bstree* root = NULL;

    for (int i = 0; i < sizeof(liczba) / sizeof(int); i++)
    {
```

```
        // liczba[i] = rand() % 41;
        // printf("%d ", liczba[i]);
        AddLeaf(liczba[i], &root, NULL);
    }
    // printf("\n--------------------\n\n");
    // printf("Inorder (rosnąco): \n");
    PrintTreeInorder(root);
    printf("\n\n");

    // printf("Preorder (od korzenia): \n");
    PrintTreePreorder(root);
    printf("\n\n");

    // printf("Postorder (do korzenia): \n");
    // PrintTreePostorder(root);
    // printf("\n\n");

    int find;
    printf("find: ");
    scanf("%d", &find);
    int search = Find(root, find);
    printf("search = %d\n", search);

    int level = Level(root, liczba, sizeof(liczba) / sizeof(int));
    printf("level = %d\n", level);





    root = DeleteTree(root);

    return 0;
}
```

```python
def height(self):
	if self.root != None:
		return self._height(self.root, 0)
	else:
		return 0

def _height(self,cur_node, cur_height):
	if cur_node == None: return cur_height
	left_height=self._height(cur_node.left_child, cur_height+1)
	right_height=self._height(cur_node.right_child, cur_height+1)
	return max(left_height, right_height)

def find(self,value):
	if self.root != None:
		return self._find(value, self.root)
	else:
		return None

def  _find(self, value, cur_node):
	if value == cur_node.value:
		return cur_node
	elif value < cur_node.value and cur_node.left_child != None:
		return self._find(value, cur_node.left_child)
	elif value > cur_node.value and cur_node.right_child != None:
		return self._find(value, cur_node.right_child)

def delete(self, value):
	return self._delete(self.find(value))

def _delete(self, node):
	if node == None or self.find(node.value)==None:
		print("Node to be deleted not found in the tree!")
		return None

	def min_value_node(n):
		current = n
		while current.left_child != None:
			current = current.left_child
		return current

	def num_children(n):
		num_children = 0
		if n.left_child != None: num_children += 1
		if n.right_child != None: num_children += 1
		return num_children

	node_parent = node.parent
	node_children = num_children(node)
	if node_children == 0:
		if node_parent != None:
			if node_parent.left_child == node:
				node_parent.left_child = None
			else:
				node_parent.right_child = None
		else:
			self.root = None

	if node_children == 1:
		if node.left_child != None:
			child = node.left_child
		else:
```

```
				child = node.right_child
			if node_parent != None:
				if node_parent.left_child == node:
					node_parent.left_child = child
				else:
					node_parent.right_child = child
			else:
				self.root = child
			child.parent = node_parent

		if node_children == 2:
			successor = min_value_node(node.right_child)
			node.value = successor.value
			self._delete(successor)

	def search(self,value):
		if self.root != None:
			return self._search(value, self.root)
		else:
			return False

	def _search(self,value, cur_node):
		if value == cur_node.value:
			return True
		elif value < cur_node.value and cur_node.left_child != None:
			return self._search(value, cur_node.left_child)
		elif value > cur_node.value and cur_node.right_child != None:
			return self._search(value, cur_node.right_child)
		return False
```