Pb1

Implementaţi un arbore binar de căutare cu chei numere întregi. Utilizaţi o structură NOD, care are un câmp de tip int, ce stochează cheia nodului şi trei câmpuri de tip pointer la NOD pentru fiul stâng, fiul drept şi părintele nodului. De asemenea structura NOD dispune de un constructor care setează câmpul int la o valoare transmisă prin parametru şi câmpurile de tip pointer la NOD le iniţializează cu pointer nul. Utilizaţi apoi o structură de tip **SearchTree**, care are ca membru rădăcina *root* de tip pointer la NOD În plus structura trebuie să aibă metodele:

- INSERT(int *key*) - inserează un nou nod în arbore cu cheia *key*. Dacă cheia deja există, nu se va insera. (0.25 p)
- MAXIM(NOD *x) / MINIM(NOD *x)- returnează nodul cu cheia maximă / minimă din subarborele de rădăcină x(0.25 p ambele funcţii)
- SUCCESOR(NOD *x) / PREDECESOR(NOD *x) - returnează nodul care este succesorul / predecesorul nodului x (0.25 p ambele funcţii)
- FIND(int *key*) - returnează nodul cu cheia *key* dacă există sau pointer nul altfel. (0.25 p)
- DELETE(int *key*) - şterge din arbore nodul cu cheia *key* dacă există (0.25 p)
- ERASE(NOD *x) - şterge din arbore nodul x (care a fost mai întâi identificat prin FIND) (0.25 p)
- PRINT_TREE(int opt) - afişază arborele în preordine (dacă opt=1), inordine (dacă opt=2), în postordine (dacă opt=3), pe niveluri (dacă opt=4). (0.5 p dintre care 0.25 pentru primele 3 afişări şi 0.25 pentru a 4-a).
- CONSTRUCT - construieşte un AB căutare pornind de la un vector de chei. (0.25p)
- EMPTY() - verifică dacă arborele este vid. (0.25 p)
- CLEAR() - şterge toate nodurile din arbore (0.25 p)

Structura trebuie să dispună de un constructor care iniţializează rădăcina cu pointer nul. Funcţiile Maxim/minim/succesor/predecesor/find returnează un pointer. În funcţia *main* se declară o variabilă de tip **SearchTree** şi se foloseşte un *menu* implementat cu ajutorul unei instrucţiuni *switch*, prin care utilizatorul să poată selecta oricare dintre operaţiile de inserţie, căutare, ştergere, minim, maxim, succesor, predecesor, afişare în cele 4 moduri - la alegere. (1 p)

```
#include <iostream>
#include <queue>
#include<vector>

struct NOD {
	int key;
	NOD* st, * dr, * p;
	NOD(int val) {
		key = val;
		st = nullptr;
		dr = nullptr;
		p = nullptr;
	}
};


struct searchTree {
	NOD* root = nullptr;
	void insert(int key) {
		NOD* z = new NOD(key);
		NOD* x = root;
		NOD* y = nullptr;
		while (x != nullptr) {
			y = x;
			if (z->key < x->key)
				x = x->st;
			else if (z->key > x->key)
				x = x->dr;
		}
		z->p = y;
		if (y == nullptr)
			root = z;
		else {
			if (z->key < y->key)
				y->st = z;
```

```
                else
                        y->dr = z;
        }
}
NOD* minim(NOD* root) {
        NOD* x = root;
        NOD* y = x;
        while (x != nullptr) {
                y = x;
                x = x->st;
        }
        return y;
}
NOD* maxim(NOD* root) {
        NOD* x = root;
        NOD* y = x;
        while (x != nullptr) {
                y = x;
                x = x->dr;
        }
        return y;
}
NOD* succesor(NOD* x) {
        NOD* y;
        if (x->dr != nullptr)
                y = minim(x->dr);
        else {
                y = x->p;
                while (y != nullptr && x == y->dr) {
                        x = y;
                        y = y->p;
                }
        }
        return y;
}
NOD* predecesor(NOD* x) {
        NOD* y;
        if (x->st != nullptr)
                y = maxim(x->st);
        else {
                y = x->p;
                while (y != nullptr && x == y->st) {
                        x = y;
                        y = y->p;
```

```
                }
        }
        return y;
}
NOD* find(int val) {
        if (root == nullptr)
                return nullptr;
        NOD* x = root;
        while (x != nullptr && x->key != val) {
                if (x->key < val)
                        x = x->dr;
                else
                        x = x->st;
        }
        return x;
}
void transplant(NOD* z, NOD* y) {
        y = succesor(z);
        if (z->p == nullptr)
                root = y;
        else {
                if (z == z->p->st)
                        z->p->st = y;
                else
                        z->p->dr = y;
        }
        if (y != nullptr)
                y->p = z->p;
}
void Delete(NOD* z) {
        if (z->st == nullptr)
                transplant(z, z->dr);
        else
                if (z->dr == nullptr)
                        transplant(z, z->st);
                else {
                        NOD* y = succesor(z);
                        if (y != z->dr) {
                                transplant(y, y->dr);
                                y->dr = z->dr;
                                z->dr->p = y;
                        }
                        transplant(z, y);
                        y->st = z->st;
```

```
                        z->st->p = y;
                }

}
void erase(NOD* z) {
        NOD* found = find(z->key);
        if (found == nullptr)
                return;
        if (found->st == nullptr) {
                if (found->p == nullptr)
                        root = found->dr;
                else if (found == found->p->st)
                        found->p->st = found->dr;
                else
                        found->p->dr = found->dr;
                if (found->dr != nullptr)
                        found->dr->p = found->p;
                delete found;
                return;
        }
        if (found->dr == nullptr) {
                if (found->p == nullptr)
                        root = found->st;
                else if (found == found->p->st)
                        found->p->st = found->st;
                else
                        found->p->dr = found->st;
                if (found->st != nullptr)
                        found->st->p = found->p;
                delete found;
                return;
        }
        NOD* succ = succesor(found);
        NOD* succParent = succ->p;

        if (succParent != found) {
                if (succ->dr != nullptr)
                        succ->dr->p = succParent;
                succParent->st = succ->dr;
                succ->dr = found->dr;
                found->dr->p = succ;
        }

        if (found->p == nullptr)
```

```
                root = succ;
        else if (found == found->p->st)
                found->p->st = succ;
        else
                found->p->dr = succ;
        succ->p = found->p;
        found->st->p = succ;
        succ->st = found->st;

        delete found;
}
void construct(std::vector<int>keys) {
        for (int key : keys)
                insert(key);
}
bool empty() {
        if (root == nullptr)
                return true;
        return false;
}
void clear(NOD* node) {
        if (node == nullptr)
                return;
        clear(node->st);

        clear(node->dr);

        delete node;
}

void RSD(NOD* root) {
        if (root == nullptr)
                return;
        std::cout << root->key << " ";
        RSD(root->st);
        RSD(root->dr);
}
void SRD(NOD* root) {
        if (root == nullptr)
                return;
        SRD(root->st);
        std::cout << root->key << " ";
        SRD(root->dr);
}
```

```
void SDR(NOD* root) {
	if (root == nullptr)
		return;
	SDR(root->st);
	SDR(root->dr);
	std::cout << root->key << " ";
}
void afisarePeNiveluri(NOD* root) {
	if (root == nullptr)
		return;
	std::queue<NOD*> queue;
	queue.push(root);
	while (!queue.empty()) {
		int size = queue.size();
		for (int i = 0; i < size; i++) {
			NOD* current = queue.front();
			std::cout << current->key << " ";
			queue.pop();
			if (current->st)
				queue.push(current->st);
			if (current->dr)
				queue.push(current->dr);
		}
		std::cout << std::endl;
	}
}

void printTree(int opt) {
	switch (opt) {
	case 1:
		RSD(root);
		break;
	case 2:
		SRD(root);
		break;
	case 3:
		SDR(root);
		break;
	case 4:
		afisarePeNiveluri(root);
		break;
	default:
		break;
	}
```

```
                std::cout << std::endl;
        }
};

int main() {
        searchTree sT;
        int optiune;
        std::cout << "Optiunea 1 pentru inserarea unui nod nou" << std::endl;
        std::cout << "Optiunea 2 pentru cautarea unui nod" << std::endl;
        std::cout << "Optiunea 3 pentru stergerea unui nod" << std::endl;
        std::cout << "Optiunea 4 pentru maximul subarborelui cu radacina intr-un nod dat" <<
std::endl;
        std::cout << "Optiunea 5 pentru minimul subarborelui cu radacina intr-un nod dat" <<
std::endl;
        std::cout << "Optiunea 6 pentru a afla succesorul unui nod dat" << std::endl;
        std::cout << "Optiunea 7 pentru a afla predecesorul unui nod dat" << std::endl;
        std::cout << "Optiunea 8 pentru a afisa arborele (exista 4 modalitati)" << std::endl;
        std::cout << "Optiunea 0 pentru a parasi meniul" << std::endl;

        do {
                int x;
                std::cout << "Alegeti optiunea: ";
                std::cin >> optiune;
                switch (optiune) {
                case 0:
                        break;
                case 1:
                        std::cout << "Ce nod doriti sa inserati? ";
                        std::cin >> x;
                        sT.insert(x);
                        break;
                case 2:
                        std::cout << "Ce nod doriti sa cautati? ";
                        std::cin >> x;
                        std::cout<<"Se afla la adresa: "<<sT.find(x)<<std::endl;
                        break;
                case 3:
                        std::cout << "Ce nod doriti sa stergeti? ";
                        std::cin >> x;
                        sT.erase(sT.find(x));
                        break;
                case 4:
                        std::cout << "Subarborele cu radacina in nodul ";
                        std::cin >> x;
```

```
                        std::cout << "Maximul subarborelui: "<<sT.maxim(sT.find(x))->key <<
std::endl;
                        break;
                case 5:
                        std::cout << "Subarborele cu radacina in nodul ";
                        std::cin >> x;
                        std::cout << "Minimul subarborelui: " << sT.minim(sT.find(x))->key <<
std::endl;
                        break;
                case 6:
                        std::cout << "Succesorul nodului: ";
                        std::cin >> x;
                        std::cout << "Este: " << sT.succesor(sT.find(x))->key << std::endl;
                        break;
                case 7:
                        std::cout << "Predecesorul nodului: ";
                        std::cin >> x;
                        std::cout << "Este: " << sT.predecesor(sT.find(x))->key << std::endl;
                        break;
                case 8:
                        std::cout << "Cum doriti sa afisati arborele? (1: RSD 2: SRD 3: SDR 4: Pe
niveluri) ";
                        std::cin >> x;
                        sT.printTree(x);
                        break;
                default:
                        std::cout << "Optiune invalida"<<std::endl;
                        break;
                }
        } while (optiune != 0);

        return 0;
}
```

Pb2.

Implementaţi un arbore AVL cu chei numere întregi (pentru template 0.25 p suplimentar). Utilizaţi o structură NOD care dispune de un câmp informaţie, un câmp înălţime şi câmpuri de tip pointer pentru fiii stâng şi drept şi pentru părinte. De asemenea structura NOD trebuie să dispună de un constructor care iniţializeă câmpul informaţie cu valoarea transmisă prin parametru, câmpul câmpul înăţime cu 1 şi câmpurile de tip pointer cu pointer nul. Utilizaţi o structură AVL care dispune de un membru de tip pointer la NOD, numit *root*. În plus dispune de funcţiile:

- INSERT(int *key*) - inserează un nou nod în arbore cu cheia *key*. Dacă cheia deja există, nu se va insera. (0.25 p)
- INSERT_REPARE - reface balansare după inserţie (1 p)
- MAXIM(NOD *x) / MINIM(NOD *x)- returnează nodul cu cheia maximă / minimă din subarborele de rădăcină x (0.25 ambele funcţii).
- SUCCESOR(NOD *x) / PREDECESOR(NOD *x) - returnează nodul care este succesorul / predecesorul nodului x (0.25 p ambele funcţii).
- FIND(int *key*) - returnează nodul cu cheia *key* dacă există sau pointer nul altfel. (0.25 p)
- DELETE(int *key*) - şterge din arbore nodul cu cheia *key* dacă există (0.25 p)
- ERASE(NOD *x) - şterge din arbore nodul x (care a fost mai întâi identificat prin FIND) (0.25 p)
- DELETE_REPARE(NOD *x) - reface balansarea arborelui după ştergere - (1p)
- ROT_ST, ROT_DR - funcţiile de rotaţie - (0.25 p)
- PRINT_TREE(int opt) - afişază arborele în preordine (dacă opt=1), inordine (dacă opt=2), în postordine (dacă opt=3), pe niveluri (dacă opt=4). (0.5 p dintre care 0.25 pentru primele 3 afişări şi 0.25 pentru a 4-a). Trebuie afişat şi factorul de balansare pentru fiecare nod.
- CLEAR() - şterge toate nodurile din arbore (0.25 p)

- EMPTY() - verifică dacă arborele este vid. (0.25 p)
- CONSTRUCT - construieşte un arbore AVL pornind de la un vector de chei. (0.25 p)

Structura trebuie să dispună de un constructor care iniţializează rădăcina cu pointer nul. Funcţiile Maxim/minim/succesor/predecesor/find returnează un pointer. În funcţia *main* se declară o variabilă de tip AVL şi se foloseşte un *menu* implementat cu ajutorul unei instrucţiuni *switch*, prin care utilizatorul să poată selecta oricare dintre operaţiile de inserţie, construire (cu construct), căutare, ştergere, minim, maxim, afişare în cele 4 moduri - la alegere, golire (clear), Merge (atunci trebuie doi arbori). (0.75 p)

```
#include <iostream>
#include <queue>
#include<vector>

struct NOD {
	int key, height;
	NOD* st, * dr, * p;
	NOD(int val) {
		key = val;
		height = 1;
		st = nullptr;
		dr = nullptr;
		p = nullptr;
	}
};


struct searchTree {
	NOD* root = nullptr;
	NOD* minim(NOD* root) {
		NOD* x = root;
		NOD* y = x;
		while (x != nullptr) {
			y = x;
			x = x->st;
		}
		return y;
	}
	NOD* maxim(NOD* root) {
		NOD* x = root;
		NOD* y = x;
```

```
        while (x != nullptr) {
                y = x;
                x = x->dr;
        }
        return y;
}
NOD* succesor(NOD* x) {
        NOD* y;
        if (x->dr != nullptr)
                y = minim(x->dr);
        else {
                y = x->p;
                while (y != nullptr && x == y->dr) {
                        x = y;
                        y = y->p;
                }
        }
        return y;
}
NOD* predecesor(NOD* x) {
        NOD* y;
        if (x->st != nullptr)
                y = maxim(x->st);
        else {
                y = x->p;
                while (y != nullptr && x == y->st) {
                        x = y;
                        y = y->p;
                }
        }
        return y;
}
NOD* find(int val) {
        if (root == nullptr)
                return nullptr;
        NOD* x = root;
        while (x != nullptr && x->key != val) {
                if (x->key < val)
                        x = x->dr;
                else
                        x = x->st;
        }
        return x;
}
```

```
void transplant(NOD* z, NOD* y) {
	y = succesor(z);
	if (z->p == nullptr)
		root = y;
	else {
		if (z == z->p->st)
			z->p->st = y;
		else
			z->p->dr = y;
	}
	if (y != nullptr)
		y->p = z->p;
}
void Delete(NOD* z) {
	if (z->st == nullptr)
		transplant(z, z->dr);
	else
		if (z->dr == nullptr)
			transplant(z, z->st);
		else {
			NOD* y = succesor(z);
			if (y != z->dr) {
				transplant(y, y->dr);
				y->dr = z->dr;
				z->dr->p = y;
			}
			transplant(z, y);
			y->st = z->st;
			z->st->p = y;
		}

}
void construct(std::vector<int>keys) {
	for (int key : keys)
		insert(key);
}
bool empty() {
	if (root == nullptr)
		return true;
	return false;
}
void clear(NOD* node) {
	if (node == nullptr)
		return;
```

```
	clear(node->st);

	clear(node->dr);

	delete node;
}

void RSD(NOD* root) {
	if (root == nullptr)
		return;
	std::cout << root->key << " ";
	RSD(root->st);
	RSD(root->dr);
}
void SRD(NOD* root) {
	if (root == nullptr)
		return;
	SRD(root->st);
	std::cout << root->key << " ";
	SRD(root->dr);
}
void SDR(NOD* root) {
	if (root == nullptr)
		return;
	SDR(root->st);
	SDR(root->dr);
	std::cout << root->key << " ";
}
void afisarePeNiveluri(NOD* root) {
	if (root == nullptr)
		return;
	std::queue<NOD*> queue;
	queue.push(root);
	while (!queue.empty()) {
		int size = queue.size();
		for (int i = 0; i < size; i++) {
			NOD* current = queue.front();
			std::cout << current->key << " ";
			queue.pop();
			if (current->st)
				queue.push(current->st);
			if (current->dr)
				queue.push(current->dr);
		}
```

```
                std::cout << std::endl;
        }
}

void printTree(int opt) {
        switch (opt) {
        case 1:
                RSD(root);
                break;
        case 2:
                SRD(root);
                break;
        case 3:
                SDR(root);
                break;
        case 4:
                afisarePeNiveluri(root);
                break;
        default:
                break;
        }
}
int height(NOD* node) {
        if (node == nullptr)
                return 0;
        return node->height;
}
int max(int a, int b) {
        return (a > b) ? a : b;
}
NOD* createNode(int key) {
        NOD* newNode = new NOD(key);
        return newNode;
}
int getBalance(NOD* node) {
        if (node == nullptr)
                return 0;
        return height(node->dr) - height(node->st);
}
NOD* rightRotate(NOD* x) {
        NOD* y = x->st;
        NOD* T2 = y->dr;
        y->dr = x;
        x->st = T2;
```

```
        y->p = x->p;
        x->p = y;
        x->height = max(height(x->st), height(x->dr)) + 1;
        y->height = max(height(y->st), height(y->dr)) + 1;

        return y;
}
NOD* leftRotate(NOD* x) {
        NOD* y = x->dr;
        NOD* T2 = y->st;
        y->st = x;
        x->dr = T2;
        y->p = x->p;
        x->p = y;
        x->height = max(height(x->st), height(x->dr)) + 1;
        y->height = max(height(y->st), height(y->dr)) + 1;

        return y;
}
NOD* insert(NOD* node, int key, NOD* parent) {
        if (node == nullptr) {
                NOD* newNode = createNode(key);
                newNode->p = parent;
                return newNode;
        }

        if (key < node->key)
                node->st = insert(node->st, key, node);
        else if (key > node->key)
                node->dr = insert(node->dr, key, node);
        else
                return node;

        node->height = 1 + max(height(node->st), height(node->dr));
        int balance = getBalance(node);

        if (balance < -1 && key < node->st->key)
                return rightRotate(node);

        if (balance > 1 && key > node->dr->key)
                return leftRotate(node);

        if (balance < -1 && key > node->st->key) {
                node->st = leftRotate(node->st);
```

```
                return rightRotate(node);
        }

        if (balance > 1 && key < node->dr->key) {
                node->dr = rightRotate(node->dr);
                return leftRotate(node);
        }

        return node;
}
void insert(int key) {
        root = insert(root, key, nullptr);
}

NOD* balanceAfterDeletion(NOD* node) {
        if (node == nullptr)
                return nullptr;

        node->height = 1 + max(height(node->st), height(node->dr));

        int balance = getBalance(node);

        if (balance > 1 && getBalance(node->st) >= 0)
                return rightRotate(node);

        if (balance > 1 && getBalance(node->st) < 0) {
                node->st = leftRotate(node->st);
                return rightRotate(node);
        }

        if (balance < -1 && getBalance(node->dr) <= 0)
                return leftRotate(node);

        if (balance < -1 && getBalance(node->dr) > 0) {
                node->dr = rightRotate(node->dr);
                return leftRotate(node);
        }

        return node;
}

void erase(NOD* z) {
        NOD* found = find(z->key);
        if (found == nullptr)
```

```
        return;
    if (found->st == nullptr) {
        if (found->p == nullptr)
            root = found->dr;
        else if (found == found->p->st)
            found->p->st = found->dr;
        else
            found->p->dr = found->dr;
        if (found->dr != nullptr)
            found->dr->p = found->p;
        delete found;
        root = balanceAfterDeletion(root);
        return;
    }
    if (found->dr == nullptr) {
        if (found->p == nullptr)
            root = found->st;
        else if (found == found->p->st)
            found->p->st = found->st;
        else
            found->p->dr = found->st;
        if (found->st != nullptr)
            found->st->p = found->p;
        delete found;
        root = balanceAfterDeletion(root);
        return;
    }
    NOD* succ = succesor(found);
    NOD* succParent = succ->p;

    if (succParent != found) {
        if (succ->dr != nullptr)
            succ->dr->p = succParent;
        succParent->st = succ->dr;
        succ->dr = found->dr;
        found->dr->p = succ;
    }

    if (found->p == nullptr)
        root = succ;
    else if (found == found->p->st)
        found->p->st = succ;
    else
        found->p->dr = succ;
```

```
                succ->p = found->p;

                if (succ->st != nullptr)
                        succ->st->p = succ;
                succ->st = found->st;

                delete found;
                root = balanceAfterDeletion(root);
                return;
        }


};


int main()
{
        searchTree sT;
        int optiune;
        std::cout << "Optiunea 1 pentru inserarea unui nod nou" << std::endl;
        std::cout << "Optiunea 2 pentru cautarea unui nod" << std::endl;
        std::cout << "Optiunea 3 pentru stergerea u+nui nod" << std::endl;
        std::cout << "Optiunea 4 pentru maximul subarborelui cu radacina intr-un nod dat" <<
std::endl;
        std::cout << "Optiunea 5 pentru minimul subarborelui cu radacina intr-un nod dat" <<
std::endl;
        std::cout << "Optiunea 6 pentru a afla succesorul unui nod dat" << std::endl;
        std::cout << "Optiunea 7 pentru a afla predecesorul unui nod dat" << std::endl;
        std::cout << "Optiunea 8 pentru a afisa arborele (exista 4 modalitati)" << std::endl;
        std::cout << "Optiunea 0 pentru a parasi meniul" << std::endl;

        do {
                int x;
                std::cout << "Alegeti optiunea: ";
                std::cin >> optiune;
                switch (optiune) {
                case 0:
                        break;
                case 1:
                        std::cout << "Ce nod doriti sa inserati? ";
                        std::cin >> x;
                        sT.insert(x);
                        break;
                case 2:
```

```
                std::cout << "Ce nod doriti sa cautati? ";
                std::cin >> x;
                std::cout << "Se afla la adresa: " << sT.find(x) << std::endl;
                break;
            case 3:
                std::cout << "Ce nod doriti sa stergeti? ";
                std::cin >> x;
                sT.erase(sT.find(x));
                break;
            case 4:
                std::cout << "Subarborele cu radacina in nodul ";
                std::cin >> x;
                std::cout << "Maximul subarborelui: " << sT.maxim(sT.find(x))->key <<
std::endl;
                break;
            case 5:
                std::cout << "Subarborele cu radacina in nodul ";
                std::cin >> x;
                std::cout << "Minimul subarborelui: " << sT.minim(sT.find(x))->key <<
std::endl;
                break;
            case 6:
                std::cout << "Succesorul nodului: ";
                std::cin >> x;
                std::cout << "Este: " << sT.succesor(sT.find(x))->key << std::endl;
                break;
            case 7:
                std::cout << "Predecesorul nodului: ";
                std::cin >> x;
                std::cout << "Este: " << sT.predecesor(sT.find(x))->key << std::endl;
                break;
            case 8:
                std::cout << "Cum doriti sa afisati arborele? (1: RSD 2: SRD 3: SDR 4: Pe
niveluri) ";
                std::cin >> x;
                sT.printTree(x);
                break;
            default:
                std::cout << "Optiune invalida" << std::endl;
                break;
            }
        } while (optiune != 0);
```

```
    return 0;
}
```

Pb6.

B-secvenţă: Numim B-secvenţă un şi de n numere $a_1, a_2, \ldots, a_n$ cu următoarele proprietăţi:

- $a_1 < a_2 < \ldots < a_j$ şi $a_j > a_{j+1} > \ldots > a_n$
- fiecare element, cu excepţia maximului apare de cel mult 2 ori în şir: o dată în partea crescătoare şi eventual o dată în partea descrescătoare a şirului
- toate elementele din partea descrescătoare se regăsesc şi în partea crescătoare.

Se citeşte o astfel de secvenţă S dintr-un fişier. Apoi se realizează K operaţii în modul următor: - pentru fiecare operaţie se citeşe o valoare *val*. Această valoare se inserează în secvenţa S, numai dacă după inserţie se păstrează proprietăţile definite mai sus. De asemenea după fiecare operaţie se primeşte un mesaj, care indică dacă operaţia a putut fi efectuată şi se afişează noua secvenţă. Folosiţi *set* din STL. (2p)

```
#include <iostream>
#include <fstream>
#include <set>

void citeste_fisier(std::set<int>& cresc, std::set<int>& desc) {

    std::ifstream fin("Text.txt");
    int curent, anterior;
    fin >> anterior;
    while (fin >> curent) {
        if (anterior < curent) cresc.insert(anterior);
        else desc.insert(anterior);

        anterior = curent;
    }
    desc.insert(curent);
    auto penultim = desc.begin();
    std::advance(penultim, desc.size() - 2);
    for (auto it = desc.begin(); it != penultim; it++)
```

```
                cresc.insert(*it);
}

bool try_insert(int val, std::set<int>& cresc, std::set<int>& desc) {
        auto it = *(desc.rbegin());
        if (val == it)
                return 0;
        if (cresc.find(val) == cresc.end()) {
                cresc.insert(val);
                return 1;
        }
        else {
                if (desc.find(val) == desc.end()) {
                        desc.insert(val);
                        return 1;
                }
                else return 0;
        }
}

void afisare_sir(std::set<int>& cresc, std::set<int>& desc) {
        for (auto it = cresc.begin(); it != cresc.end(); it++) std::cout << *it << " ";
        for (auto it = desc.rbegin(); it != desc.rend(); it++) std::cout << *it << " ";
}

int main() {
        int k, val;
        std::set<int> cresc;
        std::set<int> desc;
        citeste_fisier(cresc, desc);

        std::cout << "Sir : ";
        afisare_sir(cresc, desc);
        std::cout << std::endl<<"Introduceti K : ";
        std::cin >> k;

        for (int i = 0; i < k; i++) {
                std::cout <<std::endl<< "Valoare : ";
                std::cin >> val;
                bool succes = try_insert(val, cresc, desc);
                if (succes) {
                        std::cout <<std::endl<< "Valoarea a fost inserata. Noul sir este ";
                        afisare_sir(cresc, desc);
```

```
            }
            else std::cout <<std::endl<< " Valoarea nu a putut fi inserata";
        }
        return 0;
}
```

Pb7.

Dicţionar: Se citeşte dintr-un fişier un text în care cuvintele sunt separate prin spaţii. Pot exista şi semne de punctuaţie. Să se afişeze cuvintele citite în ordine alfabetică. Fiecare cuvânt se afişază o singură dată, având alături numărul de apariţii în text. Utilizaţi **map** din STL. Semnele de punctuaţie se ignoră. (1p)

```
#include <iostream>
#include <fstream>
#include<map>
#include<string>

std::string removePunctuation(std::string& word) {
        std::string result;
        for (auto c : word)
                if (std::isalpha(c))
                        result += std::tolower(c);
        return result;
}

std::map<std::string, int> countWords() {
        std::ifstream fin("Text.txt");
        std::map<std::string, int> wordCounts;
        std::string word, cleanedWord;
        while (fin >> word) {
                cleanedWord = removePunctuation(word);
                if (!cleanedWord.empty()) {
                        wordCounts[cleanedWord]++;
                }
        }
        fin.close();
        return wordCounts;
}

void display(std::map<std::string, int>wordCounts) {

        for (auto p : wordCounts)
                std::cout << p.first << ':' << p.second << std::endl;
```

```
}

int main() {
	std::map<std::string, int> wordCounts = countWords();
	display(wordCounts);
	return 0;
}
```