```
template <class K, class V>
class BKUTree
{
public:
        class AVLTree;
        class SplayTree;

        class Entry
        {
        public:
                K key;
                V value;

                Entry(K key, V value) : key(key), value(value) {}
        };

private:
        AVLTree *avl;
        SplayTree *splay;
        queue<K> keys;
        int maxNumOfKeys;

public:
        BKUTree(int maxNumOfKeys = 5)
        {
                this->avl = NULL;
                this->splay = NULL;
                this->maxNumOfKeys = maxNumOfKeys;
        }
        ~BKUTree() { this->clear(); }

        void add(K key, V value)
        {
                if (this->avl == NULL || this->splay == NULL)
                {
                        this->splay = new SplayTree();
                        this->avl = new AVLTree();
                }
                this->splay->add(key, value);
                this->avl->add(key, value);
                this->splay->root->corr = this->avl->insertNode;
                this->splay->root->corr->corr = this->splay->root;
                int size = keys.size();
                if (size >= maxNumOfKeys)
                        keys.pop();
                keys.push(key);
        }

        void remove(K key)
        {
```

```
                this->splay->remove(key);
                this->avl->remove(key);

                vector<K> KEY;
                while (keys.size() != 0)
                {
                        if (keys.front() != key)
                                KEY.push_back(keys.front());
                        keys.pop();
                }

                for (auto it : KEY)
                        keys.push(it);
                KEY.clear();

                if (this->splay->root != NULL)
                        keys.push(this->splay->root->entry->key);

                int n = keys.size();
                if (n > this->maxNumOfKeys)
                        keys.pop();
        }

        V search(K key, vector<K> &traversedList)
        {
                if (this->splay->root->entry->key == key)
                {
                        int n = keys.size();
                        if (n >= maxNumOfKeys)
                                keys.pop();
                        keys.push(key);
                        return this->splay->root->entry->value;
                }

                vector<K> KEY;
                V result;
                while (keys.size() != 0)
                {
                        KEY.push_back(keys.front());
                        keys.pop();
                }

                if (find(KEY.begin(), KEY.end(), key) != KEY.end())
                {
                        typename BKUTree<K, V>::SplayTree::Node *target =
this->splay->search(this->splay->root, key, traversedList);
                        if (target == NULL)
                                throw "Not found!";
                        this->splay->Splay(target, 1);
                        if (target->parent == NULL)
```

```
                        this->splay->root = target;
                result = target->entry->value;
        }
        else
        {
                typename BKUTree<K, V>::AVLTree::Node *target =
this->avl->search(this->splay->root->corr, key, traversedList);
                if (target == NULL)
                {
                        target = this->avl->search(this->avl->root, key,
traversedList);
                        if (target == NULL)
                                throw "Not found!";
                }
                this->splay->Splay(target->corr, 1);
                if (target->corr->parent == NULL)
                        this->splay->root = target->corr;
                result = target->entry->value;
        }

        for (auto i : KEY)
                keys.push(i);
        int n = keys.size();
        if (n >= maxNumOfKeys)
                keys.pop();
        keys.push(key);
        KEY.clear();
        return result;
}

void traverseNLROnAVL(void (*func)(K key, V value))
{
        this->avl->traverseNLR(func);
}
void traverseNLROnSplay(void (*func)(K key, V value))
{
        this->splay->traverseNLR(func);
}

void clear()
{
        this->avl->clear();
        this->splay->clear();
        delete avl;
        delete splay;
        avl = splay = NULL;
        while (!keys.empty())
        {
                keys.pop();
        }
```

```
                this->maxNumOfKeys = 5;
        }

        ////////////////////////////////////
        ///       CLASS SPLAYTREE        ///
        ////////////////////////////////////
        class SplayTree
        {
        public:
                class Node
                {
                        Entry *entry;
                        Node *left;
                        Node *right;
                        Node *parent;
                        typename AVLTree::Node *corr;
                        friend class SplayTree;
                        friend class BKUTree<K, V>;
                        Node(Entry *entry = NULL, Node *parent = NULL, Node *left =
NULL, Node *right = NULL)
                        {
                                this->entry = entry;
                                this->left = left;
                                this->parent = parent;
                                this->right = right;
                                this->corr = NULL;
                        }
                };

        public:
                Node *root;

                SplayTree() : root(NULL){};
                ~SplayTree() { this->clear(); };

                void rotateLeft(Node *&root)
                {
                        Node *tempPtr = root->right;
                        root->right = tempPtr->left;
                        tempPtr->left = root;
                        root = tempPtr;
                        root->parent = root->left->parent;
                        root->left->parent = root;
                        if (root->left->right != NULL)
                                root->left->right->parent = root->left;
                        if (root->parent != NULL)
                        {
                                if (root->parent->left == root->left)
                                        root->parent->left = root;
                                else
```

```
                        root->parent->right = root;
        }
}
void rotateRight(Node *&root)
{
        Node *tempPtr = root->left;
        root->left = tempPtr->right;
        tempPtr->right = root;
        root = tempPtr;
        root->parent = root->right->parent;
        root->right->parent = root;
        if (root->right->left != NULL)
                root->right->left->parent = root->right;
        if (root->parent != NULL)
        {
                if (root->parent->right == root->right)
                        root->parent->right = root;
                else
                        root->parent->left = root;
        }
}
void zigzag(Node *root)
{
        Node *ptr = root->right;
        rotateRight(ptr);
        rotateLeft(root);
}
void zagzig(Node *root)
{
        Node *ptr = root->left;
        rotateLeft(ptr);
        rotateRight(root);
}
void zigzig(Node *root)
{
        rotateRight(root);
        rotateRight(root);
}
void zagzag(Node *root)
{
        rotateLeft(root);
        rotateLeft(root);
}
void Splay(Node *&root, bool SplayOnce)
{
        if (root->parent == NULL)
        {
                return;
        }
        Node *Parent = root->parent;
```

```
        Node *grand = Parent->parent;
        if (grand == NULL)
        {
                //ZIG
                if (root == Parent->left)
                        rotateRight(Parent);
                //ZAG
                else
                        rotateLeft(Parent);
        }
        else
        {
                if (Parent == grand->left)
                {
                        //ZIGZIG
                        if (root == Parent->left)
                                zigzig(grand);
                        //ZAGZIG
                        else
                                zagzig(grand);
                }
                else
                {
                        //ZAGZAG
                        if (root == Parent->right)
                                zagzag(grand);
                        //ZIGZIG
                        else
                                zigzag(grand);
                }
        }
        if (!SplayOnce)
                Splay(root, 0);
}

void add(Entry *entry)
{
        if (this->root == NULL)
                this->root = new Node(entry);
        else
        {
                Node *walker = this->root;
                Node *prev = walker;
                while (walker != NULL)
                {
                        prev = walker;
                        if (walker->entry->key > entry->key)
                                walker = walker->left;
                        else if (walker->entry->key < entry->key)
                                walker = walker->right;
```

```
                        else
                                throw "Duplicate key";
                }
                if (prev->entry->key > entry->key)
                {
                        Node *node = new Node(entry, prev);
                        prev->left = node;
                        Splay(node, 0);
                        this->root = node;
                }
                else
                {
                        Node *node = new Node(entry, prev);
                        prev->right = node;
                        Splay(node, 0);
                        this->root = node;
                }
        }
}
void add(K key, V value)
{
        add(new Entry(key, value));
}

Node *search(Node *root, K key, vector<K> &traversedList)
{
        if (root == NULL)
                return NULL;
        traversedList.push_back(root->entry->key);
        if (root->entry->key > key)
                return search(root->left, key, traversedList);
        else if (root->entry->key < key)
                return search(root->right, key, traversedList);
        else
        {
                traversedList.pop_back();
                return root;
        }
}
V search(K key)
{
        vector<K> traversedList;
        Node *target = search(root, key, traversedList);
        traversedList.clear();
        if (target == NULL)
                throw "Not found!";
        Splay(target, 0);
        this->root = target;
        return target->entry->data;
}
```

```
void remove(K key)
{
        vector<K> traversedList;
        Node *nodeDel = search(root, key, traversedList);
        traversedList.clear();
        if (nodeDel == NULL)
                throw "Not found!";
        Splay(nodeDel, 0);
        this->root = nodeDel;
        if (this->root->left == NULL)
        {
                if (this->root->right != NULL)
                {
                        this->root->right->parent = NULL;
                        this->root = this->root->right;
                        delete nodeDel;
                }
                else
                {
                        delete this->root;
                        this->root = NULL;
                }
        }
        else
        {
                if (this->root->right == NULL)
                {
                        this->root->left->parent = NULL;
                        this->root = this->root->left;
                        delete nodeDel;
                }
                else
                {
                        this->root->left->parent = NULL;
                        Node *walker = this->root->left;
                        while (walker->right != NULL)
                                walker = walker->right;
                        Splay(walker, 0);
                        walker->right = this->root->right;
                        this->root->right->parent = walker;
                        delete nodeDel;
                        this->root = walker;
                }
        }
}

void traverseNLR(Node *root, void (*func)(K key, V value))
{
```

```
                        if (root == NULL)
                                return;
                        (*func)(root->entry->key, root->entry->value);
                        traverseNLR(root->left, func);
                        traverseNLR(root->right, func);
                }
                void traverseNLR(void (*func)(K key, V value))
                {
                        traverseNLR(this->root, func);
                }

                void clear(Node *root)
                {
                        if (root == NULL)
                                return;
                        clear(root->left);
                        clear(root->right);
                        delete root;
                }
                void clear()
                {
                        clear(root);
                }
        };

        ////////////////////////////////////
        ///       CLASS AVLTREE           ///
        ////////////////////////////////////
        class AVLTree
        {
        public:
                class Node
                {
                        Entry *entry;
                        Node *left;
                        Node *right;
                        int balance;
                        typename SplayTree::Node *corr;
                        friend class AVLTree;
                        friend class BKUTree<K, V>;
                        Node(Entry *entry = NULL, Node *left = NULL, Node *right =
NULL)
                        {
                                this->entry = entry;
                                this->left = left;
                                this->right = right;
                                this->balance = 0;
                                this->corr = NULL;
                        }
                };
```

```
public:
        Node *root;
        Node *insertNode;

        AVLTree() : root(NULL), insertNode(NULL){};
        ~AVLTree() { this->clear(); };

        void rotateLeft(Node *&root)
        {
                Node *tempPtr = root->right;
                root->right = tempPtr->left;
                tempPtr->left = root;
                root = tempPtr;
        }
        void rotateRight(Node *&root)
        {
                Node *tempPtr = root->left;
                root->left = tempPtr->right;
                tempPtr->right = root;
                root = tempPtr;
        }
        void leftBalance(Node *&root, bool &taller)
        {
                switch (root->left->balance)
                {
                case 1:
                        rotateRight(root);
                        root->balance = 0;
                        root->right->balance = 0;
                        taller = 0;
                        break;
                case 2:
                        rotateLeft(root->left);
                        rotateRight(root);
                        switch (root->balance)
                        {
                        case 0:
                                root->left->balance = 0;
                                root->right->balance = 0;
                                break;
                        case 1:
                                root->left->balance = 0;
                                root->right->balance = 2;
                                break;
                        case 2:
                                root->left->balance = 1;
                                root->right->balance = 0;
                                break;
                        }
```

```
                root->balance = 0;
                break;
        }
        taller = 0;
}
void rightBalance(Node *&root, bool &taller)
{
        switch (root->right->balance)
        {
        case 2:
                rotateLeft(root);
                root->balance = 0;
                root->left->balance = 0;
                taller = 0;
                break;
        case 1:
                rotateRight(root->right);
                rotateLeft(root);
                switch (root->balance)
                {
                case 0:
                        root->left->balance = 0;
                        root->right->balance = 0;
                        break;
                case 1:
                        root->left->balance = 0;
                        root->right->balance = 2;
                        break;
                case 2:
                        root->left->balance = 1;
                        root->right->balance = 0;
                        break;
                }
                root->balance = 0;
                break;
        }
        taller = 0;
}
void insertAvlTree(Node *&root, Entry *value, bool &taller)
{
        if (root == NULL)
        {
                root = new Node(value);
                taller = 1;
                this->insertNode = root;
                return;
        }
        if (value->key < root->entry->key)
        {
                insertAvlTree(root->left, value, taller);
```

```
                if (taller)
                {
                        switch (root->balance)
                        {
                        case 0:
                                root->balance = 1;
                                break;
                        case 1:
                                leftBalance(root, taller);
                                break;
                        case 2:
                                root->balance = 0;
                                taller = 0;
                                break;
                        }
                }
        }

        else if (value->key > root->entry->key)
        {
                insertAvlTree(root->right, value, taller);
                if (taller)
                {
                        switch (root->balance)
                        {
                        case 0:
                                root->balance = 2;
                                break;
                        case 1:
                                root->balance = 0;
                                taller = 0;
                                break;
                        case 2:
                                rightBalance(root, taller);
                                break;
                        }
                }
        }
        else
                throw "Duplicate key";
}
void add(Entry *entry)
{
        bool taller = 1;
        insertAvlTree(root, entry, taller);
}
void add(K key, V value)
{
        add(new Entry(key, value));
}
```

```
Node *deleteRightBalance(Node *root, bool &shorter)
{
        if (root->balance == 1)
        {
                root->balance = 0;
                return root;
        }
        else if (root->balance == 0)
        {
                root->balance = 2;
                shorter = false;
                return root;
        }
        else
        {
                if (root->right->balance == 1)
                {
                        rotateRight(root->right);
                        rotateLeft(root);
                        if (root->balance == 1)
                        {
                                root->right->balance = 2;
                                root->left->balance = 0;
                        }
                        else if (root->balance == 2)
                        {
                                root->right->balance = 0;
                                root->left->balance = 1;
                        }
                        else
                        {
                                root->right->balance = 0;
                                root->left->balance = 0;
                        }
                        root->balance = 0;
                }
                else
                {
                        rotateLeft(root);
                        if (root->balance == 2)
                        {
                                root->left->balance =
root->right->balance = 0;
                                root->balance = 0;
                        }
                        else
                        {
                                root->balance = 1;
                                root->left->balance = 2;
```

```
                                        root->right->balance = 0;
                                        shorter = false;
                                }
                        }
                }
                return root;
        }
        Node *deleteLeftBalance(Node *root, bool &shorter)
        {
                if (root->balance == 2)
                {
                        root->balance = 0;
                        return root;
                }
                else if (root->balance == 0)
                {
                        root->balance = 1;
                        shorter = false;
                        return root;
                }
                else
                {
                        if (root->left->balance == 2)
                        {
                                rotateLeft(root->left);
                                rotateRight(root);
                                if (root->balance == 2)
                                {
                                        root->left->balance = 1;
                                        root->right->balance = 0;
                                }
                                else if (root->balance == 1)
                                {
                                        root->right->balance = 2;
                                        root->left->balance = 0;
                                }
                                else
                                {
                                        root->right->balance = 0;
                                        root->left->balance = 0;
                                }
                                root->balance = 0;
                        }
                        else
                        {
                                rotateRight(root);
                                if (root->balance == 1)
                                {
                                        root->right->balance =
root->left->balance = 0;
```

```
                                        root->balance = 0;
                                }
                                else
                                {
                                        root->balance = 2;
                                        root->left->balance = 0;
                                        root->right->balance = 1;
                                        shorter = false;
                                }
                        }
                }
                return root;
        }
        Node *AVLDelete(Node *root, K deleteKey, bool &shorter, bool
&success)
        {
                if (root == NULL)
                {
                        success = shorter = 0;
                        return root;
                }
                else if (deleteKey < root->entry->key)
                {
                        root->left = AVLDelete(root->left, deleteKey,
shorter, success);
                        if (shorter)
                                root = deleteRightBalance(root, shorter);
                }
                else if (deleteKey > root->entry->key)
                {
                        root->right = AVLDelete(root->right, deleteKey,
shorter, success);
                        if (shorter)
                                root = deleteLeftBalance(root, shorter);
                }
                else
                {
                        if (root->left == NULL)
                        {
                                Node *tmp = root->right;
                                success = shorter = true;
                                delete root;
                                return tmp;
                        }
                        else if (root->right == NULL)
                        {
                                Node *tmp = root->left;
                                success = shorter = true;
                                delete root;
                                return tmp;
```

```
                                }
                                else
                                {
                                        Node *findmax = root->left;
                                        while (findmax->right != NULL)
                                        {
                                                findmax = findmax->right;
                                        }
                                        root->entry = findmax->entry;
                                        root->corr = findmax->corr;
                                        findmax->corr->corr = root;
                                        root->left = AVLDelete(root->left,
findmax->entry->key, shorter, success);
                                        if (shorter)
                                        {
                                                root = deleteRightBalance(root,
shorter);
                                        }
                                }
                        }
                        return root;
                }
                void remove(K key)
                {
                        bool shorter = 0;
                        bool success = 0;
                        root = AVLDelete(root, key, shorter, success);
                }

                Node *search(Node *root, K key, vector<K> &traversedList)
                {
                        if (root == NULL)
                                return NULL;
                        if (traversedList.size() > 0)
                        {
                                if (root->entry->key == traversedList[0])
                                        return NULL;
                        }
                        traversedList.push_back(root->entry->key);
                        if (root->entry->key > key)
                                return search(root->left, key, traversedList);
                        else if (root->entry->key < key)
                                return search(root->right, key, traversedList);
                        else
                        {
                                traversedList.pop_back();
                                return root;
                        }
                }
                V search(K key)
```

```
		{
			vector<K> traversedList;
			Node *target = search(root, key, traversedList);
			if (target != NULL)
			{
				traversedList.clear();
				return target->entry->value;
			}
			else
			{
				traversedList.clear();
				throw "Not Found";
			}
		}

		void traverseNLR(Node *r, void (*func)(K key, V value))
		{

			if (r == NULL)
				return;
			(*func)(r->entry->key, r->entry->value);
			traverseNLR(r->left, func);
			traverseNLR(r->right, func);
		}
		void traverseNLR(void (*func)(K key, V value))
		{
			traverseNLR(root, func);
		}

		void clear(Node *root)
		{
			if (root == NULL)
				return;
			clear(root->left);
			clear(root->right);
			delete root;
		}
		void clear()
		{
			clear(root);
			delete (insertNode);
			insertNode = NULL;
		}
	};
};
```