**VE281 — Data Structures and Algorithms**

*Programming Assignment 3*

Instructor: Hongyi Xin

— UM-SJTU-JI (Fall 2020)

**Notes**

- Due Date: 12/7
- Submission: on JOJ

# 1 Introduction

In this project, you are asked to implement a STL-like K-D Tree (K-Dimensional Tree) data structure. Though there is no similar data structure in STL, you will follow the same type of APIs in other container.

The K-D Tree will be able to handle basic operations: initialization, insertion, deletion and finding minimum / maximum node on a dimension. There are also some advanced features of a K-D Tree, such as range search and nearest neighbor search, which are very useful in data analysis of high dimensional data. Luckily, they are not required in this project due to the hardness of the implementation.

Similar to the hash table you have implemented in project 2, key-value pairs are saved in the K-D Tree. The key is a high-dimensional data point, represented by `std::tuple`, in which the data type on each dimension can be customized. The value type is a template parameter of any data type, which does not affect the behavior of the K-D Tree.

You will continue to study the STL-like iterators in this project. You need to implement iterators in a tree structure. It is similar to the iterator in `std::map` or `std::set` (based on RB-tree).

# 2 Programming Assignment

## 2.1 The KDTree Template

### 2.1.1 Overview

In the project starter code, we define a template class for you:

```
// An abstract template base
template<typename...>
class KDTree;

// A partial template specialization
template<typename ValueType, typename... KeyTypes>
class KDTree<std::tuple<KeyTypes...>, ValueType> {
public:
    typedef std::tuple<KeyTypes...> Key;
    typedef ValueType Value;
    typedef std::pair<const Key, Value> Data;
    static inline constexpr size_t KeySize = std::tuple_size<Key>::value;
    static_assert(KeySize > 0, "Can not construct KDTree with zero dimension");
};
```

Note that parameter pack (...) is widely used in this project, it is a feature in the C++11 standard for more flexible template programming. Basically it means a pack of any number of template parameters (can be 0, 1 or more). Usually if the "..." is before a parameter name, it is a parameter pack; if the "..." is after a parameter name, it is an unpacking of the parameter pack.

At first we define an abstract template class with only one parameter pack as template argument, and the name of the parameter pack is omitted since we don't need it. Then we define a partial template specialization of the abstract template, so that we only accept `<std::tuple<KeyTypes...>, ValueType>` as template parameter. If other types of template parameter are provided, the compiler will fall back to the abstract template base and throw an compile error since `class KDTree` is an incomplete type.

For example, if you want a 2D Tree, each dimension is an integer and the saved value is also an integer, the actual type of the tree is `KDTree<std::tuple<int, int>, int>`.

Here the typedefs, `Key` and `Value`, are the types of the key-value pair stored in the K-D Tree. `KeySize` is the dimension of the key, and we use a `static_assert` to ensure the dimension is at least one during compilation. You can try to compile `KDTree<std::tuple<>, int>` and find what happened.

For ease of your implementation, we assume all data types in `Key` can be compared by `std::less` and `std::greater`, so that you don't need to write customized comparators for the K-D Tree.

Don't be too afraid of these new grammars in C++11, at least we've already defined all of the classes and functions for you and you don't need to write anything related to parameter packs.

### 2.1.2 Internal Data Structures

We've already defined the internal data structure (node) for you in this project.

```
struct Node {
    Data data;
    Node *parent;
    Node *left = nullptr;
    Node *right = nullptr;
}
```

The parent of the root node should be `nullptr`, and the tree only need to save the root node. It's a very trivial definition, but it should be enough for the whole project.

### 2.1.3 Iterators

In hash table, we use a self-defined iterator containing two STL iterators (of vector and list). Iterators for these linear data structure can be simple implemented: for a vector, you only need to advance the index; for a list, you only need to make it pointing to the next node. However, when iterating a tree, it's different that you need to follow a certain tree traversal order.

The definition of the iterator is also trivial. You only need to record a pointer to the K-D Tree, and a pointer to the current node.

```
class Iterator {
private:
```

```
    KDTree *tree;
    Node *node;
}
```

We also provide the begin and end methods for you. The begin method finds the left most leaf of the tree, and the end method uses nullptr as the current node.

Your task is to write the increment and the decrement method in the Iterator class. You should use a depth-first in-order traversal of the tree to increment the iterator, which means, when you have a full iteration of the tree and print each node, the order of the output should be the exactly same as an depth-first in-order traversal.

Here's a detailed explanation about the increment method. When a increment occurs, if the current node has a right subtree, the next node should be the left most leaf in the right subtree; otherwise (if the current node doesn't have a right subtree), you should move up (by parent pointer) and find the first ancestor node so that the current node is in the left subtree of the ancestor node. When you increment the right most leaf node in the tree, you'll find that the node is not in any of its ancestors' left subtree, so you should end the loop and set the next node as nullptr.

The decrement method is a reverse of the increment method, think about how to implement it by yourself.

The behavior of doing an increment on the end iterator is an undefined behavior. Similarly, doing an decrement on the begin iterator is also an undefined behavior. For ease of debugging, you can throw an error if these operations happened, but we won't test your code with these cases. Note that doing an decrement on the end iterator is allowed, which will return the right most leaf node.

If all of your implementation is correct, range-for[1] loops will be automatically supported for the K-D Tree. Try this to evaluate your code:

```
for (auto &item : kdTree) {
    cout << item.second << endl;
}
```

### 2.1.4 The Dynamic Methods

There are three "dynamic" methods implemented in the starter code: findMinDynamic, findMaxDynamic and eraseDynamic. They are only for your reference, and you do not need to call them in your implementation. You can try to understand these functions and use them in the testing.

## 2.2 Operations

### 2.2.1 Initialization

To initialize an empty K-D Tree, you can set the root to nullptr.

To initialize a K-D Tree with another K-D Tree, you should traverse and make a deep copy of all nodes in that tree.

To initialize a K-D Tree with a vector of data points, a trivial idea is to insert the data points one by one, the time complexity is $\mathcal{O}(kn \log n)$ obviously. However, it is very likely to form a not balanced tree and lead to a poor performance. A better idea is to find the median point of the current dimension so that the data points can be equally partitioned into the left and right subtree.

```
function KDTree(data, parent, depth):
    if data is empty then
        return null;
    end
    dimension ← depth mod k;
    median ← the median point of data on dimension;
    partition data into left, median and right;
    node.key ← median;
    node.parent ← parent;
    node.left ← KDTree(left, node, depth + 1);
    node.right ← KDTree(right, node, depth + 1);
end
```

**Algorithm 1:** Construction of tree.

Before inserting the data, you should make sure there is no duplication of key. A simple method is to run a stable sort (`std::stable_sort`) on the data, and then use `std::unique` to remove duplicate keys with reverse iteration (so that the latest value of the same key is preserved).

Hint: You can use `rbegin` and `rend` for reverse iteration, and get the corresponding forward iterator by `it.base()`.

Recall the linear time selection algorithm, the time complexity of finding the median and partitioning the vector is $\mathcal{O}(kn)$, according to the Master theorem, the overall time complexity is also $\mathcal{O}(kn \log n)$. If there are even number of elements in a vector, use the left one as the median point, this may lead to some unbalance to the tree, but mostly it can be ignored.

Hint: you can use STL functions to efficiently find the median and partition the vector. Check `std::nth_element`. You may also need the `compareKey` function to compare tuples on a certain dimension, it is already implemented in the starter code.

```cpp
template<size_t DIM, typename Compare>
static bool compareKey(const Key &a, const Key &b, Compare compare = Compare()) {
    if (std::get<DIM>(a) != std::get<DIM>(b)) {
        return compare(std::get<DIM>(a), std::get<DIM>(b));
    }
    return compare(a, b);
}
```

**You should use this function whenever two keys need to be compared on a certain dimension to ensure a strict ordering of keys in the tree, including initialization, insertion, deletion and etc.**

Additionally, you'll need to implement both the copy constructor and overload the = operator, such that the following statements initiate deep copying:

```cpp
KDTree t2(t1);
KDTree t3;
t3 = t1;
```

### 2.2.2 Insertion and Find Minimum / Maximum

The pseudocode is omitted here because detailed explanations for these operations are already in the lecture slides. Think carefully about what's the difference of finding minimum and maximum, do not directly copy the code.

We'll briefly explain the template parameter for dimension here. The `findMin` method has the following definition:

```cpp
template<size_t DIM_CMP, size_t DIM>
Node *findMin(Node *node) {
    constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
    // TODO: implement this function
}
```

The first template parameter `DIM_CMP` means the dimension where nodes should be compared on, the second template parameter `DIM` means the dimension of the current node ($depth \mod k$ in the tree). We help you define the next dimension `DIM_NEXT` which can be used as template parameter recursively or used in other template methods.

For example, you can use `findMin<DIM_CMP, DIM_NEXT>(node->left)` to recursively find the minimum node in the left subtree on `DIM_CMP`; you can also use `compareNode<DIM_CMP, std::less<> >` to compare the nodes on `DIM_CMP`.

### 2.2.3 Deletion

The deletion operation is a bit more complex, we'll also provide the pseudocode for it.

Notice that in deletion, we first search for the minimum element in the right subtree, before proceeding to the maximum element in the left subtree.

```
function Delete(node, key, depth):
    if node is null then
        return null;
    end
    dimension ← depth mod k;
    if key = node.key then
        if node is a leaf then
            delete node directly;
            return null;
        else if node has right subtree then
            minNode ← the minimum node on dimension in the right subtree node.right;
            node.key ← minNode.key;
            node.value ← minNode.value;
            node.right ← Delete(node.right, minNode.key, depth + 1);
        else if node has left subtree then
            maxNode ← the maximum node on dimension in the left subtree node.left;
            node.key ← maxNode.key;
            node.value ← maxNode.value;
            node.left ← Delete(node.left, maxNode.key, depth + 1);
        end
    else
        if key < node.key on dimension then
            node.left ← Delete(node.left, key, depth + 1);
        else
            node.right ← Delete(node.right, key, depth + 1);
        end
    end
    return node;
end
```

**Algorithm 2:** Deletion of node.

Hint: In order to find the minimum / maximum on the current dimension in the left / right subtree, the comparison dimension should be the current dimension, and the starting dimension should be next of the current dimension. Think carefully about how to call the `findMin` and `findMax` method.

## 2.3 Study the Time Complexity versus *k*

In binary trees, the time complexity of the operations are only based on $n$. However, you should also take $k$ into consideration in K-D Trees. Your task is to study the performance of the finding minimum operation.

Use different $k$ values to construct K-D Trees. You may find it difficult to define high dimensional trees. We provide a helper function with you to resolve this issue.

```cpp
template<size_t SIZE, typename T>
auto vector2tuple(std::vector<T> &vec) {
    T t = T();
```

```
    if (!vec.empty()) {
        t = vec.back();
        vec.pop_back();
    }
    if constexpr (SIZE > 0) {
        return std::tuple_cat(vector2tuple<SIZE - 1, T>(vec),
                              std::make_tuple<T>(std::move(t)));
    } else {
        return std::make_tuple<>();
    }
}
```

The `vector2tuple` helper function can transform a vector of any type into a tuple of the type of a predefined size. Now you can simply define a 10-D int tree with a keyword `decltype`, which can get the type of any variable during compilation time.

```
std::vector<int> vec;
auto tuple = vector2tuple<10>(vec);
KDTree<decltype(tuple), int> kdTree;
```

You can also use vectors to initialize the tree with `vector2tuple`. We suggest that you use the same input to initialize the trees (for small dimensions you can only use the first $k$ dimensions in the input).

The main purpose of this study is to find the relationship of time complexity and $k$, so you can fix $n$ as a large integer and compare the results for different $k$.

Besides, try to induce the time complexity of finding minimum with respect to both $n$ and $k$. Compare the theoretical time complexity with your results.

Write about **one page** on the report about your findings, plots and explanations.

### 2.3.1 Hints

- The performance of programs on Windows is usually not stable, so you should do the experiments on a Unix based system.
- You may want to write another program to do this study.
- You can use the C++11 pseudo-random number generation library to generate more randomized numbers (instead of using `std::rand`).
- You can use the C++11 chrono library to get more accurate runtime of functions than `std::clock`.
- (Optional) You can use GNU Perf (only available on Linux) to find the bottleneck of your implementation.
- Although the major factor that affects the runtime is the size of the input array, however, the runtime for an algorithm may also weakly depend on the detailed input array. Thus, for each size, you should generate a number of arrays of that size and obtain the mean runtime on all these arrays. Also, for fair comparison, the same set of arrays should be applied to all the data structures.
- You should try at least 5 representative sizes.

# 3 Implementation Requirements and Restrictions

## 3.1 Requirements

- You must make sure that your code compiles successfully on a Linux operating system with `g++` and the options `-std=c++1z -Wconversion -Wall -Werror -Wextra -pedantic`.
- You should not change the definitions of the functions in `kdtree.hpp`.
- You can define helper functions, don't forget to mark them as `protected` or `private`.
- You should only hand in one file `kdtree.hpp`.
- You can use any header file defined in the `C++17` standard. You can use cppreference as a reference.

You only need to implement the methods (functions) marked with "TODO" in the file `hashtable.hpp`. Here is a list of the methods (functions):

- `increment` (in iterator)
- `decrement` (in iterator)
- `find`
- `insert`
- `findMin`
- `findMax`
- `erase`
- two constructors, destructor and `operator=`

Please refer to the descriptions of these functions in the starter code.

## 3.2 Memory Leak

You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call valgrind, which can tell whether you have any memory leaks. (You need to install valgrind first if your system does not have this program.) The command to check memory leak is:

```
valgrind --leak-check=full <COMMAND>
```

You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

```
./main < input.txt
```

causes memory leak, then `<COMMAND>` should be "`./main < input.txt`". Thus, the command will be

```
valgrind --leak-check=full ./main < input.txt
```

# 4 Grading

Your program will be graded along five criteria:

## 4.1 Functional Correctness

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program.

## 4.2 Implementation Constraints

We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points.

## 4.3 General Style

General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm.

## 4.4 Performance

We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, you will lose the performance score for those test cases.

## 4.5 Report on the performance study

Finally, we will also read your report and grade it based on the quality of your performance study.

# 5 Acknowledgement

The programming assignment is co-authored by Yihao Liu, an alumni of JI and the chief architect of JOJ.

# References

[1] Range-based for loop - cppreference: `https://en.cppreference.com/w/cpp/language/range-for`

## Appendix

**kdtree.hpp**

```cpp
#include <tuple>
#include <vector>
#include <algorithm>
#include <cassert>
#include <stdexcept>

/**
 * An abstract template base of the KDTree class
 */
template<typename...>
class KDTree;

/**
 * A partial template specialization of the KDTree class
 * The time complexity of functions are based on n and k
 * n is the size of the KDTree
 * k is the number of dimensions
 * @typedef Key         key type
 * @typedef Value       value type
 * @typedef Data        key-value pair
 * @static  KeySize     k (number of dimensions)
 */
template<typename ValueType, typename... KeyTypes>
class KDTree<std::tuple<KeyTypes...>, ValueType> {
public:
    typedef std::tuple<KeyTypes...> Key;
    typedef ValueType Value;
    typedef std::pair<const Key, Value> Data;
    static inline constexpr size_t KeySize = std::tuple_size<Key>::value;
    static_assert(KeySize > 0, "Can not construct KDTree with zero dimension");
protected:
    struct Node {
        Data data;
        Node *parent;
        Node *left = nullptr;
        Node *right = nullptr;

        Node(const Key &key, const Value &value, Node *parent) : data(key, value), parent(parent) {}

        const Key &key() { return data.first; }

        Value &value() { return data.second; }
    };

```

```cpp
public:
    /**
     * A bi-directional iterator for the KDTree
     * ! ONLY NEED TO MODIFY increment and decrement !
     */
    class Iterator {
    private:
        KDTree *tree;
        Node *node;

        Iterator(KDTree *tree, Node *node) : tree(tree), node(node) {}

        /**
         * Increment the iterator
         * Time complexity: O(log n)
         */
        void increment() {
            // TODO: implement this function
        }

        /**
         * Decrement the iterator
         * Time complexity: O(log n)
         */
        void decrement() {
            // TODO: implement this function
        }

    public:
        friend class KDTree;

        Iterator() = delete;

        Iterator(const Iterator &) = default;

        Iterator &operator=(const Iterator &) = default;

        Iterator &operator++() {
            increment();
            return *this;
        }

        Iterator operator++(int) {
            Iterator temp = *this;
            increment();
            return temp;
        }
```

```cpp

        Iterator &operator--() {
            decrement();
            return *this;
        }

        Iterator operator--(int) {
            Iterator temp = *this;
            decrement();
            return temp;
        }

        bool operator==(const Iterator &that) const {
            return node == that.node;
        }

        bool operator!=(const Iterator &that) const {
            return node != that.node;
        }

        Data *operator->() {
            return &(node->data);
        }

        Data &operator*() {
            return node->data;
        }
    };

protected:                      // DO NOT USE private HERE!
    Node *root = nullptr;       // root of the tree
    size_t treeSize = 0;        // size of the tree

    /**
     * Find the node with key
     * Time Complexity: O(k log n)
     * @tparam DIM current dimension of node
     * @param key
     * @param node
     * @return the node with key, or nullptr if not found
     */
    template<size_t DIM>
    Node *find(const Key &key, Node *node) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        // TODO: implement this function
    }

```

```cpp
    /**
     * Insert the key-value pair, if the key already exists, replace the value only
     * Time Complexity: O(k log n)
     * @tparam DIM current dimension of node
     * @param key
     * @param value
     * @param node
     * @param parent
     * @return whether insertion took place (return false if the key already exists)
     */
    template<size_t DIM>
    bool insert(const Key &key, const Value &value, Node *&node, Node *parent) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        // TODO: implement this function
    }

    /**
     * Compare two keys on a dimension
     * Time Complexity: O(1)
     * @tparam DIM comparison dimension
     * @tparam Compare
     * @param a
     * @param b
     * @param compare
     * @return relationship of two keys on a dimension with the compare function
     */
    template<size_t DIM, typename Compare>
    static bool compareKey(const Key &a, const Key &b, Compare compare = Compare()) {
        if (std::get<DIM>(a) != std::get<DIM>(b)) {
            return compare(std::get<DIM>(a), std::get<DIM>(b));
        }
        return compare(a, b);
    }

    /**
     * Compare two nodes on a dimension
     * Time Complexity: O(1)
     * @tparam DIM comparison dimension
     * @tparam Compare
     * @param a
     * @param b
     * @param compare
     * @return the minimum / maximum of two nodes
     */
    template<size_t DIM, typename Compare>
    static Node *compareNode(Node *a, Node *b, Compare compare = Compare()) {
        if (!a) return b;
```

```
        if (!b) return a;
        return compareKey<DIM, Compare>(a->key(), b->key(), compare) ? a : b;
    }

    /**
     * Find the minimum node on a dimension
     * Time Complexity: ?
     * @tparam DIM_CMP comparison dimension
     * @tparam DIM current dimension of node
     * @param node
     * @return the minimum node on a dimension
     */
    template<size_t DIM_CMP, size_t DIM>
    Node *findMin(Node *node) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        // TODO: implement this function
    }

    /**
     * Find the maximum node on a dimension
     * Time Complexity: ?
     * @tparam DIM_CMP comparison dimension
     * @tparam DIM current dimension of node
     * @param node
     * @return the maximum node on a dimension
     */
    template<size_t DIM_CMP, size_t DIM>
    Node *findMax(Node *node) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        // TODO: implement this function
    }

    template<size_t DIM>
    Node *findMinDynamic(size_t dim) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        if (dim >= KeySize) {
            dim %= KeySize;
        }
        if (dim == DIM) return findMin<DIM, 0>(root);
        return findMinDynamic<DIM_NEXT>(dim);
    }

    template<size_t DIM>
    Node *findMaxDynamic(size_t dim) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        if (dim >= KeySize) {
            dim %= KeySize;
```

```cpp
        }
        if (dim == DIM) return findMax<DIM, 0>(root);
        return findMaxDynamic<DIM_NEXT>(dim);
    }

    /**
     * Erase a node with key (check the pseudocode in project description)
     * Time Complexity: max{O(k log n), O(findMin)}
     * @tparam DIM current dimension of node
     * @param node
     * @param key
     * @return nullptr if node is erased, else the (probably) replaced node
     */
    template<size_t DIM>
    Node *erase(Node *node, const Key &key) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        // TODO: implement this function
    }

    template<size_t DIM>
    Node *eraseDynamic(Node *node, size_t dim) {
        constexpr size_t DIM_NEXT = (DIM + 1) % KeySize;
        if (dim >= KeySize) {
            dim %= KeySize;
        }
        if (dim == DIM) return erase<DIM>(node, node->key());
        return eraseDynamic<DIM_NEXT>(node, dim);
    }

    // TODO: define your helper functions here if necessary

public:
    KDTree() = default;

    /**
     * Time complexity: O(kn log n)
     * @param v we pass by value here because v need to be modified
     */
    explicit KDTree(std::vector<std::pair<Key, Value>> v) {
        // TODO: implement this function
    }

    /**
     * Time complexity: O(n)
     */
    KDTree(const KDTree &that) {
        // TODO: implement this function
```

```
    }

    /**
     * Time complexity: O(n)
     */
    KDTree &operator=(const KDTree &that) {
        // TODO: implement this function
    }

    /**
     * Time complexity: O(n)
     */
    ~KDTree() {
        // TODO: implement this function
    }

    Iterator begin() {
        if (!root) return end();
        auto node = root;
        while (node->left) node = node->left;
        return Iterator(this, node);
    }

    Iterator end() {
        return Iterator(this, nullptr);
    }

    Iterator find(const Key &key) {
        return Iterator(this, find<0>(key, root));
    }

    void insert(const Key &key, const Value &value) {
        insert<0>(key, value, root, nullptr);
    }

    template<size_t DIM>
    Iterator findMin() {
        return Iterator(this, findMin<DIM, 0>(root));
    }

    Iterator findMin(size_t dim) {
        return Iterator(this, findMinDynamic<0>(dim));
    }

    template<size_t DIM>
    Iterator findMax() {
        return Iterator(this, findMax<DIM, 0>(root));
```

```
    }

    Iterator findMax(size_t dim) {
        return Iterator(this, findMaxDynamic<0>(dim));
    }

    bool erase(const Key &key) {
        auto prevSize = treeSize;
        erase<0>(root, key);
        return prevSize > treeSize;
    }

    Iterator erase(Iterator it) {
        if (it == end()) return it;
        auto node = it.node;
        if (!it.node->left && !it.node->right) {
            it.node = it.node->parent;
        }
        size_t depth = 0;
        auto temp = node->parent;
        while (temp) {
            temp = temp->parent;
            ++depth;
        }
        eraseDynamic<0>(node, depth % KeySize);
        return it;
    }

    size_t size() const { return treeSize; }
};
```