Mini-Projects

Introduction

You will have until **January 21st, 2026**, to complete and submit your mini-project, and you will present your project on **January 28th, 2026**, the day of your exams. The mini-project will entail you implementing one non-conventional data structure from scratch in C++, verifying its correctness using unit testing, and demonstrating the operation of your data structure during the evaluation, as well as reporting the overall status of your mini-project.

Through this mini-project, I hope to teach you to consider the trade-offs that are made when designing data structures. This will allow you to think critically about the data structures you use later in your career, and use alternative implementations where desired/necessary, or perhaps even roll your own, if no viable alternatives exist.

For this project, you are expected to learn about your data structure on your own. A brief explanation will be provided, along with some references that you can use as a starting point for your search but figuring out all the details of your data structure is up to you. You may encounter sources that say that there are multiple options to solve a certain problem. If this is the case, it is up to you to decide which option you think is best. If this happens, make sure to write down somewhere why you made the decision that you did. This is one of the trade-offs mentioned in the previous paragraph, and you should be able to argue why the trade-offs you made are the right ones.

The rest of this document contains a list of potential data structures that you can choose from for your mini-project. You will have to indicate your choice of data structure through a Blackboard quiz (no grades will be given) and submit your choice before the start of the first session after the projects have been presented (Deadline: **November 26th at 13:45**).

Project

The data structures listed in this document all have a comparable complexity; this also means that all data structures will be graded equally. It is not the case that some data structures will get you more points than others because they are more or less complex than others.

You can use implementations available online of your data structure as a learning resource, but you are not allowed to directly copy-paste (parts of) online implementations of your mini-project. If you get caught plagiarizing others' work, you will receive a grade of 0 for the lab. This rule also counts for projects from students from previous years.

You are expected to deliver a project written in C++, using the CMake build system that we used throughout the labs. Your project should compile with at least one of the big three compilers (GCC, Clang or MSVC).

For each data structure, you will be given a specification in the form of a set of declarations. These may be class declarations or function declarations. Since these are declarations, they should be placed in header files, and their implementations should be placed in source files. You are allowed to take the declarations in the specification, and spread them out over multiple header files to keep your project organized. These declarations are not always complete, you may have to add some class members or methods or fill in some return types to make everything work. You should ensure that the classes, methods, functions, ... specified in the specification have the exact same name, parameters, qualifiers, ... in your solution.

Evaluation Day

At the end of the semester, you will present your mini-project during the examination period. You will do so using a 2-slide presentation in English, and a live demonstration of your choosing. In your presentation you will have 1 slide mention what data structure you implemented, and you will detail the work you have finished, and 1 slide detailing the work you were not able to finish. Lecturers will not ask questions about your work.

The format of your live demonstration is left up to you, and you are encouraged to be creative! Your demonstration must be executed live, this means that pre-recorded videos etc. will not be accepted. Beyond this you are free to choose how you demonstrate your data structure: Using an interactive program (CLI or GUI), a simulation with a live visualization, a practical use-case, …

On the day of the evaluation, during the examination period, you have to be present in the classroom 15 minutes before your assigned timeslot. When it is time, you will present your work, and when you are done presenting, you can leave. You are allowed to stay and watch the other students' presentations if you want, as long as you do so without disturbing them.

Helpful Tips

Below is a list of helpful tips you can use throughout the project to make your life a little easier.

1. The data structures listed in this project do not exist in the C++ standard library, but they can usually be created by combining one or more data structures from the standard library, with a couple of helper classes and functions to act as the glue. Make use of this!
2. Most containers in the C++ standard library define type aliases for their iterators. You can make use of these to avoid having to define your own iterators! (std::vector, std::map, ...)
3. Most search engines allow you to limit your searches to a single website. You can do this by appending "site:www.website.com" to your search query, for example "std::vector site:cppreference.com" will take you straight to std::vector's documentation. This is very useful for example if you only want results from cppreference.com or stackoverflow.com
4. To keep a software project organized (In any language) its best to stick to having 1 file per class. If you have a very small utility class that is not needed by any other part of your code, you don't need to put it in a separate file of course.
5. In C++, there are headers (.hpp, .h, ...) and source files (.cpp, .cc, .cxx., ...). It is best that for each source file, you have 1 corresponding header file that declares (but does not define) the symbols in the source file. Try to learn the differences between both and what goes where!
6. Use version control systems! I won't be going into the details of git in this subject, but the use of some form of version control can help you a lot during your project. For example, say you are near the deadline, and you want to implement an extra feature, but you don't know if you'll be able to get it working on time. At this point, you can create a separate branch with your new feature, and merge it when it is complete. Didn't finish the feature on time? No worries! Just submit the branch that doesn't have the new feature!
7. When using any kind of hash-based data structure (std::unordered_map, std::unordered_set, ...) your key type must have operator== implemented (For checking key-equality in case of a hash-collision), and you must have specialized std::hash for your key type (To compute actual hash values).
8. Make sure to document the rules with regards to iterator invalidation for your data structure if your data structure has methods returning iterators (begin() and end())
9. CLion supports the use of the \`clang-format\` code formatting tool and the \`clang-tidy\` linting tool (clang-format, clang-tidy). These tools can be activated by placing an empty \`.clang-format\` or \`.clang-tidy\` configuration file at the root of your project, and selecting "Enable clang-format" on the bottom right of your screen where it normally says "Tab" or "2 Spaces". With these tools, you can automatically format your code (Ctrl + Alt + L) according to the configuration in your configuration files, helping you to keep your project clean and organized!
10. C++ compilers are very advanced pieces of software. When they emit a warning, they are usually correct. With this in mind, I recommend that you add some compiler flags in your CMakeLists.txt to turn on as many warnings as possible, and convert them to errors. For Clang and GCC, the correct flags are -Wall -Wextra -Wpedantic -Werror, for MSVC the correct flags are /W4 /WX. Keep in mind that, for MSVC at least, you will probably get some false positives. If this is the case, you can disable a warning locally, or you can disable a single warning for your entire project.
11. If you want to build a basic 2D visualization in C++ for your demonstration the SDL2 library is an excellent choice. LazyFoo has a great tutorial series on how to use SDL2 for basic 2D graphics.

Contents

Evaluation Criteria

Your project will be evaluated on the criteria listed in the table below.

No.	Criterion	Weight
1	Did you implement the required functionality? (See specification of your data structure for details)	7
2	Does your interface match the interface given in the assignment?	7
3	Does your data structure work correctly?	7
4	Do you have at least 2 unit tests for each of the functions/methods mentioned in the assignment? One unit test should check a "normal" example, while another should check an edge case.	6
5	Did you package your data structure into a separate library?	3
6	Did your presentation only have 2 slides? (An additional title and ending slide are allowed)	2
7	Did your demonstration showcase all the functionality you were required to implement? (See specification of your data structure for details)	2
8	Did the combination of your presentation and demonstration take 5 minutes or less?	2
9	Do all your operations have the correct time-complexity? (See specification of your data structure for details, if no desired time-complexity is given, this can be ignored)	2
10	Does your data structure have the correct space-complexity? (See specification of your data structure for details)	2

Your final score for the lab is computed as a weighted average, using your score and the weights listed above.

Quadtree

A quadtree is a tree data structure used to recursively partition a two-dimensional space. It is often used in games to simplify collision checking. If we were to check every single object against every single other objects, we'd have on the order of O(N^2) collision checks to do! Using quad trees, we can significantly cut down on this, by only considering the objects that are close to each other.

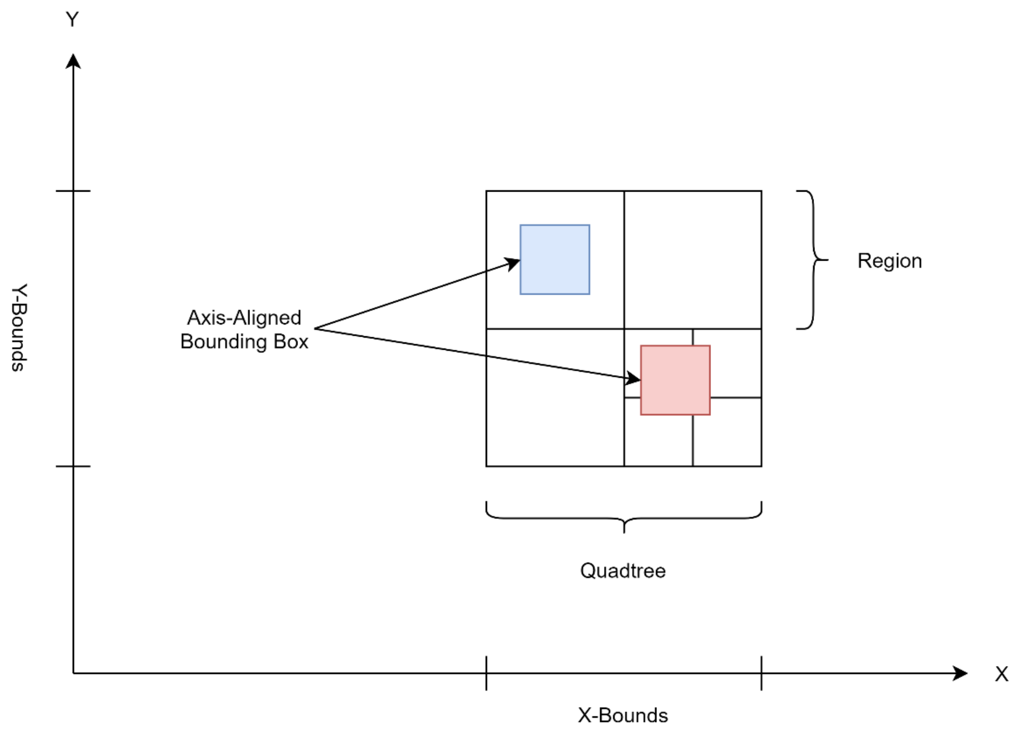

Figure 1: An illustration of a Quadtree with some terminology.

Quadtrees recursively subdivide a 2D space, until some criterion is met. In our implementation, we will subdivide until there are at most `region_capacity` objects in a single region. `region _capacity` Is given in the Quadtree's constructor, and can be different for each quadtree instance.

In our lab, we will build a region quadtree with square subspaces that covers a finite part of a 2D space. The bounds of the space that our quadtree covers are given in its constructor using the `bounds` argument. The quadtree should be able to store some arbitrary metadata (such as an ID) and an Axis-Aligned Bounding Box (AABB) for each entry. The type of the metadata should be parameterized using a template parameter.

Quadtrees also have a three-dimensional equivalent, called Octrees. In this lab, we will ignore octrees, but many of the algorithms that work for octrees, also work for quadtrees. Thus, octrees provide a useful resource if you want to learn about quadtrees.

Resources

Quadtree (Wikipedia)

Octree (Wikipedia)

A Simple and Efficient Quadtree Implementation (Github, Java)

Quick Tip: Use Quadtrees to Detect Likely Collisions in 2D Space (gamdevelopment.tutplus.com, Java)

Specification

You can add as many fields, methods, functions, files, … as you want, but at the bare minimum, this should be implemented. There are also a couple of things that you still need to fill in in this specification.

```
// This class represents an axis-aligned bounding box
// In a 2D problem, this is simply a rectangle whose sides are
// parallel to the X- and Y-axis
class AxisAlignedBoundingBox
{
    // This friend function (A free function that can
    // access private fields) should check if two AABBs overlap
    // It returns true if there is overlap,
    // it returns false if there isn't
    friend bool collides(
        const AxisAlignedBoundingBox& one,
        const AxisAlignedBoundingBox& two
    );
}

// This class is our actual quadtree
// It stores AABB's together with some metadata
// Space Complexity: O(N)
template <typename MetadataType>
class Quadtree
{
    public:
        // Constructor
        // `bounds` specifies the edges of the region that
        // the quadtree covers.
        // `region_capacity` specifies the maximum number of objects
        // in a single region.
        Quadtree(
            const AxisAlignedBoundingBox& bounds,
            unsigned int region_capacity
        );

        // This method inserts the given metadata and
        // AABB into the quadtree.
        void insert(
            const AxisAlignedBoundingBox& aabb,
            const MetadataType& meta
        );

        // This method should return a std::unordered_set of
        // all items inside the given AABB.
        // The given AABB may span multiple square regions of
        // the quadtree.
        // The worst-case time-complexity of this method should be
        // O(log(N)) for a Quadtree with N leaf nodes
        // TODO: You should decide the element type of the
        // std::unordered_set
        //     Your set should contain the AABB and the Metadata of
        // all objects in the given region.
        std::unordered_set<auto> query_region(
            const AxisAlignedBoundingBox& aabb
        ) const;

        // TODO: You should decide the return type for this method
        // This method should return an iterator that runs over
        // every element in the quadtree.
        auto begin();

        // TODO: You should decide the return type for this method
        // This method should return an iterator pointing
        // one past the last element in the quadtree.
        // When iterating from begin() to end(),
        // we should visit all elements in the quadtree.
        // There is no requirement with regards to
        // the order that you visit these elements in.
        auto end();
};
```

L-System Interpreter

L-Systems are an incredibly simple, and an incredibly powerful tool for generating realistic looking plant geometry. They work by iteratively performing string replacements according to a specific set of rules (productions). The strings produced by L-Systems can then be interpreted using Turtle graphics, to yield plant-like images.

At every iteration, an L-System interpreter looks through the current string and replaces each symbol with the replacement indicated by the matching production. The generation process starts from an initial string, usually called an "axiom". The set of used symbols is given by the L-System's alphabet. How replacements are carried out, is determined by the productions in the L-System. Each production is a single rule that says which character to look for, and what string to replace it with.

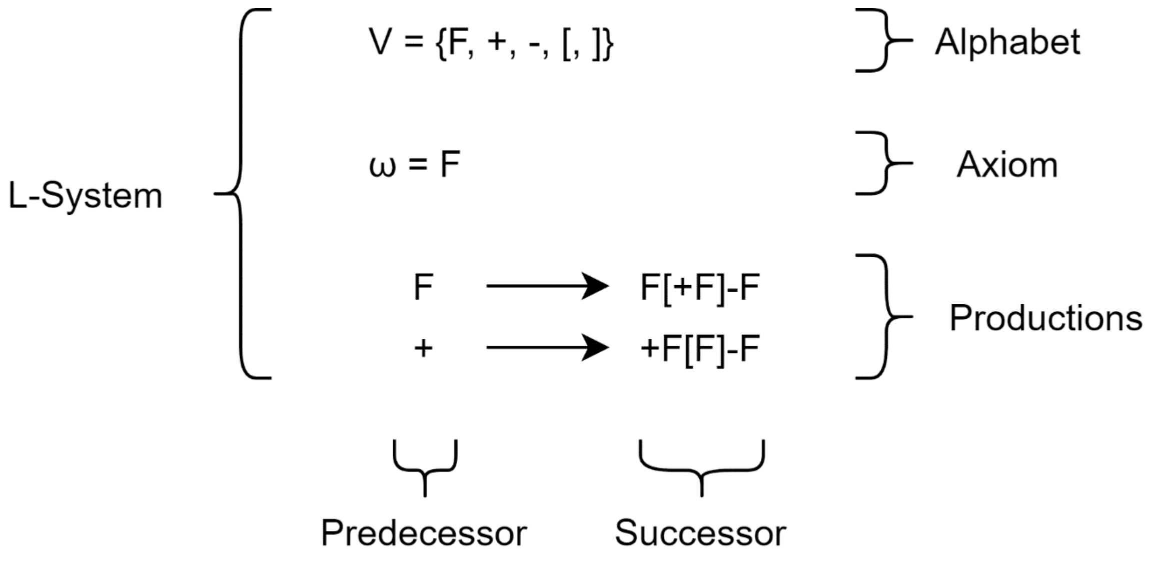

Figure 2: An illustration of an L-System, with some terminology.

You DO NOT have to consider the geometric interpretation of these symbols. You just need to write the necessary classes to generate strings

Tip: For your demonstration, you could use python's built-in "turtle" package! You can use your C++ data structure to evaluate the L-System, and then store the output in a file, that you read in using a Python script built on the "turtle" package!

You can also use online tools like this one to verify the correct operation of your L-System.

Resources

The Algorithmic Beauty of Plants (Book, by the inventor of L-Systems)

Plants are Recursive!!: Using L-Systems to Generate Realistic Weeds by Sher Minn Chong (Conference Talk, YouTube)

Interactive L-System Generator (GitHub, JavaScript)

L-Systems in Unity (GitHub, C#)

A blogpost about L-Systems and their geometric representation through Turtle graphics (Blog post, Github.io)

Specification

You can add as many fields, methods, functions, files, … as you want, but at the bare minimum, this should be implemented. There are also a couple of things that you still need to fill in in this specification.

```
// This class represents a single production.
// A production is a replacement rule, it tells us which symbol
// to look for, and what (sequence of) symbol(s) to replace it with.
// The original symbol is usually called a “predecessor”.
// The (sequence of) symbol(s) is usually called a “successor”.
template <typename SymbolType>
class Production
{
};

// This function verifies that all symbols in the production
// (Predecessor and all symbols in the successor) are
// in the given alphabet.
// If all symbols are in the alphabet, this function returns true,
// otherwise, this function returns false.
// The time-complexity of this function should be be O(N)
// where N is the total number of symbols in the production
// (predecessor + successors)
template <typename SymbolType>
bool isValidProduction(
    const Production<SymbolType>& production,
    const std::unordered_set<SymbolType>& alphabet
);

// This class represents the actual L-System.
// It contains an axiom (Initial state), A set of productions
// (Replacement rules) and an alphabet (List of allowed symbols).
template <typename SymbolType>
class LSystemInterpreter
{
    public:
    // This constructor takes an axiom, a set of productions and
    // an alphabet and stores these in the fields.
    // It should check that all productions are valid using
    // the “isValidProduction” function above.
    // If any of the productions are invalid,
    // it should throw an exception.
    // The constructor should also check that
    // every production has a unique predecessor.
    // It should also check that there is a Production for
    // each symbol in the alphabet.
    // If there is a symbol in the alphabet without a production,
    // you can decide what to do:
    //   1. Throw an exception
    //   2. Add an identity production (A -> A)
    LSystemInterpreter(
        const std::vector<SymbolType>& axiom,
        const std::unordered_set<Production<SymbolType>>& productions,
        const std::unordered_set<SymbolType>& alphabet
    );

    // After `operator()` has been called one or more times,
    // the L-system will have accumulated an internal state.
    // This method should reset this internal state,
    // so the next call to `operator()` starts from
    // the L-system's axiom again.
    void reset();

    // This function should execute a single iteration of the L-System.
    // When this method is called twice,
    // the second call should use the result of the first call
    // as its starting point.
    // This allows us to use `std::generate()` to iteratively
    // execute the L-System.
    //
    // Be careful when applying your productions!
    // Your productions should always be applied at the same time!
    // An example:
    // We have 2 productions: A -> AB, B -> A, and an axiom "ABA"
    // We can apply the first production first, and get:
    // "ABBAB"
    // And then we can apply the second production, to get:
    // "AAAAA"
    // THIS IS WRONG!
    //
    // You should always execute all productions on the original text!
    // So, in this case, the correct output would be:
    // "ABAAB"
    // (The first and last A's were replaced by AB,
    // and the middle B by A)
    std::vector<SymbolType> operator() () const;
};
```

N-Dimensional Pareto Front

A Pareto front is a data structure commonly used in multi-objective optimization and decision making scenarios. It contains a set of solutions, each of which represents an optimal trade-off between N objectives. Only solutions which are not Pareto-dominated will be kept in a Pareto front.

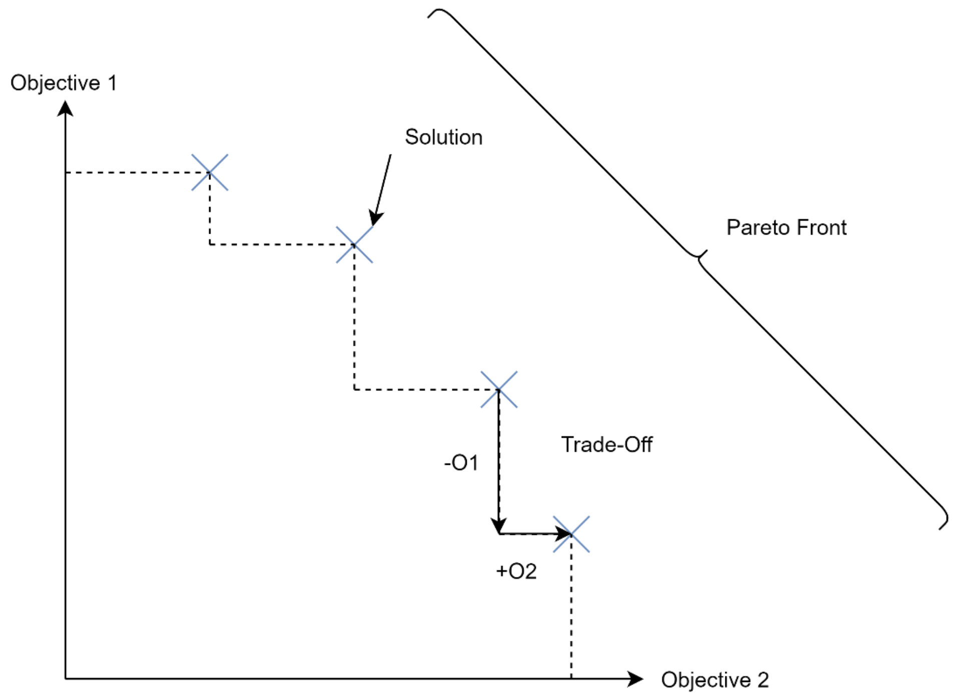

Figure 3: An example of a 2-dimensional Pareto-front

Each point on the Pareto front is called a single solution, and represents one possible way of solving a problem. Each axis of the Pareto front represents one objective, or one evaluation criterion, that we use to evaluate solutions and compare them against each other.

A solution is part of a Pareto front if, and only if, it is not Pareto dominated by any other solution in the front. A solution "A", Pareto dominates another solution "B", if "A" scores equal to "B" on all objectives, and if "A" scores better than "B" on at least one objective.

An important issue that arises with the use of Pareto fronts is choosing a solution, after all, we can only implement one solution to a problem, not all of them. In this case, we will consider linear preference vectors to select the best solution. A linear preference vector is an N-dimensional vector, given by a user, that reflects the importance of the different objectives considered. Its elements usually range from 0 to 1, and also sum to 1. For example, when we have a 2-objective problem, and the user values the first objective about twice as much as the second objective, their preference vector could be [0.33, 0.67].

Selecting the correct solution is done by taking the dot product of a solution's score vector, and the user's preference vector. For example, say we have a solution with score vector [10, 20] and another solution with score vector [20, 10]. Using the preference vector from the previous example, we can calculate a combined score (scalar value) for both solutions: (10 * 0.33) + (20 * 0.67) = 16.7 for the first solution and (20 * 0.33) + (10 * 0.67) = 13.3 for the second solution. Since the first solution has the highest scalar value here, we consider that to be "the best" solution.

Resources

[Pareto Front (Wikipedia)]

[Pareto Optimality (Springer Publishing)]

[Generate and Plot Pareto Front (MATLAB Documentation)]

[Pareto front (The Azimuth Project)]

[Spatial Containers, Pareto Fronts and Pareto Archives (GitHub, C++)]

Specification

You can add as many fields, methods, functions, files, … as you want, but at the bare minimum, this should be implemented. There are also a couple of things that you still need to fill in in this specification.

```
// ScoreVector is a type alias for a vector of doubles.
// This will make the rest of the code slightly easier to read.
// You can also replace this type alias by a full class with members,
// methods, ... if you think that would be useful.
using ScoreVector = std::vector<double>;

// PreferenceVector is a type alias for a vector of doubles.
// Same note from ScoreVector applies.
using PreferenceVector = std::vector<double>;

// This class represents the entire Pareto front.
// It should store a set of solutions, along with each solution’s score.
// The solution type can vary, and is determined by a template
// parameter.
template <typename SolutionType>
class ParetoFront
{
    public:
    // This constructor creates an empty Pareto front with
    // a fixed number of objectives.
    // (The number of objectives never changes throughout
    // the Pareto Front’s life.)
    ParetoFront(int ndimensions);

    // This method inserts a single solution,
    // along with its score into the Pareto front.
    // If the given solution was part of the pareto front
    // (it wasn’t dominated by any solution currently in the front),
    // this method should return true, otherwise,
    // the method should return false.
    bool insert(ScoreVector&& score, const SolutionType& solution);

    // This method uses a given linear preference vector to return the
    // best solution.
    // Note that the solution is returned as a const reference!
    // If the user wishes to modify it, they need to copy it first!
    const SolutionType& query(const PreferenceVector& preference) const;
    // TODO: You should decide the return type for this method
    // This method should return an iterator that runs over
    // every solution in the Pareto front.
    auto begin();


    // TODO: You should decide the return type for this method
    // This method should return an iterator pointing
    // one past the last solution in the Pareto front.
    // When iterating from begin() to end(),
    // we should visit all elements in the Pareto front.
    // There is no requirement with regards to
    // the order that you visit these elements in.
    auto end();
};
```

Bloom Filter

A Bloom filter is a probabilistic data structure used to check if a value is a member of a certain set. One of the main use-cases for this is to prevent expensive operations from being performed on elements that are not part of the given set.

Using a bloom filter requires requires multiple different hash functions. We could try to come up with a number of unique and good hash functions, but a much easier way of achieving this is by generating a hash iteratively, starting from some initial value “c”. That way, by selecting a different “c” for each hash function, we can essentially generate an unlimited number of hash functions.

Cuckoo filters are an extension of Bloom filters that we will not consider in this lab, still, resources on Cuckoo filters often start from bloom filters, so I’ve added a few here for reference.

Resources

Bloom Filter (Wikipedia)

Cuckoo Filter: Simplification and Analysis (arXiv)

Cuckoo Filter: Better than Bloom (ACM Library, connect to UA VPN if unable to access PDF)

efficient/cuckoofilter (GitHub)

bitly/dablooms (GitHub)

Space efficient machine learning feature stores using probabilistic data structures – a benchmark (Zalando Engineering)

Specification

```
// This class is used to compute an element’s hash.
// The class itself does not define any implementation for its operator().
// This implementation is provided by on a per-class basis
// through template specialization.
// At the very least, your project should contain
// a specialization for std::string,
// you are free to add other specializations as you see fit.
// This class is designed to operate similar to std::hash,
// except for the existence of an initialization constant.
template <typename ElementType>
class Hasher
{
    public:
    // This is the constant that will be used
    // as an initial value while hashing.
    std::size_t c;

    // This constructor creates a hasher with
    // a given initialization constant c.
    Hasher(std::size_t c);

    // Similar to std::hash,
    // we use operator() to perform the actual hashing.
    // This allows us to write code snippets like this:
    //
    // const Hasher<std::string> h (0xDEADBEEF);
    // const std::size_t string_hash = h(“This is a String!”);
    //
    std::size_t operator() (ElementType e) const;
};

// This class is our actual Bloom filter, it makes use of Hasher to
// figure out where all of its contents go.
// Space complexity: O(1)
template <typename ElementType>
class BloomFilter
{
    // This constructor takes the size of the bloom filter in bits,
    // and a list of hash initialization constants for its hash functions.
    // Note that in C++, it is impossible to
    // store or address individual bits.
    // You can check this for yourself by calling “sizeof(bool)”.
    // sizeof() Is a function that returns
    // the size of its argument in bytes.
    // Normally, this should return 1, indicating that
    // each “bit” actually takes up an entire byte of storage.
    // Keep this in mind when designing your bit array and
    // remember that the argument to this constructor is in BITS,
    // not in BYTES.
    BloomFilter(
        std::size_t bit_size,
        std::vector<std::size_t> hash_initialization_constants
    );

    // This method inserts element e into the Bloom filter.
    // Time complexity: O(K) where K is the number of hash functions used.
    void insert(const ElementType& e);

    // This method queries the Bloom filter to
    // check if an element might be stored in it.
    // This method should return true if
    // the element is possibly in the Bloom filter,
    // or false if the element is definitely not in the Bloom filter.
    bool query(const ElementType& e) const;

    // This method returns true if
    // every single bit in the Bloom filter has been set to one.
    // In this case, the Bloom filter has become useless,
    // and should be re-built.
    bool is_full() const;
};
```

Sparse Matrix

Matrices are an important tool in engineering and mathematics. They are used in an almost uncountable number of applications for computing various things. They consist of a set of rows, with each row containing the same number of values. We can define mathematical operations between matrices such as addition, subtraction, and multiplication.

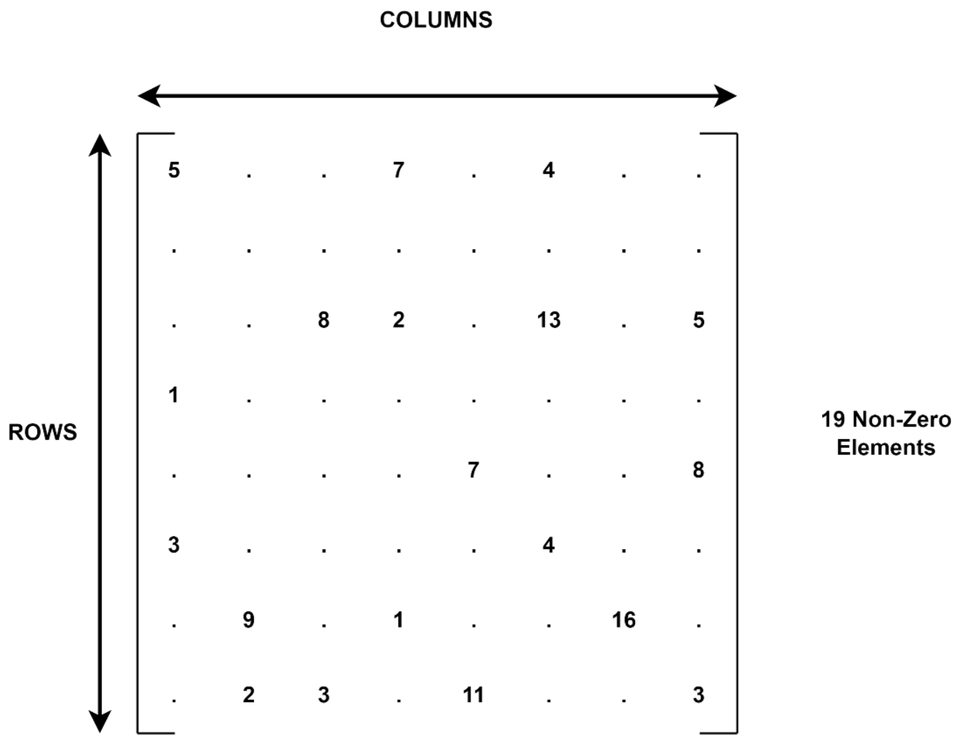

Figure 4: An example of a sparse matrix.

In this lab, we will be implementing a sparse matrix using COO format. Sparse matrices are matrices where the majority of the elements are zero, with only a few non-zero elements. A great example of a sparse matrix is an identity matrix, where every element, except those on the main diagonal is zero.

Resources

Sparse Matrix (Wikipedia)

Sparse Matrices | Coordinate (COO) Format | Intro & Implementation in C (YouTube)

Sparse Matrices – GNU Scientific Library Documentation (C)

torch.sparse (Pytorch documentation)

Specification

```
// This class represents our sparse matrix.
// It takes 1 template argument,
// which reflects the type of element the matrix will be made up of
// (int, float, double, ...)
// Operations between two matrices (Such as multiplication)
// are only allowed between matrices with the same template argument.
//
// A sparse matrix of size N x M with Z non-zero elements
// (Z <= N x M) should have a space complexity of O(Z).
template <typename ElementType>
class SparseMatrix
{
    public:
    // Our matrix has a fixed size, specified by the `rows` and
    // `columns` arguments here.
    SparseMatrix(std::size_t rows, std::size_t columns);

    // We use operator() to access individual elements of the matrix.
    // Note that this method returns a const reference.
    // This means that the element must exist
    // (Otherwise, you can’t return a reference to it),
    // and the element will not be modified
    // (Because the reference is const).
    // Note that the method itself is not marked const.
    // This is to accommodate the case where
    // an element that has not been accessed before is accessed.
    // In this case, you must assume the element is zero,
    // and return a const-reference to a default-constructed object
    // of type ElementType.
    // (We will make the assumption that default constructing
    // an element yields the equivalent of the value 0).
    // Note that the method is marked `const`.
    // This is required to distinguish it from the non-const overload.
    // This will prevent you from modifying the underlying matrix,
    // however, this shouldn’t be an issue.
    const ElementType& operator()(std::size_t  row, std::size_t column)
const;

    // This is another overload of the operator().
    // This one, however returns a mutable reference (non-const),
    // which means the user may assign to or otherwise modify
    // the value returned.
    // Thus, it is important that this value already exists,
    // and is in its final place.
    // When accessing an element that hasn’t been accessed before,
    // the same rules apply as in the previous overload of operator().
    ElementType& operator()(std::size_t row, std::size_t column);

    // In this case, operator* implements matrix-matrix multiplication.
    // For 2 sparse matrices A and B of size
    // N x M and M x K respectively, each with
    // Y or Z non-zero elements respectively,
    // the time-complexity of this operation should be
    // O((K * Y) + (N * Z)).
    // Before this method starts calculating,
    // it should verify that both matrices have compatible shapes
    // (`left` should have the same number of columns as
    // `right` has rows).
    // If this is not the case, the method should throw an exception.
    // We only allow this operation between matrices with
    // the same template argument.
    template<typename U>
    friend operator*(
        const SparseMatrix<U>& left,
        const SparseMatrix<U>& right
    );

    // This method should return an iterator that iterates over
    // all elements in the matrix,
    // regardless of whether they are non-zero or not.
    // When dereferenced, the elements in the iterator can be
    // mutable or immutable.
    // TODO: You should decide the return type of this method.
    auto begin();

    // This method should return an end iterator,
    // that can be compared against the iterator returned by begin().
    // When dereferenced, the elements in the iterator can be
    // mutable or immutable.
    // TODO: You should decide the return type of this method.
    auto end();
};
```