Project 5 specifications

September 2025

1 Overview and purpose

A 'set' data structure is widely used in real-world applications for its ability to store unique elements and support efficient membership testing. In database management, sets are used to eliminate duplicate entries and perform fast lookups, ensuring data integrity and quick retrieval. In search engines, sets help manage the indices of unique keywords, enabling rapid search query processing. In network security, sets are utilized to track unique IP addresses and detect anomalies or repeated access attempts. In machine learning, sets assist in feature selection by maintaining a collection of unique features or parameters. In social media platforms, sets manage user connections and relationships, facilitating quick determination of mutual friends or common interests. In compiler design, sets are employed in parsing and lexical analysis to ensure that tokens and syntax rules are unique and efficiently processed. Overall, the 'set' data structure is crucial for maintaining uniqueness, optimizing search operations, and enhancing performance in a variety of computational and data management tasks.

2 Project description

You will implement a class that uses a linked list to store elements of a "set" of integers. Integers may be added to the set and removed from the set, but there is only ever one copy of each unique integer in the set. Internally, you will store the integers in a linked list. When a new integer is added into the list, a new node is inserted into the linked list, and when an integer is removed from the set, the node containing it will be removed from the linked list.

```
#include <initializer_list>
#include <iostream>

// Class declarations
class Set;
class Node;

// Function declarations
// Prints elements of the set (this code is given below)
ostream &operator<<( ostream &out, Set const &rhs );

// Class definitions
class Set {
  public:
    // This is new and will be clearly explained
    Set( std::initializer_list<int> initial_values );
   ~Set();

    // The instructions will describe exactly what to do
    Set( Set const &orig );
    Set( Set      &&orig );

    // The instructions will describe exactly what to do
    Set &operator=( Set const &orig );
    Set &operator=( Set      &&orig );

    // Size operations
    bool        empty() const;
    std::size_t size()  const;

    // Clear all items out of the set
    void clear();

    // Find the value item in the set
    //  - Return the address of the node if found,
    //    and nullptr otherwise.
    Node *find( int const &item ) const;

    // Insert the item into the set if it
    // is not already in the set.
    //  - Return 1 if the item is new,
    //    and 0 otherwise.
    std::size_t insert( int const &item );
    // Insert all the items in the array
    // from array[begin] to array[end - 1]
    // that are not already in the set.
    //  - Return the number of items inserted
    //    into the set.
    std::size_t insert( int         const array[],
                        std::size_t const begin,
                        std::size_t const end );

    // Remove the item from the set and
    // return the number of items removed.
    std::size_t erase( int const &item );

    // Move any items from 'other', whose values
    // do not appear in 'this', to 'this'.
    // Leave any items that already appear
    // in both sets, in both sets.
    std::size_t merge( Set &other );

    // Set operations (Automatic Assigment)
    Set &operator|=( Set const &other );     // union
    Set &operator&=( Set const &other );     // intersection
    Set &operator^=( Set const &other );     // symmetric difference
    Set &operator-=( Set const &other );     // minus

    // Set operations (Binary)
    Set operator|( Set const &other ) const; // union
    Set operator&( Set const &other ) const; // intersection
    Set operator^( Set const &other ) const; // symmetric difference
    Set operator-( Set const &other ) const; // minus

    // Returns 'true' if 'this' set is a
    // superset of 'other' set; that is,
    // all entries in the 'other' set are
    // also in this set.
    bool operator>=( Set const &other ) const;

    // A superset but not equal to.
    bool operator>( Set const &other ) const;

    // Is 'this' a subset of the other set?
    bool operator<=( Set const &other ) const;

    // A subset but not equal to.
    bool operator<( Set const &other ) const;

    bool operator==( Set const &other ) const;
    bool operator!=( Set const &other ) const;
  private:
    Node *p_head_;

  friend ostream &operator<<( ostream &out, Set const &rhs );
};
```

```
class Node {
  public:
    Node( int new_value, Node *new_next );
    int   value() const;
    Node *next()  const;

  private:
    int   value_;
    Node *next_;

  // Allow any member function in the class
  // 'Set' to access or modify the member
  // variables of any instance of this class.
  friend class Set;
};

// You are given this function that prints out
// a set to the output stream.
std::ostream &operator<<( std::ostream &out, Set const &rhs ) {
  out << "{";

  if ( !rhs.empty() ) {
    out << rhs.p_head_->value();

    for ( Node *ptr{ rhs.p_head_->next() }; ptr != nullptr; ptr = ptr->next() ) {
      out << ", " << ptr->value();
    }
  }

  out << "}";

  return out;
}
```

2.1 Development strategies

Run-time is not relevant to this project: you do not have to optimize your code so as to execute more efficiently. You are welcome to optimize your code but this is not necessary.

First, implement the node class and all of its member functions. This should be straight-forward from the course notes, as they are the same member functions as for a linked list.

Second, start with a skeleton, where each member function is defined, but where the default value of the return type is returned. So for example,

```
  bool Set::empty() const {
    return false;
  }

  std::size_t size() const {
    return 0;
  }
```

```
    Node *find() const {
      return nullptr;
    }

    // Any member function that returns a set
    // by reference should return *this;
    //  v---- the & indicates a return by reference
    Set &Set::operator|=( Set const &other ) const {
      return *this;
    }

    // Any member function that returns a set
    // not by reference, should have a local
    // variable declared to be a set and then
    // return it.
    //  v---- no & indicates a return by value (copy)
    Set Set::operator|( Set const &other ) const {
      Set result{};
      return result;
    }
```

Next, we will present the member functions in such an order that previously implemented member functions can often be used to simplify the implementation of subsequent member functions. Many of the functions above can be implemented using the `insert(...)` and `erase(...)` member functions. These, in addition to `count(...)`, the various constructors and assignment operators should be the only ones that need to manipulate the nodes in the linked list.

2.2 The constructor

Under normal circumstances, we would have a constructor that simply initializes the set by initializing the internal linked list:

```
    Set::Set():
    p_head_{ nullptr } {
      // Empty constructor
    }
```

All that is done here is that the linked list is initialized to `nullptr`, and then after you create a set, you can then start inserting values into it:

```
    int main() {
      Set my_set{};
      my_set.insert( 3 );
      my_set.insert( 5 );
      my_set.insert( 9 );
      // and so on...
```

This is a lot to type, so it would be nicer if we could use the following declaration and initialization:

```
    int main() {
      // Create a set containing 3, 5 and 9
      Set my_set{ 3, 5, 9 };
      // and so on...
```

For this, we will introduce the class `std::initializer_list`:

```
    class Set {
      public:
        Set( std::initializer_list<int> initial_values );
      // Other components of the class definition...
    };
```

The `<int>` just tells the compiler that this will be an initializer list of integer values.

Now, this data structure is like an array that is initialized with some values:

```
    // Create an array of capacity of six and initialize
    // those entries with these six values
    int some_data[6]{ 653, 659, 661, 673, 677, 683 };
```

We have seen how to walk through the entries of an array with a `for` loop. We can do something similar with an "initializer list":

```
    Set::Set( std::initializer_list<int> initial_values ):
    p_head_{ nullptr } {
      for ( int const &value : initial_values ) {
        insert( value );
      }
    }
```

This is not on the examination, so you can ignore this, or read the following to understand how this works: This is a "range-based loop", but can also be described as a "for in" loop, so "for" each value "in" the "initializer list" container, do something with that value. In this case, we are calling the `insert(...)` function which you will have to write.

Humorously enough, if you have a skeleton for each member function, for example,

```
std::size_t Set::insert( int const &item ) {
  return 0;
}
```

then the constructor will simply initialize `p_head_` and even though it may call `insert(...)` many times, the set will remain empty. Then, when you finally implement the `insert(...)` member function, your constructor will also work correctly.

2.3 The empty member function

The first member function you should implement is `bool empty()`. The function `bool empty() const` returns `true` if the set is empty, and `false` otherwise. This can most easily be checked if `p_head_` is assigned `nullptr`, so once this is implemented, you should see

```
int main() {
  Set my_set{};
  // This should now print '1' for 'true'.
  std::cout << my_set.empty() << std::endl;
  return 0;
}
```

2.4 The size member function

The member function `std::size_t size() const` returns the number of objects in the set, and this can be achieved through one of two means:

1. Authoring a for loop that traverses the linked list counting the number of nodes.
2. Introducing a member variable `size_t size_` that is incremented each time a new item is placed into the set, and decremented each time an existing item in the set is erased from the set.

The choice is yours. You have seen how you can walk through a linked list and count the entries, and you can implement a similar member function here.

2.5 The find member function

The member function `Node *find( int const item ) const` traverses through the linked list and if there is a node containing the value `item`, the address of that node is returned, otherwise `nullptr` is returned. Thus, to check if the value 42 is in the set, you will test

```
int main() {
  Set my_set{};

  if ( my_set.find( 42 ) == nullptr ) {
    std::cout << "42 is not in the set..."
              << std::endl;
  }

  return 0;
}
```

2.6 The insert member function

The member function `std::size_t insert( int const item )` checks if the `item` is already in the set. Of course, for this, you can use the above `find(...)` member function. If it is already in the set, just return `0`. Otherwise, you will put a new node into the linked list (wherever you want, at the head, at the tail, or anywhere else in the linked list) and 1 is returned. The value returned is the number of items inserted into the set.

Now the following should work:

```
int main() {
  Set my_set{ 42 };  // Your constructor should work
```

```
      Node *p_search{ my_set.find( 42 ) };

      if ( p_search == nullptr ) {
        std::cout << "42 is not in the set..."
                  << std::endl;
      } else {
        std::cout << "We found "
                  << p_search.value()
                  << std::endl;
      }

      // Oops, we have a memory leak!!!
      return 0;
    }
```

You will notice that we have added a node storing 42 into this linked list, but the program exits before the memory is deallocated. Thus, we need two more member functions: `void clear()` and the destructor.

2.7 The clear member function

The function `void clear()` removes all of the items in the set, which in turn is the process of deleting all the nodes in the linked list. In general, `p_head_` should be assigned `nullptr` when this function returns. This is similar to clearing a linked list as taught in class.

2.8 The destructor

The destructor just calls `clear()`. For almost all students, this would be sufficient. In the unlikely case your constructor allocated additional memory (perhaps creating sentinels for your linked list), your destructor would also have to deallocate this memory, too.

2.9 Testing

Now is the time to test your code, before you go on. Here is one test you may want to try:

```
    Set my_data_1{ 1, 3, 5, 2, 4, 8, 5, 3, 1 };
    // This should print '6'
    std::cout << my_data_1.size() << std::endl;
    // This should print '0' ('false')
    std::cout << my_data_1.empty() << std::endl;
    assert( my_data_1.find( 0 ) == nullptr );
    assert( my_data_1.find( 1 )->value() == 1 );
    assert( my_data_1.find( 5 )->value() == 5 );
    assert( my_data_1.find( 6 ) == nullptr );
    assert( my_data_1.find( 8 )->value() );
```

Notice what happens: `Node *find(...)` returns either `nullptr` or a pointer to a node. If it is a pointer to a node, then what is returned is an address, so on that return value, you can ask it for the value at that address, so `my_data_1.find( 5 )->value()` should return 5 because 5 is indeed in the set. If you were to try `my_data_1.find( 7 )->value()`, because the address returned is `nullptr`, this would result in an error.

2.10 The copy constructor

The copy constructor `Set( Set const &orig )` initializes this set as empty, and then calls `insert(...)` on each item found in the set `orig`.

Remember that you can iterate through the nodes in `orig` by using

```
    for ( Node *ptr{ orig.p_head_ }; ptr != nullptr; ptr = ptr->next() ) {
      // Call 'insert(...)' with the value returned by ptr->value()
    }
```

The keyword `private` is there to indicate that users of the class cannot access the private member variables; however, the author of the class (the one implementing these member functions) should be able to correctly access and manipulate the member variables of any instance of this class.

2.11 The move constructor

The move constructor `Set( Set &&orig )` is only called if the compiler determined that the instance of the class being copied (`orig`) will be deallocated immediately after this constructor is called. Thus, initialize this just like in the constructor. Then, we can use the nodes in the linked list in `orig` in this newly constructed set. Thus, all we need to do is call `std::swap( p_head_, orig.p_head_ )` and do the same with any other member variables you chose to declare.

2.12 The assignment operator

The assignment operator `Set &operator=( Set const &rhs )` makes this a copy of `rhs` and then returns `*this`. First, check if `this == &rhs`. If it is equal, the user is awkwardly assigning a set to itself... While this is unusual, in this case, we can just immediately return `*this`. Otherwise, call `clear()` and then call `insert(...)` on each item found in the `rhs`.

2.13 The move operator

The move operator `Set &operator=( Set &&rhs )` (also called the move assignment operator) is only called if the compiler determined that the instance of the class being assigned (`rhs`) will be deallocated immediately after this assignment is called. Thus, we can use the linked list inside of `rhs` to do the assignment. Thus, the easiest approach is to just call `std::swap( p_head_, orig.p_head_ )` and do the same with any other member variables you chose to declare. Then, what used to be this set will be deallocated when the `rhs` either goes out of scope or is deallocated. Then return `*this`.

2.14 The insert member function for a range

The member function

```
      std::size_t insert( int         const array[],
                          std::size_t const begin,
                          std::size_t const end );
```

attempts to insert all the entries in the array between `array[begin]` and `array[end - 1]`. The number returned is the number of items in that range that were successfully inserted into the set.

2.15 The erase function

The function `std::size_t erase( int const item )` checks if the `item` is in the set. If it is not `0` is returned. Otherwise, the node in the linked list containing that value is removed from the linked list, its memory is deallocated and `1` is returned. The value returned is the number of items removed from the set.

2.16 The merge function (the most complex function)

The function `std::size_t merge( Set &other )` checks each item in the `other` set and if it is in both sets, it is left in the `other` set. If the item is in `other` but not in this set, the node storing that item is moved from that linked list to this linked list. Where you place it in this linked list is up to you. This routine should not deallocate or allocate any memory, but should only move existing nodes. The value returned is the number of nodes that were moved from the `other` set into this set.

2.17 Set operations

There are four set operations we will implement:

1. The union of two sets is a set that contains all entries that are in one set **or** the other. Thus, the union of $\{1,2,3\}$ and $\{3,4,5\}$ is the set $\{1,2,3\} \cup \{3,4,5\} = \{1,2,3,4,5\}$.
2. The intersection of two sets is a set that contains all entries that are in one set **and** the other. Thus, the intersection of $\{1,2,3\}$ and $\{3,4,5\}$ is the set $\{1,2,3\} \cap \{3,4,5\} = \{3\}$.
3. The symmetric difference of two sets is a set that contains all entries that are in one set **or** the other, but not both. In other words, a set that contains all entries that are exclusively in one set or the other (**xor**). Thus, the symmetric difference of $\{1,2,3\}$ and $\{3,4,5\}$ is the set $\{1,2,3\}\Delta\{3,4,5\} = \{1,2,4,5\}$.
4. One set minus another set is a set that contains all entries that are in the first set that are not also in the other. entries that are in one set **and** the other. Thus, $\{1,2,3\}$ minus $\{3,4,5\}$ is the set $\{1,2,3\} - \{3,4,5\} = \{1,2\}$.

Because these operations parallel the bit-wise and minus operations, we will use operator overloading.

2.17.1 Automatic assigning set operations

1. For the union of two sets (`|=`), call `insert(...)` on each item that is in the right-hand side `rhs`. Then return `*this`.
2. For the intersection of two sets (`&=`), call `erase(...)` on each item in this set that you do not `find(...)` in the set of the right-hand side `rhs`. Then return `*this`.
3. For the symmetric difference of two sets (`^=`), for each item in the right-hand side `rhs` that you `find(...)` in this set, `erase(...)` it, otherwise `insert(...)` it. Then return `*this`.
4. For the set minus operation (`-=`), for each entry in the right-hand side `rhs`, `erase(...)` it from this set. Then return `*this`.

Recall that you can walk through all the entries of this object with the loop

```
    for ( Node *ptr{ p_head_ }; ptr != nullptr; ptr = ptr->next_ ) {
      // You can now access ptr->value()
    }
```

and that you can walk through all the entries of the argument with the loop

```
    for ( Node *ptr{ rhs.p_head_ }; ptr != nullptr; ptr = ptr->next_ ) {
      // You can now access ptr->value()
    }
```

Remember that within a member function of a class, you can access the member variables of any instance of the class, not just the member variables of `this`. This is the design of `private`, for the author of the class should have the insight and understanding of the private members variables and functions to use them correctly; it is other users of the class that should not have access to the member variables.

2.17.2 Binary set operations

For each automatic operator, there is a corresponding binary operation. Each of these will be implemented as follows:

1. Create a local variable `tmp` that is a copy of `*this`.
2. Use the corresponding automatic assignment operation to perform the operation of the right-hand side on `tmp`.
3. Return `tmp`.

For example:

```
Set Set::operator&( Set const &rhs ) const {
  // Create a copy of 'this'.
  Set tmp{*this};
  // Make 'tmp' the intersection of
  // 'tmp' and the right-hand side.
  tmp &= rhs;
  // Return the temporary set that is now the
  // intersection of this and 'rhs'.
  return tmp;
}
```

This way, if your automatic assigning set operations are working correctly, then so shall these.

2.18 The subset and superset operators

The function `bool operator>=( Set const &rhs ) const` returns `true` if each item in the `rhs` set is also in this set, and `false` otherwise. That is, this set is a superset of the `rhs` set.

The function `bool operator>( Set const &rhs ) const` returns `true` if this set is a superset of the `rhs` set but not equal to it. If you have determined that this set is a superset, to check that they are not equal, just compare the sizes.

The function `bool operator<=( Set const &rhs ) const` returns `true` if the `rhs` set is a superset of this set, so just return `rhs >= *this`. If this is true, we say that this set is a subset of the `rhs` set.

The function `bool operator<( Set const &rhs ) const` returns `true` if the `rhs` set is a superset of this set, so just return `rhs > *this`.

2.19 The equality operator

The equality operator `bool operator==( Set const &rhs ) const` returns `true` if this set is a superset of the `rhs` and the `rhs` is a superset of this set.

2.20 The inequality operator

The inequality operator `bool operator!=( Set const &rhs ) const` returns the negation of the equality operator.

2.21 Development strategies

The Set class has four major functions that are repeatedly called by other Set functions and operators: `find()`, `insert()`, `clear()`, and `erase()`.

2.21.1 Build the basics first

First, complete a skeleton such that your code compiles. Second, implement and test `find()`, and `insert()`. These functions allow you to complete the constructor (`Set{}`). Third, complete and test the basic functions `empty()` and `size()`. With `empty()` completed you can now print a Set using `std::cout` using the code provided.

2.21.2 Complete clear, the copy constructor and the assignment operator

Next you need to deallocate memory by implementing `clear()`. and the destructor (`~Set`). Once these are complete you have all the necessary functions to complete the copy and move constructors (`Set(Set const &)` and `Set(Set &&)`), as well as the assignment and move operators (`=`).

2.21.3 Erase is the most challenging

The last of the often called functions is `erase()`. Code and test `erase()` then use a combination of insert, erase and out to convince yourself that erase is working. With this last function complete you now have all the tools required to complete the operators. Most operators iterate through sets calling a combination of the four functions listed in this section.

2.22 Privacy is not security

This is optional.

Just because users cannot access private member variables directly in their programs, it is still possible for users to circumvent all the features of a class that are intended to provide encapsulation. If you're interested, try running this program:

```
int main() {
  Set data{};
  data.insert( 42 );
  std::cout << "Empty: " << data.empty() << std::endl;
  std::cout << "Size:  " << data.size()  << std::endl;
  std::cout << "Find:  " << data.find( 42 )  << std::endl;

  // Magic...
  *(double **)( &data ) = nullptr;

  // Suddenly, 'p_head_ == nullptr'
  // is once  again 'true'...
  std::cout << "Empty: " << data.empty() << std::endl;
  std::cout << "Size:  " << data.size()  << std::endl;
  std::cout << "Find:  " << data.find( 42 )  << std::endl;

  return 0;
}
```

If you're curious what is happening, `&data` is the address of the object `data`. Then `(double **)( &data )` tells the compiler to treat this address as if it was a local variable storing the address of an address that stores the address of a double. This is called "casting", and while useful in some special circumstances, is also potentially dangerous. Then, we assign to what is at the address the value `nullptr`. The compiler is fine with this, as assigning a pointer a value of `nullptr` is completely acceptable:

```
    double *p_datum{ new double{ 4.2 } };
    delete p_datum;
    p_datum = nullptr;
```

The problem is that what is stored at that address is the member variable `p_head_`, so when you overwrite that memory location with `nullptr`, you're also changing the value of `p_head_`.