COMP 2012 Object-Oriented Programming and Data Structures

Assignment 3 Mixian Order System Challenge

Disclaimer & Academic Integrity

All characters appearing in this work are fictitious. Any resemblance to real persons, living or dead, is purely coincidental. No such event is provided by HKUST officials or affiliated parties.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to ensure you understand what is considered plagiarism and the penalties. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copyee) is not just receiving a zero on your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, up to and including expulsion.

Situation & Introduction

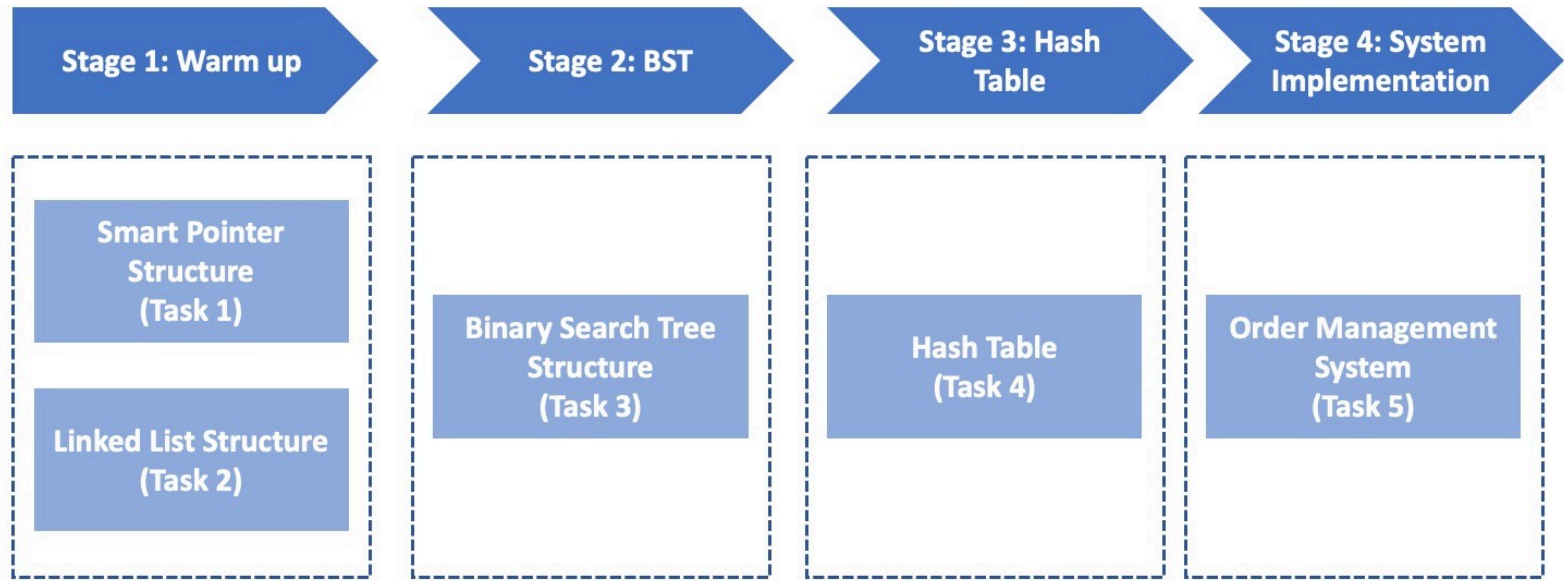
You are Chris Wong, a student who is studying COMP 2012. You notice that there is a new restaurant called "TamGor DiDi" in HKUST. Since many students are ordering their favorite "Choose Your Own Mixian" product, they want a system to analyze the orders each day so they can prepare inventory more effectively. As an outstanding student, you are invited to help them build the system and potentially receive free "Mixian" upon completion.

The Order Management System includes several functions:

1. Import order information, and store the data according to their `order id`, `spicy level`, and `soup type`
2. Store all order information in a BST with different keys for aggregation.

3. Put all order information in a hash table with order id as the key.

Here is the recommended sequence of completing the tasks:



After reading the introduction, you can download the skeleton file HERE and start the challenge! **Please note that this program uses a CLI (Command Line Interface) to interact with users. For more information, jump to the Resource & Testing section to learn more.**

Stage 1: Warm Up (Task 1 & Task 2)

Pre-requisite: None
Target File: SmartPtr-impl.h, LinkedList.cpp

In this stage, you will complete the backbone of the system – the `SmartPtr` and `LinkedList` class. You have learned about `LinkedList` in the previous COMP 2011 course, but not `SmartPtr`. The `SmartPtr` Template class is a wrapper data structure that helps us to manage **heap-allocated** memory by preventing users from directly calling `new` or `delete`, resulting in a lower risk of memory leaks or undefined behavior.

IMPORTANT: In this assignment, you are required to use Smart Pointer as the return type of all heap-allocated memory. To ensure the integrity of the class, there will be no hidden test cases in this stage.

Task 1: SmartPtr Class

Please read the description before moving on to the actual tasks.

Smart Pointer is a wrapper data structure that stores a raw pointer. It helps us keep track of how many owners have access to this pointer. If the reference count drops to 0, it means that no functions are using the pointer, allowing us to safely call `delete` to deallocate the memory. It is a simplified version of `std::shared_ptr` in the C++ standard library.

```
template <typename T>
class SmartPtr {
    // .... Only show the data member of this class
    // A raw pointer inside this container
    T* ptr;
    // A reference count for the raw pointer, showing how many SmartPtr objects are pointing t
    unsigned int* ref_count;
};
```

As you can see above, there are only two data members: `ptr` and `ref_count`. `ptr` is a raw pointer representing type `T`. While `ptr` can be `nullptr`, **the SmartPtr object itself is still defined**. Other than `operator *`, all other functions can be safely called.

The ref_count is a heap-allocated data member, which is responsible for keeping track of how many SmartPtr objects containing the specified pointer. ref_count is defined only when ptr is not nullptr. Otherwise, ref_count will be nullptr because we do not need to record how many objects are pointing to nullptr.

Memory management of the raw ptr will be handled through the constructor(s), destructor, set(), and unset(). When a new object is created and passed through the SmartPtr constructor, the object will store the new object pointer and set ref_count to 1. For example:

```
SmartPtr<float> sp {42.5};
```

The sp now holds the pointer to the object which has value 42.5, with ref_count set to 1, since only sp is holding the object.

If you need to pass this object to another function, a copy constructor is provided to share ownership of the pointer. For example

```
SmartPtr<float> sp2 {sp};
```

The sp2 object holds the same object as sp, but the ref_count is incremented by 1 since both sp and sp2 are holding the pointer to 42.5.

When both sp and sp2 are out of scope or unset themselves using the unset() function, the corresponding ref_count will be 0, indicating that no SmartPtr is holding the heap-allocated object. When ref_count reaches 0, we can safely deallocate the memory by calling delete. Note that SmartPtr objects are allocated on the stack so that they can destruct themselves after going out of scope.

You are required to implement all member functions in SmartPtr-impl.h file.

Constructors & Destructors

Task 1.1

```
SmartPtr();
```

A default constructor is needed since there is no object. You should set ptr to nullptr. ref_count should also be set to nullptr because ptr is nullptr. We should treat this constructor as storing a nullptr.

Task 1.2

```
explicit SmartPtr(T* p);
```

A raw pointer p is passed to the constructor. Assume that the pointer p is pointing to a variable or object of type T that was created using the new operator, storing the raw pointer and set ref_count to 1. **We assume that all pointers provided to this constructor point to objects created using the new operator and are not currently managed by another SmartPtr object.** An explicit keyword is added to prevent implicit conversion and potential undefined behaviors. Note that here p can possibly be nullptr.

Task 1.3

```
SmartPtr(T& val);
```

The conversion constructor allocates memory for a value type T by using the copy constructor. It creates a new object instance of T by copying val and sets ref_count to 1. **We assume that class T has its copy constructor.** Here val must be a reference to a valid object, because a reference to nullptr is undefined in C++. The reason to copy val is because we do not know if the memory referenced by val is allocated on the stack or heap.

Task 1.4

```
SmartPtr(const SmartPtr<T>& sp);
```

The copy constructor of SmartPtr does not allocate new memory. If it stores a valid pointer, copy and store the same pointer, and increment ref_count by 1, since there is one more SmartPtr instance sharing the same pointer. The heap memory pointed by ref_count is shared by the new SmartPtr when we copy-construct it from another SmartPtr. If sp stores nullptr, it just creates a new SmartPtr object with nullptr, which is basically the same with a default constructor.

Task 1.5

```
~SmartPtr();
```

The destructor of a `SmartPtr` instance. If the underlying `ptr` is `nullptr`, we can simply destruct the object safely. If `ptr` is not `nullptr`, we decrement `ref_count` by 1 as there is now one less `SmartPtr` object pointing to the variable/object. If `ref_count` is 0 after the decrement, we should deallocate the memory since the pointer is no longer being referred to by anyone. `ref_count` should also be deleted if we delete `ptr`.

Member Functions

Task 1.6

```
void set(T& val);
```

This function allocates memory for a value type `T` by using the copy constructor. It creates a new object instance of `T` by copying `val` and set `ref_count` to 1. The function also needs to ensure that the previous `ptr` content has been unset properly. The previously pointed `ptr` should be deallocated as specified in the destructor. **Note that self-assignment is possible (i.e., `val` is exactly the object `ptr` points at).** Same with above, `val` must be a reference to a valid object, because a reference to `nullptr` is undefined in C++. `ref_count` should also be deleted if we delete `ptr`.

To do that, you should **always** copy `val` first, then destruct current `ptr` and `ref_count` , and finally assign the new `ptr`.

Task 1.7

```
void unset();
```

Set `ptr` to `nullptr`. The memory originally pointed to by `ptr` will be deallocated when no other instances of `SmartPtr` hold the same address. Same with above, `ref_count` should also be deleted if we delete `ptr`.

Operator Overloading

Task 1.8

```
SmartPtr<T>& operator=(const SmartPtr<T>& sp);
```

The assignment operator stores the same address in `ptr` as in `sp.ptr`. The memory originally pointed by `ptr` will be deallocated if no other instance is using that memory, increment or decrement the `ref_count` accordingly if the memory is still being used by another instance of `SmartPtr`. For the newly pointed `ptr`, we should increment the `ref_count` accordingly as we have more `SmartPtr` objects. This operator should support cascading assignment, i.e., `sp = sp1 = sp2;`.

Note that `sp` should be copied before deleting the old `ptr` if no other instances are pointing to the old `ptr`.

Task 1.9

```
bool operator==(const SmartPtr<T>& sp) const;
bool operator!=(const SmartPtr<T>& sp) const;
```

Compare the `ptr` member of two `SmartPtr` using the correpsonding operators.

Task 1.10

```
T& operator*() const;
```

Return by reference the dereferenced pointer `ptr`. We do not check whether `ptr` is `nullptr` or not (like a raw pointer). This overloads the `dereference operator *SmartPtr`, but this does NOT overload the `arrow operator SmartPtr->`. The arrow operator is overloaded by `T* operator->() const` below.

Task 1.11

```
T* operator->() const;
```

Return the address stored in `ptr`. Same with above, this overloads the arrow operator but NOT the dereference operator.

Task 1.12

```
explicit operator bool() const;
```

Return `true` if `ptr` is not `nullptr`. Note that it is a **new operator** not covered in the lecture note. The operator defines how a `SmartPtr` object can be converted into a `bool`, so that a `SmartPtr` object can be directly used in an if-statement. For example, suppose `sp` has a valid `ptr`, then `if (sp)` will evaluate as True. The advantage of making it `explicit` is that `if (sp)` will still work, but unintentional code such as `if (f(smart_ptr))`, where `f()` is a function that accepts a bool, will not work unless you add an explicit cast to bool. Also note that the `SmartPtr::isNull()` function, which checks if `ptr` is a `nullptr`, is implemented by calling this bool operator.

Task 2 : LinkedList Class

The `linkedList` class provides an interface for us to perform collision resolution in a `BST` and a `HashTable`. You only need to implement one member function in this class.

Different from the singly-linked lists implemented in this course, which only record the next pointer, and thus, can only go forward from the head node, the `LinkedList` we are using here is actually a doubly-linked list, so we can go both forward and backward from a node. We recommend you to check the definition of `LLNode` in `LinkedList.h` for more details.

The `LinkedList` class has the following public member functions:

- `LinkedList` cannot contain smart pointer containing `nullptr`.

Member Function	Explanation
`LinkedList()`	Default constructor of `LinkedList` object
`LinkedList(const LinkedList& ll)`	Copy constructor of `LinkedList` object, require a deep copy
`~LinkedList()`	Destructor of `LinkedList` object
`void add(SmartPtr<Order> order)`	Add the data to the end of the Linked List. `order` will not contain `nullptr`.
`void remove(SmartPtr<Order> order)`	Remove the data if it exists in the Linked List. `order` will not contain `nullptr`.
`int length() const`	Return the number of nodes in the Linked List
`bool isEmpty() const`	Return `true` if the linked list is empty.
`bool contains(SmartPtr<Order> order) const`	Return `true` if the linked list contains the same data. `order` will not contain `nullptr`.
`SmartPtr<LLNode> getHead() const`	Get the head node of the Linked List
`LinkedList& operator=(const LinkedList& ll)`	Overloaded operator= to perform deep a copy
`void clear()`	Reset the whole linked list to empty.

The `LLNode` class has the following public member functions:

Member Function	Explanation
`LLNode(SmartPtr<Order> order)`	Constructor given an `Order` object. Note that order cannot be `nullptr`, otherwise, we will throw an exception using C++ throw, which will cause the program to print an error message ("You are providing a nullptr order!") and exit directly.
`~LLNode()`	Default destructor
`SmartPtr<Order> getOrder()`	Return the `Order` information stored in the current `LLNode`. Note that the `Order` cannot contain a `nullptr`.
`SmartPtr<Order> getNext()`	Get the next node in the linked list.
`SmartPtr<Order> getPrev()`	Get the previous node in the linked list.
`void setNext(SmartPtr<Order> next)`	Set the next node as `next`
`void setPrev(SmartPtr<Order> prev)`	Set the previous node as `prev`

There are a few public functions in `LLNode`. You should also take a look at them.

Task 2.1

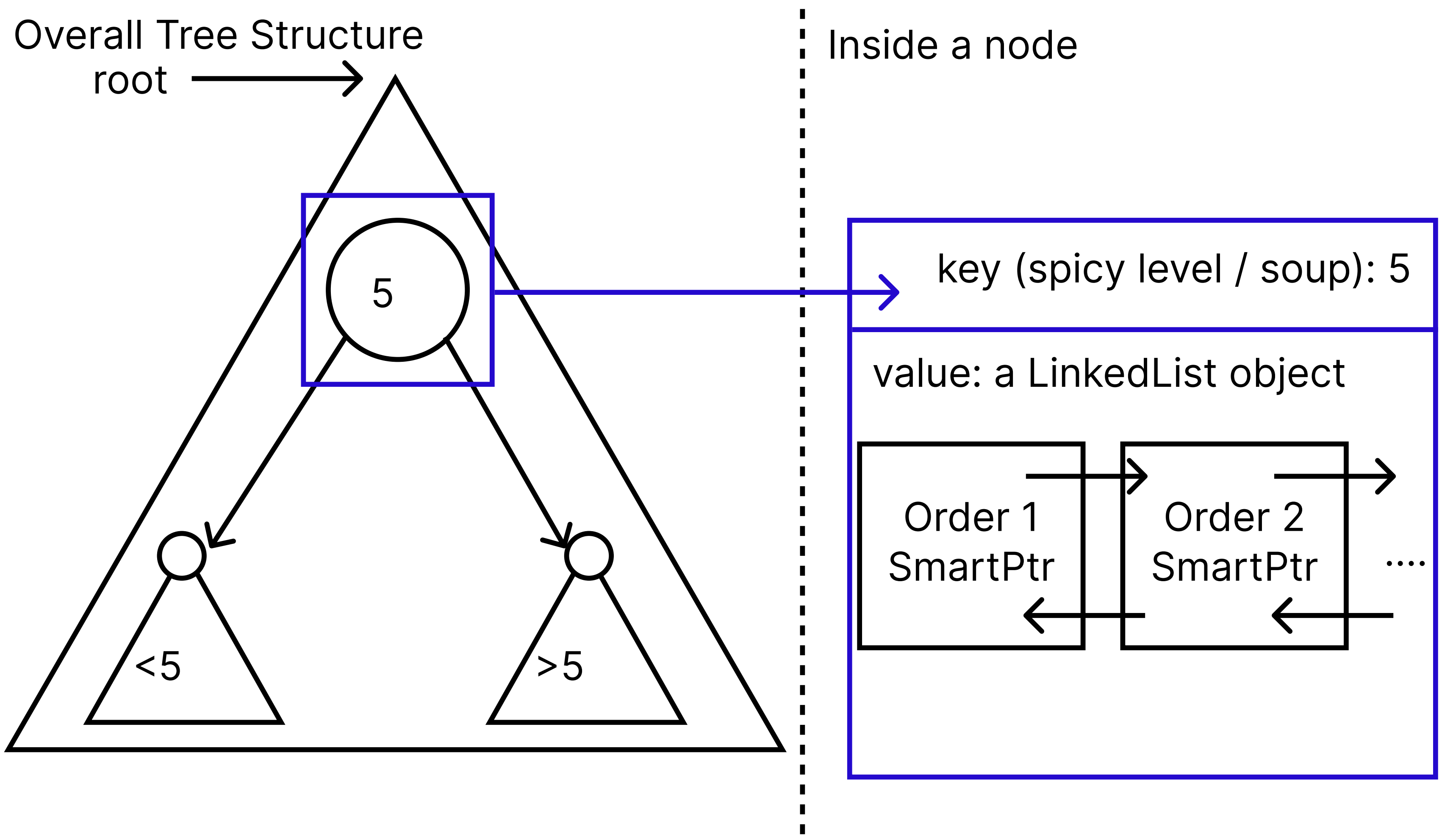
```
SmartPtr<Order> operator[](int index) const
```

You only need to implement the accessor for `operator[]`. It works like a 0-indexed array. Return the element in the linked list if the range is valid. If it is out-of-bound or a negative number, return a `SmartPtr` instance representing `nullptr`.

Stage 2: Binary Search Tree

Pre-requisite: Stage 1 (Task 1 & Task 2)
Target File: BST-Impl.h

In this stage, you are required to implement a Binary Search Tree (BST) to store the spicy level / soup type, and ordered smart pointers. Unlike the BST in the lecture slides which only has one value in each node, we will store **a pair of (key, LinkedList)** in each node. We refer to the `spicy level/soup type` as the `key` and the `LinkedList` as the `value` of this tree. Since there will be multiple orders with the same key, we use `LinkedList` to store all the orders with that key. The following image visualizes the structure of the BST:



Task 3.1

```
bool contains(K key, SmartPtr<Order> order) const
```

Return `true` only if the BST contains the same key and value. `order` will not contain `nullptr`.

Task 3.2

```
int size() const
```

Return the number of `Order` in the BST. It should be a recursive function. **Note: We are NOT counting the number of nodes in the BST, but the number of `Order` in the BST. You can assume `Order` in the BST is unique.**

Task 3.3

```
void add(K key, SmartPtr<Order> order)
```

Add the `SmartPtr<Order>` instance to the BST. If the key does not exist in the BST, create a new node with the key and value. If the key exists in the BST, add the `SmartPtr<Order>` to the `LinkedList` in the node. Use `BSTNode<S>::add(SmartPtr<Order> order)` to add the `SmartPtr<Order>` to the `LinkedList`. `order` will not contain `nullptr`. You can assume that no duplicate orders exist in the test cases.

Task 3.4

```
const BST<K>* find(K key) const
```

Return the pointer to the node with the same key. If the key does not exist in the BST, return `nullptr`. **Note: We do not use `SmartPtr` here because the `BST<K>` instance is allocated on the stack.**

Task 3.5

```
const BST<K>* findMin() const
```

Return the minimum node of the tree in the `BST<K>`. **Note: We do not use `SmartPtr` here because the `BST<K>` instance is allocated on the stack.**

Task 3.6

```
void _remove(K key)
```

Remove the node with the same key in the BST. You should follow the rules specified in the lecture slides. This function will **replace the node with the minimum node from the right side of the BST if its left and right are both not `nullptr`.** Check more rules about BST node deletion in the [lecture slide].

Task 3.7

```
void remove(const K& key, SmartPtr<Order> order)
```

Remove the `SmartPtr<Order>` in the `LinkedList` with the same key in the BST. If the `LinkedList` is empty after removing the `SmartPtr<Order>`, remove the node with the same key from the BST. `order` will not contain `nullptr`. Note that you might need to call the `_remove` function (Task 3.6) if necessary.

Task 3.8

```
Pair<K, SmartPtr<Order>> operator[](int index) const;
```

Return a `Pair<K, SmartPtr<Order>>` object of the corresponding tree as a 0-indexed array. For example, index = 0 means **the first element of the first node of the tree**. If the index is out of bounds or negative, return a default constructed `Pair`. To read more about the `Pair` class, refer to `Pair.h`.

Suppose we have a tree structured as follows:

Denoted as <Key, Length of the Linked List>

```
         <5, 5>
<3, 3>           <7, 2>
   <4, 2>
```

- From index 0 to 2, it should return `3` as key, with the corresponding element of the linked list as the value. For example, `bst[0]` should return `Pair <3, the first element of the linked list>`. And `bst[2]` should return `Pair <3, the last element of the linked list>`. Since the linked list only has length `3`, the corresponding index range is `[0, 2]`.
- From index 3 to 4, it should return `4` as key. The value should follow the same procedure specified above.
- From index 5 to 9, it should return `5` as key. The value should follow the same procedure specified above.
- From index 10 to 11, it should return `7` as key. The value should follow the same procedure specified above.
- For all other out-of-bounds indices, return the default-constructed pair.

Task 3.9

```
std::vector<SmartPtr<Order>> rangeSelect(const K& lower, const K& upper, bool reverse = false)
```

Return a `vector` (check here for more details) that contains all the `SmartPtr<Order>` instances in the range `[lower, upper]` inclusive. The order of adding the record into the vector should be the same as follows:

1. It should be sorted by key `K` in increasing order.
2. If the key is the same, the order should follow the order in the BST's Linked List.

You can assume that `K` has defined operator `==`, `!=`, `<`, `>`, `<=`, `>=`.

If `reverse` parameter is set to true, then the whole sorting will be reversed. For example, `result[0]` should be the last element if `reverse=true` is set.

About C++ `std::vector`

The `std::vector` is a container from the C++ Standard Library that allows you to store and manage a collection of elements. It is similar to an array, but with additional features that make it more flexible and dynamic. Check more details here.

In this PA, we will need to use the following basic operations of `std::vector` (**We have included the header files for you, so we do NOT accept answers that rely on additional header files**):

1. Declaring a Vector

You can declare a vector by specifying the type of elements it will store using the template syntax:

```
std::vector<T> numbers;
```

This creates an empty vector of `T`. You can replace `T` with any other data type, such as `double`, `char`, or even user-defined types.

2. `push_back()`

The `push_back()` function adds an element to the **end** of the vector (i.e., vector is a queue). The syntax is:

```
vector.push_back(value);
```

Input: A single element of the type `T` stored in the vector.

Output: None. The vector is modified by adding the element at the end.

3. `resize()`

The `resize()` function changes the size of the vector, either increasing or decreasing the number of elements. The syntax is:

```
vector.resize(new_size);
```

Input: A new size for the vector (an **unsigned** integer).

Output: None. The vector's size is changed to the specified new size.

4. `size()`

The `size()` function returns the number of elements currently in the vector. The syntax is:

```
int vector.size();
```

Input: None.

Output: The number of elements in the vector (an **unsigned** integer).

5. Indexing Operator

You can access elements in the vector using the indexing operator `[]`, similar to arrays. The syntax is:

```
vector[index];
```

Input: An integer index (zero-based) to access the element.

Output: The element at the specified index in the vector.

Example Usage

```
#include <vector>
using namespace std;

int main() {
    vector<int> numbers;

    // Adding elements using push_back.
    // The vector is now [10, 20]
    numbers.push_back(10);
    numbers.push_back(20);

    // Resizing the vector
    // The vector is now [10, 20, _, _, _]
    numbers.resize(5);

    // Accessing elements using the index operator
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Vector size: " << numbers.size() << std::endl;
}
```

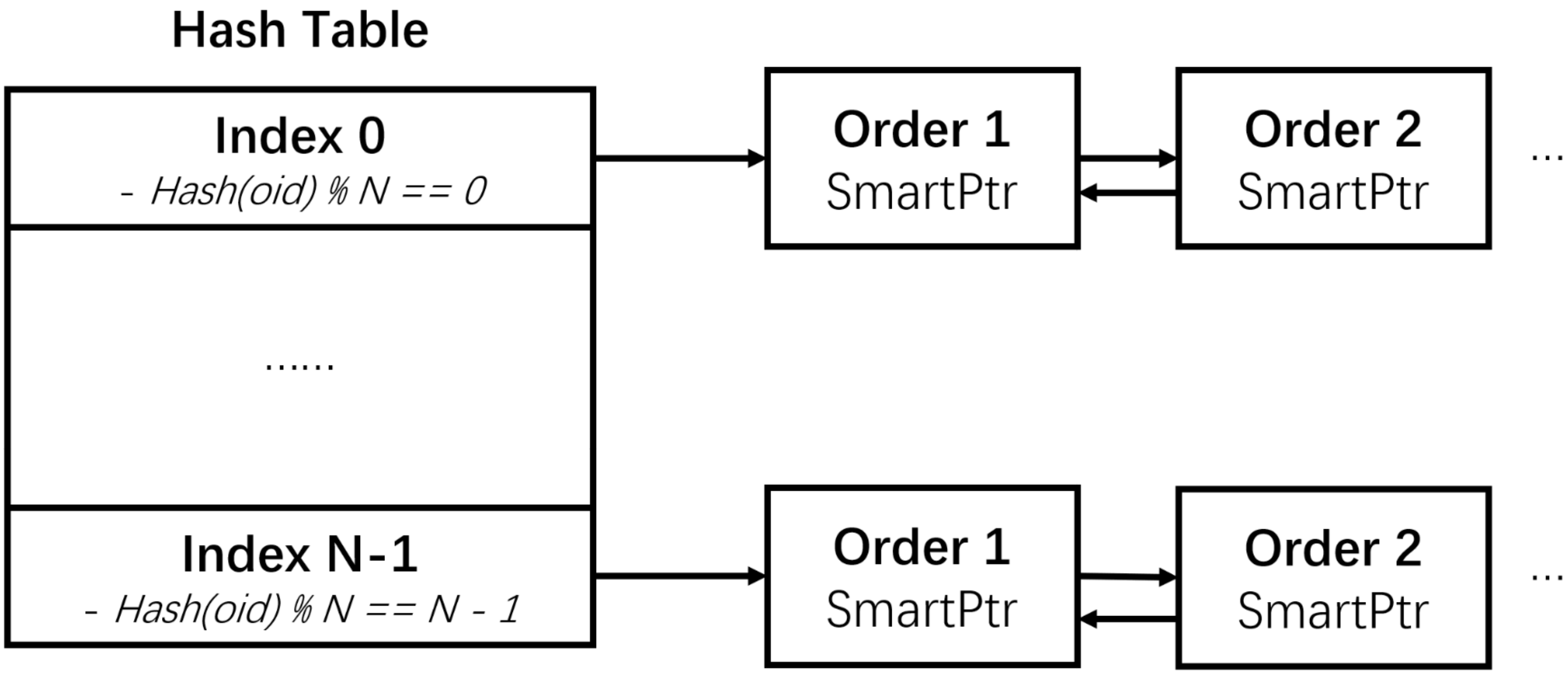
In this example:

- We first create an empty vector of integers.
- Then, we add two numbers (10 and 20) using `push_back()`.
- We resize the vector to have 5 elements (the extra elements will be uninitialized, so be careful when accessing them).
- We access the first element using the index operator `numbers[0]` and print the size of the vector.

Stage 3: Hash Table

Pre-requisite: Stage 1 (Task 1 & Task 2)
Target File: HashTable.cpp

In this stage, you are required to implement a special HashTable for `SmartPtr<Order>`. The hash table stores all records (`SmartPtr<Order>`). Since the hash table holds those records, the corresponding `Order` objects will not be destructed as long as the hash table still exists.



The hash table is defined as a `std:vector<LinkedList>` with a pre-defined size. Given a specified `order id`, we first call the hash function to find the index in the hash table. If the `LinkedList` in the corresponding index is not empty, it suggests a collision occures, and you need to insert the new order to the end of the `LinkedList`. If it is empty, no collision happens before, and you should set the order as the head of the `LinkedList`.

In this stage, to verify your implementations of `HashTable` with the public test cases, we suggest you to first implement the `OrderSystem::hashFunc` hash function in Task 5.1.

Task 4.1

```
HashTable(int size, int (*hash)(int));
```

The constructor of the `HashTable` class takes `size`, which is the number of cells in the hash table, and `hash`, which is the hash function. You should initialize the `std::vector<LinkedList> hashTable` with `size` cells. You can assume that the hash function will always return `[0, INT_MAX - 1]`.

Note: `HashTable` definitely cannot have `SmartPtr<Order>` that contains `nullptr`, otherwise we cannot hash its order ID.

Task 4.2

```
HashTable::HashTable(const HashTable& ht)
```

The copy constructor of the `HashTable` class should deep copy every content in the object `ht`.

Task 4.3

```
void add(SmartPtr<Order> order)
```

Add the `order` to the hash table if it does not already exist in the table. You should use the `oid` of the `Order` as the key to find the cell in the `table`. Add the `order` to the corresponding `LinkedList`. The index of the hash table is determined by the `hash result % size of the table`. Note that there should be no duplicated `orders` in the hash table. `order` will not contain `nullptr`.

Task 4.4

```
SmartPtr<Order> get(int key) const
```

Get the `order` with the same key (i.e., order ID) in the hash table. If the key does not exist in the hash table, return a default-constructed `SmartPtr<Order>` object. Note the `HashTable::contains(int key)`, which checks whether this is an order in the hash table containing the specified key, is implemented by calling this function.

Task 4.5

```
void remove(SmartPtr<Order> order)
```

Remove the `order` with the same key in the hash table. `order` will not contain `nullptr`. You do not have to delete empty LinkedLists after removing the specified order.

Task 4.6

```
SmartPtr<Order> operator[](int key) const;
```

Return the `order` record if the corresponding `oid` exists. If it does not exist in the hash table, return a default-constructed `SmartPtr<Order>` instance.

Stage 4: System Implementation

Pre-requisite: Stage 1 (Task 1 & Task 2), Stage 2 (Task 3), Stage 3 (Task 4)
Target File: OrderSystem.cpp, OrderSystem-impl.h

Welcome to the final stage! It is almost there! In this stage, you will implement the system using the data structures you have implemented in the previous stages.

Task 5.1

```
OrderSystem() // in OrderSystem.cpp
static int hashFunc(int oid) // in OrderSystem.cpp
```

The constructor of the Ordersystem Class. Even though it is a default constructor, we need to give the hash table a size and a hash function. For the size, Chris decides to use her lucky number `769`, and the hash function is `OrderSystem::hashFunc`.

You should also implement `OrderSystem::hashFunc` using the following formula:

```
oid * oid + 13 * oid - 7
```

You need to ensure that the function only returns a value in the range `[0, INT_MAX - 1]`, or a non-negative integer. If it exceeds integer representation, use modulo to map back to the range. For example, if the hash calculated is `INT_MAX`, it will return to 0 since `INT_MAX % INT_MAX` is 0. Note: The header file for `INT_MAX` has been included for you.

Note that

- Signed (not unsigned) integer overflow is undefined in C++.
- The `oid` will not exceed 100,000,000.
- Remember to consider small and large order IDs at the same time.

Task 5.2

```
OrderSystem(vector<SmartPtr<Order>> order)
```

The conversion constructor of the OrderSystem class. The required hash function and size should follow Task 5.1. You should add all orders inside the system using the `addOrder` function. `order` will not contain `nullptr`.

Task 5.3

```
~OrderSystem();
```

The destructor of the OrderSystem class. You should ensure that all the memory allocated in the system is released. This is an easy one :).

Task 5.4

```
void addOrder(SmartPtr<Order> order);
```

Add an order to the system if it does not exist. You should add an order record in the hash table and in both BSTs. `order` will not contain `nullptr`. Pay attention to duplicate orders.

Task 5.5

```
void removeOrder(SmartPtr<Order> order);
```

Remove an order from the system if it contains this record. Remove the record from the hash table, and from both BSTs. `order` will not contain `nullptr`.

Task 5.6

```
SmartPtr<Order> getOrder(int oid) const;
```

Get the order by their `oid`. If the record does not exist, return a default-constructed `SmartPtr`. Note that the `OrderSystem::containsOrder(int oid)`, which checks if the system contains an order with the specified `oid`, is implemented by calling this function.

Task 5.7

```
template <typename T>
int getNumberMixianStat(T start, T end, const BST<T>& tree) const;
```

Aggregate the number of mixian orders in the specified range [start, end] in the BST `tree`, including both `start` and `end`. You CANNOT introduce extra header files.

Task 5.8

```
template <typename T>
float getPriceSumStat(T start, T end, const BST<T>& tree) const;
```

Aggregate the total revenue of the mixian orders specified in the range [start, end] in the BST `tree`, including both `start` and `end`. You CANNOT introduce extra header files.

Resource & Testing

We have made some changes to the source code. Remember to check Change Log for recent updates!

Skeleton Code: HERE (updated Nov 24, 2024).

- For the files of the skeleton code, please refer to Stages 1-4 for more details.

Sample programs:

- Linux (x86_64): pa3
- If you encounter `permission denied`, run `chmod +x <executable file>` to add exection permission.

Command Line Interface (CLI)

This program uses the command line interface (CLI) to interact with the user. The following commands are supported:

- `--mode MODE`, `-m MODE`: Specify the mode of the program. The mode can be `test`, `main`, or `generate`. **This option is required.**
- `--testcase <int>`, `-t <int>`: Specify the test case number to run. This option is only available in `test` mode. Acceptable input is `1-24`. **This option is required in `test` mode.**
- `--src <string>`, `--source <string>`, `-s <string>`: Specify the path of the source `txt` file to import students in interactive mode. This option is only available in the `main` mode. **This option is required in `main` mode.**
- `--output <string> <int>`, `-o <string> <int>`: Specify the path of the output `txt` file and the number of students to generate. This option is only available in the `generate` mode. **This option is required in `generate` mode.**

Example 1

If you would like to run the program with test case `5`, you should type the following:

```
./pa3 -m test -t 5
```

Example 2

If you would like to run the program with the source file `output.txt`, you should type the following:

```
./pa3 -m main -s ./output.txt
```

You can specify actions by giving the corresponding action id:

```
1. Add an order to the order management system
2. Remove an order from the order management system
3. Check if an order is in the order management system
4. Get an order from the order management system
5. Print the data structures in the order management system
6. Trigger getNumberMixianStat function (Using spicyLevel tree)
7. Trigger getNumberMixianStat function (Using soup tree)
8. Trigger getPriceSumStat function (Using spicyLevel tree)
9. Trigger getPriceSumStat function (Using soup tree)
10. Exit
```

Example 3

If you would like to generate 1000 random students and save the result to `output.txt`, you should type the following:

```
./pa3 -m generate -o ./output.txt 1000
```

After generating the output file, you can use the following command to run the program with the generated file:

```
./pa3 -m main -s ./output.txt
```

Sample Output & Grading Scheme

Your finished program should produce the same output as our sample output for all given test cases. User input, if any, is omitted in the files. Please note that sample output, naturally, does not show all possible cases. It is part of the assessment for you to design your own test cases to thoroughly test your program. **Remember to remove any debugging message you might have added before submitting your code.**

There are 24 given test cases which can be found in the given main function. These 24 test cases are first run without memory leak checking (numbered #1 - #24 on ZINC). Then, the same 24 test cases will be run again, in the same order, with memory leak checking (these will be numbered #101 - #124 on ZINC). For example, test case #101 on ZINC is test case 1 (in the given test mode) run with memory leak checking.

Each test case run without memory leak checking (i.e., #1 - #24 on ZINC) is worth 2 marks. The second run of each test case with memory leak checking (i.e., #101 - #124 on ZINC) is worth 0.5 marks. The maximum score you can achieve on ZINC, before the deadline, will therefore be 24*(2+0.5) = 60.

About memory leak and other potential errors

Memory leak checking is done via the `-fsanitize=address,leak,undefined` option related documentation here of the latest g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to the "Your Output" tab in the test case details popup) for errors such as memory leaks. Other errors and bugs, such as out-of-bounds access, use-after-free bugs, and some undefined-behavior-related issues may also be detected. You will receive a score of 0 for the test case if any errors are found. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leaks yourself using the same options, you may follow our guide on Checking for memory leak yourself Yourself.

After the deadline

We will have 16 additional test cases that won't be revealed to you before the deadline. Together with the 24 given test cases, there will then be 40 test cases used to determine your final assignment grade. All 40 test cases will be run twice: once without memory leak checking and once with memory leak checking. The assignment total will therefore be 40*(2+0.5) = 100. Details will be provided in the marking scheme, which will be released after the deadline.

Here is a summary of the test cases for your information.

Main thing to test	Number of test cases in main before deadline (given test cases)	Number of test cases in main after deadline (given+hidden test cases)
Stage 1 (Task 1-2)	10	10
Stage 2	5	11
Stage 3	4	8
Stage 4	5	11

Submission & Deadline

Deadline: 23:59:00 on 30/11/2024

ZINC Submission

Please submit the following files to ZINC by zipping them. ZINC usage instructions can be found here.

```
- SmartPtr-impl.h
- LinkedList.cpp
- BST-Impl.h
- HashTable.cpp
- OrderSystem.cpp
- OrderSystem-impl.h
```

Notes:

- The compiler used on ZINC is `g++11`. However, there shouldn't be any noticeable differences if you use other versions of the compiler to test on your local machine.
- **We will check for memory leak and address issues in the final grading**, so make sure your code has no memory leaks. You are strongly recommended to test your code on the CS Lab 2 machine to ensure it will work on ZINC.
- You may submit your file multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect that you may not receive the grading result report quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts you cannot finish, please provide a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have completed. Empty implementations can be like:

```cpp
int SomeClass::SomeFunctionICannotFinishRightNow()
{
  return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded()
{
}
```

Reminders

Make sure you actually upload the correct version of your source files - we only grade what you upload. Some students in the past submitted an empty file or a wrong file or an executable file which is resulted in zero marks. So **you must double-check the files you have submitted**.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum of 100 points) for every minute you are late. For instance, since the deadline for Assignment 3 is 23:59:00 on **Nov 30**, if you submit your solution at 1:00:00 on **Nov 30+1**, there will be a penalty of -61 points for your assignment. However, the lowest grade you may receive for an assignment is zero: any negative score after deductions due to late penalties (and any other penalties) will be reset to zero.

Frequently Asked Questions

Q: My code doesn't work / there is an error. Here is the code. Can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own, and we cannot complete the tasks for you. We might provide some very general hints, but we shall not fix the problem or debug the problem for you.

Q: Can I add extra helper / global functions?
A: You may do so in the files that you can submit. However, this implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?
A: No. All libraries have been included in the provided header files.

Q: Can I use global or static variables such as `static int x`?
A: No, except the ones we have provided.

Q: Can I use `auto`?
A: No.

Q: In Task 1.4, what if `sp` stores `nullptr`?
A: If `sp` stores `nullptr`, we still need to initialize the newly created `SmartPtr` object with `nullptr`, similarly with the default constructor.

Changelog

- **17th November, 2024**
  1. In line 268 of `test.cpp`, change `std::cout << "sp4: ";` to `std::cout << "sp6: ";`.
- **19th November, 2024**
  1. In `LinkedList.cpp`, we modify the **provided code** of `LinkedList::~LinkedList()` and `LinkedList::clear()` to fix potential memory leak brought by cycle reference.
  2. In line 61 of `LinkedList.h`, we delete `SmartPtr getHead() const;` which is not useful in this assignment.
- **20th November, 2024**
  1. In lines 3-4 of `HashTable.cpp`, we provide additional notes for your reference.
  2. In lines 153, 156, 172, 175 `interactive-main.cpp`, we modify the input manner to support order soup type inputs.
- **24th November, 2024**
  1. In `MakeFile`, we add support to detect changes to header files, so you do not need to run make clean every time you change the header file.
  2. In line 299 `test.cpp`, convert `std:cout` to `std::cout`.
  3. In `test.cpp`, modify `parseOrders` as suggested by [@351](#).

Further Notes

This programming assignment users [CLI11](#) as the Command Line Interface parser. The teaching team does NOT own any rights to CLI11.