The Skip List Problem

# Specifications

The purpose of this program is to make use of the SkipList created in program 1 to implement an STL compliant container class. The description of this programs functionality is defined below.

* Create an ADT with the storage structure like linked lists while providing search performance comparable to arrays.
* The algorithm will consist of two dynamically allocated arrays called heads\_ and tails\_ which will be pointers - the access point to the internal storage of the SkipList.
* The internal part of the SkipList is made up of vectors, each vector corresponding to a node to be added to the SkipList.
* Each node in the SkipList Is initially added to level 0 assuming that there are no duplicates which is handled within methods. There is then a coin flip algorithm that adds the nodes from level 0 to levels 1, 2, 3, etc. There is a 50-50 chance that the node is added to the next level but the probability gets slimmer with each increase in level.
* Once a node has been added into the SkipList, , it  gets a previous, next, up, and down pointers to connect it to its surrounding nodes. All nodes at level 0 have their down pointer set to nullptr. Similarly, nodes at the highest level do not have an up pointer as it is set to nullptr.
* From there, there are several methods to *insert, erase/delete*, and *contains* that perform different operations on the SkipList.

Assumptions

* The assignment dictates creation of a new node for each level. This requires storing the same data multiple times which works for integers but can have side effects if complex ADTs are used.
* The random number generator will generate a fair distribution of odds for each coin toss.
* No duplicates are allowed and any attempts to add duplicates simply ignores the input values and returns to the user result indicating that no addition operation happened.

Program Inputs

For this assignment, there is no external input provided by the user. The input is provided as part of the program compilation and is validated for type correctness at compile time.

Program Output

The program output will be printed using the ostream insertion operator defined in the SkipList class

The program will create an output of this sort:

After Adding 7:

Level: 4 - - empty

Level: 3 - - empty

Level: 2 - - empty

Level: 1 - - empty

Level: 0 - - 7

The variable type outputted to each level is determined at runtime through the template which allows any sort of primitive type (int, double, string, etc.).

Special Cases

Adding a node to an empty level needs to be treated differently since it requires modification to the head and tail. Similarly, addition of nodes before the current head and after the current fail require special handling.

Worst Cases:

The random number generator favors one odd for some duration and limits all added nodes to level 0 generating the search performance equivalent to traditional linked lists.

# Design

* For this program I will be creating 2 files: SkipList.cpp and SkipList.h
* SkipList.h will contain all the headers for the methods I will describe below so they are accessible to the SkipList.cpp file which will implement those methods. This file will also contain the SkipList Container class as well as the nested iterator class
* SkipList.cpp needs to be able to access the headers in the .h file so I will #include “SkipList.h” in the .cpp file
* I will also have a Main.cpp file where I will call methods to make sure I am getting the correct output.

**Template<T>: defined at beginning of header file.**

**Struct SkipListNode in container class:   
SkipList(initializer\_list<value\_type>)**

Initializes the SkipList with the given value type at construction time.

**Explicit SkipList(MaxLevel\_ = 1)**

* constructor creates a node, sets the data member variable to the integer value passed into the constructor, and sets all the other pointer member variables to nullptr.
* Default value of maxLevels\_ is set to 1 at construction time.

**Int data\_;**

* which is the data for the nodes – holds the primitive type

**SkipListNode\* next\_;**

* Links the next node at the same level

**SkipListNode\* prev\_;**

* Links the previous node at the same level

**SkipListNode\* upLevel\_;**

* Links nodes one level up

**SkipListNode\* downLevel\_;**

* Links nodes one level down

**ALL** nodes are set to nullptr at construction time in

SkipListNode is implemented as a nested struct – no need for private members

**Private member variables/Methods of container class:**

- **maxLevel\_:** int an integer representing the maximum number of layers that the SkipList can have

- **heads\_[]:** SkipListNode\* a dynamically allocated array that will store the head pointers pointing toward the start (or head) of each of the SkipList’s layers.

- **tails\_[]:** SkipListNode\* a dynamically allocated array that will store the head pointers pointing toward the end (or tail) of each of the SkipList’s layers.

- addBefore(newNode: SkipListNode\*, nextNode: SkipListNode\*): void Takes in 2 instances of SkipListNode and adds the first (new Node) instance to the SkipList before the second (pre-existing Node) instance. No return value.

- alsoHigher() const: bool Returns true 50% of the time and will be used to determine if a SkipListNode that is being inserted will also be inserted into higher levels of the SkipList

**Public functions of container class:**

+ **virtual ~SkipList()** Destructor clears the SkipList by traversing through each layer of the SkipList and deallocating the SkipNodes

+ **bool insert(item: int):** parameter – value determined at construction time and returns bool. Takes in a value and inserts a new SkipListNode containing the integer into the SkipList depending on the primitive types position in the list. Returns true if insertion was successful. SkipListNodes with the given integer value are inserted into multiple levels of the SkipList depending on probability. (public method)

+ bool erase(item: int): parameter – value determined at construction time and returns bool. Takes in an value and removes the SkipListNodes in the SkipList that contain that primitive type. Returns true if removal was successful. Searches for the SkipListNodes from top layer to bottom, removes them from the SkipList, and deallocates the memory associated them.

+ bool contains(item: int) const: parameter – value determines at construction time and returns bool. Takes in a value and searches for a SkipListNode contains the given primitive type. Returns true if such a SkipListNode is found. Starts by traversing through the top layer of the SkipList and makes its way down until it either finds the SkipListNode or until it hits the bottom layer can still has not found it.

**+** bool alsoHiger() const

Uses a random number generator to toss a coin with 50% probability of returning true and 50% probability of returning false. Declare it static so the generator does not ger re-initialize. This method will determine if a newly added node moves up levels in the SkipList.

**+**addbefore()

parameters – SkipListNode\* which is the new node to be added. SkipListNode\* which is the next node used as a reference. Returns – nothing (void).given a node to be added to the SkipList, place it before the next node.

+ ==Operator: takes 2 SkipLists to compare and returns a bool. returns true if the items in level 0 of SkipList *X* and *Y* are equal and false if they are not equal.

+ !=Operator takes 2 SkipLists and returns true if they are not equal at level 0 and returns false if they are equal at level 0.

+size() which is going to be declared as a constant. Takes a SkipList and returns the integer count of the number items at level 0 of that SkipList.

+empty(): traverses through level 0 of the SkipList and if head and tail pointers are both null then that means the list is empty do return true. Otherwise return false.

+clear(): takes a SkipList and traverses through level 0 and nulling out all pointers and deleting every node of the SkipList. Moves to upper level if current node has an up pointer. Can make use of the erase method to remove each node.

+ SkipList(const SkipList&) Copy Constructor - creates another SkipList object of the same class. Used to initialize one object from another object.

+SkipList&operator=(const SkipList&): copy assignment operator – assigns one instance of a SkipList to another.

+SkipList(SkipList&&): move constructor – allows a SkipList that will be later destroyed, to be stored into another SkipList object.

+SkipList&operator=(SkipList&&): move assignment operator – transferring a temporary SkipList to another SkipList.

**Friend Method of container class:**

operator<<(os: ostream&, list: const SkipList&): ostream& Overloaded insertion operator used to print out the contents of the SkipList level by level. It takes in an output stream and the SkipList. It returns an output stream to match the way the standard insertion works.

**Nested Iterator Class contains methods:**

**+begin():** iterates through level 0 of the SkipList and find the first non-null node. Returns iterator.

**+end():** iterates through level 0 of the SkipList and find the last non-null node. Traverses through and picks up the node before a nullptr pointer. Returns iterator.

Implementation Plan

Inset()

* Call contains() function to see if item already exists and if it does simply return false.
* Start at highest level at non-null pointer traversing through until a greater node is hit.
* At that point move node down level
* Repeat the process of checking greater and lower nodes and moving down until node hits a next node that is greater, and the previous node is smaller. This is the insertion point
* Once at insertion point, call method addBefore() passing it the new node to add and new nodes next node. This will create all the necessary pointers for the new node.
* Call alsoHigher() to test if node should be added into the next level and so on. If alsoHigher() returns true push\_back another node into the vector until alsoHigher() comes back as false.
  + Call addBefore() for nodes added to higher levels
* Case where list is empty – when traversing and all heads are nullptr – add list at level 0 connecting node pointers to head and tail.
  + Call alsoHigher() to see if added to higher levels.
* Return true once successful insertion

Erase()

* Start by calling the contains method to see if the item exists – if true – continue on with your search and deletion. If contains() method returns false – return false.
* Set Node to the first non-null head pointer starting from the highest level and moving down. If no head pointer is non-null, then the list is empty, return false.
* Continue to execute the steps below:
  + If the “Next” node after “Node” has smaller value than “item”, move “Node” to “Next”.
  + If the “Next” node is larger, then move “Node” to “Down” node. If the “Down” node is null the item does not exist.
  + If the value in “Node” matches “item”, the current nodes next pointer is set to the previous nodes next pointer. And the currents nodes previous pointer is set to the next nodes previous pointer.
  + The current nodes pointers are then set to nullptr and the whole vector can be deleted. Return true.

Contains()

* Set Node to the first non-null head pointer starting from the highest level and moving down. If no head pointer is non-null, then the list is empty, return false.
* Continue to execute the steps below:
  + If the value in “Node” matches “item”, return true.
  + If the “Next” node after “Node” has smaller value than “item”, move “Node” to “Next”.
  + If the “Next” node is larger, then move “Node” to “Down” node. If the “Down” node is null the item does not exist.

addBefore()

* Pass in the current node/node being added as first parameter and the node after the current node as the second parameter
* Link all pointers together. Setting:
  + nodeAdded->next to nextNode -> next
  + nodeAdded->prev to nextNode
  + nodeAdded->next->prev to nodeAdded
  + nodeAdded->prev->next to nodeAdded
* (pretty much taking the pointers from both sides of the node added and rearranging them to incorporate the node added so it has pointers on both sides)
* Nodes at level 0 should not have down pointer set to nullptr
* Nodes at the highest level should have up pointer set to nullptr

Size()

* creating an int variable named *count*
* starting at the level 0 head pointer and calling next\_ while the next\_ pointer does not equal nullptr.
* Every time next\_ is called and the next pointer does not equal null, add 1 to the *count* variable which will give you the size of level 0.

Testing/Implementation

1. Make sure the origional program 1 is working before adding anything else.
2. I will then define the template and container. At this point the program should have the same functionalities as program 1 so I will test it in the Main.cpp file and make sure I am able to add different types of primitive types to the SkipList.
3. Start by working on the SkipList.h file and adding the new methods to the SkipList class. After that add the nested iterator class with the begin and end methods.
4. Also add in the copy, move, and assignment operator and constructors.
5. Start by implementing the begin and end methods using breakpoints and hovering over variables to make sure the iterator is picking up the right nodes in level 0 of the SkipList.
6. I will then implement the size method and I will add a cout statement to print the number of elements at level 0. I will verify the number of elements by first inserting a set number of elements through a for-loop.
7. Next I will make sure the empty method is working which is fairly straight forward and just returns whether the SkipList is empty or not. I will simply hover over and verify that I get the correct return value.
8. I will then test the clear method and print out the list before and after calling clear to see how it effects the list. I will also follow through with break points to see how the traversal is working when moving up levels.
9. I will then also test things like the operator == and != to see if it get the correct bool value for matching and non matching SkipLists.
10. I will use the test cases provided by the professor to make sure I am getting the desired output.
11. I will test on Linux machines to make sure everything Is working properly and will use Valgrind to make sure I don’t have any memory leaks.