**Linked Lists - Skip List**

Purpose

This programming assignment implements a **skip list** and compares its performance with that of the doubly linked and the MTF lists you have implemented/used in the [Lab 5](https://canvas.uw.edu/courses/1028107/assignments/2941941).

Skip List

The **skip list** is a sorted list whose find( ) is bounded by *O( log n )* on average. As shown below, a level-6 skip list consists of six parallel lists where the lowest list includes all items in a sorted order; middle lists inherit items from their one-level lower list with a 50% possibility; and the top list includes no items. All of those lists' left and right most items are a dummy whose actual value is a negative and positive infinitive respectively.

S5: -inf <----------------------------------------------------------> +inf

^ ^

| |

v v

S4: -inf <----------> 17 <------------------------------------------> +inf

^ ^ ^

| | |

v v v

S3: -inf <----------> 17 <----------> 25 <--------------------------> +inf

^ ^ ^ ^

| | | |

v v v v

S2: -inf <----------> 17 <----------> 25 <----------> 38 <----------> +inf

^ ^ ^ ^ ^

| | | | |

v v v v v

S1: -inf <--> 12 <--> 17 <----------> 25 <--> 31 <--> 38 <----------> +inf

^ ^ ^ ^ ^ ^ ^

| | | | | | |

v v v v v v v

S0: -inf <--> 12 <--> 17 <--> 20 <--> 25 <--> 31 <--> 38 <--> 45 <--> +inf

Find Algorithm

Given a value to find, the **find( )** function starts with the left most dummy item of the top level list, (-inf of S5 in the above example), and repeats the following two steps until it reaches the item at the lowest list that includes the given value:

1. move right toward +inf while the current item < the given value,
2. shift down to the same item at the next lower list if it exists

For instance, in order to find item 31, we traverse from S5's -inf through S4's -inf, S4's 17, S3's 17, S3's 25, S2's 25, S1's 25, S1's 31, and S0's 31.

In our implementation, we have two methods such as **find( )** and **searchPointer( )**: Since **find( )**calls **searchPointer( )**method, you need to implement  **searchPointer( ) . (Correction: You do not need to implement, as it is already implemented in the code)**

template<class Object>

bool SList<Object>::**find**( const Object &obj )

// points to the level-0 item close to a given object

SListNode<Object> \*p = **searchPointer**( obj );

return ( p->next != NULL && p->item == obj ); // true if obj was found

}

/\* return a pointer to the item whose value == obj  
or return a pointer to the first item whose value > obj if we can't find the exact item.\*/

template<class Object>

SListNode<Object> \*SList<Object>::**searchPointer**( const Object &obj ) {

SListNode<object> \*p = header[Level - 1]; // start from the top left  
  
// implement your codes here  
  
 return p;

}

Insert Algorithm

Given a new object to insert, the **insert( object )** function starts with calling **searchPointer( object )**. If**searchPointer( value )** returns a pointer to the exact item, we don't have to insert this value. Otherwise, start inserting this item just in front of (i.e., on the left side of) what **searchPointer( object )** has returned. After inserting the item at the lowest level, (i.e., at S0), you have to repeat the following steps:

1. Calls **rand( ) % 2** to decide whether the same item should be inserted in a one-level higher list. Insert one when **rand( ) % 2** returns 1, otherwise stop the insertion.
2. To insert the same new item in a one-level higher list, move left toward -inf at the current level until encountering an item that has a link to the one-higher level list.
3. Shift up to the same item in the next higher list.
4. Move right just one time, (i.e., to the next item).
5. Insert the new item in front of the current item.

For instance, to insert item 23, you have to go to item 25, insert 23 in front of 25, and thereafter call **rand( ) % 2** to decide if you need to insert the same item in the next higher list. If it returns 1, you have to traverse S0's 20, S0's 17, S1's 17, and S1's 25. Insert 23 before item 25. Repeat the same sequence of operations to insert 23 on S2, S3, and S4. However, don't insert any items at the top level, (i.e., S5).

S5: -inf <------------------------------------------------------------------> +inf

^ ^

| |

v v

S4: -inf <----------> 17 <----------> <----------------------------------> +inf

^ ^ ^

| | |

v v v

S3: -inf <----------> 17 <----------> <--> 25 <--------------------------> +inf

^ ^ ^ ^

| | | |

v v v v

S2: -inf <----------> 17 <----------> <--> 25 <----------> 38 <----------> +inf

^ ^ ^ ^ ^

| | | | |

v v v v v

S1: -inf <--> 12 <--> 17 <----------> <--> 25 <--> 31 <--> 38 <----------> +inf

^ ^ ^ ^ ^ ^ ^ ^

| | | | | | | |

v v v v v v v v

S0: -inf <--> 12 <--> 17 <--> 20 <--> 23 <--> 25 <--> 31 <--> 38 <--> 45 <--> +inf

Delete Algorithm

Given an object to delete, the **remove( object )** function starts with calling **searchPointer( object )**. If**searchPointer( value )** returns a pointer to the exact item, we delete this item from the lowest up to the highest level as repeatedly traversing a pointer from the current item to its above item. For instance, to delete item 17, start its deletion from S0's 17, simply go up to S1's 17, delete it, and repeat the same operations till you delete S4's 17.

Statement of Work

* Download the following files to your project. Note that **slist.cpp.h** won't be copied, because it is what you have to design and is thus read-protected. You'll see the following files:
  + [**dlist.h**](https://canvas.uw.edu/courses/1028107/files/32574525/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574525/download?wrap=1): a doubly-linked list's header file
  + [**dlist.cpp.h**](https://canvas.uw.edu/courses/1028107/files/32574522/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574522/download?wrap=1): a doubly-linked list's template implementation
  + [**mtflist.h**](https://canvas.uw.edu/courses/1028107/files/32574527/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574527/download?wrap=1): an MTF list's header file
  + **mtflist.cpp.h**: an MTF list's template implementation (from Lab 5)
  + [**transposelist.h**](https://canvas.uw.edu/courses/1028107/files/32574524/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574524/download?wrap=1): an transpose list's header file
  + **transposelist.cpp.h**: an transpose list's template implementation (from Lab 5)
  + [**slist.h**](https://canvas.uw.edu/courses/1028107/files/32574526/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574526/download?wrap=1): a skip list's header file
  + [slist\_incomplete.cpp.h](https://canvas.uw.edu/courses/1028107/files/33609732/download?wrap=1)[View in a new window](https://canvas.uw.edu/courses/1028107/files/33609732/download?wrap=1): a skip list's template cpp file that you have to complete
  + [**driver.cpp**](https://canvas.uw.edu/courses/1028107/files/32574523/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574523/download?wrap=1): a main program for the skip list

* Complete **slist\_incomplete.cpp.h** by implementing the **insert** and **remove** functions:
* template<class Object>
* void SList<Object>::insert( const Object &obj ) {
* // right points to the level-0 item before which a new object is inserted.
* SListNode<Object> \*right = searchPointer( obj );
* SListNode<Object> \*up = NULL;
* if ( right->next != NULL && right->item == obj )
* // there is an identical object
* return;
* // Implement the rest by yourself /////////////////////////////////////////
* }
* template<class Object>
* void SList<Object>::remove( const Object &obj ) {
* // p points to the level-0 item to delete
* SListNode<Object> \*p = searchPointer( obj );
* // validate if p is not the left most or right most and exactly contains the
* // item to delete
* if ( p->prev == NULL || p->next == NULL || p->item != obj )
* return;
* // Implement the rest by yourself /////////////////////////////////////////

}

* Compile and run the driver program, (driver.cpp) in order to verify the correctness of your implementation. Before compile, change your **slist\_incomplete.cpp.h** to **slist.cpp.h**. Did you obtain the same results as follows?
* mv slist\_incomplete.cpp slist.cpp
* g++ driver.cpp
* ./a.out
* #faculty members: 10
* contents:
* -inf -inf -inf -inf -inf -inf
* berger berger berger
* cioch
* erdly erdly erdly erdly erdly
* fukuda
* jackels
* olson olson olson
* stiber
* sung
* unknown unknown
* zander zander
* +inf +inf +inf +inf +inf +inf
* deleting unknown
* #faculty members: 9
* contents:
* -inf -inf -inf -inf -inf -inf
* berger berger berger
* cioch
* erdly erdly erdly erdly erdly
* fukuda
* jackels
* olson olson olson
* stiber
* sung
* zander zander
* +inf +inf +inf +inf +inf +inf
* find \*p->item = stiber
* finding stiber = 1
* create another list
* find \*p->item = stiber
* finding stiber = 1
* #faculty members: 9
* cost of find = 104
* [mfukuda@perseus]$
* Change **driver.cpp** to **driver.cpp.old** and download [**statistics.cpp**](https://canvas.uw.edu/courses/1028107/files/32574528/download?wrap=1)[Preview the documentView in a new window](https://canvas.uw.edu/courses/1028107/files/32574528/download?wrap=1) This program is used for your performance evaluation. Compile and run **statistics.cpp** in order to compare the performance among the doubly-linked, MTF, transpose, and skip lists. Run this statistics with **10000 items**:
* g++ statistics.cpp

./a.out 10000