

Assignment 5

Due May 21, 2015 by 11:59pm

Points 100

Submitting a text entry box or a file upload

Available after May 12, 2015 at 3:30pm

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 laboratory work 5.

Skip List

The **skip list** is a sorted list whose $\text{find}()$ 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
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()**:

```
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
}

template<class Object>
SListNode<Object> *SList<Object>::searchPointer( const Object &obj ) {
    // 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.
}
```

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
S4: -inf <-----> 17 <-----> <-----> +inf
    ^             ^             ^
    |             |             |
    v             v             v
S3: -inf <-----> 17 <-----> <--> 25 <-----> +inf
    ^             ^             ^
    |             |             |
    v             v             v
S2: -inf <-----> 17 <-----> <--> 25 <-----> 38 <-----> +inf
    ^             ^             ^             ^
    |             |             |             |
    v             v             v             v
S1: -inf <--> 12 <--> 17 <-----> <--> 25 <--> 31 <--> 38 <-----> +inf
    ^     ^     ^             ^     ^     ^     ^
    |     |     |             |     |     |     |
    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](#) : a doubly-linked list's header file
 - [dlist.cpp.h](#) : a doubly-linked list's template implementation
 - [mtflist.h](#) : an MTF list's header file
 - [mtflist.cpp.h](#): an MTF list's template implementation (from Lab 5)
 - [transposelist.h](#) : an transpose list's header file
 - [transposelist.cpp.h](#): an transpose list's template implementation (from Lab 5)
 - [slist.h](#) : a skip list's header file
 - [slist_incomplete.cpp.h](#) : a skip list's template cpp file that you have to complete
 - [driver.cpp](#) : 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](#)  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
```

What to Turn in

Clearly state any other assumptions you have made. Your softcopy should include:

1. All .h and .cpp files: The professor (or the grader) will compile your program with "g++ driver.cpp" and "g++ statistics.cpp" for grading your work. Make sure that the archive includes your own slist.cpp.h, (i.e., slist_incomplete.cpp.h you have modified).
2. A separate report in .doc or .docx:
 - (2a) an output of the driver.cpp execution,
 - (2b) performance results (Don't include all the results obtained from statistics.cpp. Simply include the cost of each list execution you saw at the very end of **statistics.cpp**'s outputs, and
 - (2c) your performance consideration in a half page. (Consider the performance if **statistics.cpp** randomly accesses list items.)

. Grading Guide and Answers

Check the following grading guide to see how your homework will be graded. Key answer will be made available after the due date through [Solution](#) page.

1. Documentation (10pts)

An output of the driver.cpp execution, (i.e. execution cost of each list)

Correct (5pt) Wrong (0pt)

Performance consideration about the cost with random list node accesses.

Well considered (5pt) Not enough (0pt)

2. Correctness (70pts)

Successful compilation

Compiled(20pts) Compilation errors(0pt): If not compiled, 0 for this correctness category

Insert algorithm

Correct(25pts) 1 bug(20pts) 2 bugs(15pts) 3 bugs(10pt) 4+ bugs(0pt)

Delete Algorithm

Correct(25pts) 1 bug(20pts) 2 bugs(15pts) 3 bugs(10pt) 4+ bugs(0pt)

3. Program Organization (20pts)

Write comments to help the professor or the grader understand your pointer operations.

Proper comments

Good (10pts) Fair (7pts) Poor(3pts) No explanations(0pt)

Coding style (proper identations, blank lines, variable names, and non-

redundant code)

Good (10pts)

Fair (7pts)

Poor(3pts)

No explanations(0pt)