

# Program 4

**Due** Dec 1, 2018 by 11:59pm    **Points** 30    **Submitting** a text entry box or a file upload  
**Available** Nov 14, 2018 at 3:30pm - Dec 2, 2018 at 12:02am 17 days

This assignment was locked Dec 2, 2018 at 12:02am.

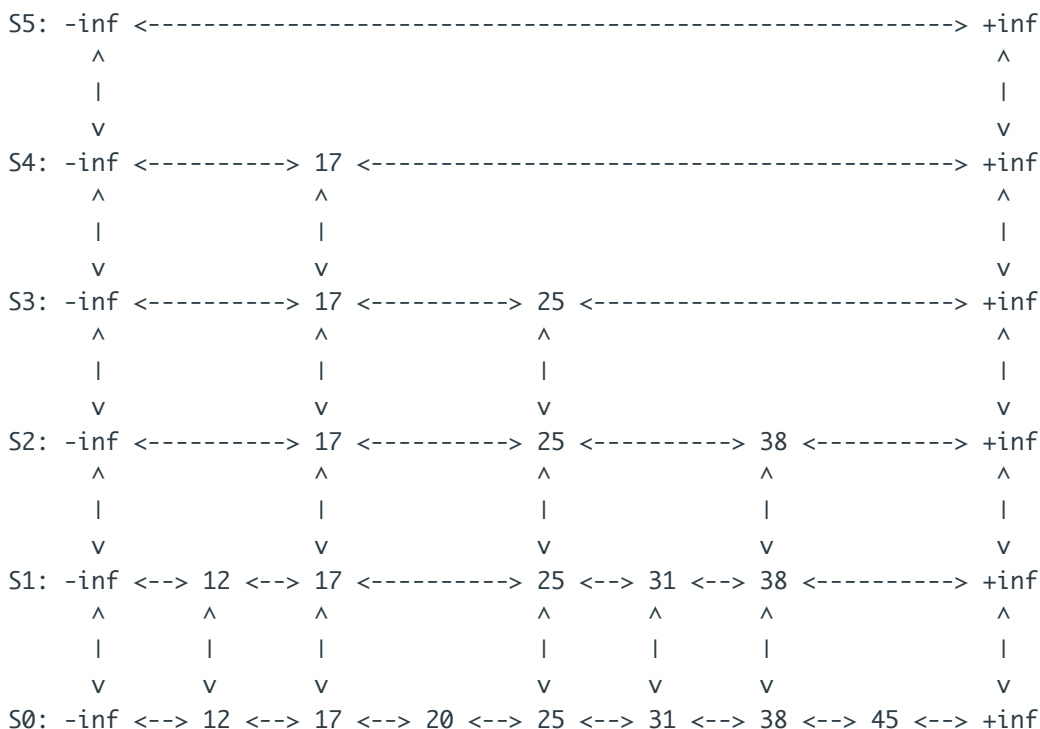
## Linked Lists - Skip List

### 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 4.

### 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.



## Initialize and delete

In this implementation, Skip lists have 6 levels (S0 to S5). Each level's left and right most items are a dummy whose actual value is a negative and positive infinitive respectively. Initialize slist is implemented in **init()** function, and it is called in constructor method. Since slist creates a number of pointers, it should clear up all items and delete pointer in destructor method, using **clear()** function.

**init() function is given.** You have to complete the **clear()** function. **clear()** function will iterate through to delete all pointers of SListNode, and at the end of each level, the leftmost dummy header point to the rightmost dummy header as next pointer.

## 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( )**:

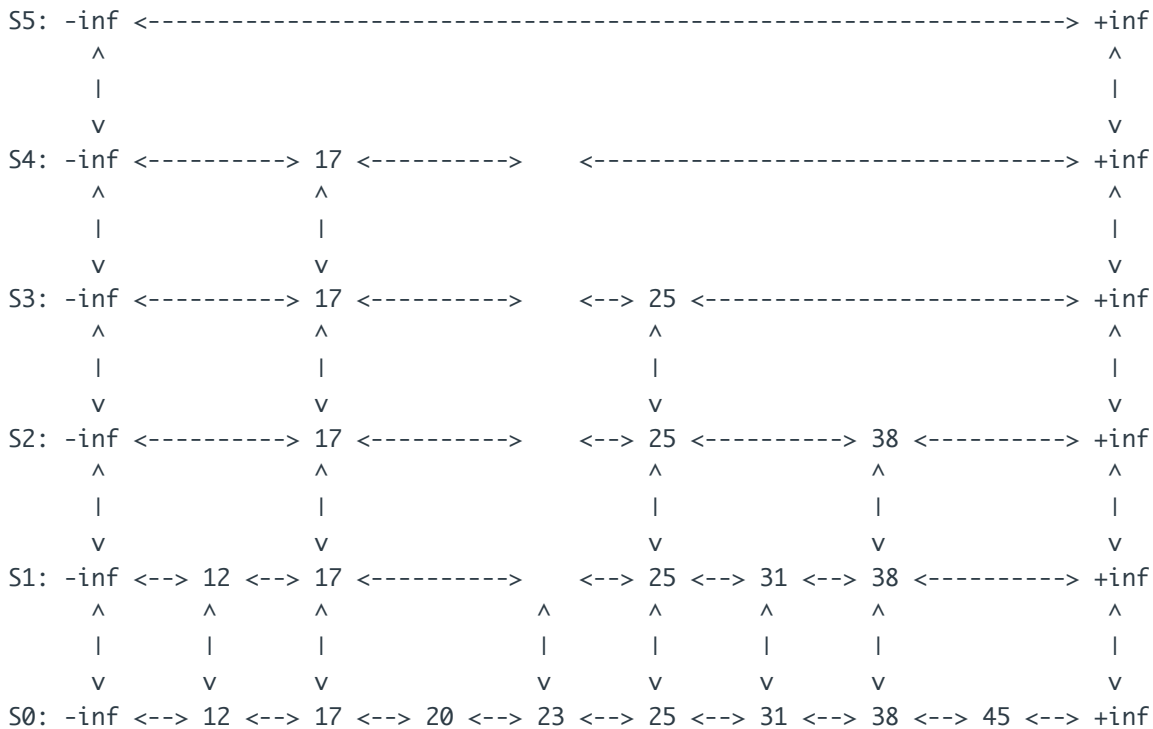
## Insert Algorithm

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).



## 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. (Files-->Programs/Program4/) 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 (Should get from Lab 4 solution)
  - transposelist.h: an transpose list's header file
  - transposelist.cpp.h**: an transpose list's template implementation (Should get from Lab4 solution)
  - 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

- statistics.cpp: a main program to compare different algorithms

- Complete **slist\_incomplete.cpp.h** by implementing the **clear**, **insert** and **remove** functions:

```
template<class Object>
void SList<Object>::clear( ) {
    // for each level, iterate from the first to last item.

    for ( int i = 0; i < LEVEL; i++ ) {

        // Implement the rest by yourself //

    } // let the right most point to it
}

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**. You should obtain the same results as [result1.txt](#)  (as far as the 0th level. The other levels can be different. Also cost can be different)

```
mv slist_incomplete.cpp slist.cpp
g++ driver.cpp
./a.out > result1.txt
```

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 **various parameters**: (sample of result2.txt is not posted, as the output will be different)

```
g++ statistics.cpp
./a.out 10 > result2.txt (Note you can change the parameter. Minimum parameter is 10)
```

**Note that this statistics program works as follows: (added 11/24/2018)**

- **initializes 1000 integers with rand numbers, which are inserted in a list,**
- **picks up only 10 items out of those. You can change this parameter, but it should be larger than 10.**
- **accesses each item of those 10 with a probability of (its index + 1) / 45**

## What to Turn in

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

1. All .h and .cpp.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 .txt, .doc or .docx or pdf:
  - (2a) result1.txt: output of the driver.cpp execution,
  - (2b) result2.txt: performance results
  - (2c) Report the results and analyze algorithms.  
(Should explain about the goal, experiment, result and analysis, and conclusion.)

## • 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 (8pts)  
An output of the driver.cpp execution, (i.e. execution cost of each list)  
Correct (1pt)                      Wrong or not submitted (0pt)  
  
An output of the statistics.cpp execution,  
Correct (1pt)                      Wrong or not submitted (0pt)  
  
Report and performance consideration about the cost with random list node accesses.(6 pts)  
goal (1 pt) + experiment (1 pt) + result and analysis ( 3 pt) + conclusion (1 pt)
2. Correctness (20 pts)  
  
Compilation errors (0 pts)  
  
Successful compilation (4 pts)  
+ Correct implementation of algorithms including clear, insert, remove (14 pts)  
+ No memory leak (2 pts)
3. Program Organization (2pts)  
Write comments to help the professor or the grader understand your pointer operations.  
  
Proper comments  
Good (1pt)              Poor(0pt)  
  
Coding style (proper identations, blank lines, variable names, and non-redundant code)  
Good (1pt)              Poor(0pt)