# Lock-based Concurrent Skiplist

Names Changok Kim, Xiaoguang Ye, Vishal Kumar
ck1334@nyu.edu, xy302@nyu.edu, vkumar04@students.poly.edu

## Abstract:

We are implementing Lockedbased non-deterministic Lazy Skiplist for map. Algorithm used performs insert, delete, find and get operation in expected logarithmic time. The space used by the datastructure is of order O(nlogn). Datastructure is simple multilevel sorted list. Every element are present in the lowest level and depending on the probability factor, some are also present on higher level. This is the invariant of skiplist. In a sense its just like a balanced binary search tree, where we have each node at leaves, and to make operation fast, some leaf nodes are also at higher level. But it is different from balanced binary tree because it does not need rebalancing on updates.

It uses the concept of Optimistic synchronization and Lazy synchronization. Because of these, the locks used in the datastructure are not bottleneck to performance. The existence of node in different level depends on probability factor and hence is non-deterministic in nature.

## Introduction:

Skiplist are balanced datastructure. As against other balanced datastructure like AVL tree, Red-black tree, all variant of B tree it does not require re-balancing and also does not incur any global change. Thus it is more suitable to the multi programmed environment.

Skiplist are sorted multilevel linked list. Each node is associated with subset of lists and have level from 0 to maximum. For simplicity, we have predefined the maximum level in our implementation. Bottom list contains all nodes and each higher list is sublist of lower list. Thus higher lists provides a shortcut into lower-level lists.

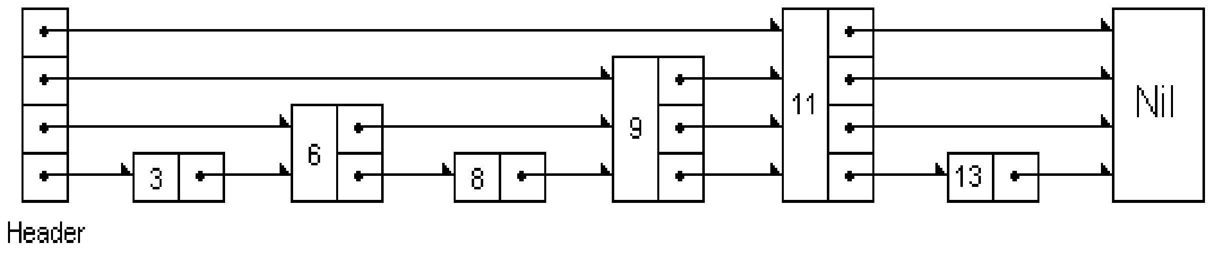

Figure 1: Skip List

To understand it better we can co-relate to subway system from Newyork. We have slow trains making stops at each station and fast trains skipping some. In order to have least commute time, one can take fast train till possible and then switch to slow trains if required. Thus it is a skip list with maximum level predefined to 1.

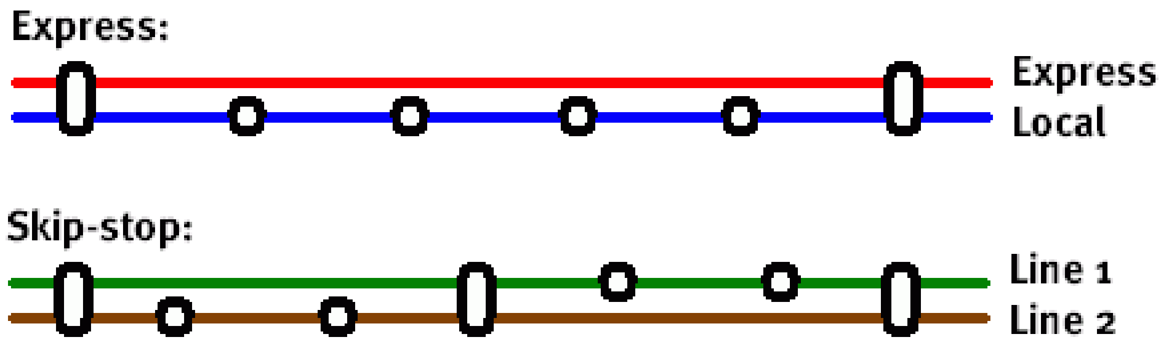

Figure 2: Subway example

We are implementing locked based version of Skiplist, though we also have lockfree version of it. To make sure locks are not the bottleneck we do not lock till we are required to. Any update on the list involves local changes thus locking on the whole datastructure is not good idea. We delay the lock till we identify the local region, take snapshot of it and then try to acquire locks for for local region. Before we complete our operation, we compare the current snapshot with the previous one and only if they are same we continue our operation. If snapshots are different we retry. This approach of locating local region and then locking is called Optimistic Synchronization. Other synchronization technique used to enhance performance is lazy-synchronization. In this approach we split remove method in two steps. First we mark it deleted by setting bit of node and then in second step we physically deleted by unlink it from rest of datastructure. Any operation between logical and physical deletion considers given node as deleted node. Same technique is used in add method where a new node is not considered to be in list till fully_linked_ bit is set. This increases the chances of two snapshots taken in optimistic synchronization to be same.

General Structure of Node :

```
struct Node<K key, V value>
{
        K key_;
        V value_;
        int hightest_level_;
        Mutex lock_;
        bool marked_;
        bool fully_linked_;
        node ** nexts_;
};
```

As earlier discussed, a node can be present in different level of lists. This is defined by probability factor. For simplicity we have taken probability factor of 1/2. This suggest half of nodes present in lower list are also present in successive higher list. The height of new node case can be defined by flip of coin i.e. increase the height of node unless we have heads.

At time of insertion mared_ bit is false and is set when logically removed. We make fully_linked_ bit set when new node is liked at each level.

Each node also has lock associated with it. This way can have locks on local region.

nexts_ has links to successive nodes for each level and helps in defining predecessor and successor relationship.

**Implementation:**

We add two sentinels head and tail. Head has the least key value and tail has the maximum key value. These are of equal height and their height is atleast one more than any other node. At start each level of head points to corresponding level of tail.

During the course of operation we maintain the invariant that a node present at higher level is also present at lower level.

We have find(), add(), remove() and get() method in out implementation.

find(key, predecessor, successor) : It strats from the highest level of head. At each level it follows the link in nexts_ till it reaches a node with key equal to or greater than the key we are looking for. At each level, nodes with nearest key value lesser than search key forms the predecessor list and nodes with nearest key value greater or equal to form successor list. We perform this operation with out any lock. The list of predecessor and successor defines the local area of changes and thus limits the locks required. Also it acts as snapshot which is compared once lock is acquired and thus is optimistic about these two list not being modified in-between.

If it finds a node with search key, it returns the top level where the search key is found. Else it returns with negative value, indicating search key is not present in the list.

add(key) : It issues find() and check if there is unmarked node the list. If so, it return with false. If node present is not fully_linked_ it simply spins till it is fully_linked_ before returning. If node present is marked_, it means some thread is trying to delete, so simply retries. As remove() does not sets marked_ flag unless node is fully linked, it is safe to check if node is unmakrked before checking fully_linked. If node is not fully_linked and unmarked_, it will become fully_linked before becoming marked.

If no such node is present, it will create a lock all nodes in predecessor

remove(key) : It calls find() and if return value is negative it return with unsuccessful status. If lfound is positive, reference of victim node is present at lfound index of successor list. We lock victim and check if it is unmarked. If not we unlock the victim, return false. If

victim is still unmarked, we set the marked bit, i.e. logically delete it. The we lock predecessor from level 0 to top and making sure predecessor ar each level still points to victim and is not logically deleted. If any condition fails we call find() again and repeat the procedure. The only difference is, we do not need to check if victim is marked or not as we have already done this.
Once we have list of predecessor all pointing to victim and unmarked, we change predecessor nexts_ link to victim nexts_. Now we release victiom lock and predecessor lock and return true.

get_value(key) : As our implementation is map, this function returns the value associated with the search key. This calls find(). If return value is negative we return with NULL value meaning search key is not present in the list. If return of found is positive value, we need to lock the search node. This is the node present at lfound index of successor list. Before returning the value, we check if node is still there by checking marked_bit. If it is set true, it means node has been logically deleted. In this case we retun with NULL value. Else we will return with the value of node.

**Problem with the current implementation :**
We are not physically deleting the nodes. This is required because C++ does not provide garbage collection mechanism. As we are implementing using normal pointers, situation where a reference memory is reclaimed. We propose two solution to it.
a. Use of smart pointers : C++ 00x standard has shared_ptr which has reference count mechanism. Using this instead of normal pointers, solves the issue. We tried this, but received segmentation fault. We were unable to find reason for it, but stack trace suggested some exception form shared_ptr library.
b. If we want to continue with simple pointers, we can have try catch block, protecting such unprotected references. If some unprotected reference points to NULL, code can understand the node has been physically deleted and proceeds accordingly.

**Test Cases Design :**
Following points are checked for proving the correctness of data structure.

a. At each level predecessor key value is less than successor key value.

b. Number of successful insert and deleted is equal to change in size of list.

**Performace :**

Following charts show you the result of compared lazy skip list against red-black tree(std::map).

Machine Specification: Energon Sever, 8 Core 1.8GHz Intel Xeon

We put same work loads to each thread and used cpu cycle as time unit. Because of variance of usage on each operation, we tested with three different scenario( 99% get, 0.9% add and 0.1% remove…) and the size of range as well.

The result shows us always Lazy SkipList perform slightly better performance than Red-Black tree.

1. Throughput in operations per each thread of 100,000 operation with fixed range of key, 0~200,000

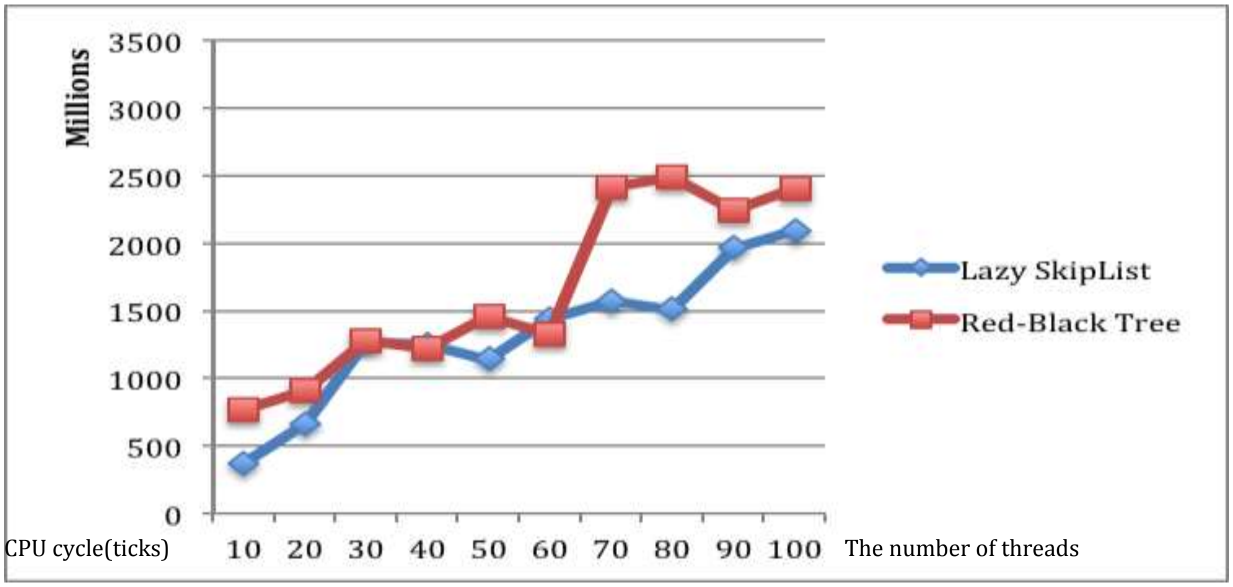

**Figure 3: operation with 9% add, 1% remove, 90% get**

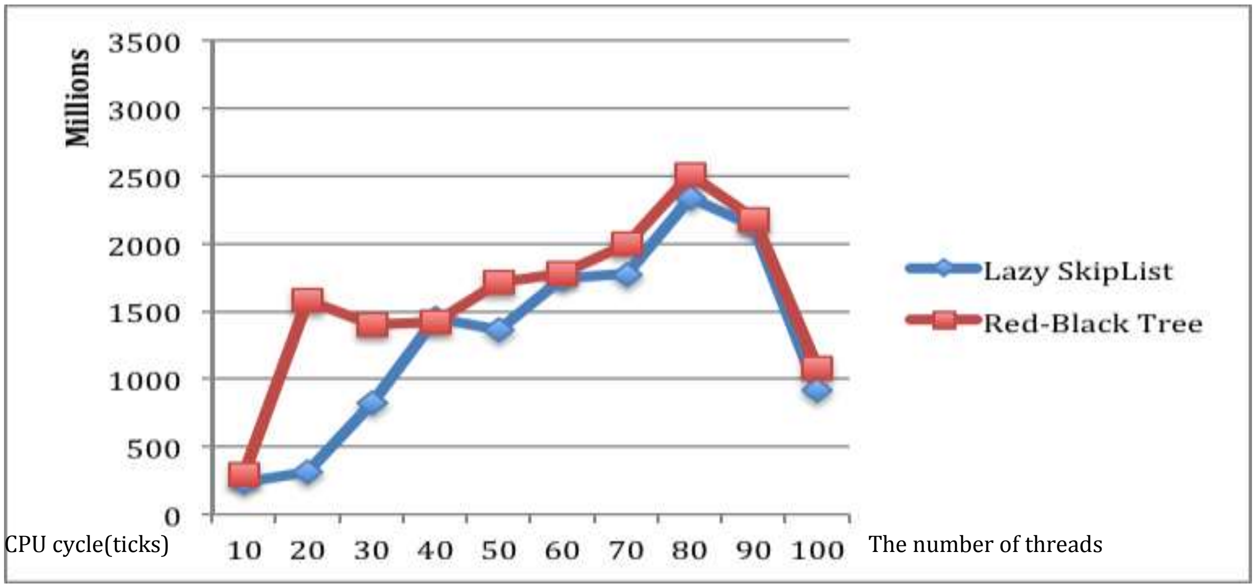

**Figure 4: operation with 20% add, 10% remove, 70% get**

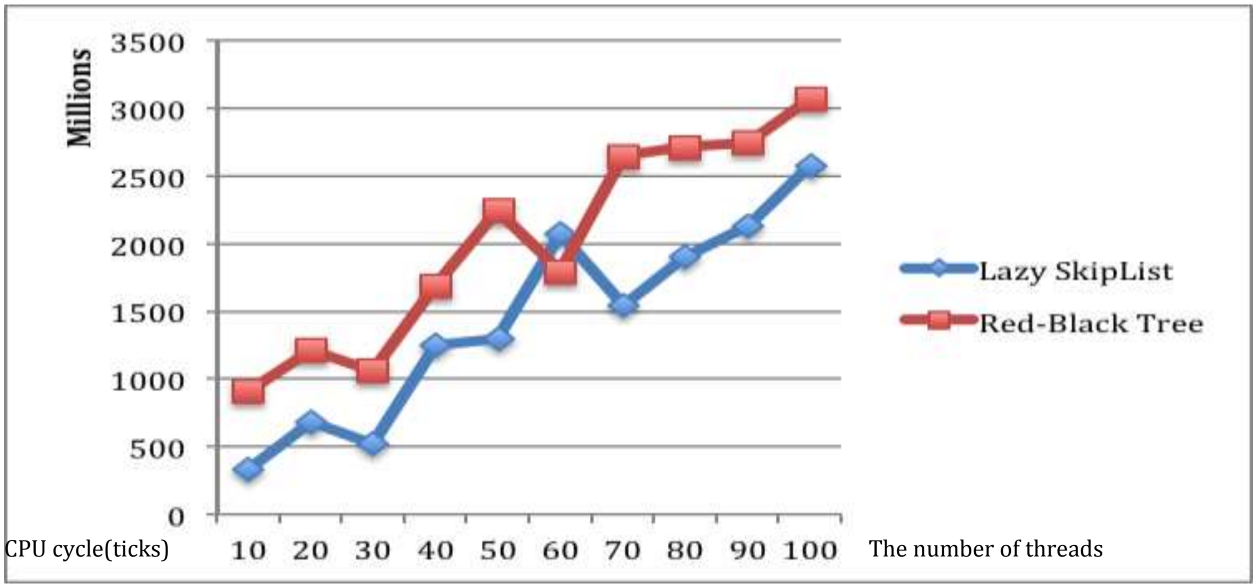

**Figure 5: operation with 50% add, 50% remove**

2. Throughput in operations per each thread of 100,000 operation with fixed range of key, 0~2,000,000

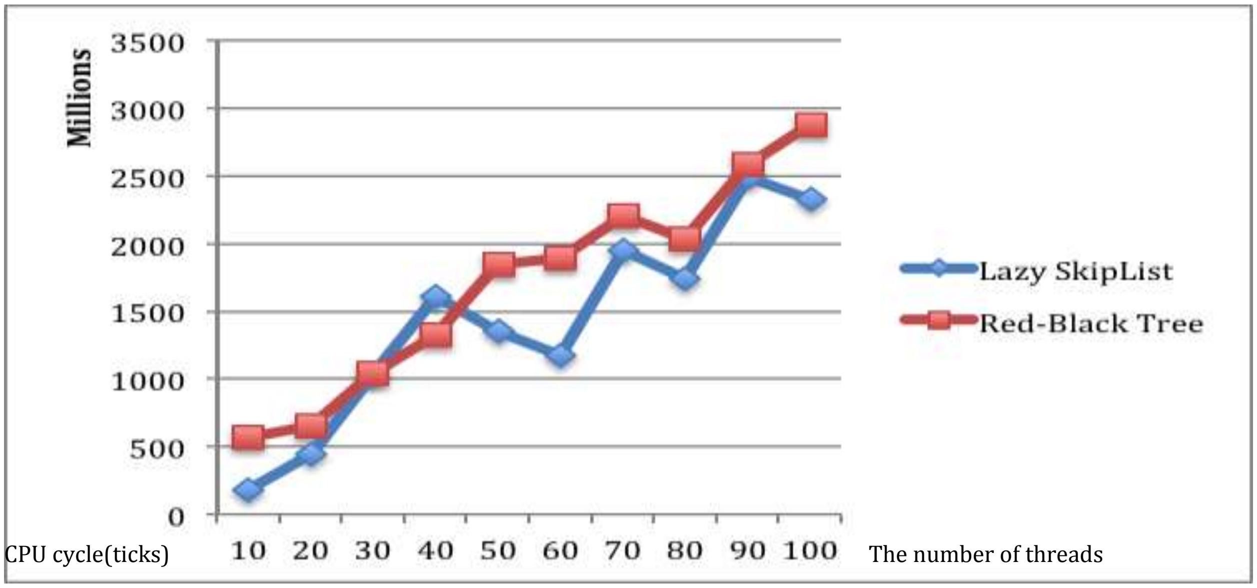

**Figure 6: operations with 9% add, 1% remove, 90% get**

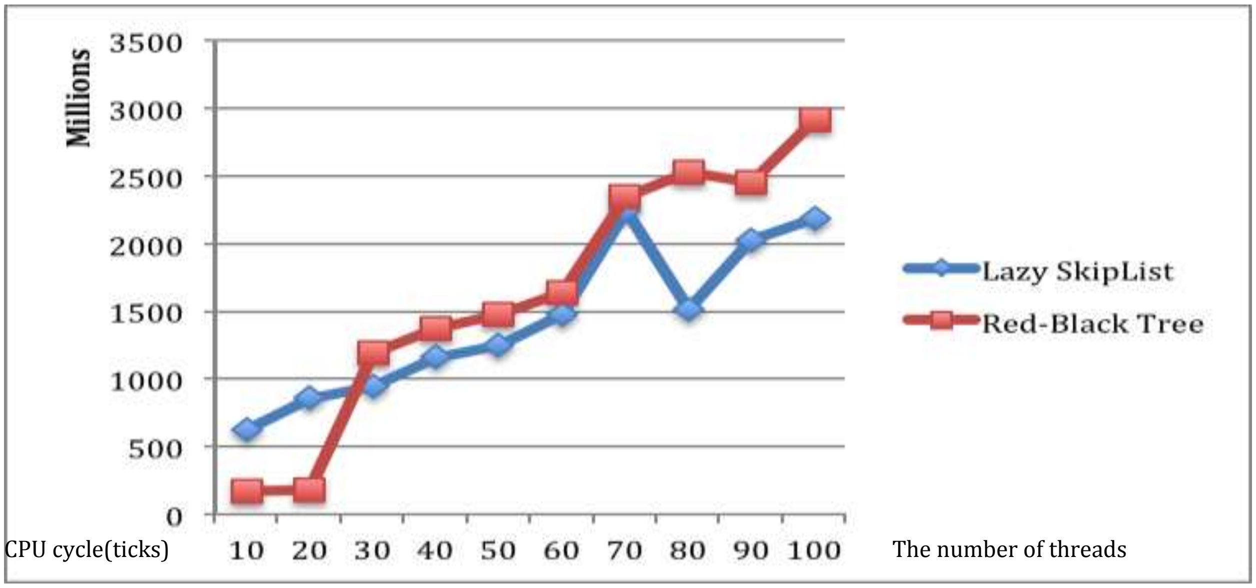

**Figure 7: operations with 20% add, 10% remove, 70% get**

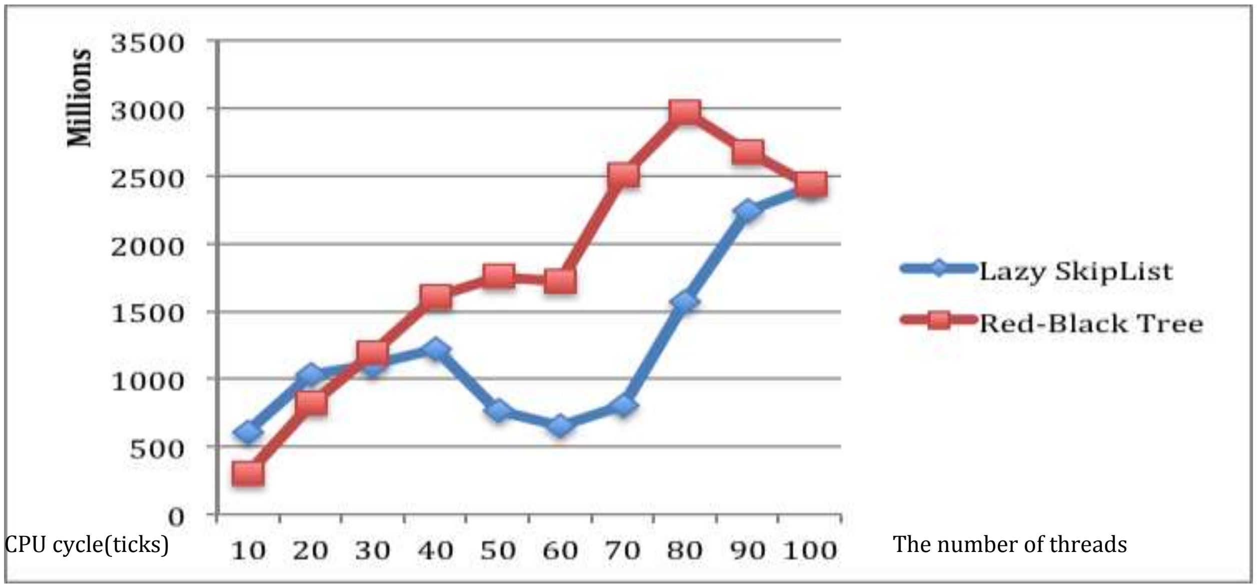

**Figure 8: operations 50% add, 50% remove**

**Conclusion :**

As evident from our test results, skip list are better in multi processor environment. Though we have used locks in red-black implemented map for each access, because of non-local changes, its hard to achieve such performance maps.

Also it is very easy to prove correctness of skiplist. Throughout the skiplist-invariant "each top list is sublist of every bottom list" is maintained. This can be easily inferred because nodes are added from bottom level to top, where as deleted from top level to bottom.

We can also see that locked based implementation is deadlock free. This can be concluded from the fact that nodes are always locked in descending order of key value, thus no circle of contention can form.

Performance results of skiplist suggest that finding the local region of update, and locking as less as possible while maintaining invariant can result in tremendous performance boost in multi processor environment.

**Sources :**

1. Art of Multiprocessor Programming, Herlihy, Shavit

2. Skip Lists : A probabilistic Altrenative to Balanced Tree , William Pugh

3. A Simple Otimistic Skip-list Alogrithm , Maurice Herlihy, Yossi Lev, Victor Luchangco, Nir Shavit

4. MIT Open Course Lecture on Skip List

5. Deterministic Skip Lists , J. Ian Murno, Thomas Papadakis, Robert Sedgewick

6. http://drdobbs.com/high-performance-computing/208801371

7. http://videolectures.net/mit6046jf05_demaine_lec12/