1 Skip Lists

- Probabilistic balance – balance doesn’t depend on input, or behavior of the skip list, but rather, a random number generator
- A “two-dimensional” data structure – doubly-linked lists connected horizontally and vertically
- All items are kept in levels – the bottom level $S_0$ contains all the items, and two special items $-\infty$ and $\infty$ at either end that indicate the ends of the skip list.
- There are multiple levels, each of which, on average, contains half of the elements from the level below
- All duplicate items (stored at various levels) are connected as a “tower” in a linked-list
- Suppose we can go left, right, up, or down – how do we find elements?

2 Searching Skip Lists

- Begin at the left (the $-\infty$) at the top level
- If the next item at the same level is greater than the key $k$ we’re looking for, go down one level
- If it isn’t greater than $k$, then go right (to the next element on the same level)
- If we end up on the bottom level and the next element is greater than $k$, then $k$ isn’t in the skip list
- Otherwise, we find the element in $O(\log n)$

3 Building Skip Lists

- How do we build a skip list to guarantee $O(\log n)$ searches?
- Make sure that each level contains only (on average) half as many elements as the level below
- To insert, place an element at the bottom level, then insert on the level above with probability 0.5
- For $n$ elements, this insertion procedure will result in $\log n$ levels, which is what allows us to have $O(\log n)$ searches
- For removal, we simply find the key to be removed, and remove it from the linked list at all of its levels
- Skip lists in practice are faster than AVL-Trees or some other balanced tree structures

4 Announcements

- Reading: Data Structures (Into Java) – 9.4, Ch. 10
5 Threading (Review)

- Allow simultaneous execution of pieces of code within one process, each of which is a thread.
- Provides behavior similar to that of multiple processes executing simultaneously, but since threads are within one process, they share state (variables, etc...).
- Need to synchronize threads to ensure that they don’t overwrite information used by another.

6 Synchronization

- Suppose we have a class that allows users (different threads) to read and write a non-zero int value, and we want to ensure that each int value gets written and read by exactly one thread, so no information is lost.

Read/Write 1 (broken)

```java
class ReadWrite {
    volatile int value = 0;

    void write(int x) {
        value = x;
    }

    int read() {
        int x = value;
        value = 0;
        return x;
    }
}
```

- What’s wrong with the above code?
- If multiple threads read at the same time, they could get the same value (violating our requirement that each value is written and read exactly once).
- If there is no value, a thread will read 0.
- If multiple threads write without a read, one could overwrite the value stored by another, losing information.

Read/Write 2 (broken)

```java
class ReadWrite {
    volatile int value = 0;

    void write(int x) {
        while (value != 0); // spin
        value = x;
    }

    int read() {
        while (value == 0); // spin
        int x = value;
    }
}
```
Now we check to see whether there is already a value on write (solving the overwrite problem?) and check whether there is no value on read (solving the “no-value” problem?)

What about multiple threads calling either method at the same time?

Let’s mutually exclude the threads from this `ReadWrite` object

Read/Write 3 (ok...)

```java
class ReadWrite {
    volatile int value = 0;

    void write(int x) {
        for (;;) {
            synchronized (this) {
                if (value == 0) {
                    value = x;
                    return;
                }
            }
        }
    }

    int read() {
        for (;;) {
            synchronized (this) {
                if (value != 0) {
                    int x = value;
                    value = 0;
                    return x;
                }
            }
        }
    }
}
```

• The `synchronized` keyword indicates that we want to use the object specified (in this case, `this`) as an object on which to mutually exclude threads – only one thread may be within a `synchronized` block at a time, and if another attempts to enter, it must wait until the thread in the block has left

• We say that a thread acquires a lock on an object to enter a `synchronized` block and releases the lock when it leaves the block (no matter how it leaves the block)

• The problem with the above example is that there is no guarantee made (by Java) that every thread will get its fair share of the time (and therefore, the lock) – it is possible that one of the threads will starve, meaning that it will not get a chance to execute

Read/Write 4 (better...)

```java
```
class ReadWrite {
    volatile int value = 0;

    void write(int x) {
        synchronized (this) {
            while (value != 0) {
                try {
                    wait(); // same as this.wait()
                } catch (InterruptedException e) { }
            }
            value = x;
            notifyAll(); // same as this.notifyAll()
        }
    }

    int read() {
        synchronized (this) {
            while (value == 0) {
                try {
                    wait(); // same as this.wait()
                } catch (InterruptedException e) { }
            }
            int x = value;
            value = 0;
            notifyAll(); // same as this.notifyAll()
            return x;
        }
    }
}

• The call to wait means “release the lock on the specified object temporarily, sit and wait for something to happen (as indicated by a notify or notifyAll on the object)"

• The call to notifyAll wakes up all threads that had called wait on the specified object, and indicate to them that they should try to reacquire the lock – only one thread may actually acquire the lock

• notify does the same thing as notifyAll, except it awakes only one thread

• A thread must have a lock on the object it is calling wait, notify, or notifyAll on.

• Since the synchronized block is the entire method, and it is on this, we can simply put synchronized in the method declaration as a keyword for the same behavior

7 Readers and Writers (multiple)

• Famous problem: The readers and writers problem – how to allow multiple readers read access and a single writer write access and properly mutually exclude them

• Write a lock that must be acquired by the readers and writers before they do anything

• Readers acquire a read lock, writers acquire a write lock

• Multiple readers may have read locks, but only one writer can have a write lock, and never can a readers and a writer both have a lock (because the write may update the data the reader is looking at)
Readers and Writers Lock

class ReadWriteLock {
    int readers = 0, writers = 0;

    synchronized void acquireReadLock() {
        try {
            while (writers > 0) {
                wait();
            }
            readers++;
        } catch (InterruptedException e) { }
    }

    synchronized void releaseReadLock() {
        readers--;
        if (readers == 0 && writers > 0) {
            notifyAll();
        }
    }

    synchronized void acquireWriteLock() {
        writers++; // why do we increment here?
        try {
            while (readers > 0 || writers > 1) {
                wait();
            }
        } catch (InterruptedException e) { writers--; }
    }

    synchronized void releaseWriteLock() {
        writers--;
        notifyAll();
    }
}