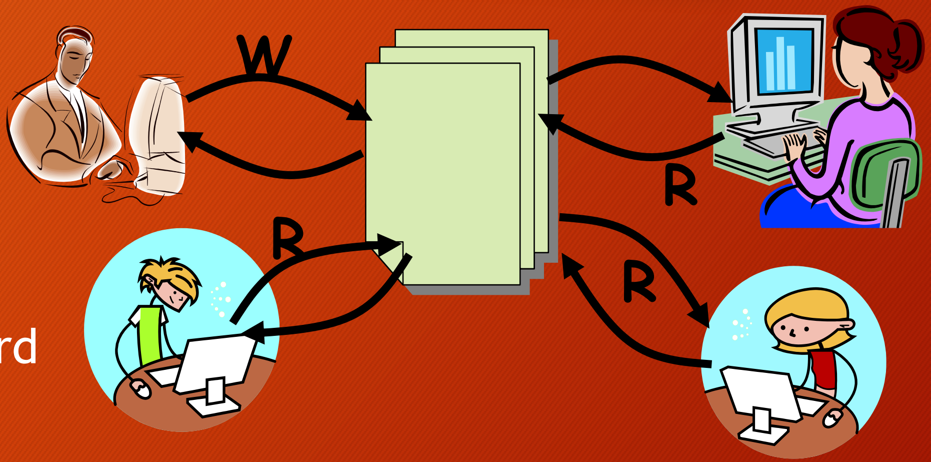
Skip Lists

A alternative to binary search trees for highly concurrent environments

Motivation: Concurrency

- Hardware is concurrent
  - Multi-core processors
  - Instruction level pipelining
  - Hardware support is abundant



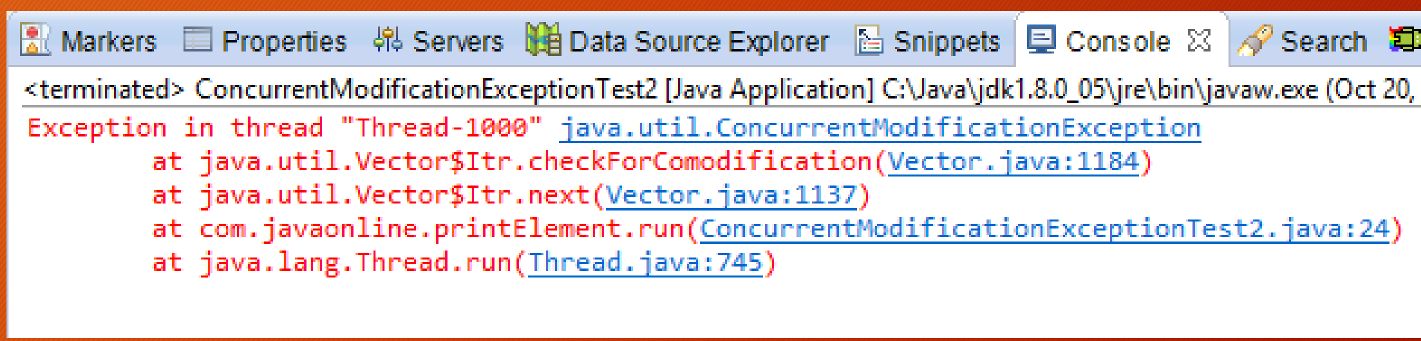
- Writing concurrent software is hard
  - We don't think in parallel
  - One copy of everything
  - Concurrency is usually an afterthought

Concurrent Data Structures

Accessing shared data is the most common operation that may need to be parallelized

Concurrent Access

- Must maintain consistency
  - Multiple readers allowed while there are no writers
  - No readers while there’s a writer
  - Only one writer at a time
  - else we get:



Locking

- Coarse-grained Locking
  - Lock the whole data structure
  - Obviously correct, also simple
  - But inefficient
- Fine-grained Locking
  - Each piece or node will have its own lock
  - Lock only a piece of the data structure
  - Only lock those nodes that need to be modified
- Lock-free Synchronization
  - Atomic CAS
  - Transactional Memory

Linked Lists vs. Binary Trees

Sorted Doubly Linked Lists

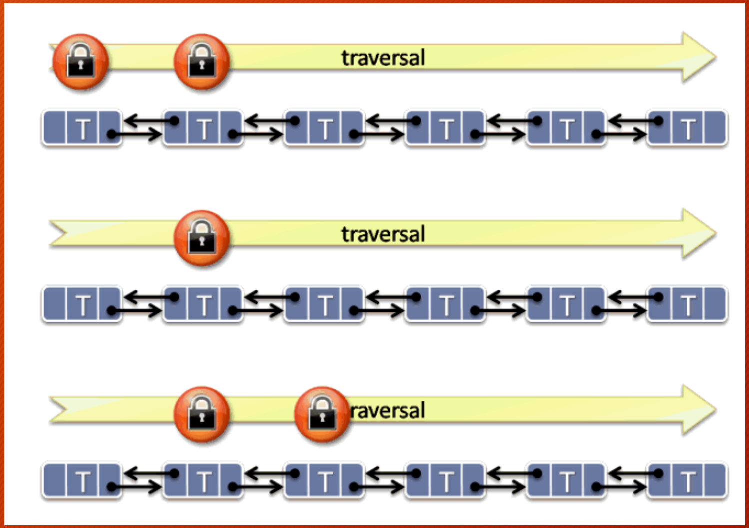
- The Good: Simple linear data structure; anybody can make one!
- The Bad: Worst case O(n) performance

Balanced Binary Search Trees

- The Good: Worst case O(logn) performance
- The Bad: Far more complex to implement

Locking a Linked List

- Hand-over-Hand locking
  - Localized updates: lock only those nodes
  - While traversing the list each thread keeps at most 2 locks
    - Lock node
    - Lock node.next
    - Unlock node
  - All locks are acquired in the same order (no deadlocks!)
  - Advantage: Many threads can safely operate on the list at the same time



Locking a Binary Tree

- Let’s try to do the same thing here too
- Each node has it’s own lock
- Acquire locks in one direction: up the tree
- But how many locks would we need?
- Too many!
  - Rebalancing and rotating may modify a large portion of the tree
  - At worst even the root lock may be required
  - Lower depth nodes become high contention resources – bottleneck!
- Solutions exist but are too complicated to be practical
  - Transactional memory
  - MCAS
  - Lock-free trees

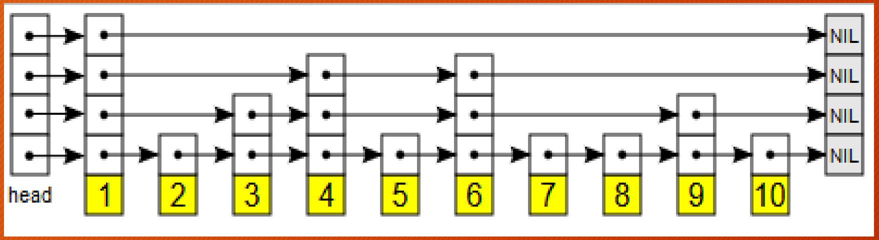
Skip Lists

What, how And why?

Introducing the Skip List

- Skip lists were first described in 1989 by William Pugh
- Non-deterministic data structures
  - Probabilistic balancing
- Expected runtime of O(logn) for search
  - Worst case is O(n)

Skip List: What it looks like



- Multileveled linked list
  - With sentinel node in the front
- Lowest level is a normal sorted linked list
- Upper levels can skip some nodes in the middle
- Each nodes level is random
  - Each node promotes itself with a probability p
  - Usually ½

Skip Lists: In Code

- Here's a sample in C
  - Holds key-value pairs
  - Might contain up to `MAX_LEVEL` levels
  - `forward` holds the pointers for each level in the list, `forward[0]` is the base pointer
  - The header is a sentinel
- `ConcurrentSkipListMap` and `ConcurrentSkipListSet` in Java are implemented using Skip Lists

```c
typedef struct sln {
    void *key, *value;
    struct sln *forward[MAX_LEVEL];
} *slnode;

typedef struct sl {
    unsigned int levelcount;
    slnode header;
} *skiplist;
```

Skip List: Search

- Start at the top
- For each level from top to bottom
  - Skip all nodes that have a smaller key
  - This is like taking a shortcut
  - Prunes the search space, probabilistically
- If search key is found then exit

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

Skip List: Insertion

- Insertion is similar to search
- Find out where the node need to go
  - From top left, skip all smaller and NIL nodes
  - Keep going down and insert at bottom level
- Afterwards, elevate the new node up a random level
  - Need to keep track of all pointer to update

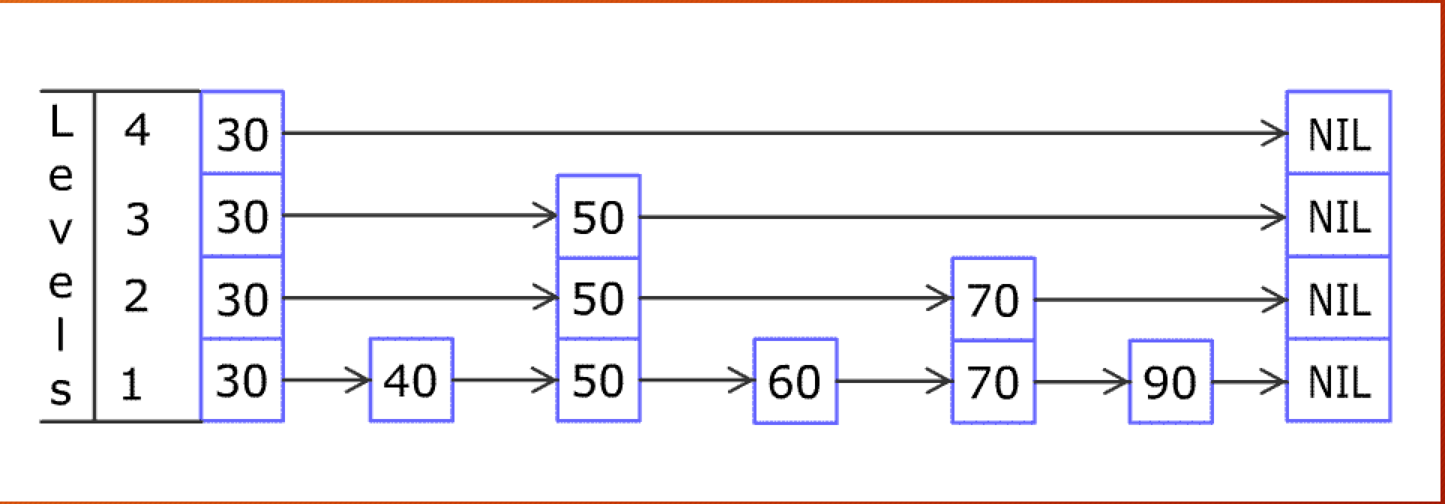
```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey ≤ x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        lvl := randomLevel()
        if lvl > list→level then
            for i := list→level + 1 to lvl do
                update[i] := list→header
            list→level := lvl
        x := makeNode(lvl, searchKey, value)
        for i := 1 to level do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x
```

Skip Lists: Deletion

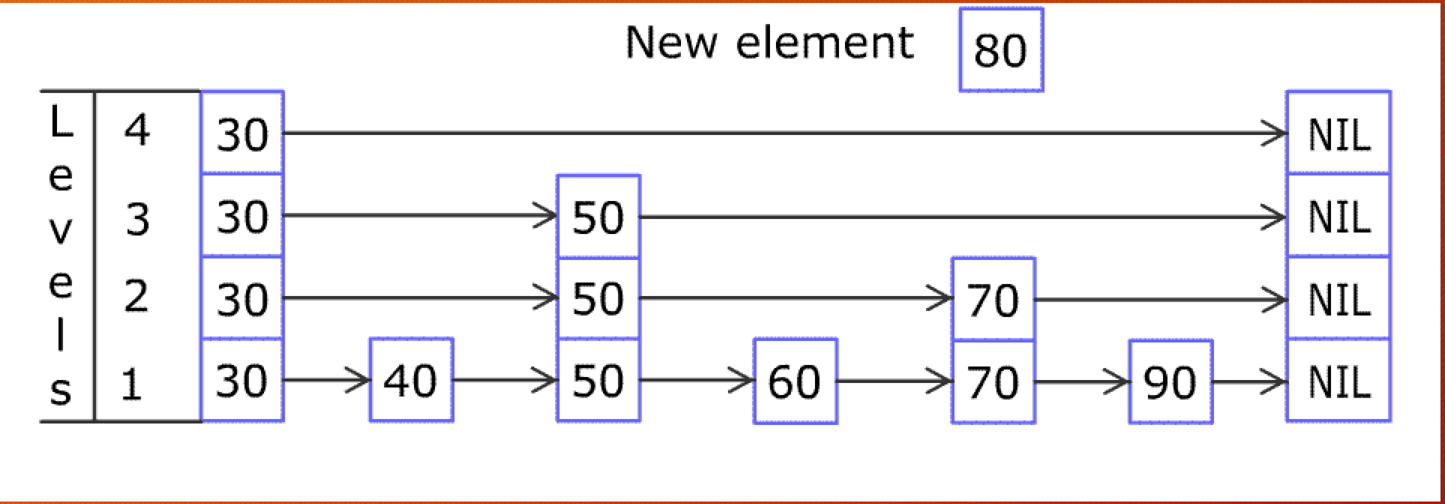
- Very similar to Insertion
- Might need to decrease level

```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
            if update[i]→forward[i] ≠ x then break
            update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and
                list→header→forward[list→level] = NIL do
            list→level := list→level – 1
```

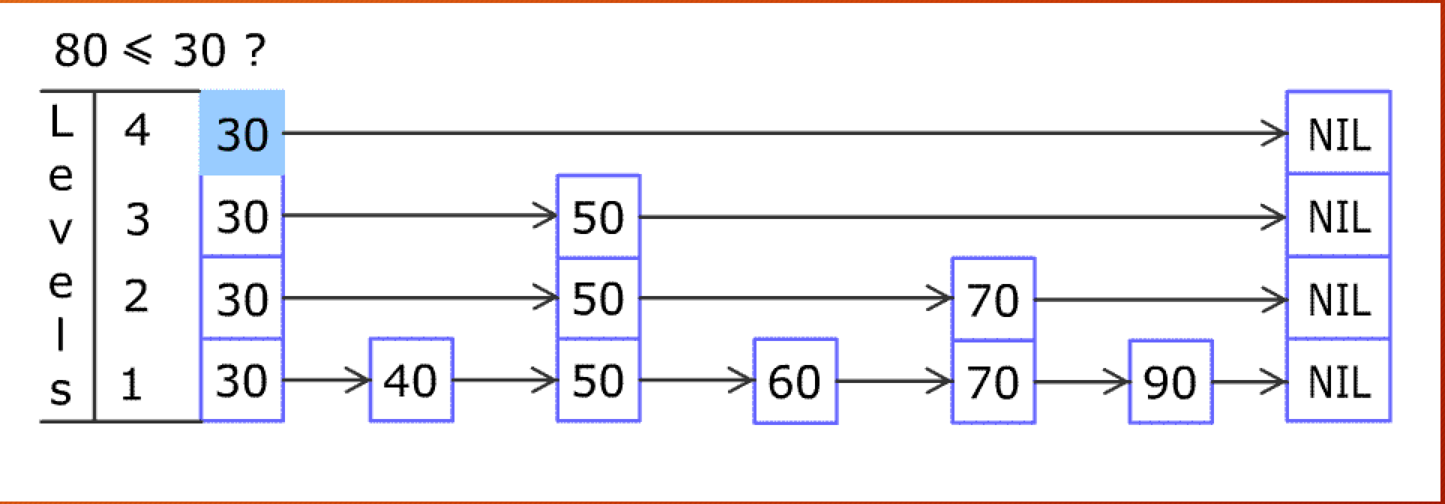
Skip List: Insertion



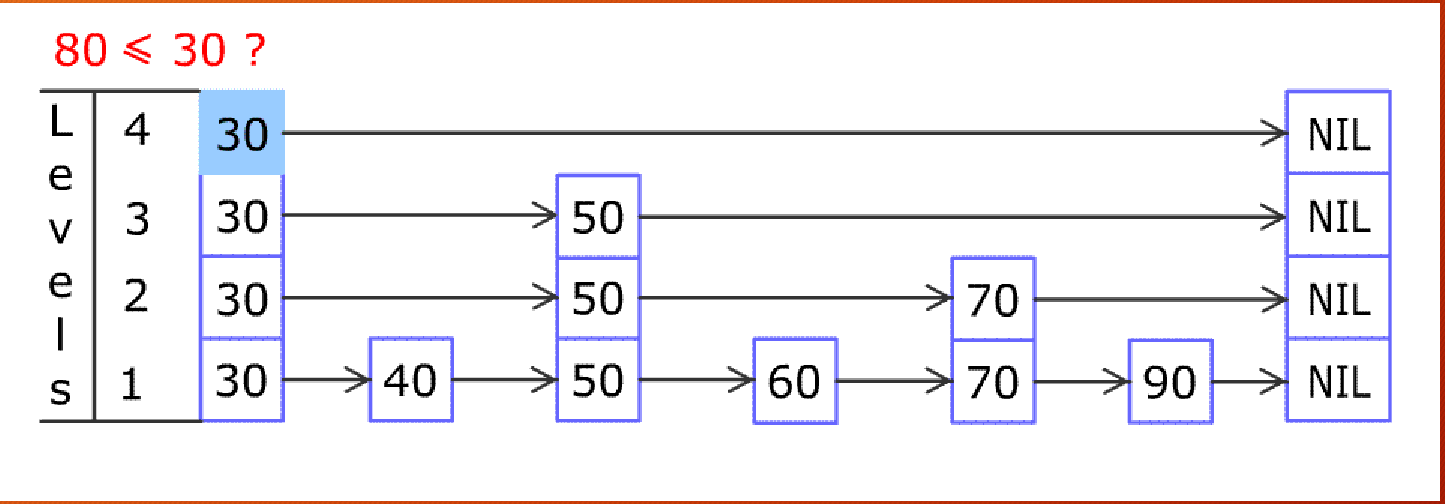
Skip List: Insertion



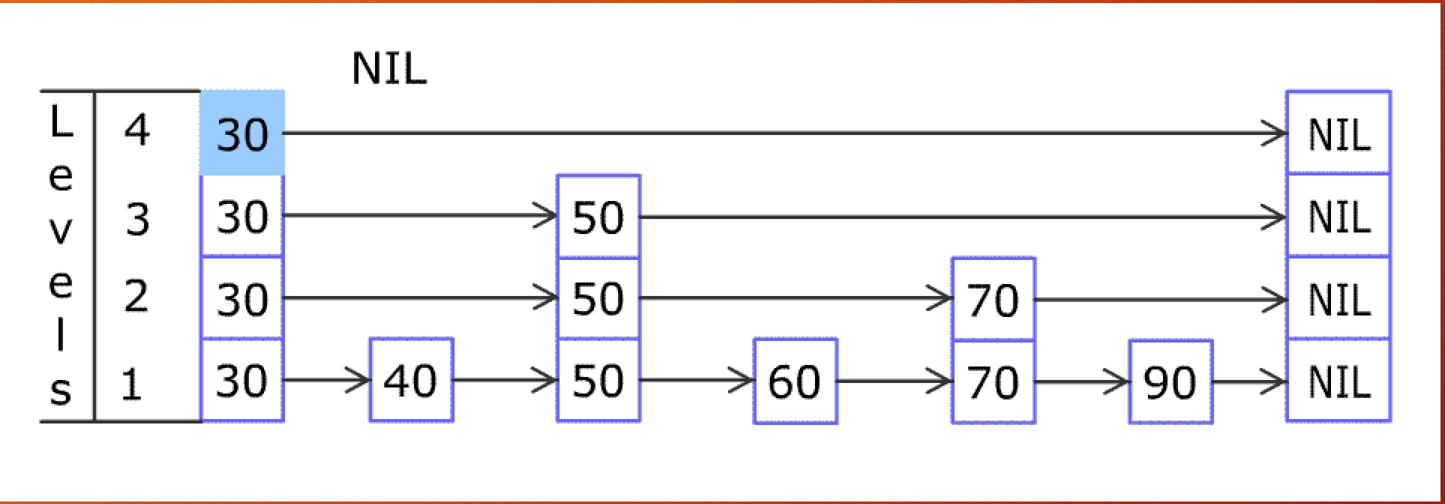
Skip List: Insertion



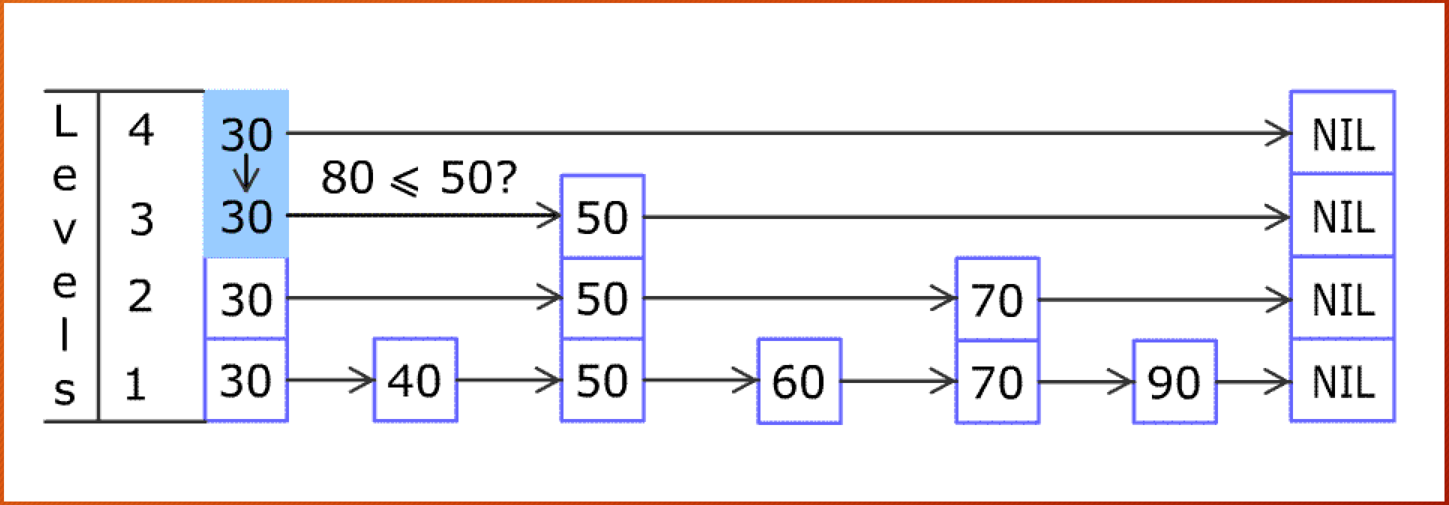
Skip List: Insertion



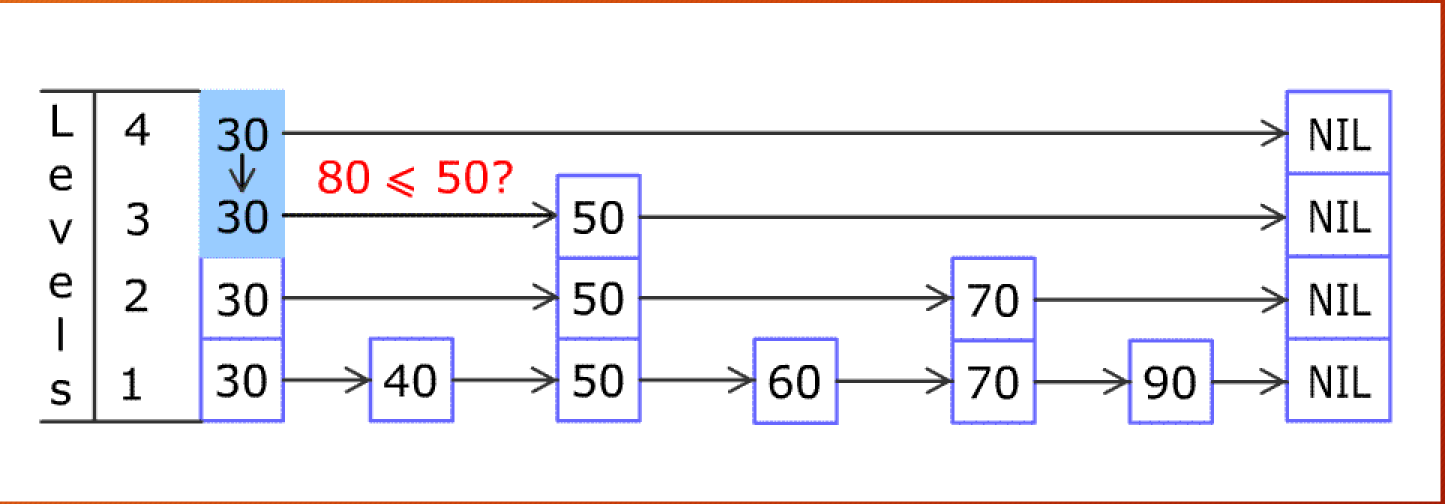
Skip List: Insertion



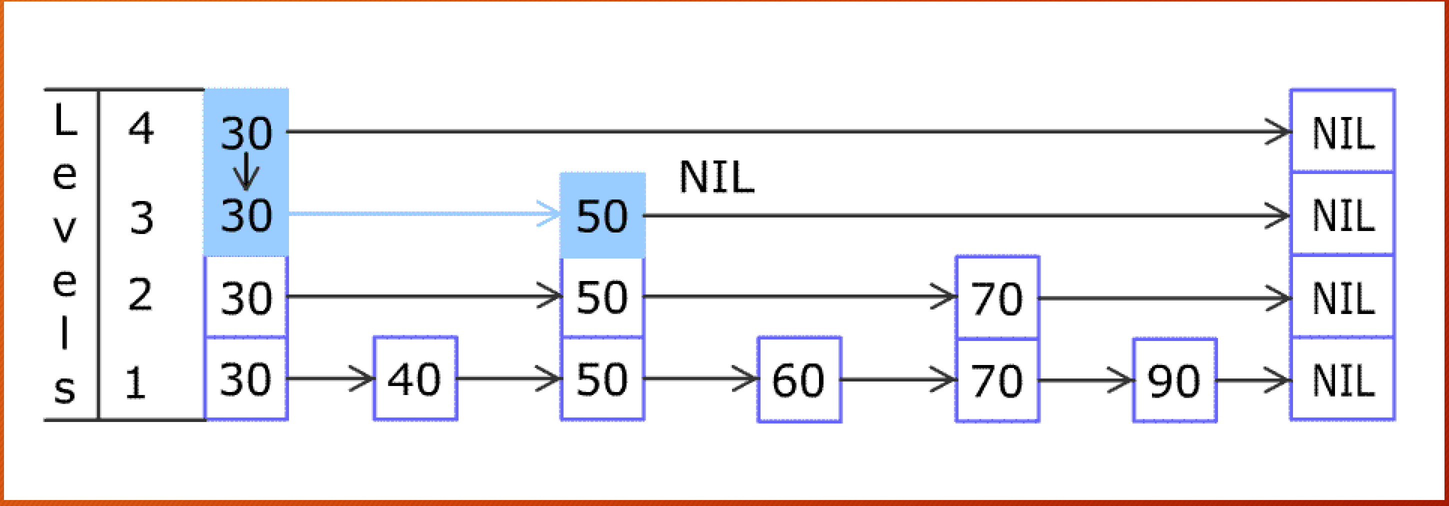
Skip List: Insertion



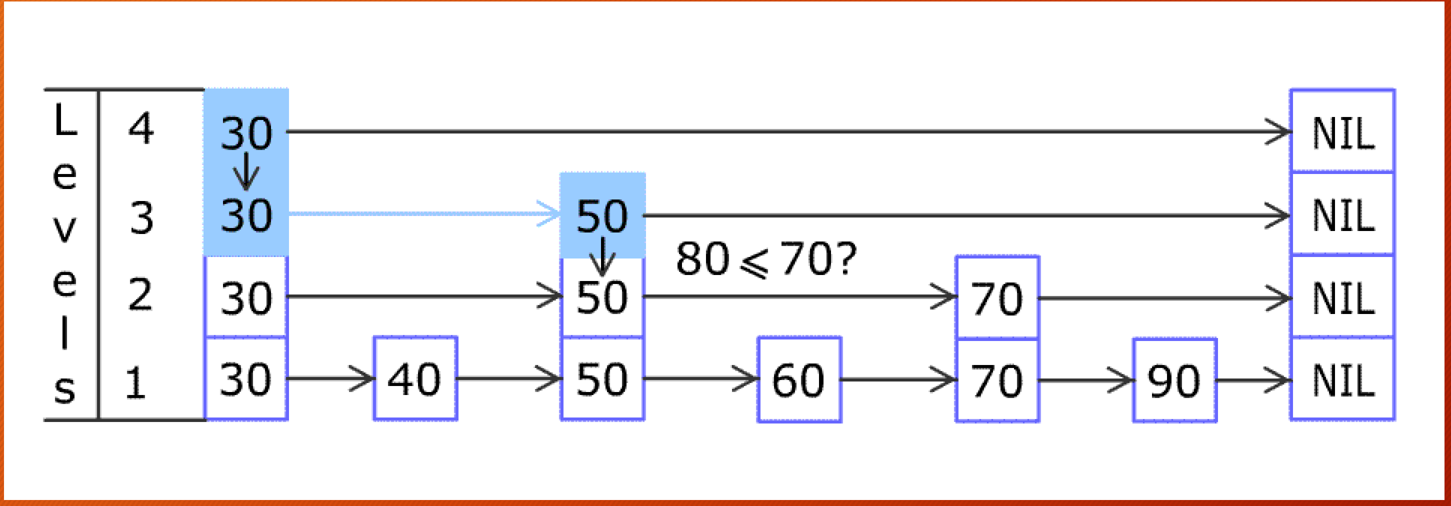
Skip List: Insertion



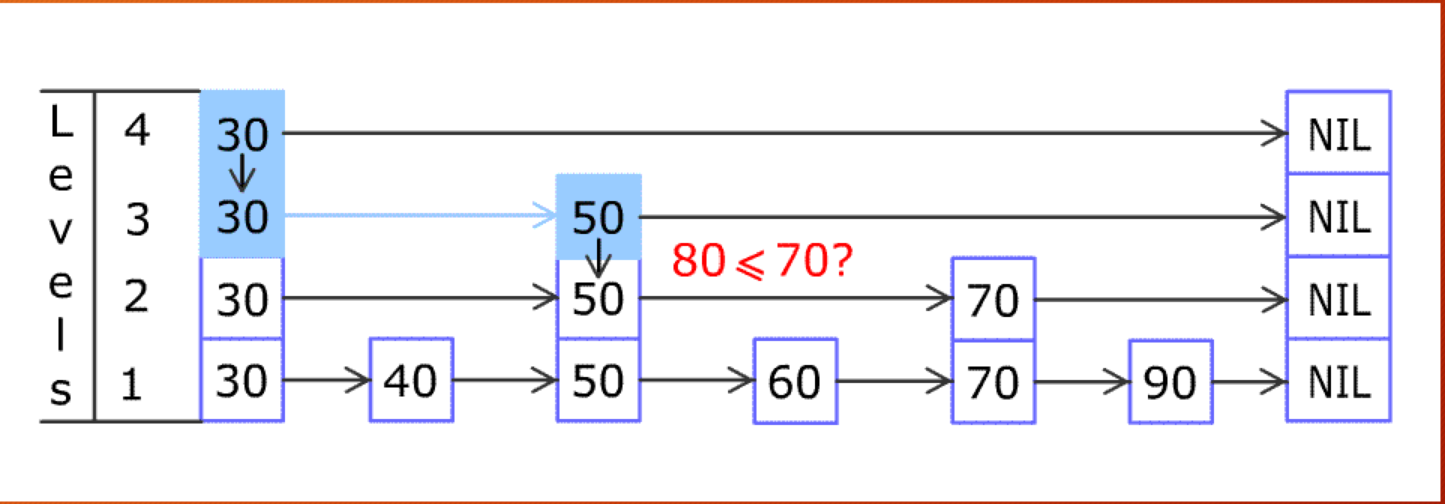
Skip List: Insertion



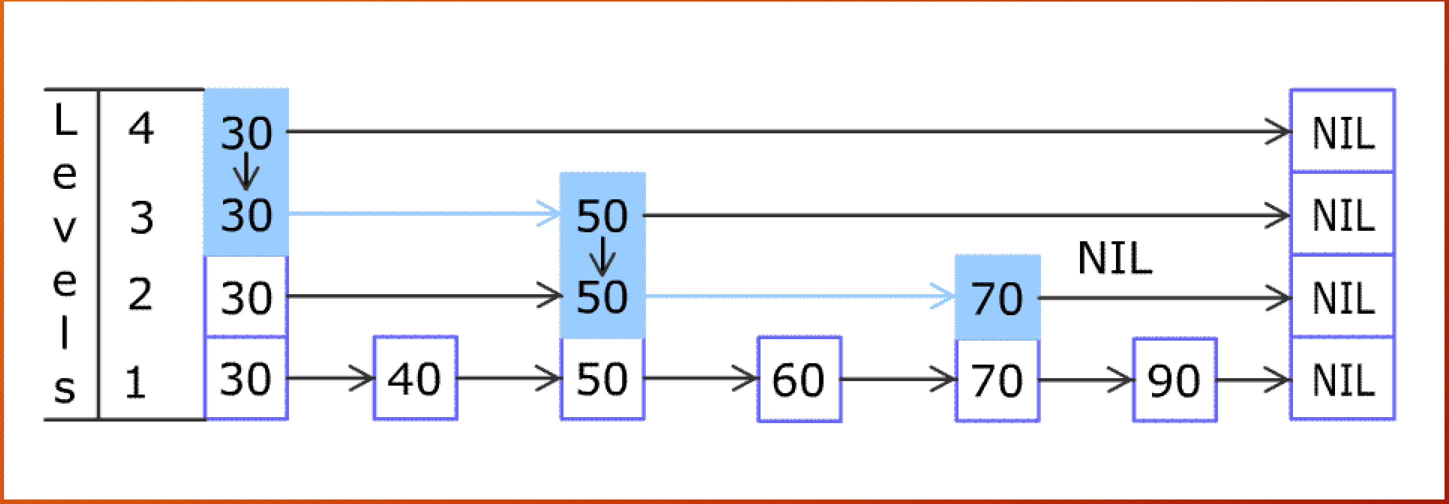
Skip List: Insertion



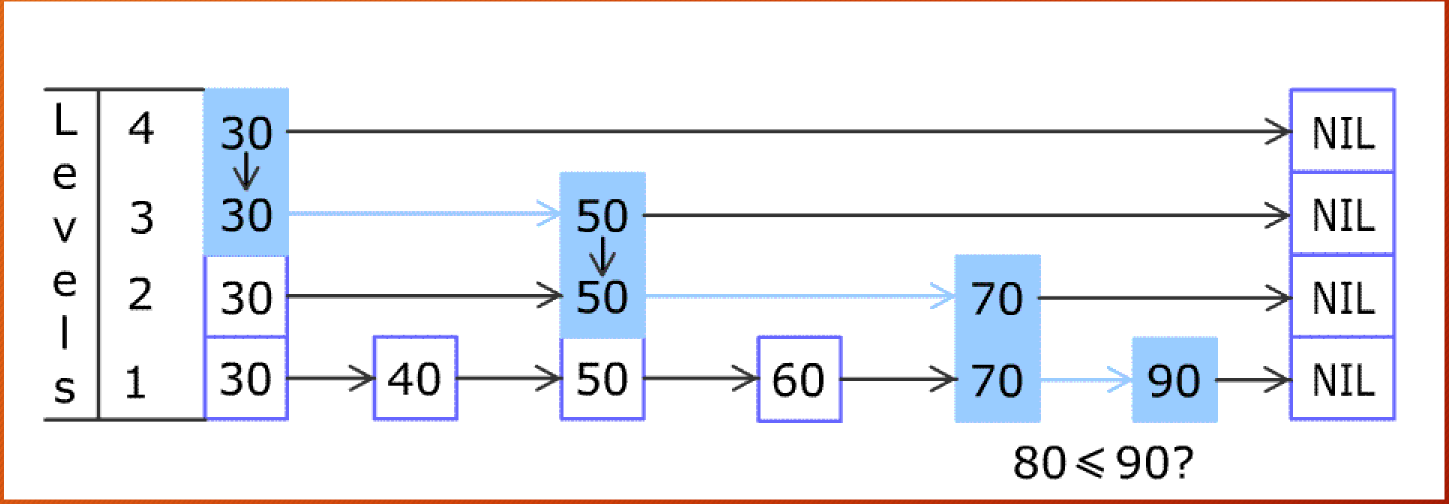
Skip List: Insertion



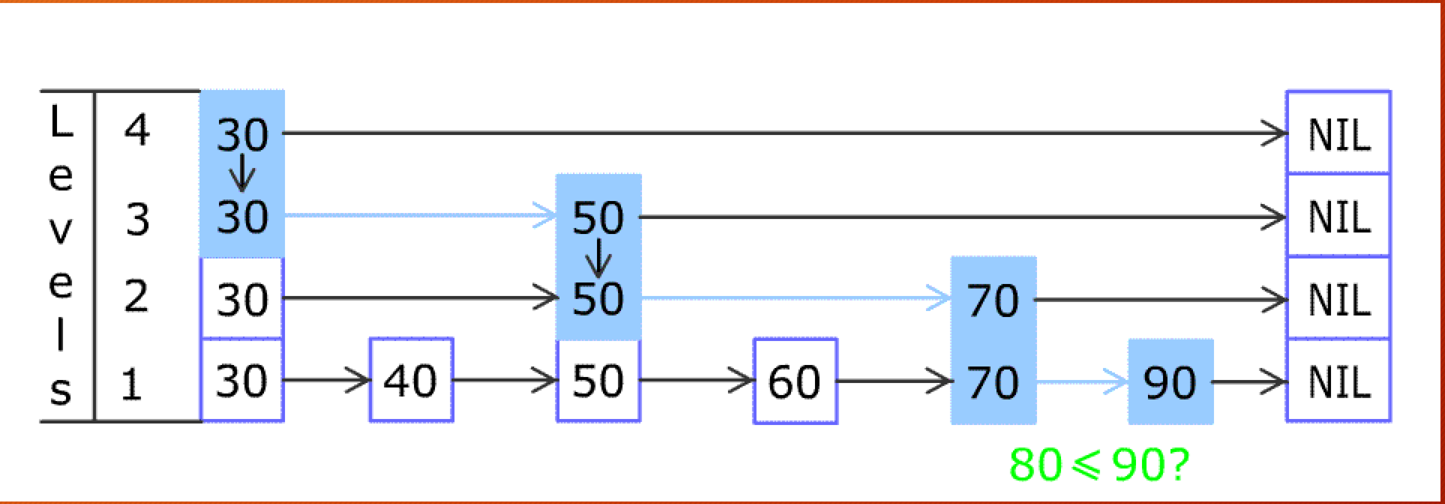
Skip List: Insertion



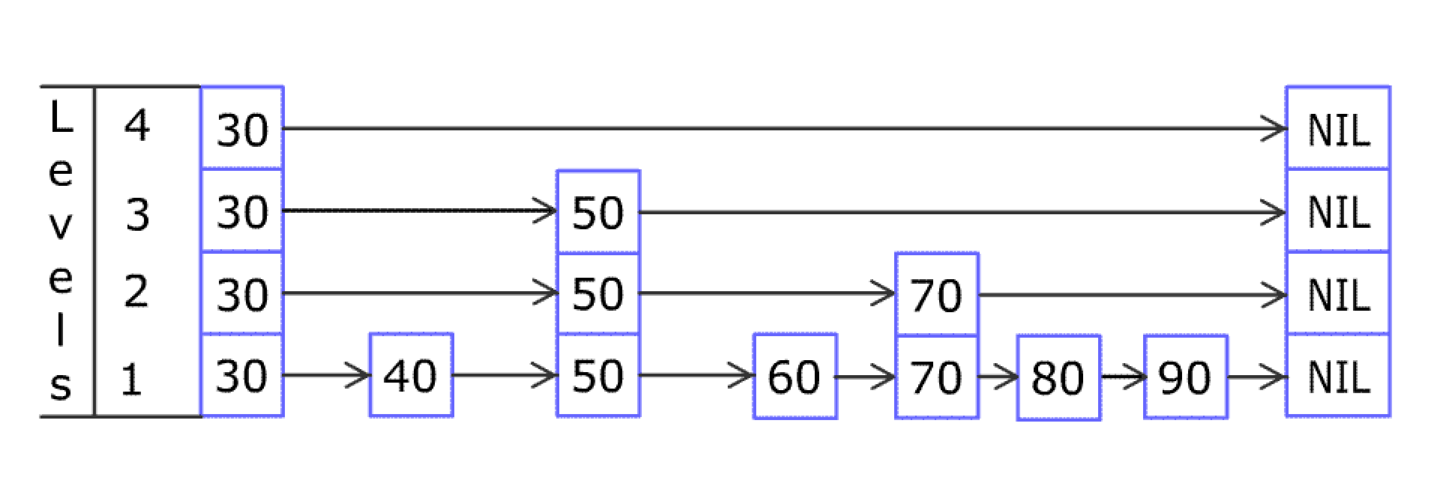
Skip List: Insertion



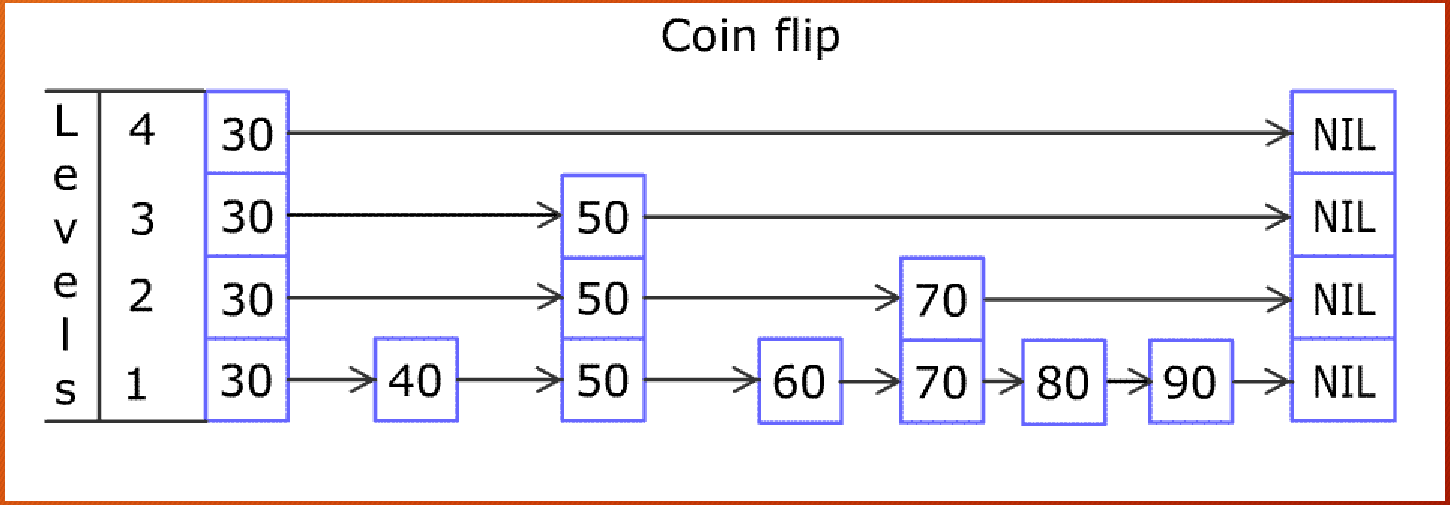
Skip List: Insertion



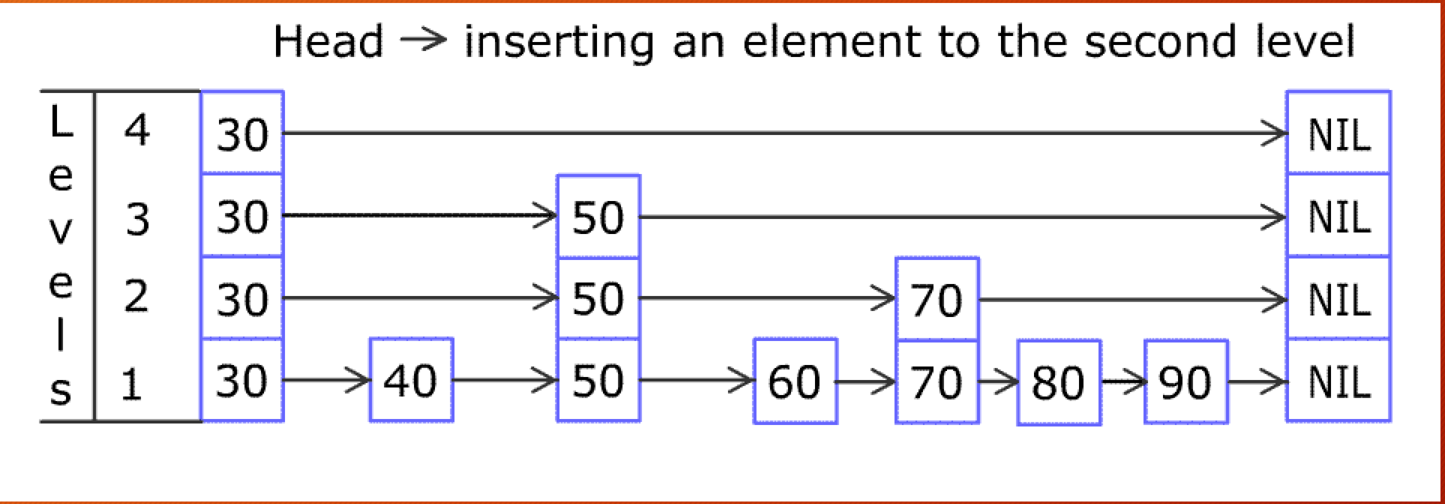
Skip List: Insertion



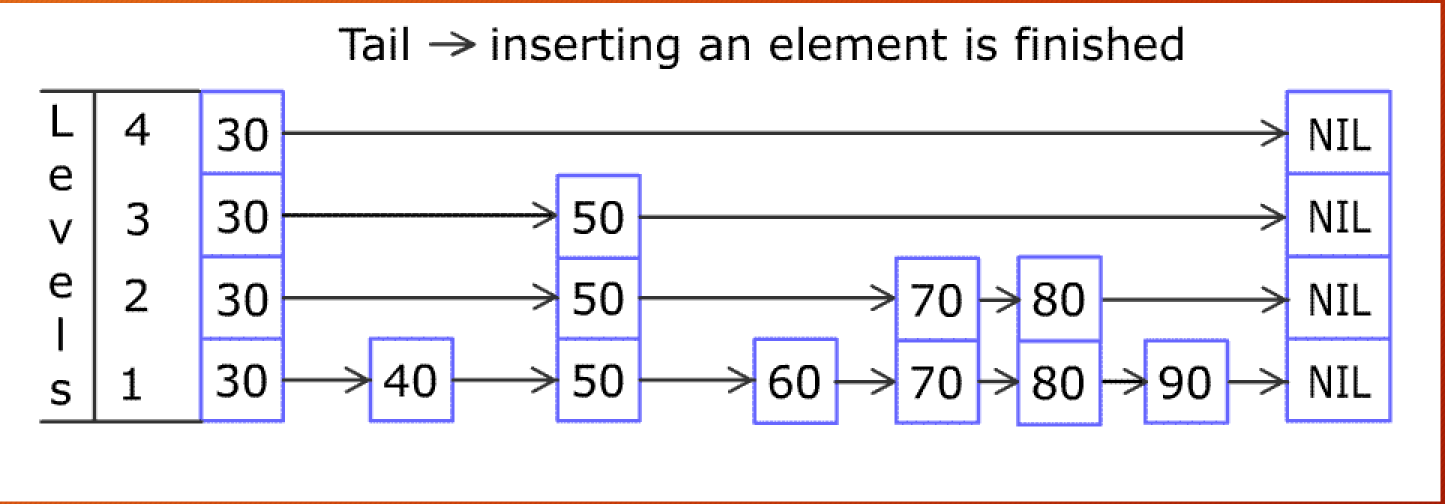
Skip List: Insertion



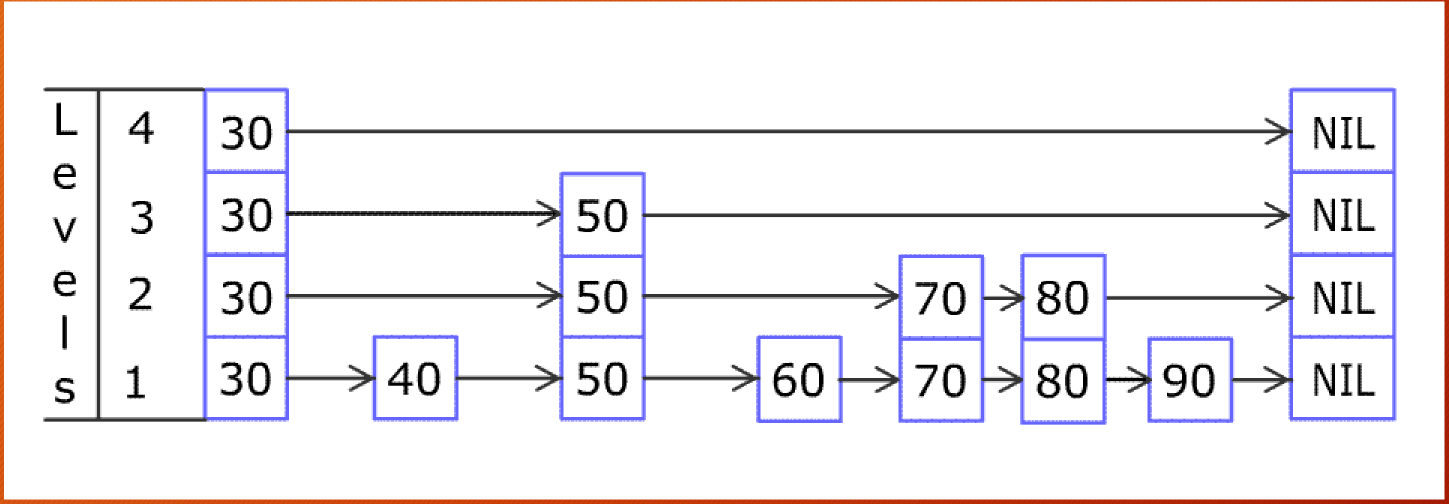
Skip List: Insertion



Skip List: Insertion



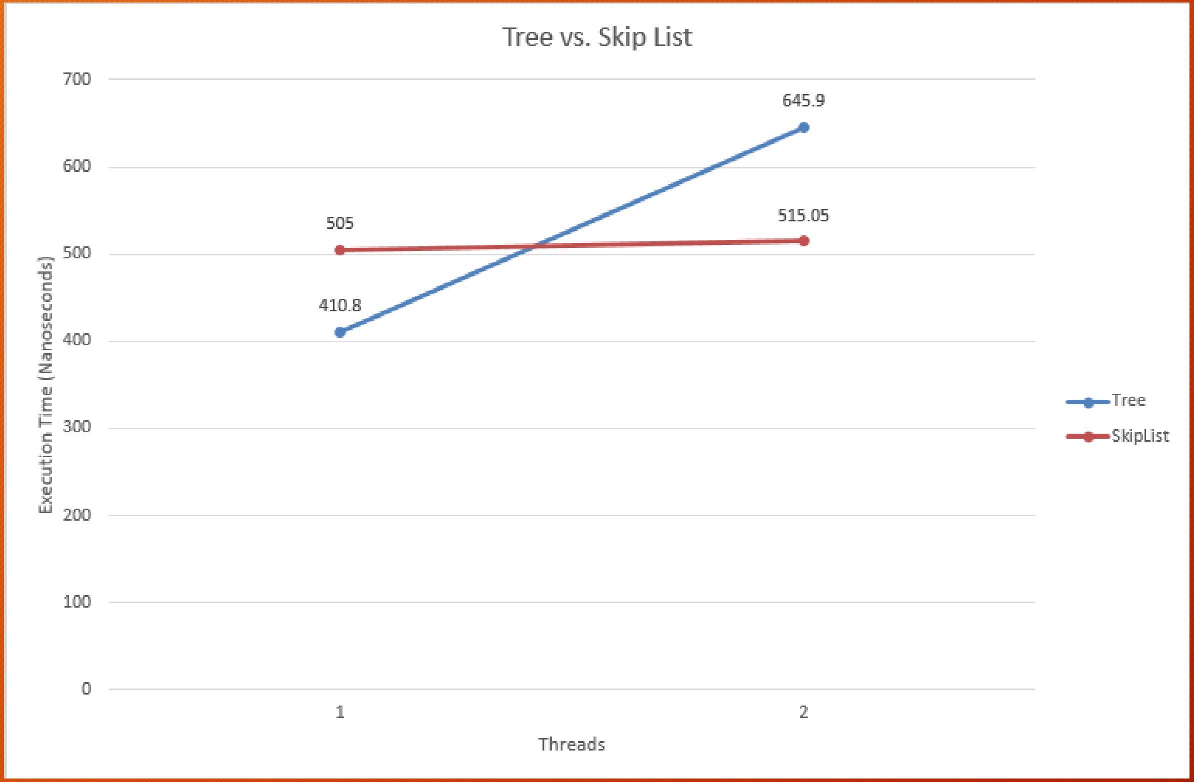
Skip List: Insertion



Skip Lists: Performance

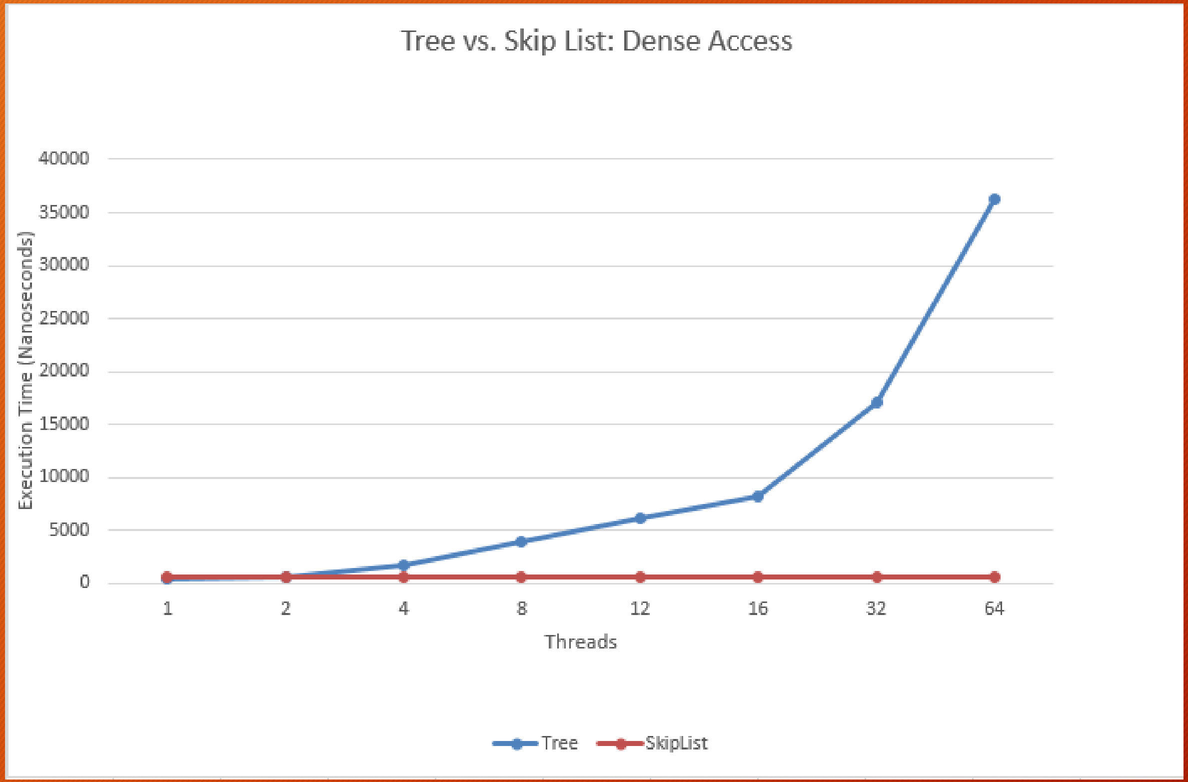
- Internally locked Skip List vs. Globally locked Tree
- Using `ConcurrentSkipListMap` and `TreeMap` from Javas standard library
- Create of 100,000 `Integer` to `String` mapping in each map
  - Both maps are identical
- Generate random integers and access the corresponding keys
  - Same entries accessed for each map
- Steadily increase number of threads accessing the Maps at a time
  - 1, 2, 4 etc.
  - Dense: each thread accesses 100,000 random entries
  - Sparse: each thread accesses 1000 random entries

Skip Lists: Performance



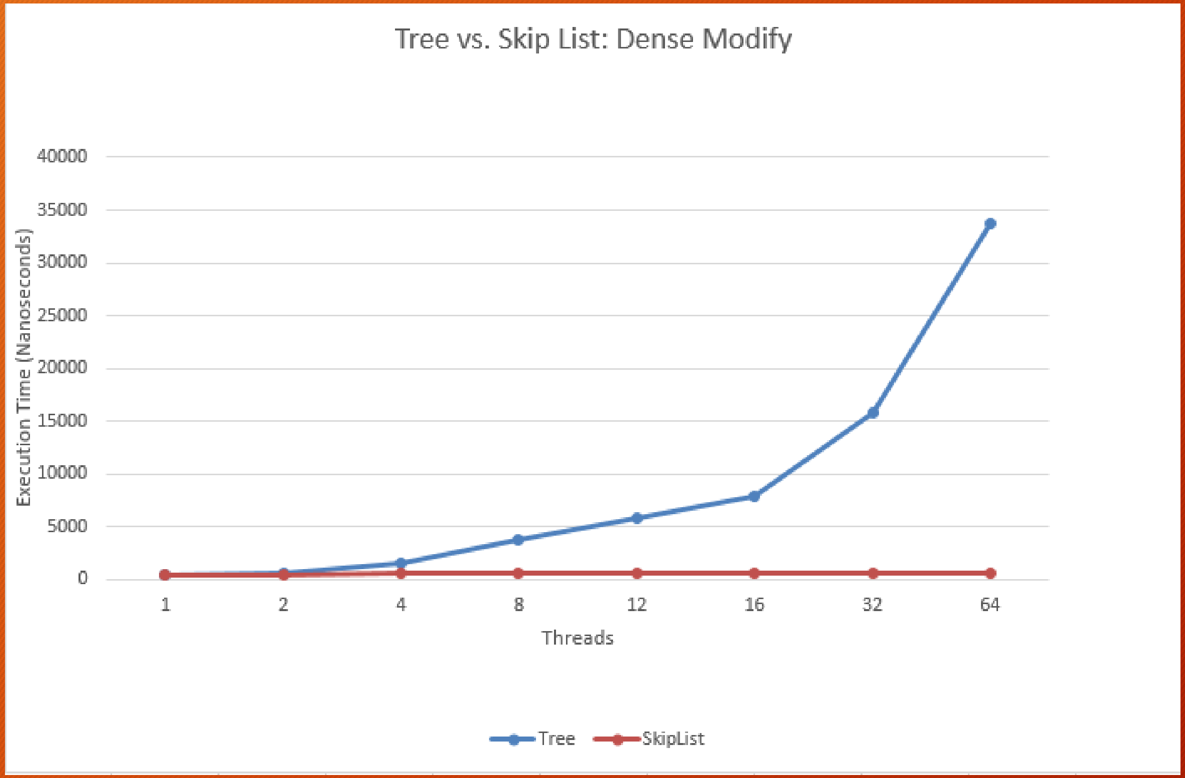
- Binary Trees are better in single threaded applications
- But even with only 2 threads Skip Lists pull ahead

Skip Lists: Performance



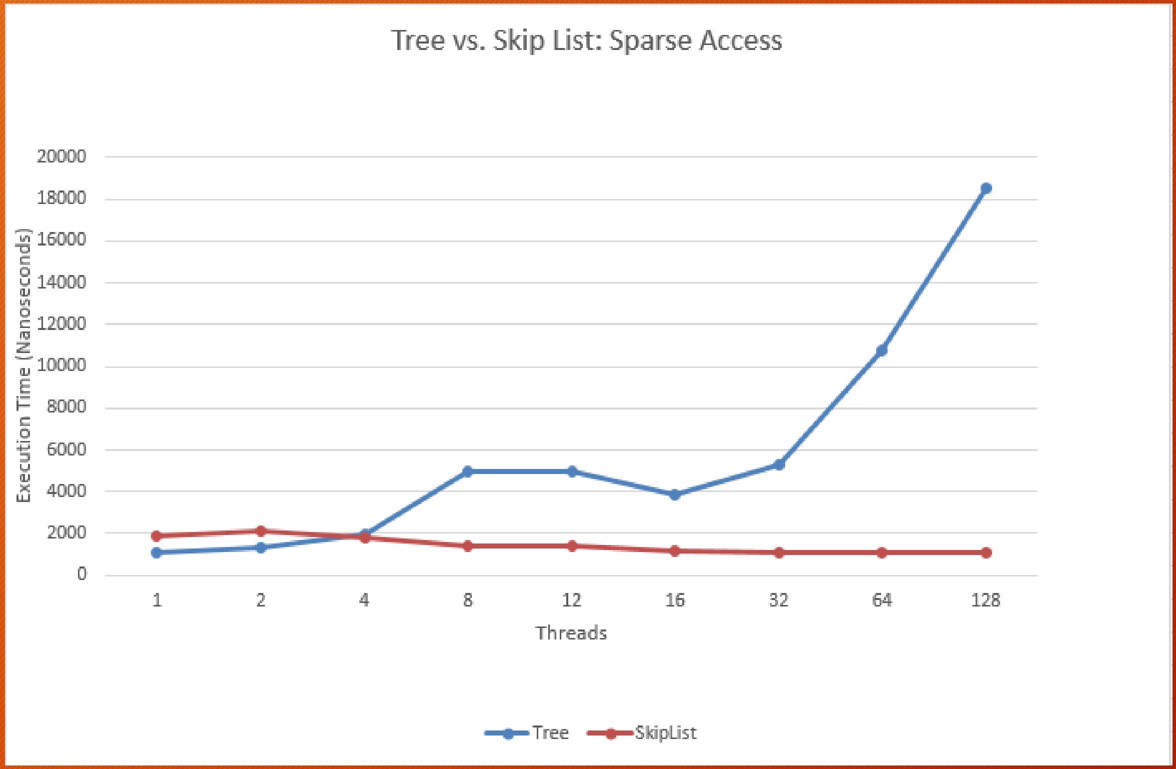
- Accessing lots of data
- It keeps getting worse

Skip Lists: Performance



- Modifying lots of data
- Similar story

Skip Lists: Performance



- Accessing little bit of data
- Somewhat better scaling

Skip Lists: Performance

- Modifying little bit of data
- Similar story

Skip Lists: Advantages

- Skip Lists are better for highly concurrent environments
  - Localized updates are better
  - Better performance for bulk data access
- Scale significantly better than Red-Black trees as parallelism increases
  - Global locks are bad; don’t scale
  - Finer locking mechanisms are rather convoluted
- Lock-free skip lists are possible!
  - Perform about as well as locking ones
- Next time, when you want a lock-free concurrency-friendly data structure consider a Skip List!

Conclusions

- References:
  - Keir Fraser "Paractical Lock-freedom" http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf
  - William Pugh "Skip lists: A probabilistic alternative to balanced trees" ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf
  - Thomas Papadakis "Skip Lists and Probabilistic Analysis of Algorithms" https://cs.uwaterloo.ca/research/tr/1993/28/root2side.pdf
  - Kier Fraser "Concurrent Programming without Locks" http://www.cl.cam.ac.uk/research/srg/netos/papers/2007-cpwl.pdf
  - http://www.drdobbs.com/parallel/choose-concurrency-friendly-data-structu/208801371
- All data and code used can be found here:
  - https://github.com/DroidX86/skip-stats
  - Including all the references, further reading and this presentation

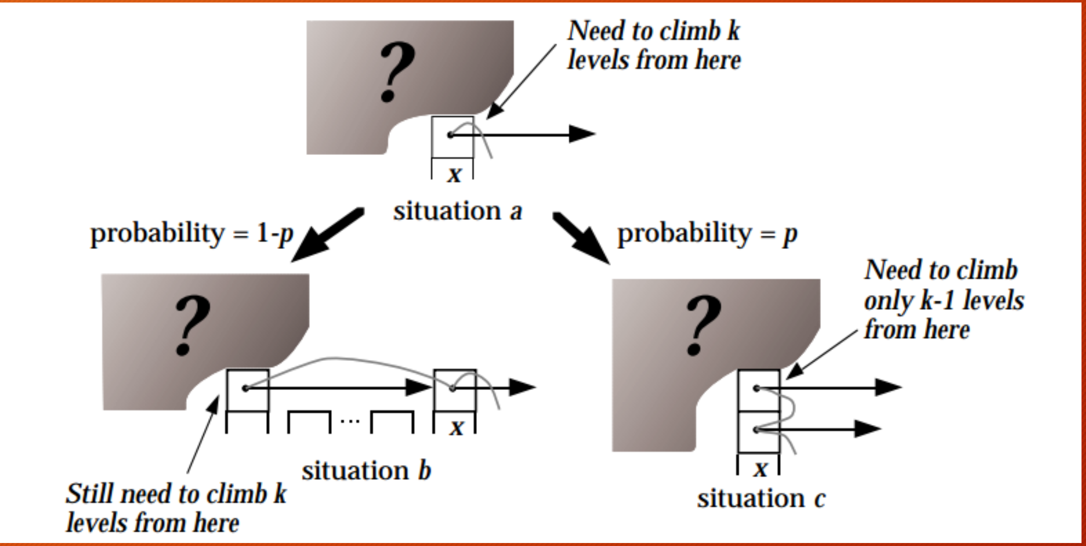
Thank You!

Rounak Das

Jadavpur University; BCSE-IV

Skip Lists: Analysis

- Runtime is dominated by time to search
- Analyze the search path backwards
- At any point we are here:



p is the probability of adding a level to a node

Skip Lists: Analysis

- Let $p = ½$ for simplicity
- Expected number of steps to walk through k levels
  - $C(k) = 1 + ½*C(k-1) + ½*C(k)$
  - Or, $C(k) = 2 + C(k-1)$
- $C(k) = 2*k$ by expansion
- Expected number of levels in the tree is $log(n)$
  - At level 1 = $n/2$
  - At level 2 = $n/4$ and
  - …
  - At level $log(n) = 1$
  - k can be $log(n)$ at most
- $C(k) = O(logn)$