Selected Sorting Algorithms

CS 165: Project in Algorithms and Data Structures

Michael T. Goodrich

UNIVERSITY of CALIFORNIA IRVINE

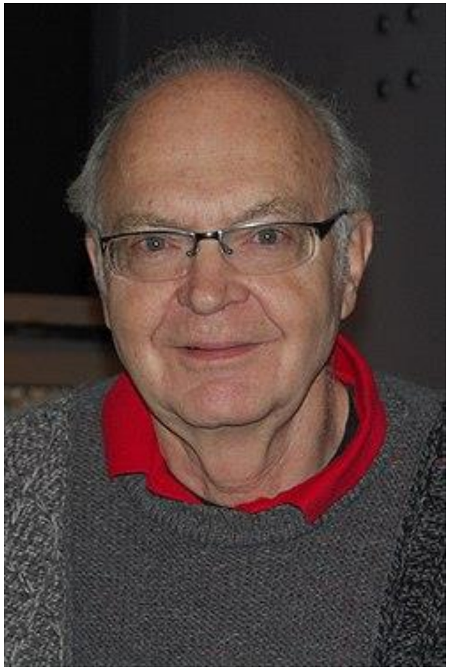
Some slides are from J. Miller, CSE 373, U. Washington

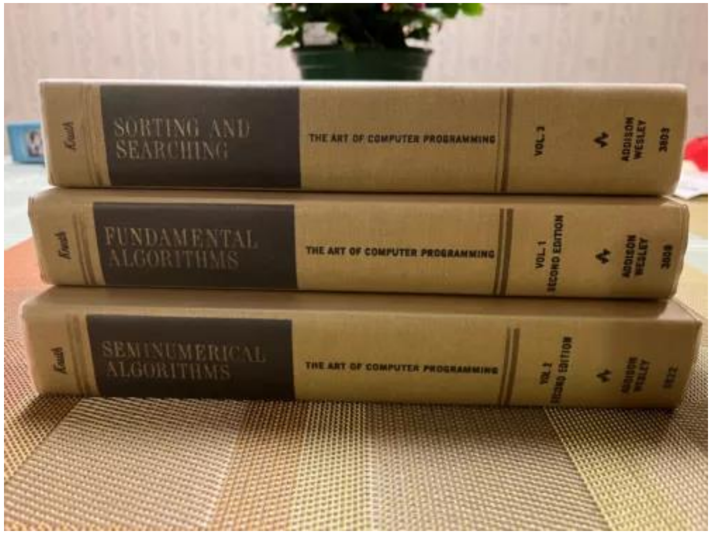
Why Sorting?

- Practical application
  - People by last name
  - Countries by population
  - Search engine results by relevance
- Fundamental to other algorithms
- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single ‘best’ sort for all scenarios
  - Knowing one way to sort just isn’t enough
- Many to approaches to sorting which can be used for other problems

Why Sorting?

- Donald Knuth, in his book "The Art of Computer Programming" (Volume 3), famously stated: "Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting or searching!".





Problem statement

There are *n* comparable elements in an array and we want to rearrange them to be in increasing order

Pre:

- An array **`A`** of data records
- A value in each data record
- A comparison function
  - <, =, >, compareTo

Post:

- For each distinct position **`i`** and **`j`** of **`A`**, if **`i < j`** then **`A[i] ≤ A[j]`**
- **`A`** has all the same data it started with

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

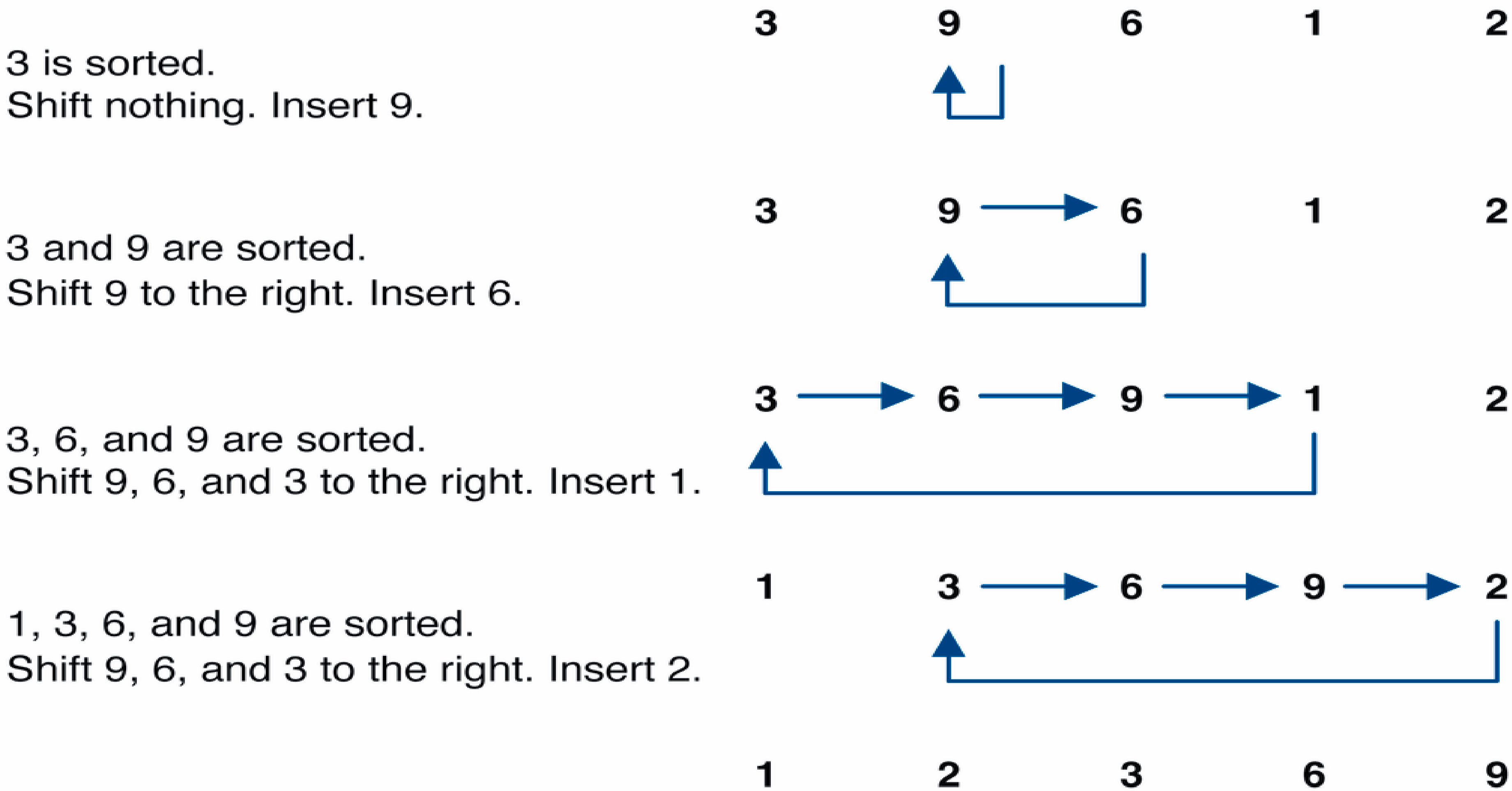
- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Simple sorting algorithm.
  - n-1 passes over the array
  - At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

	2	15	8	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 2	2	8	15	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 3	1	2	8	15	17	10	12	5
	0	1	2	3	4	5	6	7

Insertion sort example



Insertion sort code

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion-sort Analysis

- An **inversion** in a permutation is the number of pairs that are out of order, that is, the number of pairs, (i,j), such that $i<j$ but $x_i>x_j$.
- Each step of insertion-sort fixes an inversion or stops the while-loop.
- Thus, the running time of insertion-sort is $O(n + k)$, where k is the number of inversions.

Insertion-sort Analysis

- The worst case for the number of inversions, k, is…

- This occurs for a list in reverse-sorted order.
- What about the following sequence consisting of two increasing consecutive subsequences (which are called “runs”)?

(1,3,5,7,…,[n/2]-1, 2,4,6,8,…[n/2])

Shell sort description

- **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position

Shell sort example

- Idea: Sort all elements that are 5 indexes apart, then sort all elements that are 3 indexes apart, ...

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	72	82	93	95	15 swaps

Shell sort code

```
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide element i back by gap indexes
            // until it's "in order"
            int temp = a[i];
            int j = i;
            while (j >= gap && temp < a[j - gap]) {
                a[j] = a[j - gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
```

Shell sort Analysis

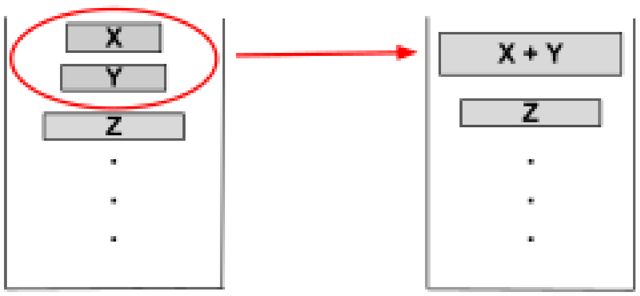
- Harder than insertion sort
- But certainly no worse than insertion sort
- Worst-case: $O(n^2)$
- Average-case: ????

Tim-sort

- Tim-sort is a recent sorting algorithm that was included in reference implementations for Java and Python.
- It uses a bunch of heuristics aimed at speeding up the running time for sorting
- As such, it took over 10 years before its running time was proved to be O(n log n) in the worst case.

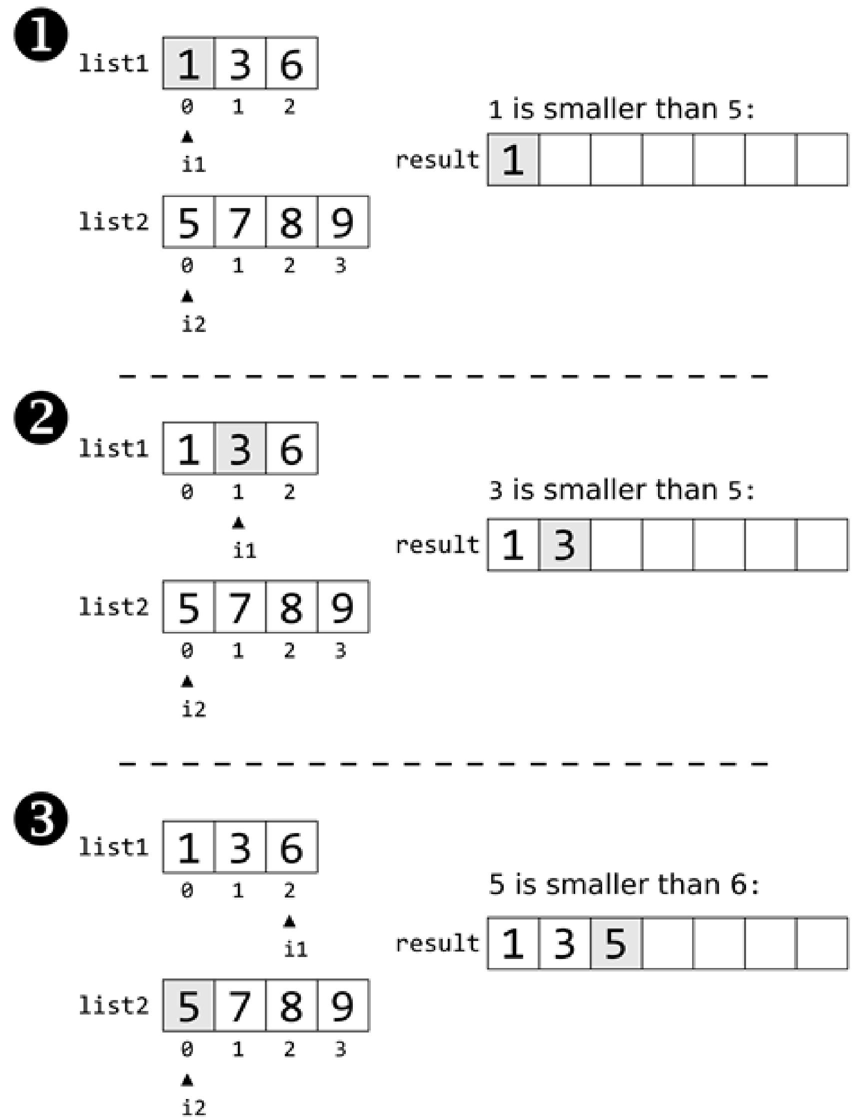
Tim-sort

- First, Tim-sort divides the input into **runs**, i.e., increasing consecutive subsequences.
- Then, it pushes the first run onto a stack, and starts processing runs left to right, pushing each new run onto the stack
- When certain conditions occur, Tim-sort merges a pair of runs that are consecutive on the stack, replacing the pair on the stack with the merged result.



Merge Algorithm

The merge method in Tim-sort is the same merge method as used in the well-known Mergesort algorithm.



Tim-sort Core Algorithm

Algorithm 3: TimSort: translation of Algorithm 1 and Algorithm 2

Input: A sequence to S to sort

Result: The sequence S is sorted into a single run, which remains on the stack.

Note: At any time, we denote the height of the stack $\mathcal{R}$ by h and its i^{th} top-most run (for $1 \leqslant i \leqslant h$) by R_i. The size of this run is denoted by r_i.

```
runs ← the run decomposition of S
R ← an empty stack
while runs ≠ ∅ do                                              // main loop of TimSort
    remove a run r from runs and push r onto R                  // #1
    while true do
        if h ⩾ 3 and r1 > r3 then merge the runs R2 and R3                 // #2
        else if h ⩾ 2 and r1 ⩾ r2 then merge the runs R1 and R2            // #3
        else if h ⩾ 3 and r1 + r2 ⩾ r3 then merge the runs R1 and R2       // #4
        else if h ⩾ 4 and r2 + r3 ⩾ r4 then merge the runs R1 and R2       // #5
        else break
while h ≠ 1 do merge the runs R1 and R2
```

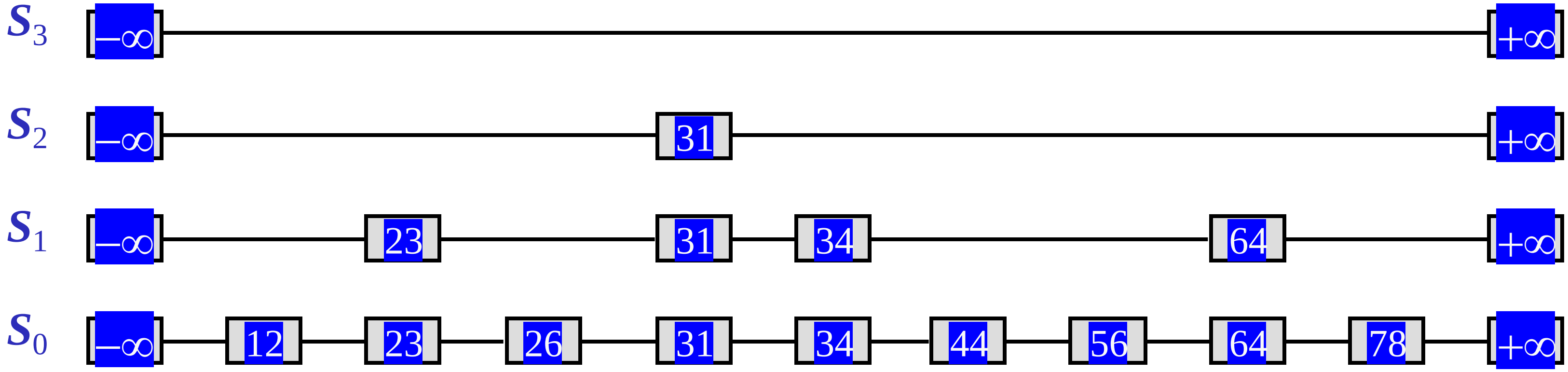
From https://arxiv.org/abs/1805.08612

Tim-sort Analysis

- The sizes of the runs on the stack grow at least as fast as the Fibonacci sequence, which is used to show that Tim-sort runs in O(n(1 + log r)) time, where r is the number of runs.
- In fact, it runs in time that is proportional to n times the binary Shannon entropy of the input sequence.

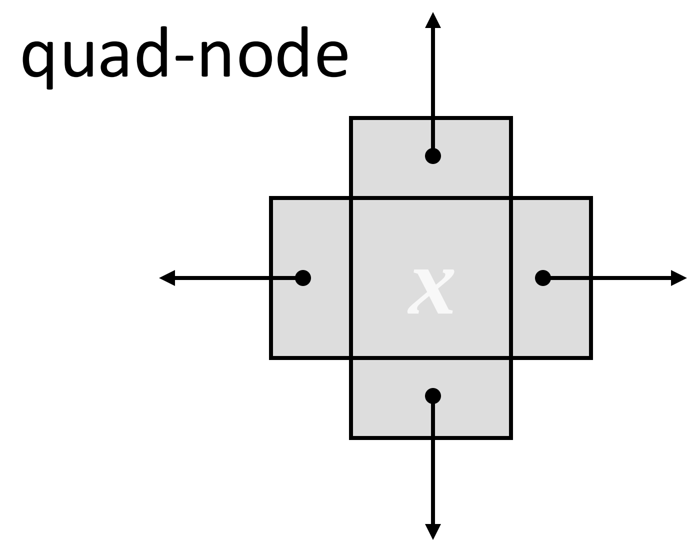
Skip List

- A skip list for a set S of distinct (key, element) items is a series of lists
  $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys
- Skip lists are one way to implement the dictionary ADT
- Java applet



Possible Implementation

- We can implement a skip list with quad-nodes
- A quad-node stores:
  - item
  - link to the node before
  - link to the node after
  - link to the node below
- Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them



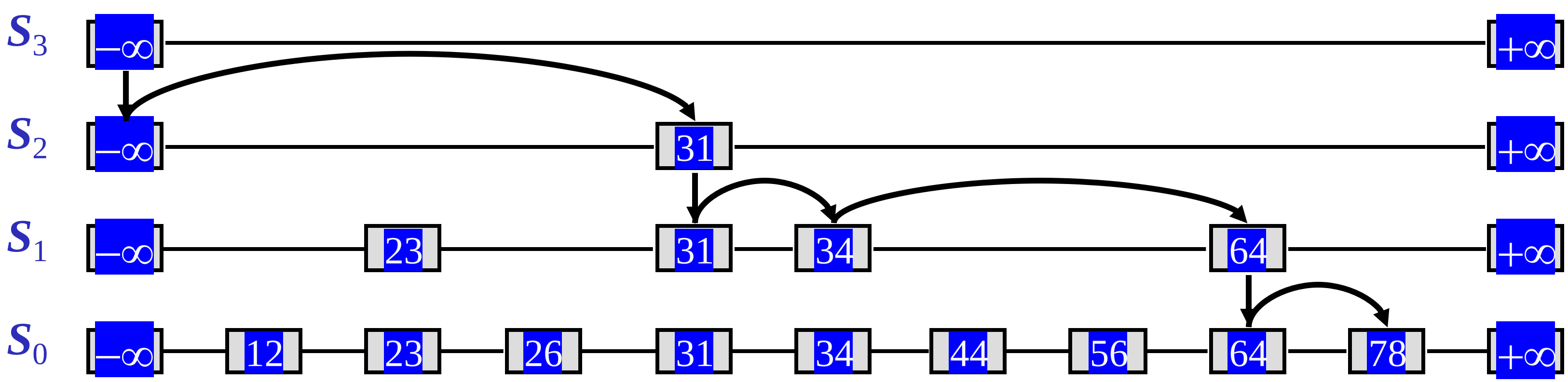
Top-Down Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(after(p))$

    $x = y$: we return $element(after(p))$

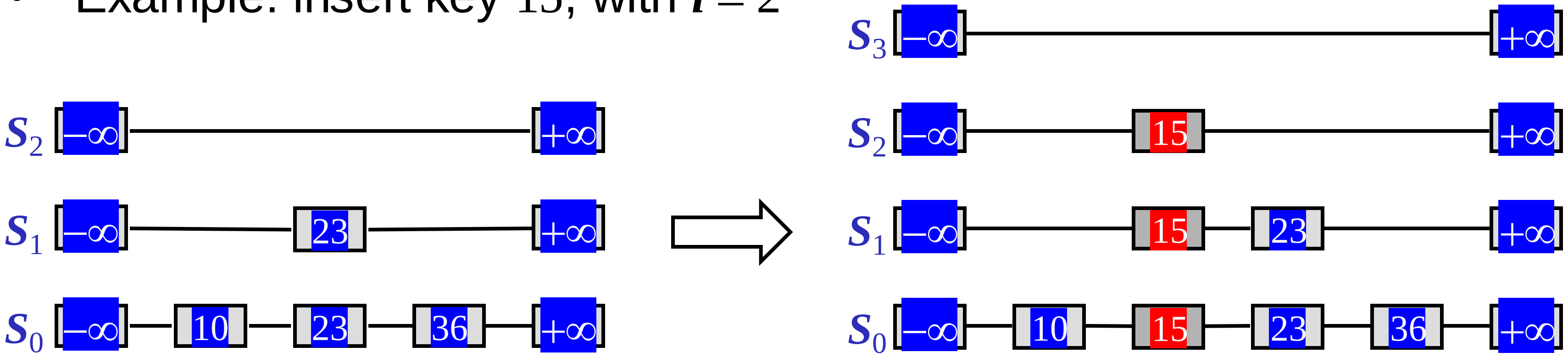
    $x > y$: we "scan forward"

    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return ***NO_SUCH_KEY***
- Example: search for 78



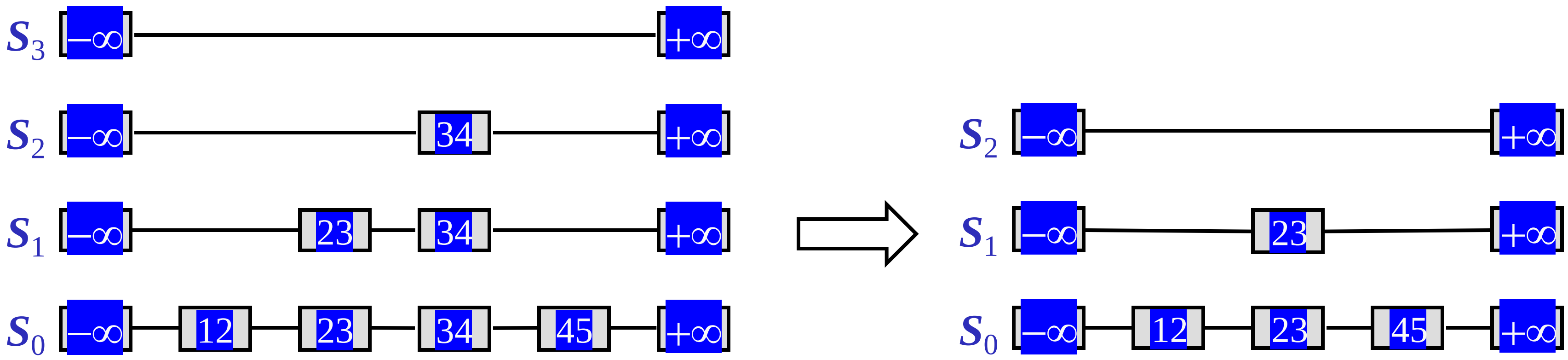
Insertion

- To insert an item (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
  - For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



Deletion

- To remove an item with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
- We use the following two basic probabilistic facts:

  Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

  Fact 2: If each of n items is present in a set with probability p, the expected size of the set is np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 2, the expected size of list S_i is $n/2^i$
- The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

Height

- The running time of the search an insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(\log n)$
- We use the following additional probabilistic fact:

  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most
  $n/2^{3\log n} = n/n^3 = 1/n^2$
- Thus a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

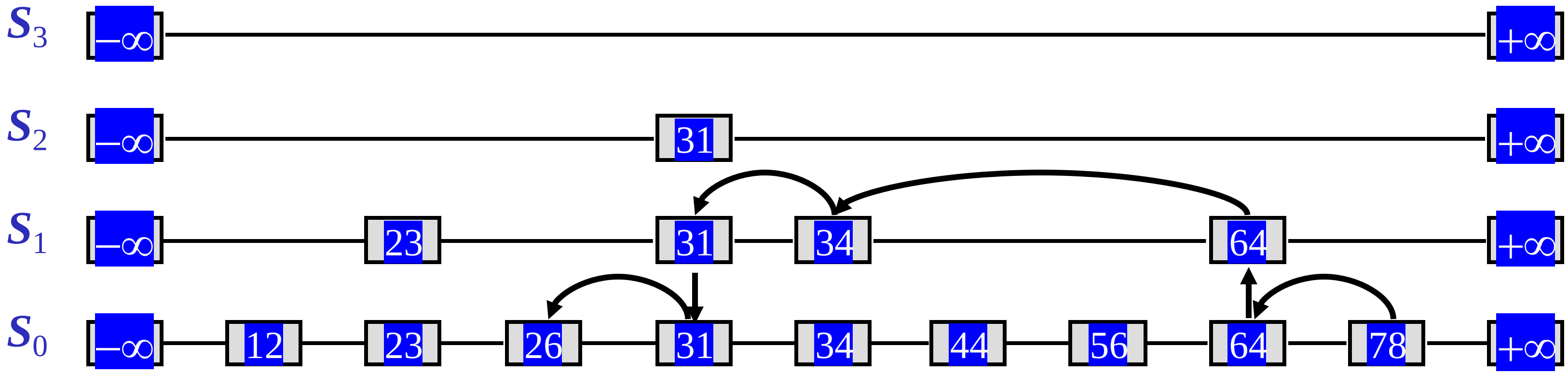
Search and Update Times

- The search time in a skip list is proportional to
  - the number of drop-down steps, plus
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability
- To analyze the scan-forward steps, we use yet another probabilistic fact:

  Fact 4: The expected number of coin tosses required in order to get tails is 2
- When we scan forward in a list, the destination key does not belong to a higher list
  - A scan-forward step is associated with a former coin toss that gave tails
- By Fact 4, in each list the expected number of scan-forward steps is 2
- Thus, the expected number of scan-forward steps is $O(\log n)$
- We conclude that a search in a skip list takes $O(\log n)$ expected time
- The analysis of insertion and deletion gives similar results

Up-Down Search

- We search for a key x in a a skip list as follows:
  - We start at the last bottom position of the top list
  - We move left and up until we reach a level with previous key less than the search key
  - Then we move down and left until we find the correct position
- Example: search for 26



Skip-List Sort

- Insert the elements x_1, x_2, …, into a skip-list, doing up-down search from the bottom-left part of the skip-list for each element x_i.
- The expected time to insert x_i is $O(\log d_i(X))$, where $d_i(X)$ is the distance in the bottom level to the place where x_i belongs.
- $d_i(X)$ is bounded by $I_i(X)$, where $I_i(X)$ is the number of inversions with x_i as the right element.

Analysis of Skip-List Sort

- The analysis of the expected running time uses the fact that the geometric mean is always at most the arithmetic mean:

$$
\begin{aligned}
& c \sum_{i=1}^{|X|} (1 + \log[d_i(X) + 1]) \\
& \quad = c|X| + c \log\left[\prod_{i=1}^{n} (d_i(X) + 1)\right] \\
& \quad = c|X| + 2c|X| \log\left(\prod_{i=1}^{|X|} [I_i(X) + 1]\right)^{1/|X|} \\
& \quad \leq c|X|\left(1 + 2\log\left[\frac{Inv(X)}{|X|} + 1\right]\right).
\end{aligned}
$$