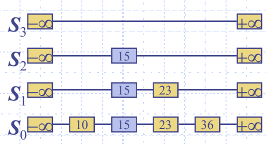
Lecture 10:
Skip Lists and Quick Selection

Law of Least Action

Wholeness Statement

Randomized algorithms are efficient with high probability, that is, their worst case behavior is highly unlikely. The field of pure consciousness is always efficient and follows the law of least action.

Skip Lists



Skip Lists

- A relatively recent data structure
  - A probabilistic alternative to balanced trees
  - A randomized algorithm with similar running times as red-black trees
    - O(log *n*) expected time for search and update operations
  - *Much* easier to code than red-black trees!
  - Fast!

Skip Lists

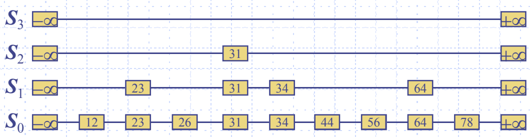
- Can be used to implement an ordered dictionary
- Simpler insertion and deletion than a bounded-depth binary search tree
  - (AVL and Red-Black trees)
- Uses about the same amount of space
- Example of a randomized algorithm

Outline and Reading

- Definition of Skip List (§3.5)
- Operations
  - Search (§3.5.1)
  - Insertion (§3.5.2)
  - Deletion (§3.5.2)
- Implementation
- Analysis (§3.5.3)
  - Space usage
  - Search and update times

What is a Skip List

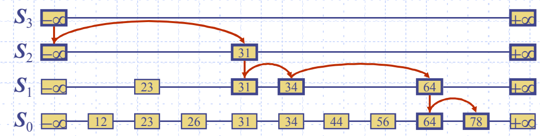
- A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \dots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains all the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,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



7

Search

- Search for a key x in a a skip list as follows:
  - Start at the first position of the top list
  - At the current position p, 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



8

Randomized Algorithms

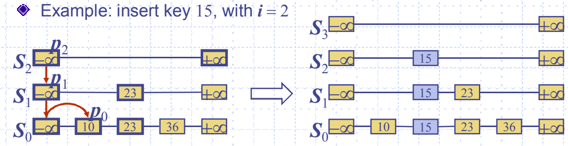
- A randomized algorithm performs coin tosses to control its execution
  - (i.e., uses random bits)
- It contains statements of the type

  $b \leftarrow random()$
  if $b = 0$
  do A …
  else { $b = 1$}
  do B …
- The algorithm's running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large
- But the worst case has very low probability
  - (e.g., it occurs when all the coin tosses give "heads")
- A randomized algorithm is used to insert items into a skip list

9

Insertion

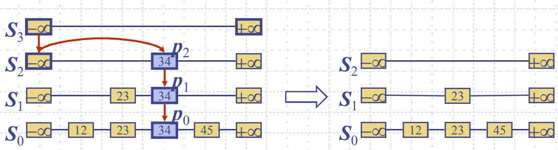
- To insert an item (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \dots, S_{i+1}$, each containing only the two special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



10

Deletion

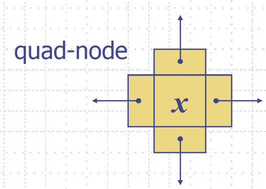
- To remove an item with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \dots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \dots, p_i$ from the lists $S_0, S_1, \dots, S_i$
 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



11

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
  - link to the node above
- Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them



12

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 and 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 $1 - 1/n^2$

Generalizing

- For c > 1, h is larger than c log n with probability at most 1 / n^{c-1}
- That is, the probability that h is smaller than c log n is 1 – 1 / n^{c-1}

Skip List Search

- To search for an element with a given key:
  - Find location in top list
    - Top list has O(1) elements with high probability
    - Each key in this list defines a range of items in next list
  - Drop down a level and recurse
- O(1) time per level on average (i.e., search forward 2 times on average)
- O(log *n*) levels with high probability
- Total time: O(log *n*)

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

Summary

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm
- In a skip list with n items
  - The expected space used is $O(n)$
  - The expected search, insertion and deletion time is $O(\log n)$
- Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability
- Skip lists are fast and simple to implement in practice

Implementing a Dictionary

- Comparison of efficient dictionary implementations

	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	• no ordered dictionary methods • simple to implement
Skip List	log n high prob.	log n high prob.	log n high prob.	• randomized insertion • simple to implement
Red Black Tree	log n worst-case	log n worst-case	log n worst-case	• complex to implement

Main Point

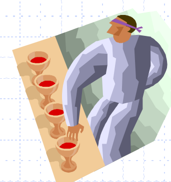
1. A skip list is a technique for efficiently implementing an ordered dictionary ADT. It uses random choices, independent of the distribution of the keys, to arrange items such that search and update times are O(log n) on average.
   The dynamism of the unified field seems chaotic when studied at the macroscopic level, yet it is a field of perfect order, responsible for the order and balance in creation.

Another Algorithm Design Pattern

- Prune and Search
  - AKA Decrease and Conquer
  - Examples:
    - binary search (earlier)
    - quick select
- Randomized Algorithms
  - Quick Sort, Skip Lists, and Quick Select

Selection

Prune-and-Search
or
Decrease-and-Conquer



The Selection Problem

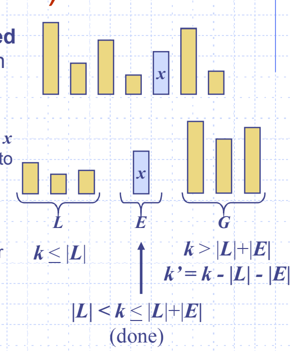
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

k=3   7 4 9 <u>6</u> 2 → 2 4 <u>6</u> 7 9

- Can we solve the selection problem faster?

Quick-Select (§ 4.7)

- Quick-select is a **randomized** selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Search: depending on k, either answer is in E, or we need to recurse in either L or G



Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

k=5, S=(7 4 9 3 2 6 5 1 8)

k=2, S=(7 4 9 6 5 8)

k=2, S=(7 4 6 5)

k=1, S=(7 6 5)

5

Quick Select

Algorithm *QuickSelect*(S, *lo, hi , k*)
  Input Unsorted Sequence S and k
  Output the k-th smallest element in S

```
p ← inPlacePartition(S, lo, hi)
j ← p – lo + 1
if j = k then
    return S.elemAtRank(p)
else if j > k then
    return QuickSelect(S, lo, p-1, k)
else
    return QuickSelect(S, p+1, hi, k-j)
```


In Place Version of Partition

Algorithm *inPlacePartition*(S, *lo, hi*)
  Input Sequence S and ranks lo and hi, 0 ≤ lo ≤ hi < S.size()
  Output the pivot is now stored at its sorted rank

```
p ← a random integer between lo and hi
S.swapElements(S.atRank( lo ), S.atRank( p ))
pivot ← S.elemAtRank(lo)
j ← lo + 1
k ← hi
while j ≤ k do
    while k ≥ j ∧ S.elemAtRank( k ) ≥ pivot do
        k ← k – 1
    while j ≤ k ∧ S.elemAtRank( j ) ≤ pivot do
        j ← j + 1

    if j < k then
        S.swapElements(S.atRank( j ), S.atRank( k ))

S.swapElements(S.atRank( lo ), S.atRank( k )) {move pivot to sorted rank}
return k
```


- What is the time complexity of Partition?
- What is the loop invariant of the outermost loop?

The loop invariant of the outer-most loop of inPlacePartition

forall i; lo+1 ≤ i < j; S.elemAtRank(i) ≤ pivot
- The values in S at ranks between lo+1 and j are less than the pivot

/\

forall i; k < i ≤ hi; S.elemAtRank(i) ≥ pivot
- The values in S at ranks between k and hi are greater or equal to the pivot

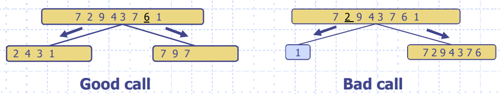
In Place Version of Partition

Algorithm *inPlacePartition*(S, *lo, hi*)
  Input Sequence S and ranks lo and hi, 0 ≤ lo ≤ hi < S.size()
  Output the pivot is now stored at its sorted rank *k* and *k* is returned;

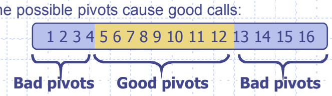
```
p ← a random integer between lo and hi
S.swapElements(S.atRank( lo ), S.atRank( p ))
pivot ← S.elemAtRank(lo)
j ← lo + 1
k ← hi
while j ≤ k do
    if S.elemAtRank( j ) ≥ pivot
        then
            S.swapElements(S.atRank( j ), S.atRank( k ))
            k ← k – 1
        else
            j ← j + 1

S.swapElements(S.atRank( lo ), S.atRank( k ))

return k
```

Expected Running Time

- Consider a recursive call of quick-select on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

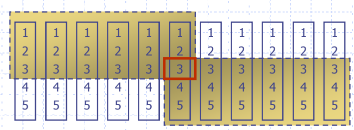
- Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function:
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$
- Let T(n) denote the expected running time of quick-select.
- By Fact #2,
  - $T(n) \leq T(3n/4) + (\text{expected \# of calls before a good call}) * bn$
- By Fact #1,
  - $T(n) \leq T(3n/4) + 2bn$
- So T(n) is O(?).

Conclusion

- We can solve the selection problem in O(n) expected time

Deterministic Selection

- We can do selection in O(n) worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide S into n/5 sets of 5 each
  - Find a median in each set
  - Recursively find the median of the "baby" medians.



- See Exercise C-4.24 for details of analysis.

Main Point

2. Prune-and-Search algorithms reduce the search space by some fraction at each step, then the smaller problem is recursively solved.
   The problem of world peace can be reduced to the smaller problem of peace and happiness of the individual. The problem can be further reduced to the much smaller problem of forming a small group (square root of 1%) practicing the TM and TM-Sidhi program together.

So far in the course

- Important basic data structures
  - Stacks and Queues
  - Lists, Vectors, Sequences, Trees, Priority Queues, Heaps
  - Dictionaries implemented based on Hash Tables, Binary Search Trees, and Skip Lists
- Important algorithms
  - Searching, sorting, and selection
- Design techniques
  - Divide-and-Conquer, Prune-and-Search, and Randomization
- Solution to recurrences
- Amortized analysis

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Selection can be done by sorting the input, then retrieving the element with the k-th smallest key (O(n log n)).

2. Applying the Prune-and-Search algorithm design strategy allows the Selection Problem to be solved in O(n)-time.

3. **Transcendental Consciousness** is the silent, unbounded home of all the laws of nature.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field follow the law of least action that governs the activities of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, one experiences the laws of nature as waves of one's own unbounded pure consciousness.