Deletion in Skip Lists

The idea for deletion is similar to that of insertion:

- Use SkipSearch to find the element to be deleted in S_0
  - If it's not there, return "not found"
- Delete the element from S_0, and as many higher lists as it's in

Code left as an exercise.

Probabilistic Analysis of Skip Lists

In the worst case, the coin always lands heads, and $S_0 = S_1 = S_2 = \cdots = S_h$

- Then the running time of Skip-Search is $O(n)$

This is very unlikely!

Claim: If $top[S] = h$, then the expected running time of a SkipSearch is $O(h)$.

Proof: Clearly we move down h times.
How often do we move across when we're searching for k?

- Suppose at ith level we move down at position x.
- That means $key[after[x]] > k$.
- Each key beyond x that we scan at level $i-1$ could not have been put at level i.
  - coin landed tails for that item – probability 1/2
- thus we scan an average of two items at level $i-1$
- $E(\#\text{ items scanned}) = 2h$ (across) $+\ h$ (down)

What is the probability that $top[S] = h$?

$$
\begin{aligned}
& \Pr(top[S] \geq h) \\
= \; & \Pr(h \text{ heads in a row for some element}) \\
\leq \; & \tfrac{n}{2^h}
\end{aligned}
$$

$$
\begin{aligned}
& E(\#\text{items scanned}) \\
= \; & \Sigma_{h \geq 1} 3h \Pr(top[S] = h) \\
= \; & \Sigma_{h=1}^{3 \lg n} 3h \Pr(top[S] = h) + \Sigma_{h > 3 \lg n} 3h \Pr(top[S] = h) \\
\leq \; & 9 \lg n \, \Sigma_{h=1}^{3 \lg n} \Pr(top[S] = h) + \Sigma_{h > 3 \lg n} 3h \Pr(top[S] = h) \\
\leq \; & 9 \lg n + \Sigma_{h > 3 \lg n} 3h \tfrac{n}{2^h} \\
\leq \; & 9 \lg n + \Sigma_{h > 3n \lg n} \tfrac{h}{2^h} \\
\leq \; & 9 \lg n + 3n \, \Sigma_{h > 3 \lg n} \tfrac{1}{2^{h/2}} \quad [\text{since } h \leq 2^{h/2} \text{ for } h \geq 4] \\
= \; & 9 \lg n + \tfrac{3n}{(n^{3/2})(1-(1/\sqrt{2}))} \\
& [\Sigma_{h > 3 \lg n} \tfrac{1}{2^{h/2}} \text{ is a geometric series with } r = 1/2^{1/2}] \\
= \; & 9 \lg n + O(1/\sqrt{n}) \\
= \; & O(\lg n)
\end{aligned}
$$

Similar analysis works to show that the expected running time of SkipInsert and SkipDelete is $O(\lg n)$

Skip Lists: Discussion

Skip lists are a relatively recent innovation.

- that's why they're not discussed in CLR

They seem to work very well in practice.

- the code is simple
  - no recursion
- the probabilistic analysis does not depend on the input being "nice"
- In practice, we seem to do better by using a biased coin
  - probability of heads is, say 1/4
  - this means we use fewer pointers

Amortized Complexity

Sometimes we're interested not only in the cost of one operation, but of a *sequence* of operations.

- E.g., in a dictionary, a sequence of inserts, deletes, and searches

Even if each operation in the sequence has expected cost $O(\lg n)$, the expected cost of a sequence of n operations may be only $O(n)$. *Amortized complexity* considers the cost of a sequence of operations.

- If a sequence of n operations takes time $O(n)$, each one takes $O(1)$ on average

Example: Consider the following algorithm for implementing a queue using two stacks (Exercise 11.1-6):

- Push every enqueue onto stack 1.
- For a dequeue,
  - if stack 2 isn't empty, then pop an element off stack 2.
  - if stack 2 is empty and stack 1 isn't, then move all of stack 1 onto stack 2 and then pop an element off stack 2.
  - if both stacks 1 and 2 are empty $\rightarrow$ error

Suppose we start with an empty queue and perform N enqueues and M dequeues

- Claim: this will take at most $2N$ pushes and at most $N + M$ pops.
  - The amortized complexity: at most 2 pushes per operation and at most 1 pop

Another example: In homework problem 13.2-4, you will show that $n-1$ successive TREE-SUCCESSOR calls take time $O(n)$, although each one takes expected time $O(\lg n)$ (and worst-case time $O(n)$).

Amortized complexity seems appropriate for analyzing the cost of a sequence.

- Can always get an upper bound by considering the worst-case time for each operation separately, but may be able to do better
- Read Chapter 18 for more examples

The Disjoint-Set Data Type

A *disjoint-set* data type consists of a collection of *disjoint* sets $S_1, \ldots, S_k$.

- each set is represented by one of its elements
- the exact element depends on the representation
    - x_S is the representative element of set S
    - S_x is the set containing x

Operations on this data type:

- Make-Set(x): creates a set $\{x\}$
    - not a set with a pointer to x (typo in book)
    - x can't be in any of the other sets
- Union($x_S, x_{S'}$): replace S and S' by $S \cup S'$
- Find(x): returns x_S, if $x \in S$
    - Text calls it Find-Set

Text has a different Union:

- Union$'(x, y)$: replace S_x and S_y by $S_x \cup S_y$
    - Union$'(x, y)$ = Union(Find(x),Find(y))

An application: connected components

The disjoint-set data type turns out to be very useful in graph algorithms.

One application:

- finding connected components of an undirected graph.
- testing if two vertices are in the same connected component.

Recall a graph $G = (V, E)$

- V = vertices; E = edges
- an edge $e = (v, v')$

CONNECTED-COMPONENT(V, E)

for each vertex $v \in V$
    do MAKE-SET(v)
for each edge $(u, v) \in E$
    do if FIND(u) $\neq$ FIND(v)
        then UNION(FIND(u),FIND(v))

Complexity:

- $|V|$ MAKE-SETS
- $2|E|$ FINDS
- $\leq |E|$ UNIONS

SAME-COMPONENT(u, v)

if FIND(u) $=$ FIND(v)
    then return TRUE
    else return FALSE

Complexity: 2 FINDS

UNION/FIND also useful in finding minimum spanning tree

Quick-Find

Typical implementation of UNION/FIND:

- Assume $S_1 \cup \ldots \cup S_k \subseteq \{1, \ldots, n\}$

Model sets as doubly-linked lists (with *head* and *tail*)

- $x_S = head[S]$

Keep an array $T[1..n]$ such that $T[x] = head[S_x]$.

With this implementation:

- FIND takes constant time
  - $\text{FIND}(x) = T[x]$
- MAKE-SET takes constant time
  - easy to update S and T
- What about UNION?

UNION($x_S, x_{S'}$) could take $O(n)$:

- Combine linked lists S and S' into one list
  - put S at end of S'
  - Combining doubly-linked lists is $O(1)$
  - Problem: need to fix the array T
    - Must change pointer for the elements in S
    - This could take time $O(|S|)$

Sequence of K MAKE-SETS + M FINDS + N UNIONS takes time $O(K + M + N^2)$.

- note $N < K$

It's not too hard to find a sequence of n operations that takes time $O(n^2)$:

- make $n/2$ sets: $\{x_1\}, \ldots, \{x_{n/2}\}$
- UNION(1,2), UNION(2,3), ..., UNION($n/2-1$,$n/2$)
- After j unions, have $\{1, \ldots, j\}$ in S_j
- Require $1+\cdots+(n/2-1) = O(n^2)$ pointer changes.

An improvement

Keep track of $|S|$

- easy to do – initially 1, $|S \cup S'| = |S| + |S'|$

For $S \cup S'$, put smaller list at end

- this minimizes the number of updates to T

Union(S, S') takes time $O(\min(|S|, |S'|)$

A sequence of K Make-Sets + M Finds + N Unions takes time $O(K + M + N \lg N)$.

Proof: After j Unions, biggest $N + 1 - j$ sets have total size $\leq N + 1$. (Proof is by induction on j.)

- After N Unions, biggest set has size $\leq N + 1$

If an element switches from S to S' after Union (i.e., we put S after S') it's because $|S'| \geq |S|$

- Thus $|S' \cup S| \geq 2|S|$
- An element can switch $\leq \lg(N + 1)$ times

Can achieve $O(N \lg N)$:

- make $n/2$ sets then
- Union(1,2), Union(3,4), ... Union($n/2-1, n/2$)
  Union(2,4), Union(6,8), ...
  Union(4,8), Union(12,16), ...

Quick-Union

A different approach that does better with union: Each set S is represented by a tree (not a linked list)

- the representative element of S is $root[S]$
- for each node x, have $p[x]$ (parent of x)
  - have an array $P[1..n]$, where $P[x] = p[x]$
  - don't have pointers to children
  - for the root, have $p[x] = x$ ($p[x] = \text{NIL}$ OK too)

With this implementation:

- MAKE-SET takes constant time
- UNION($x_S, x_{S'}$) takes constant time
  - have $root[S']$ be the parent of $root[S]$
  - This gives one tree whose nodes are $S \cup S'$
  - These are not necessarily binary trees!
- What about FIND?

FIND(x) returns the root of the tree that contains x

- This takes time $O(depth(x))$
  - depth(x) = length of path from root to x

A sequence of K MAKE-SETS + M FINDS + N UNIONS takes time $O(K + M^2 + N)$.

It's not too hard to find a sequence of n operations that takes time $O(n^2)$:

- make $n/3$ sets: $\{x_1\}, \ldots, \{x_{n/3}\}$
- UNION(1,2), UNION(2,3), ..., UNION($n/3-1$,$n/3$)
- After j unions, have $\{1, \ldots, j\}$ in S_j, organized as a tree with one path.
- FIND(1), ..., FIND($n/3$) takes time $O(n^2)$.

Improving Quick-Union

Two heuristics for improving QUICK-UNION:

- when taking the union, make the root of the tree with more nodes (actually, of greater *rank*) the parent of the other root
  - rank $\geq$ length of longest path from the root to a leaf
  - easy to maintain $rank[x]$ for each node x
  - this guarantees the depth is at most $\lg N$
- *path compression*
  - when we do a FIND(x), change the parent of x to the root
  - in the process, do the same for every node on the path from x to the root
    - little overhead, since we need to visit these nodes anyway
    - this will amortize the work of changing the pointers

Improved Union-Find**: Pseudocode**

```
Make-Set(x)
p[x] ← x
rank[x] = 0
```

```
Union(x_S, x_S')
if rank[x_S] > rank[x_S']
   then p[x_S'] ← x_S
   else p[x_S] ← x_S'
        if rank[x_S] = rank[x_S']
           then rank[x_S'] = rank[x_S'] + 1
```

```
Find(x)
if x ≠ p[x]
   then p[x] ← Find(p[x])
return p[x]
```

Find(x) sets the parent of x to the root, returns the root, and recursively calls Find($p[x]$)

Analysis of Union/Find

Define

$$F(0) = 1$$
$$F(i+1) = 2^{F(i)} \text{ for } i \geq 0$$

Have

$F(1) = 2$
$F(2) = 2^{F(1)} = 4$
$F(3) = 2^{F(2)} = 2^4 = 16$
$F(4) = 2^{F(3)} = 2^{16} = 65,536$
$F(5) = 2^{F(4)} = 2^{65,536} =$ a very big number

$\lg^*(n) =$ least k such that $n \leq F(k)$
$\lg^*(n) \leq 5$ if $n \leq 2^{65,536}$

Theorem: A sequence of K Make-Sets + M Finds + N Unions takes time $O((K+M)\lg^*(K)+N)$.

Bottom line: Amortized cost of each operation is essentially constant!

The next four slides cover the proof of the theorem.

- You're not responsible for it, although you may find it interesting

Suppose we are given a sequence σ of K MAKE-SET, M FIND, and N UNION instructions. Let σ' the sequence with all the FINDs deleted.

- there is no path compression in σ'

Fact 1: After performing σ', a node of rank r has $\geq 2^r$ descendants (including itself).

Proof: Easy argument by induction. The rank of a node increases only when it acquires all the children of another node of equal rank as its children.

Fact 2: After performing σ, there are at most $K/2^r$ nodes of rank r.

Proof: First consider σ'. The rank of a node is $>$ than the rank of its children.

- Subtrees of two nodes of rank r must be disjoint
- Each subtree has 2^r nodes, so at most $K/2^r$

Performing FIND doesn't affect the rank, so the result is also true for σ.

Fact 3: The highest rank is $\leq \lg K$.

Fact 4: After performing σ, the rank of a node is $>$ than the rank of its children.

Proof: Obvious for σ'. Path compression doesn't change this fact.

The cost of $\textsc{Find}(x)$ is the number of nodes on the path from x to the root.

- if we perform $\textsc{Find}(x)$ again, the cost is 1

How do we keep track of the changing costs?

- Need some accounting gimmmicks
  - each time we visit a node during a $\textsc{Find}$, we charge either a Canadian or an American penny
  - At the end, the total number of pennies is the total running time of the $\textsc{Finds}$

Partition the ranks into *groups*:

- Group g consists of all nodes of rank $F(g-1)+1$ to $F(g)$; group 0 consists of nodes of rank 1.
- Since the highest rank is $\lg K$, there are at most $\lg^*(\lg K) + 1 = \lg^*(K)$ groups.

Fancy accounting for $\textsc{find}(x)$

- If x or x's parent is the root, or x's parent is in a different group from x, charge x one Canadian penny
- Otherwise, charge x one American penny.

Fact 5: After σ, we have been charged at most $M(2 + \lg^* K)$ Canadian pennies.

Proof: For any FIND, as we go up the path, we charge 2 for the root and the child of the root, + 1 for each time we change groups. There are $\leq \lg^* K$ groups. Thus, charge $\leq 2 + \lg^* K$ Canadian pennies for each of M FINDs.

Fact 6: If x is in group g, then at most $F(g)$ American pennies are put at node x.

Proof: Each time we charge x an American penny, we do path compression, and x gets a parent of higher rank. After $F(g)$ compressions, x's parent must be in a different group, and we don't charge American pennies any more.

Fact 7: There are at most $N(g) = K/2^{F(g-1)}$ nodes in group g.

Proof: There are $\leq N/2^r$ nodes of rank r. Therefore

$$\begin{aligned} N(g) &\leq \Sigma_{r=F(g-1)+1}^{F(g)} N/2^r \\ &\leq N\, \Sigma_{r=F(g-1)+1}^{\infty} 1/2^r \\ &= \frac{2N}{2^{F(g-1)+1}} \\ &= \frac{N}{2^{F(g-1)}} \end{aligned}$$

Fact 8: At most $KF(g)/2^{F(g-1)} = K$ American pennies are charged at nodes in group g.

Fact 9: At most $K \lg^* K$ American pennies are charged altogether.

Fact 10: At most $(K + M) \lg^* K + 2M$ pennies are charged altogether.

Thus, the total cost of M FINDs (after K MAKE-SETs) is $(K + M) \lg^* K$.