Lists and SkipList

Alon Efrat
Computer Science Department
University of Arizona

A (singly connected) link list

- Set of cells in memory. Each cell contains a key, and a pointer to the next cell.
- A pointer is the address of the next cell in memory. (in java, it is the reference)
- There is a variable (head) storing the address of the first cell
- We could think about the memory as a large array, so a possible interpretation might look like the example below:

```
Cell address: 102 104 106 108 110 112 114 116 118
Key: D A B C
Next cell: 0 null 110 118
```

- Constant time to move from a cell to the next cell
- No efficient way to move to the previous cell, or to find a key. Require linear scan.

A (doubly connected) link list

- Set of cells in memory. Each cell contains a key, and a pointer to the next cell and a pointer to the previous cell (prev)
- A pointer is the address of the next cell in memory. (in java, it is the reference)
- There is a variable (head) storing the address of the first cell
- The last element points to NULL.
- We could think about the memory as a large array, so a possible interpretation might look like the example below:

```
Cell address: 102 104 106 108 110 112 114 116 118
Key: D A B C
Next cell: 0 null 110 118
Prev cell: 118 0 null 104
```

- Constant time to move from a cell to the next cell or to the previous cell
- No efficient way to find a key. Require linear scan.

Searching a key x in a sorted linked list

```
head → ∞ → 7 → 14 → 21 → 32 → 37 → 71 → 85 → 117 → ∞
```

1. cell *p =head;
2. while (p→key < x) p=p→next;
3. return p; // (which is either equal or larger than x)

Note:
- The ∞ and ∞ elements are not “real” keys.
- They are in the list to prevent checking special cases
- Sometimes we prefer to return the element proceeding the one containing x. Then line 2 is replaced with
- while (p→next→key < x) p=p→next

Inserting a key into a Sorted linked list

To insert 35 -

1. p= find(35); // find the proceeding element – the next one is > 35
2. CELL *p1 = (CELL *) malloc(sizeof(CELL));
3. p1→key=35;
4. p1→next = p→next;
5. p→next = p1;
**Deleting a key from a sorted list**

To delete 37 -
- `p=find(37);` // Again find proceeding element
- `CELL *p1 = p→next;`
- `p→next = p1→next;`
- `free(p1);`

**Rules:**
- Consists of several levels.
- All keys appear in level 1.
- Each level is a sorted list.
- If key `x` appears in level `i`, then it also appears in all levels below level `i`.
- First element in each level has key `-∞`.
- Last element has key `+∞`.
- First element in upper level is pointed to by variable `top`.

**More rules**
- An element in level `i > 1` points (via down pointer) to the element with the same key in the level below.
- Elements in the lowest level have `down-pointer=NULL`.
- Also maintain a counter specifying the number of levels.

**Finding an element with key `x`**
- `p=top;`
- `while(1){`
  - `while (p→next→key ≤ x) p=p→next;`
  - `if (p→down == NULL) return p;`
  - `p=p→down;`
- `}`

If the key `x` is in SL, we return a pointer to the lowest element contain x.
If `x` is not in SL, return pointer to lowest predecessor.

**A “perfect” SkipList**

A SL is Perfect if between every two consecutive keys of level `i` there is exactly one key of level `i-1`.

**Scheme for creation a well-performing SL**
- Start from Level 1 (lowers level)
- For `i=2,3,...`
  - Generation of Level `i`: we scan the keys in level `i-1`.
  - Each second key is “promoted” to participate in level `i` as well.

Most SL are not perfect.
Hard to maintain.
**Search in a “perfect” SkipList**

Another example

```c
p=top;
while(1){
    while (p\rightarrow next\rightarrow key \leq x )
        p=p\rightarrow next;
    if (p\rightarrow down == NULL ) return p
    p=p\rightarrow down;
}
```

**Inserting new element x**

(the resulting SL will not be perfect)

- Determine \( k \geq 1 \) defined as the number of levels in which \( x \) participates (explained later how).
- Perform `find(x)`, but once the search path is in one of the lowest \( k \) levels:
  - \( x \) is inserted after the elements at which the search path branches down or terminates.
  - The `next-pointer` behave like a “standard” linked list
  - The `down pointer(s)` point between themselves.

**Example - inserting 119. \( k=2 \)**

**Inserting an element - cont.**

- If \( k \) is larger than the current number of levels, add new levels (and update \( top \), and \( num\_of\_levels \) counter)
- Example - insert(119) when \( k=4 \)
- Heuristic: Add at most one new level (not needed for the analysis)

**Determining k**

- \( k \) - the number of levels at which an element \( x \) participate.
- Use a random function `OurRnd()` --- returns 1 or 0 (True/False) with equal probability.
  - \( k=1 \);
  - While ( \( OurRnd()==1 \) ) \( k++ ; \)

**Deleteing a key x**

- Find \( x \) in all the levels it participates, using `find(x)`.
- During the “find”, delete \( x \) from each level it participates using the standard “delete from a linked list” method.
- If one or more of the upper levels become empty, remove them (and update \( top \) and \( num\_of\_levels \))

**“expected” space requirement**

- **Claim**: The expected number of elements is \( O(n) \).
- The term “expected” here refers to the experiments we do while tossing the coin (or calling `OurRnd()`). No assumption about input distribution.
- So imagine a given set, given set of operations insert/del/find, but we repeat many time the experiments of constructing the SL, and count the #elements.
Facts about SL

- **Def**: The height of the SL is the number of levels
- **Claim**: The expected number of levels is \( O(\log n) \)
- (here \( n \) is the number of keys)
- **“= Proof”** (A rigorous proof coming later)
  - The number of elements participate in the lowest level is \( n \).
  - Since the probability of an element to participates in level 2 is \( \frac{1}{2} \), the expected number of elements in level 2 is \( n/2 \).
  - Since the probability of an element to participates in level 3 is \( 1/4 \), the expected number of elements in level 3 is \( n/4 \).
  - ...
  - The probability of an element to participate in level \( j \) is \( (1/2)^{j-1} \) so number of elements in this level is \( n/2^{j-1} \).
  - So after \( \log(n) \) levels, no element is left.

More facts

- **Thm**: The expected time for find/insert/delete is \( O(\log n) \)
- **Proof** For all Insert and Delete, the time is \( \leq \) expected #elements scanned during find(\( x \)) operation.
- Will show: Need to scan expected \( O(\log n) \) elements.

Bounding time for insert/delete/find

- Putting it together: The expected number of elements scanned in each level is \( O(1) \)
- There are \( O(\log n) \) levels
- Total time is \( O(\log n) \)
- As stated, getting bounds for time for insert/delete are similar

How likely is it to see a `too-tall` SL?

- We will prove a bound on the height. Similar bounds could be proven for similar properties.
- The question what is `too-tall` is up to the user.
- Of course, the larger \( n \) is, the more level we expect to see. So lets ask the user to pick a value \( Z \).
- We will compute the how likely is it that the the number of levels is is at least \( Z \log n \), where \( Z=1,2,3... \)
- That is, we estimate the probability that the height of the SL is
  - \( \log n \)
  - \( 2 \log n \)
  - \( 3 \log n \)
  - \( 4 \log n \)
  - ...

To reduce the worst case scenario, we verify during insertion that \( k \) (the number of levels that an element participates) in) is \( \leq \log n \)

“Conclusion”: The expected storage is \( O(n) \)
Example: In a roulette, the result is a number

Similarly, for 3 Events $A_1$, $A_2$, $A_3$. The probability that at least one of them happens

Example: In a roulette, the result is a number $k$ between 1..38

Event $A$: $k$ is even.

Event $B$: $k$ is divided by 3.

Pr($A$ or $B$) = Pr($A$) + Pr($B$) - Pr($A$) \cap Pr($B$)

Assume that $A$, $B$ are two events. Let

- Pr($A$) be the probability that $A$ happens,
- Pr($B$) be the probability that $B$ happens,
- Pr($A$ \cap $B$) is the probability that either event $A$ happens or event $B$ happens (or both).

So probably that at least one of them happened is

Pr($A$) + Pr($B$) - Pr($A$) \cap Pr($B$)

Similarly, for 3 Events $A_1$, $A_2$, $A_3$. The probability that at least one of them happens

Pr($A_1$ \cup $A_2$ \cup $A_3$) \leq Pr($A_1$) + Pr($A_2$) + Pr($A_3$)

But how likely is that the SL is too tall?

Assume the keys in the SL are \{x_1, x_2, ..., x_n\}

* The probability that x_i participates in $\geq k+1$ levels is $2^{-k}$.
  * (same probability for all x_i).

Define $A_i$ is the event that x_i participates in $\geq k+1$ levels.

Pr($A_i$) = $2^{-k}$.

Define $A_i$ is the event that x_i participates in $\geq k+1$ levels.

Pr($A_i$) = $2^{-k}$ (for every j)

If the height of SL $\geq k+1$ then

- at least one of the x_i participates in $\geq k+1$ levels.
- The probability that any x_i (one or more) participates in $\geq k+1$ levels is $\leq$ Pr($A_1$) + Pr($A_2$) + ... + Pr($A_n$) = $n \cdot 2^{-k}$

This is the probability that the height of the SL is $\geq k+1$.

So how likely is it that the height of SL is $> Z \log n$?

The probability that any x_i participates in $> k$ levels is $\leq n \cdot 2^{-k}$

If none of the x_i’s is at level $\geq k$ then the height is $\leq k$.

Recall $y^{ab} = (y^a)^b = (y^b)^a$

Write $k = Z \log_2 n$, and $2^{\log_2 n} = (n)^5$

Want to find: The probability that the height is Z times log_2 n.

That is, Twice log_2 n, 3 time log_2 n, 4 times log_2 n ...

But how likely is that the SL is tall?

The probability that any x_i participates in at least k levels is $\leq n \cdot 2^{-k}$

Ignore the $' +1'$

If none of the x_i’s is at level $\geq k$ then the height is $\leq k$.

Write $k = Z \log_2 n$, and $2^{\log_2 n} = (n)^5$