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
- The last element points to NULL.
- We could think about the memory as a large array, so a possible interpretation might looks like the example below:

```
head
A → B → C → D → NULL
```

Memory Snapshot:

<table>
<thead>
<tr>
<th>Cell address</th>
<th>102</th>
<th>104</th>
<th>106</th>
<th>108</th>
<th>110</th>
<th>112</th>
<th>114</th>
<th>116</th>
<th>118</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Next cell</td>
<td>0 null</td>
<td>110</td>
<td>118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
</tbody>
</table>
A (singly connected) linked 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
- The last element points to NULL.
- We could think about the memory as a large array, so a possible interpretation might looks like the example below:

```
B
C
D
A
NULL
```

<table>
<thead>
<tr>
<th>Cell address</th>
<th>102</th>
<th>104</th>
<th>106</th>
<th>108</th>
<th>110</th>
<th>112</th>
<th>114</th>
<th>116</th>
<th>118</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>D</td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Next cell</td>
<td>0</td>
<td>null</td>
<td>110</td>
<td>118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>104</td>
</tr>
</tbody>
</table>

- 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 looks like the example below:

<table>
<thead>
<tr>
<th>Cell address</th>
<th>102</th>
<th>104</th>
<th>106</th>
<th>108</th>
<th>110</th>
<th>112</th>
<th>114</th>
<th>116</th>
<th>118</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>D</td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Next cell</td>
<td>0 null</td>
<td>110</td>
<td></td>
<td>118</td>
<td></td>
<td></td>
<td></td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>Prev cell</td>
<td>118</td>
<td>0 null</td>
<td>106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

- Constant time to move from a cell to the next cell or to the previous cell
- No efficient to find a key. Require linear scan.
Searching a key \( x \) in a sorted linked list

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

Note:

- The \( -\infty \) and \( \infty \) 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 (\( p \rightarrow \text{next} \rightarrow \text{key} < x \))  \( p = p \rightarrow \text{next} \)
To insert 35 -

- p = find(35); // find the proceeding element – the next one is > 35
- CELL *p1 = (CELL *) malloc(sizeof(CELL));
- p1->key = 35;
- p1->next = p->next;
- p->next = p1;
To insert 35 -

- \( p = \text{find}(35); \) // find the proceeding element – the next one is > 35
- \( \text{CELL} \,*p1 = (\text{CELL} *) \text{malloc(sizeof(CELL))}; \)
- \( p1 \rightarrow \text{key} = 35; \)
- \( p1 \rightarrow \text{next} = p \rightarrow \text{next}; \)
- \( p \rightarrow \text{next} = p1; \)
To insert 35 -

- p = find(35); // find the proceeding element – the next one is > 35
- CELL *p1 = (CELL *) malloc(sizeof(CELL));
- p1→key=35;
- p1→next = p→next ;
- p→next = p1 ;
inserting a key into a Sorted linked list

To insert 35 -

- \( p = \text{find}(35); \) // find the proceeding element – the next one is > 35
- \( \text{CELL} \ *p1 = (\text{CELL} \ *) \text{malloc}() \text{sizeof}()\text{CELL}(); \)
- \( p1 \rightarrow \text{key} = 35; \)
- \( p1 \rightarrow \text{next} = p \rightarrow \text{next} ; \)
- \( p \rightarrow \text{next} = p1 ; \)
To insert 35 -

- \( p = \text{find}(35); \) // find the proceeding element – the next one is > 35
- \( \text{CELL } *p1 = (\text{CELL } *) \text{malloc}(<\text{sizeof}(\text{CELL})>); \)
- \( p1 \rightarrow \text{key}=35; \)
- \( p1 \rightarrow \text{next} = p \rightarrow \text{next} ; \)
- \( p \rightarrow \text{next} = p1 ; \)
To insert 35 -

- p = find(35); // find the proceeding element – the next one is > 35
- CELL *p1 = (CELL *) malloc(sizeof(CELL));
- p1→key=35;
- p1→next = p→next ;
- p→next = p1 ;
To delete 37 -
- \( p = \text{find}(37); \) // Again find proceeding element
- \( \text{CELL } *p1 = p \rightarrow \text{next}; \)
- \( p \rightarrow \text{next} = p1 \rightarrow \text{next}; \)
- \( \text{free}(p1); \)
To delete 37 -

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

- p = find(37); // Again find proceeding element
- CELL *p1 = p → next;
- p → next = p1 → next;
- free(p1);
SKIP LIST - A data structure for maintaining keys in a sorted order

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 $-\infty$.
- Last element has key $+\infty$.
- First element in upper level is pointed to by variable $top$. 

```
Level 3
   -∞       21   37   ∞
   ↓   ↓   ↓          ↓
Level 2
   -∞   7    21   37    71   85   117   ∞
   ↓   ↓   ↓   ↓   ↓   ↓   ↓   ↓
Level 1
   -∞   7    14   21   32   37    71    85   117   ∞
```

```
- 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 $\text{down-pointer}=\text{NULL}$
- Also maintain a counter specifying the number of levels.
An empty SkipList
Finding an element with key $x$

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

If the key $x$ is in SL, we return a pointer to the lowest element containing $x$. If $x$ is not in SL, return pointer to lowest predecessor.
Finding an element with key $x$

- $p$=top;
- while(1){
  - while ($p$➔next➔key $\leq$ $x$) $p$=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 \( i=2,3 \ldots \) Generation of Level \( i \):
  
  we scan the keys in level \( i-1 \).
  Each second key is “promoted” to participate in level \( i \) as well.

-∞ \( \rightarrow \) \( -∞ \) \( \rightarrow \) \( 7 \) \( \rightarrow \) \( 4 \) \( \rightarrow \) \( ∞ \)

-∞ \( \rightarrow \) \( 7 \) \( \rightarrow \) \( 14 \) \( \rightarrow \) \( 21 \) \( \rightarrow \) \( 32 \) \( \rightarrow \) \( 37 \) \( \rightarrow \) \( 40 \) \( \rightarrow \) \( 71 \) \( \rightarrow \) \( 117 \) \( \rightarrow \) \( ∞ \)

-∞ \( \rightarrow \) \( 21 \) \( \rightarrow \) \( 37 \) \( \rightarrow \) \( 71 \) \( \rightarrow \) \( ∞ \)

-∞ \( \rightarrow \) \( 71 \) \( \rightarrow \) \( ∞ \)

next-pointer

down-pointer
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
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 \(i=2,3\ldots\)
  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->next->key <= x) {
        p = p->next;
    }
    if (p->down == NULL) return p;
    p = p->down;
}
```
Search in a “perfect” SkipList

Another example

```cpp
p = top;
while(1) {
    while (p ➔ next ➔ key ≤ x)
        p = p ➔ next;
    if (p ➔ down == NULL) return p
    p = p ➔ down;
}
```
Search in a “perfect” SkipList

Another example

\[ p = \text{top} ; \]
\[ \text{while}(1) \{ \]
\[ \quad \text{while} \ (p\rightarrow next\rightarrow key \ \leq \ x) \]
\[ \quad \quad p = p\rightarrow next; \]
\[ \quad \text{if} \ (p\rightarrow down == \text{NULL}) \quad \text{return} \ p \]
\[ \quad p = p\rightarrow down ; \]
\[ \} \]
Inserting new element $x$
(the resulting SL will not be perfect)
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)
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 \( \text{find}(x) \), 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.
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 $\text{find}(x)$, 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.

![Diagram showing levels and pointers](image-url)
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 $\text{find}(x)$, 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.
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 $\text{find}(x)$, 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.
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 $\text{find}(x)$, 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. $k=2$
• 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)
- 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 $\text{find}(x)$.
- 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 $\text{top}$ and $\text{num_of_levels}$).
Deleteing a key $x$

- Find $x$ in all the levels it participates, using $\text{find}(x)$.
- 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 $\text{top}$ and $\text{num_of_levels}$).
Deleting a key \( x \)

- Find \( x \) in all the levels it participates, using \( \text{find}(x) \).
- 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 \( \text{top} \) and \( \text{num_of_levels} \) ).
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)
- “\( \approx \) **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 \( \frac{1}{4} \), 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.
Facts about SL

- **Claim**: The expected number of elements is $O(n)$.
- (here $n$ is the number of keys)
- “$\approx$ Proof” (Real proof – later)
  - The total number of elements is
    $$n + n/2 + n/4 + n/8 \ldots \leq n(1 + 1/2 + 1/4 + 1/8 \ldots) = 2n$$

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

“Conclusion”: The expected storage is $O(n)$
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.
**Thm:** Expected time for `find` operation is $O(\log n)$

- **Proof** – we know that there are $O(\log n)$ levels. Will show that we spend $O(1)$ time in each level.
- Assume during find($x$), we scanned $t$ elements, (for $t>8$) in level $r$. Assume first that $r$ is not the upper level.
  - (the search visited $b$, branched down to $b_1$ and then visited $b_2...b_8$
    (not sure what happened before or after)

All smaller than $x$

None of these 7 elements reached level $r+1$ (why?)

The probability that none of these 7 elements reached level $r+1$ is $1/2^7$. For larger value of 7 – very slim.
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 let's ask the user to pick a value \( Z \).
- We will compute the how likely is it that the \( \text{number of levels is} \) is at least \( Z \log_2 n \), where \( Z=1,2,3... \)

That is, we estimate the probability that the height of the SL is

- \( \log_2 n \)
- \( 2 \log_2 n \)
- \( 3 \log_2 n \)
- \( 4 \log_2 n \)
- ...
Reminder from probability

- Assume that \( A, B \) are two events. Let
  - \( \Pr(A) \) be the probability that \( A \) happens,
  - \( \Pr(B) \) be the probability that \( B \) happens
  - \( \Pr(A \cup B) \) is the probability that either event \( A \) happens or event \( B \) happens (or both).
- So probably that at least one of them happened is
  \[
  \Pr(A) + \Pr(B) - \Pr(A \cap B) \leq \Pr(A) + \Pr(B)
  \]
Similarly, for 3 Events \( A_1, A_2, A_3 \). The probability that at least one of them happens
  \[
  \Pr(A_1 \cup A_2 \cup A_3) \leq \Pr(A_1) + \Pr(A_2) + \Pr(A_3)
  \]
Example: In a roulette, the result is a number \( k \) between 1..38
  - Event \( A \): \( k \) is even. \( \Pr(A) = \Pr(k \text{ is even}) = 19/38 = 0.5 \)
  - Event \( B \): \( k \) is divided by 3. \( \Pr(B) = 12/38 = 0.315 \)
  - \( \Pr(A \text{ or } B) = \Pr(A \cup B) = \Pr((k \text{ is divided by 2}) \text{ or } (k \text{ is divided by 3})) \leq 0.5 + 0.315 = 0.815 \)
Pick your favorite number $k$.
What is the probability that the SL has $>k$ levels?

$$\Pr( \text{height of the SkipList } \geq k ) =$$

$$\Pr\left\{ (x_1 \text{ participates in more than } k \text{ levels }) \text{ OR } (x_2 \text{ participates in more than } k \text{ levels }) \text{ OR } (x_3 \text{ participates in more than } k \text{ levels }) \text{ OR } \cdots \right\}$$

/*Apply the principle from the previous slide*/

$$= \Pr(x_1 \text{ participates in more than } k \text{ levels }) + \Pr(x_2 \text{ participates in more than } k \text{ levels }) + \Pr(x_3 \text{ participates in more than } k \text{ levels }) + \cdots$$

$$= \frac{1}{2^k} + \frac{1}{2^k} + \frac{1}{2^k} + \cdots = \frac{n}{2^k}$$
Pick your favorite number $k$. What is the probability that the SL has $>k$ levels?

Answer: $\leq n/2^k$

$\Pr(\text{height of the SkipList } \geq k) =$

$\Pr\{ (x_1 \text{ participates in more than } k \text{ levels} ) \text{ OR }$

$(x_2 \text{ participates in more than } k \text{ levels} ) \text{ OR }$

$(x_3 \text{ participates in more than } k \text{ levels} ) \text{ OR }$

$\cdots$

$(x_n \text{ participates in more than } k \text{ levels} ) \}$

$\leq$

/*Apply the principle from the previous slide*/

$\Pr(x_1 \text{ participates in more than } k \text{ levels}) +$

$\Pr(x_2 \text{ participates in more than } k \text{ levels}) +$

$\Pr(x_3 \text{ participates in more than } k \text{ levels}) +$

$\cdots$

$\Pr(x_n \text{ participates in more than } k \text{ levels}) =$

$1/2^k +$

$1/2^k +$

$1/2^k +$

$\cdots$

$1/2^k = n/2^k$
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 $\leq n/2^k$.
- If none of the $x_i$'s is at level $\geq k$ then the height is $\leq k$.
- Recall $2^{(ab)}=(2^a)^b = (2^b)^a$.
- Write $k = (\log_2 n) Z$.
- Therefore, $2^k = 2^{(\log_2 n) \cdot Z} = (2^{\log_2 n})^Z = n^Z$.

So the probability of seeing a SkipList with more than $Z \log n$ levels is $\leq n/2^k = n/n^Z = 1/n^{z-1}$.

- Let's play with some examples, to see if this is good news or bad news.
- Let's pick $n=1000$.
- The probability that the height $> 7 \log_2 n$ is $\leq 1/1000^6=1/10^{18}$ ... So the probability that the height $\leq 7 \log_2 n$ is $\geq 1-1/10^{18}$.
- The probability that the height $< 10 \log_2 n$ is $\geq 1-1/10^{27}$.

Conclusion: In this case (and in many other randomized algorithms) the probability of success is so high, that practically we can ignore it (higher chance of a lightning strike).