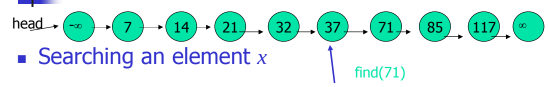
SkipList

Alon Efrat
Computer Science Department
University of Arizona

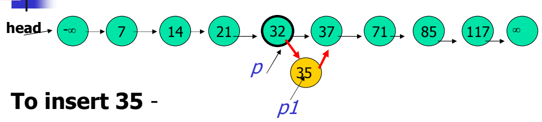
Searching a key in a Sorted linked list



- Searching an element *x*
- cell **p =head ;*
- while *(p->next->key < x)   p=p->next ;*
- return p ;
- Note: we return the element **proceeding** either the element containing *x*, or the largest element with a key smaller than *x* (if *x* does not exists)

2

inserting a key into a Sorted linked list

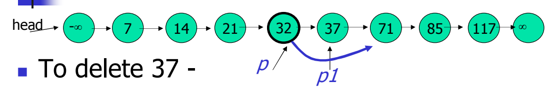


To insert 35 -

- *p*= find*(35);*
- CELL **p1* = (CELL *) malloc(sizeof(CELL));
- *p1->key=35;*
- *p1->next* = *p->next ;*
- *p->next = p1 ;*

3

deleteing a key from a sorted list



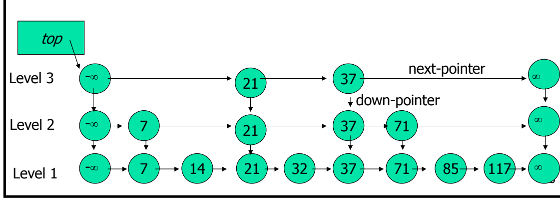
- To delete 37 -
- *p=find(37);*
- CELL **p1 =p->next;*
- *p->next = p1->next ;*
- free(*p1*);

4

SKIP LIST - A data structure for maintaining keys in a sorted order

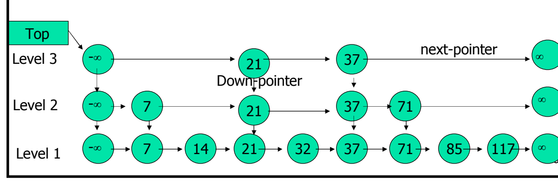
Rules:

- Consists of several **levels.**
- All keys appear in level ∞
- 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

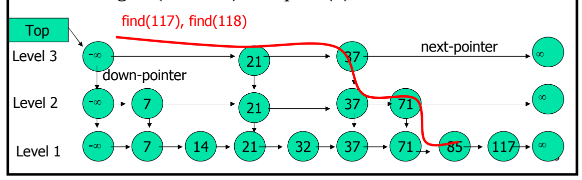
- An element in level ***i >1*** points (via down pointer) to the element with the same key in the level below.
- Elements in the lowest level have ***down-pointer=NULL***
- We also have a counter specifying the number of levels.



An empty SkipList

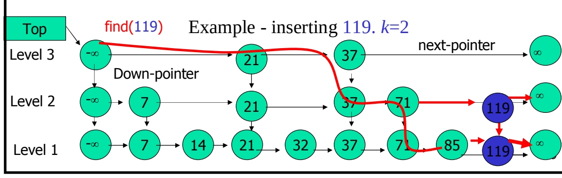
Finding an element with key *x*

- *p=top* ;
- while(1){
  - while (*p->next->key* < ***x***) *p=p->next*;
  - if (*p->down* == *NULL*) **return** *p->next*
  - *p=p->down* ;
- }
- Observe that we return the element in the lowest level containing ***x***, (if exists) , or *pred(**x**)* if ***x*** is not in the SList



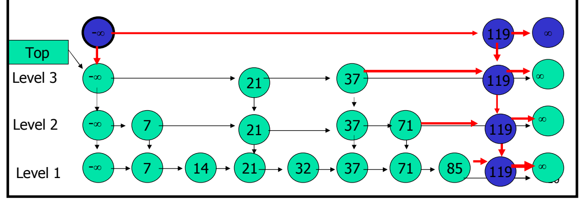
Inserting new element *x*

- Determine ***k***, defined as the number of levels in which ***x*** participates (explained later how)
- Do 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 "standrad" linked list
  - The *down pointer* points between themselves.



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***

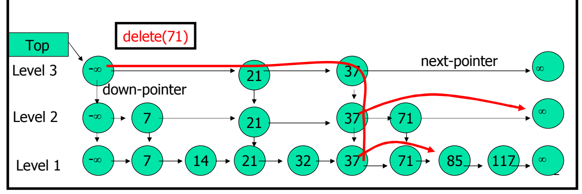


Determining *k*

- ***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())* ***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*)



Facts about SL

- **Claim:** The expected number of levels is O(log *n*)
- (here *n* is the number of keys)
- **"≅ Proof"** (a rigorous proof requires the use of random variables)
  - The number of elements participate in the lowest level is ***n***.
  - Since the probability of an element to participates in level 2 is ½, 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 participates in level *j* is $1/2^{j-1}$ so $n/2^{j-1}$
  - So after log(*n*) levels, no element is left.

Facts about SL

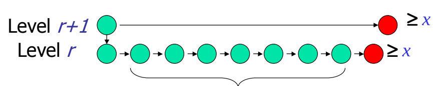
- **Claim:** The expected number of elements is O(*n*).
- (here *n* is the number of keys)
- **"≅ Proof"** (a rigorous proof requires the use of random variables)
  - The total number of elements is

    $n+n/2+n/4+n/8\ldots \le 2n$

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

Facts about SL

- **Thm**: The expected number of elements scanned by a find operation is O(log *n*)
- **≅Proof** – we know that there are *O(* log *n)* levels. Will show – 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.



None of these 7 elements reached level *r+1*

The probability that none of these 7 elements reached level *r+1* is $1/2^t$. For larger value of 7 – very slim.

Facts about SL

- **Thm**: The expected time for find/insert/delete is O(log *n*)
- **Proof** For all 3 operations, the time is bounded by the number of elements need to be scan during find(*x*) operation, which is O(log *n*)