Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- Data structures
  - Skip lists
    - Definition
    - Search
    - Insertion/Deletion
    - **Randomized data structure**

LIST

Introduction and motivations

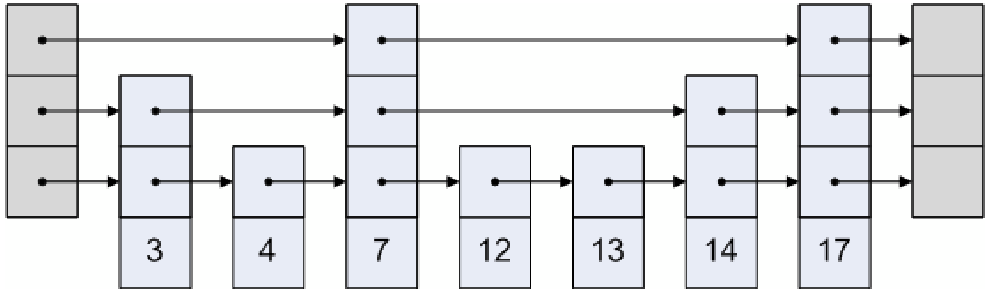
- Array: Static
  - Good for a direct access of the elements
- Linked Lists: Dynamic
  - Good for insertion, deletion of elements
  - Problem:
    - Search:
      - It depends on where we are in the list
      - Direct access: limited to Next element, or Next/Previous element
    - Do we need to visit all the elements in a list to find one?
      - We can skip some elements

Introduction and motivations

- Linked lists
  - Benefits:
    - Easy to insert & delete in O(1) time
      - No need to estimate total memory needed
  - Drawbacks:
    - Difficult to search in less than O(n) time
      - Cannot use binary search
      - Hard to *jump* to the middle

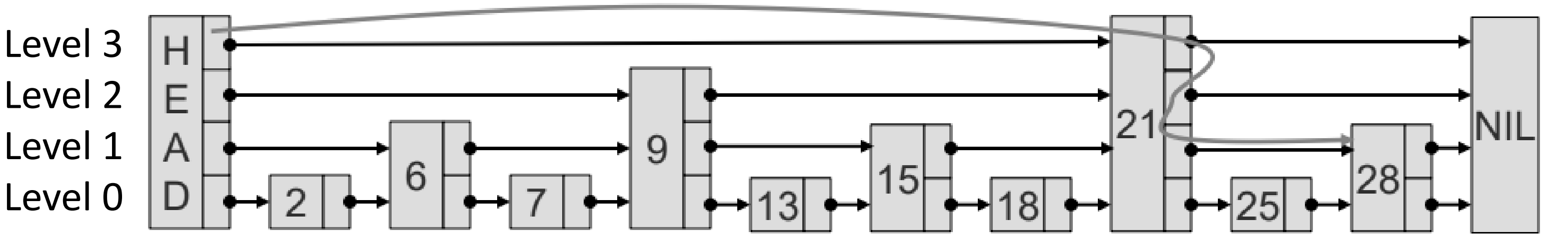
Skip list

- Generalization of linked list
  - Bill Pugh (1990)
- Key features
  - Efficient (with high probability)
    - Expected search time is O(log n)
  - Randomized
    - use random coin flips to build the data structure
  - Easy to implement



Skip list

- Principle
  - Start with a sorted linked list
    - Add another layer linking every other element
    - Repeat for that layer, …
  - Hierarchy of sorted linked lists
    - Base: all the elements are connected
  - Skip list = sorted linked list with shortcuts
    - Example: access 28

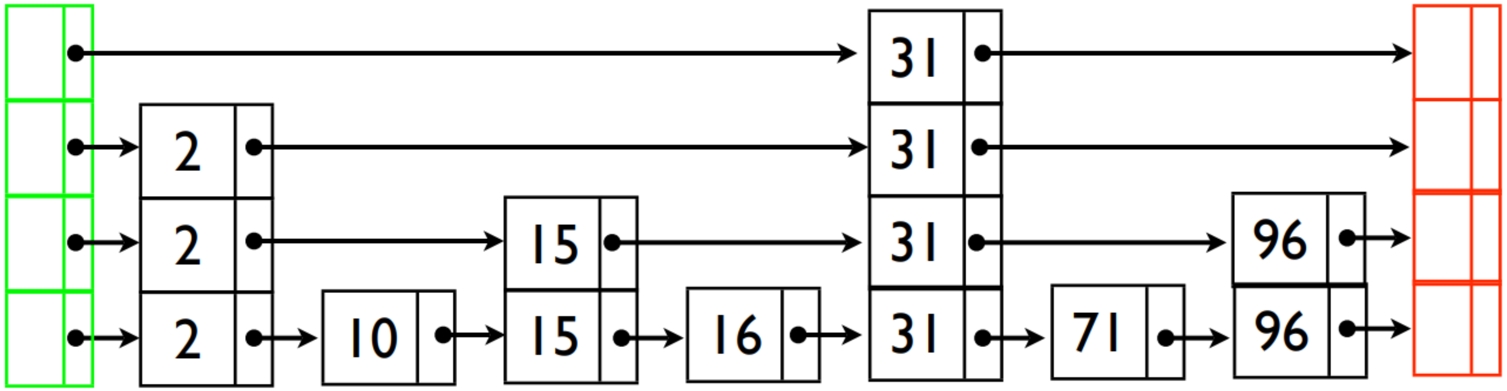


Perfect skip list

- We started with a normal linked list (level 0)
- Then
  - ➢every other node in level 0 (2^{nd} node from original list) and added them to level 1
  - ➢every other node in level 1 (4^{th} node from the original list) and raised it to level 2
  - ➢every other node in level 2 (8^{th} node from the original list) and raised it to level 3
- → $O(\log_2(n))$ levels
- Why
  - ➢At each level, we visit at most 2 nodes
    - At any node, x, in level i, you sit between two nodes (p,q) at level i+1 and you will need to visit at most 1 other node in level i before descending
  - ➢There are O(log(n)) levels
    - →visit at most O(2*log(n)) levels = O(log(n))

Skip list

- Example
  - A “perfect” skip list



head

Dummy header

tail

Terminal sentinel

Skip list

- Sorted skip list
  - Keys in sorted order.
    - O(log n) levels
  - Each higher level contains half the elements of the level below it.
  - Head and Tail nodes are in every level!
  - Nodes have variable sizes:
    - Data + between 1 and O(log n) pointers
    - Pointers point to the start of each node
- Skip lists
  - It is because higher level lists let you **skip** over many items

Search

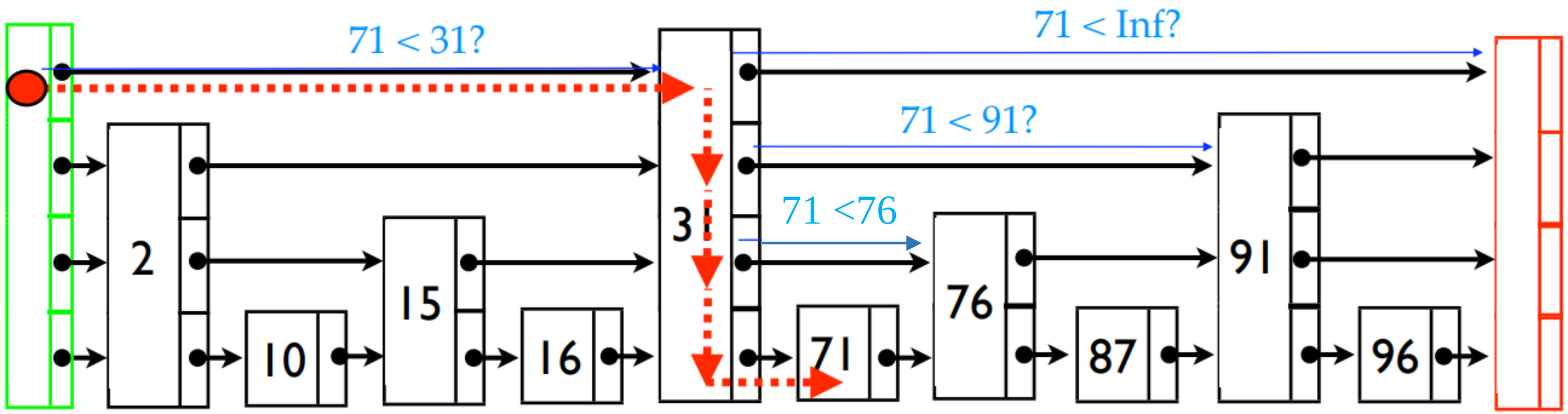
- To search for an item
  - scan along the shortest list **until** passing the desired item.
  - then drop down to a slightly more complete list at **one level lower**.
  - Finally, do a sorted sequential searching

Search

- Example
  - ➢ Target = 71
    - ○ Comparisons
    - ○ Skip lists

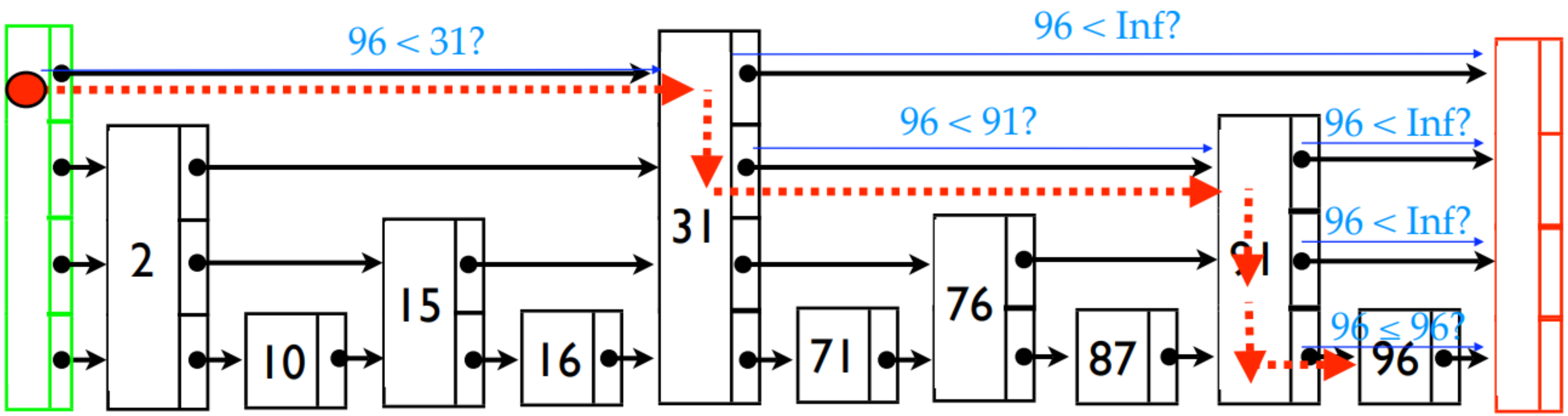
Pseudo code: search(x)

If x == key
    done
else If k < next key
    go down a level
If k ≥ next key
    go right in the current list



Search

- Example:
  - ➢Target = 96



Search

- Remarks and analysis
  - O(log n) levels
    - As we cut the # items by 2 at each level
  - Visit at most 2 nodes per level:
    - Target + Comparison to go down a level
  - If more visits then you could have done it on 1 level higher up.
    - → search time = O(log n).

Perfect vs. Random Skip list

- Perfect
  - ➢ What happens when you add or delete elements?
    - To maintain perfect balance
- Random skip list
  - ➢ Need to decide when to promote a node to some *i* level
  - ➢ Use **randomization**, with a chance of 50%, to decide when to promote a node

Insert and Delete

- Need to rearrange the entire data structure
  - Perfect Skip Lists
    - They are too structured to support efficient updates.
- Principle:
  - Relax the requirement that each level have **exactly half** the items of the previous level
- Instead:
  - We design the structure so that we expect 1/2 the items to be carried up to the next level
- Because Skip Lists are a **randomized** data structure
  - /!\ The same sequence of inserts / deletes may produce different structures!!
    - It depends on the outcome of random coin flips.

Randomization

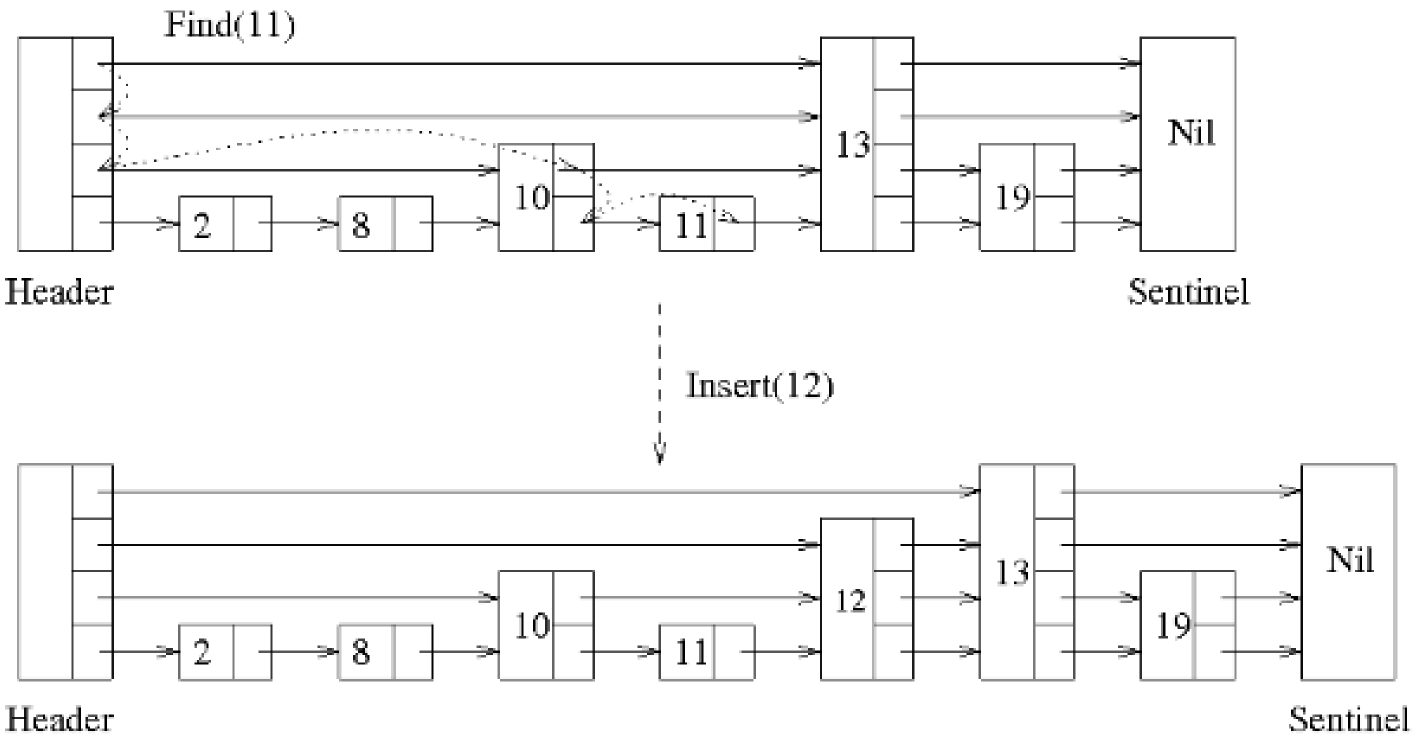
- Allows for some **imbalance**
  - Notion that we will retrieve in the trees ! ☺
- Expected behavior (over the random choices)
  - **same** as with perfect skip lists.
- Principle:
  - Each node is promoted to the next higher level with probability 1/2
  - Expect 1/2 the nodes at level 1
  - Expect 1/4 the nodes at level 2
  - Expect 1/2^i the nodes at level i
- → expect # of nodes at each level is the same as with perfect skip lists.
  - In addition, expect the promoted nodes will be well distributed **across the list**

Randomization

- As nodes are inserted they are repeating trials of probability **p** (stopping when the first unsuccessful outcome occurs)
  - → it means we will not have an "every other" node promotion scheme
  - but the expected number of nodes at each level matches the non-randomized version
- Warning:
  - This scheme introduces the chance of some very high levels!
    - → usually cap the number of levels at some MAXIMUM value
    - However the expected number of levels is still log2 (n)

Example

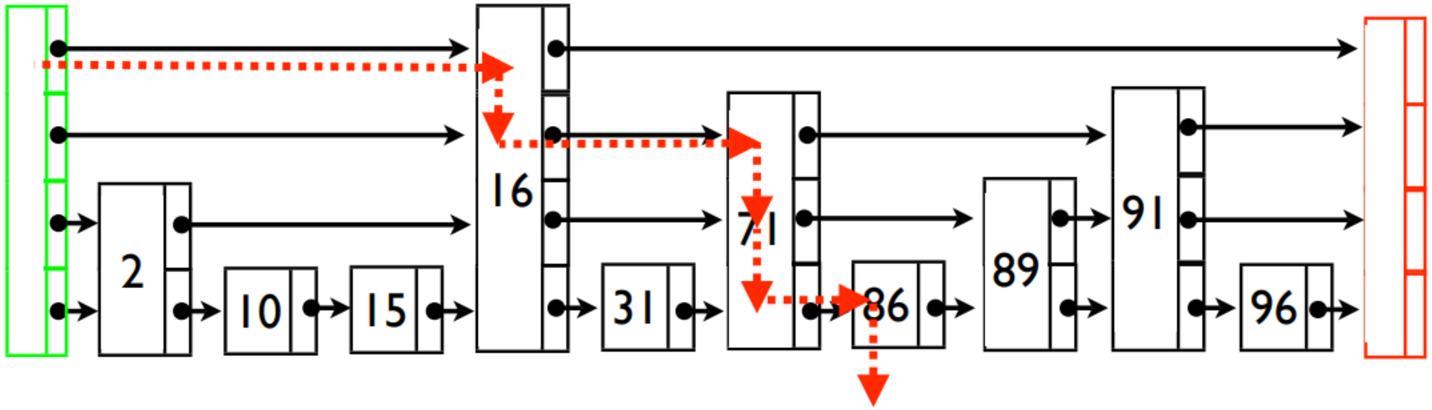
- .

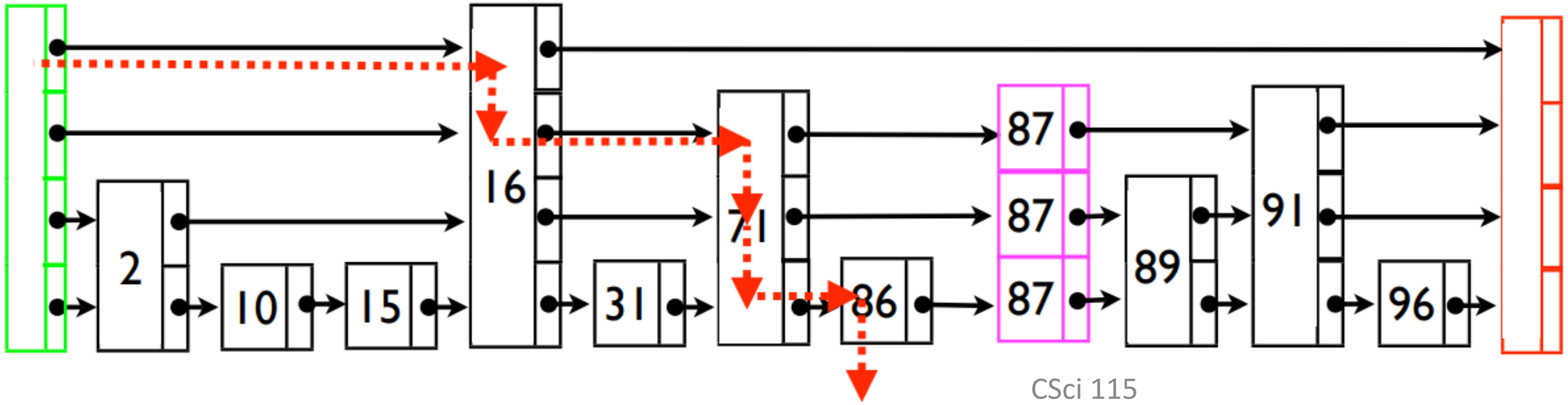


How we decide the level of 12 → Randomization

Insertion

- Randomized skip list
  - Example: Insert 87

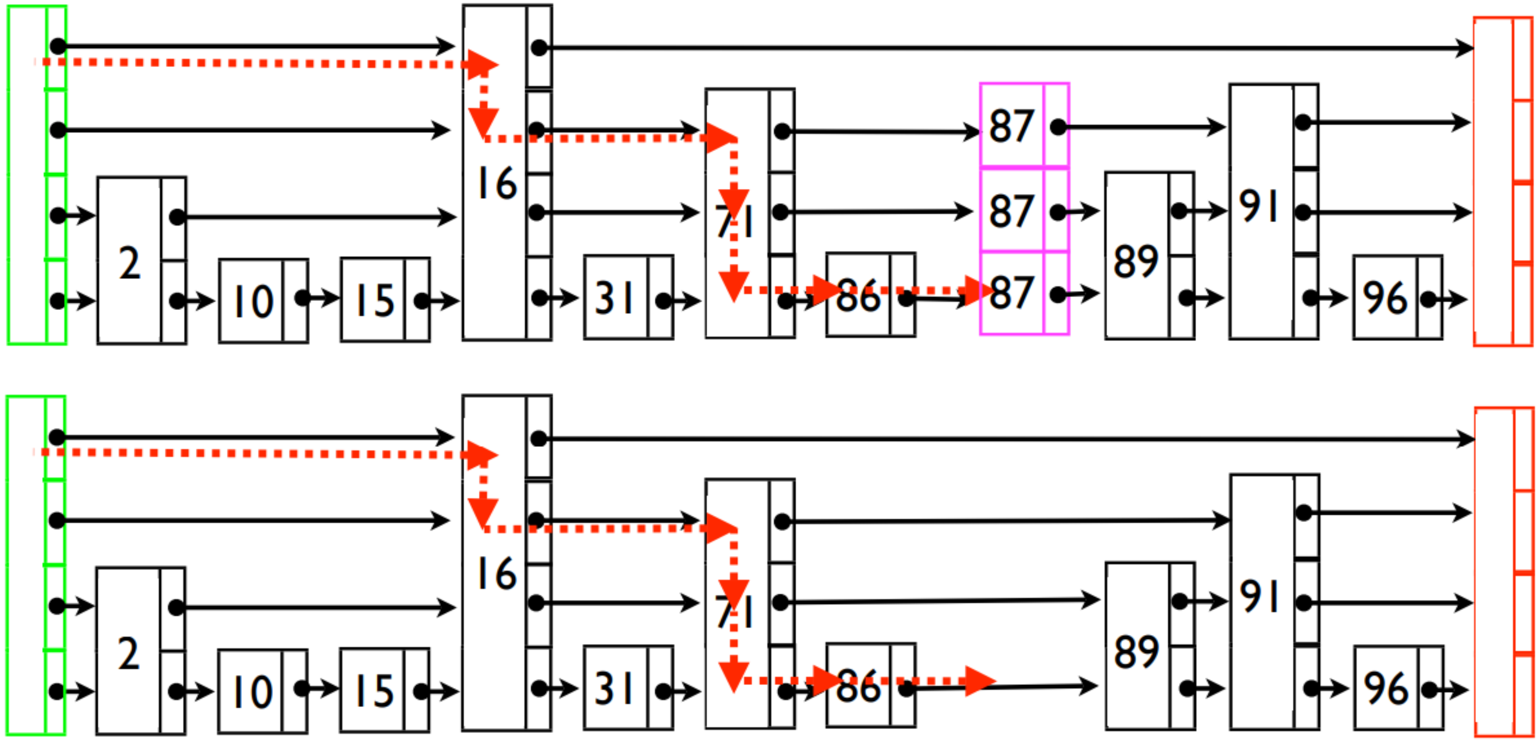




Pseudo code:
Find k Insert node in level 0
i = 1
while (FLIPCOIN() == "heads") {
    insert node into level i
    i++
}

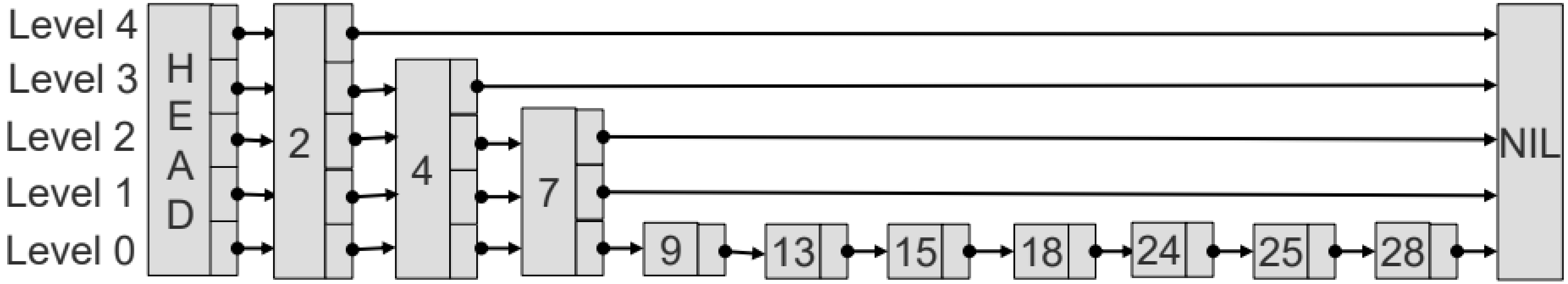
Delete

- Randomized skip list
  - Example: Remove 87



Worst case

- A worst case skip list
  - All the same height
  - Just ascending or descending order of height
- But highly unlikely possibilities
  - the skip list will just be a linked list or
  - the skip list will have every node at every level



Search time

- Search time with the randomized approach
  - Start at the node and walk backwards to the head node counting our expected number of steps
  - if we can move up a level we do so that we take the "faster" path and only move left if we can't move up
- Backward analysis
  - 2 options
    - Probability of **Case A**: p
      - we added each level independently with probability p
    - Probability of **Case B**: 1-p
    - the top level at level 0
    - the current level where we found our search node = level k
      - expected max k = log2 (n)

Search time

- Define a recurrence relationship of the cost of walking back to level 0
  - Base case:
    - $C(0) = O(1)$
      - Only expect 1 node + head node at level 0
  - Recursive case: ($p=0.5$)
    - $C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$
      - $1+C(k)$ = Case B and its probability is $(1-p)$
      - $1+C(k-1)$ = Case A and its probability is p
    - $C(k) = 1/p + C(k-1)$
    - $= 1/p + 1/p + C(k-2)$
    - $= 1/p + 1/p + 1/p + C(k-3)$
    - $= k/p = \log_2 (N) / p = O(\log_2 (N))$

Conclusion

- Skip list
  - Efficient data structure
  - Probabilistic data structure
  - "expected" O(log(n)) for insert, remove, and search operations
- Perfect and Randomized skip list
- Node structures are of variable size
  - The size of a node doesn't change after its creation
  - Useful assumption:
    - Knowledge about the maximum number of levels in advance

Algorithm	Average	Worst case
Space	$O(n)$	$O(n \log n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Questions ?

- Reading:
  - ➢Csci 115 book: Section 5.4 (Skip list)
  - ➢Pugh, W. Skip lists: A probabilistic alternative to balanced trees, Communications of the ACM, vol. 33, no. 6, pp. 668–676, 1990.