# Skip Lists

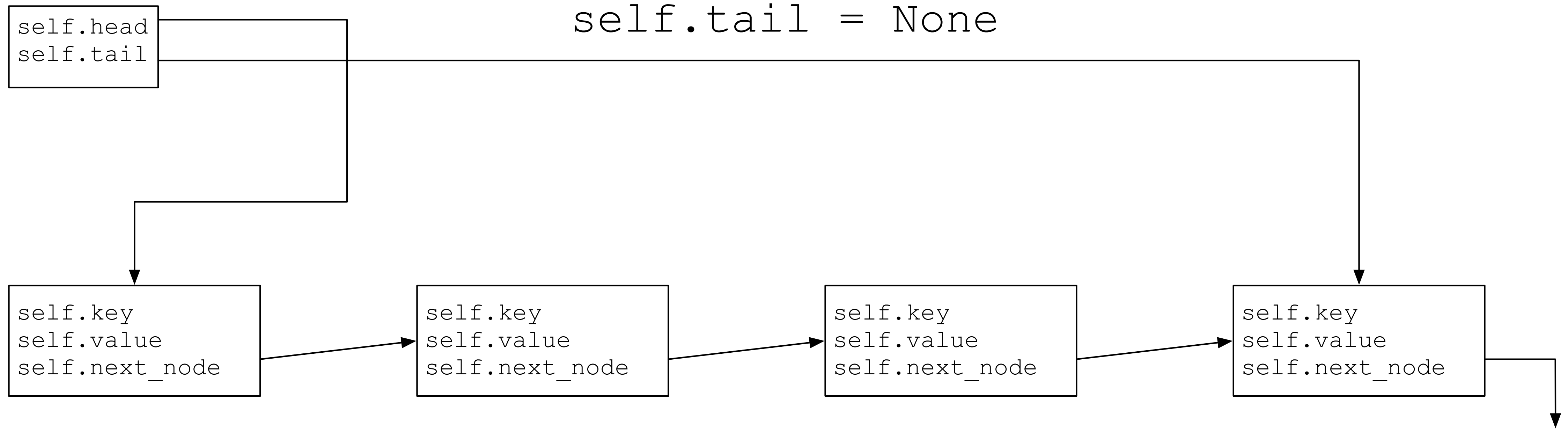
Thomas Schwarz, SJ

# Linked Lists

- Standard data structure for storing key-value records

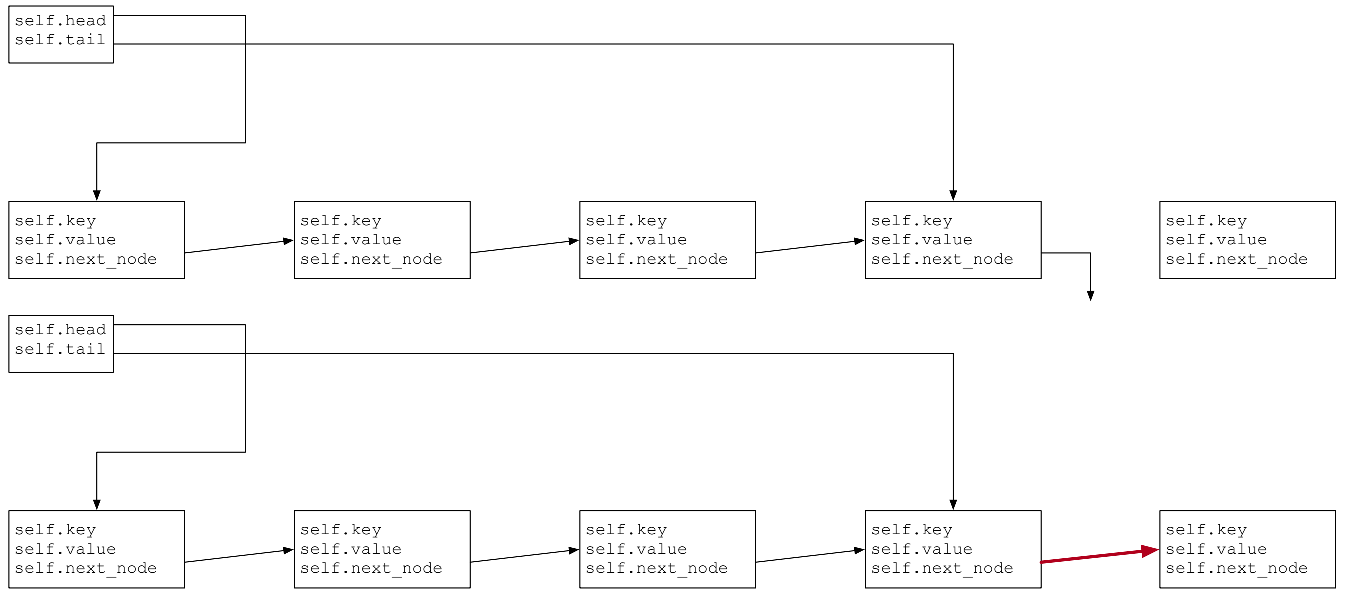
```
class Node:
    def __init__(self, key, value, next_node):
        self.key = key
        self.value = value
        self.next_node = next_node

class List:
    def __init__(self):
        self.head = None
        self.tail = None
```



# Linked List

- Tail inserts:
    - Insert after the tail:
        - If there is no tail:
            - List is empty, so create a new node and set head and tail to it
        - Otherwise:
            - Create a new node and change the link in the tail



# Linked List

```
def insert(self, key, value):
        new_node = Node(key, value, None)
        if self.tail:
            self.tail.next_node = new_node
            self.tail = new_node
        else:
            self.head = new_node
            self.tail = new_node
```

## Linked List

- Update by Value
  - Run through the list:
    - Set current pointer to head
    - Check value
    - Follow current pointer

```
def update_by_value(self, old_value, new_value):
        current = self.head
        while(current):
            if current.value == old_value:
                current.value = new_value
                return
            current = current.next_node
```

# Linked List

- Update based on key

```
def update(self, key, new_value):
        current = self.head
        while(current):
            if current.key == key:
                current.value = new_value
                return
            current = current.next_node
```

# Ordered Linked List

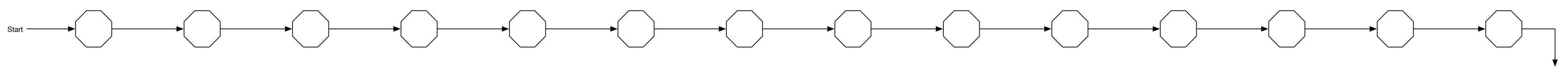
- Ordered Linked List:
  - Insert in order:
    - First find insertion point: the node before
      - This means looking at the next node
      - Three cases:
        - Insert before head (and update head)
        - Insert after tail (and update tail)
        - Insert in the middle

# Ordered Linked List

```
def insert(self, key, value):
        new_node = Node(key, value, None)
        if not self.head:
            self.head = new_node
            self.tail = new_node
        elif self.head.key > key:
            new_node.next_node = self.head
            self.head = new_node
        else:
            current = self.head
            while current.next_node and current.next_node.key < key:
                current = current.next_node
            new_node.next_node = current.next_node
            current.next_node = new_node
            if not new_node.next_node:
                self.tail = new_node
```

# Ordered Linked List

- Start with a normal **ordered** linked list
    - Nodes consist of key plus pointer to data plus link to next node

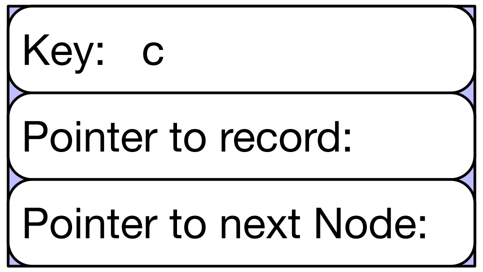


# Ordered Linked List

- How do we find a record with a given key *c*?
  - Use the pointer to the list in order to find the first node
  - Compare the key of the node with *c*
  - If they are equal, you found the record
  - If they are not equal, continue until you find the record
  - If you get to the null pointer at the very end or if you find a node with key > *c* then the record is not there

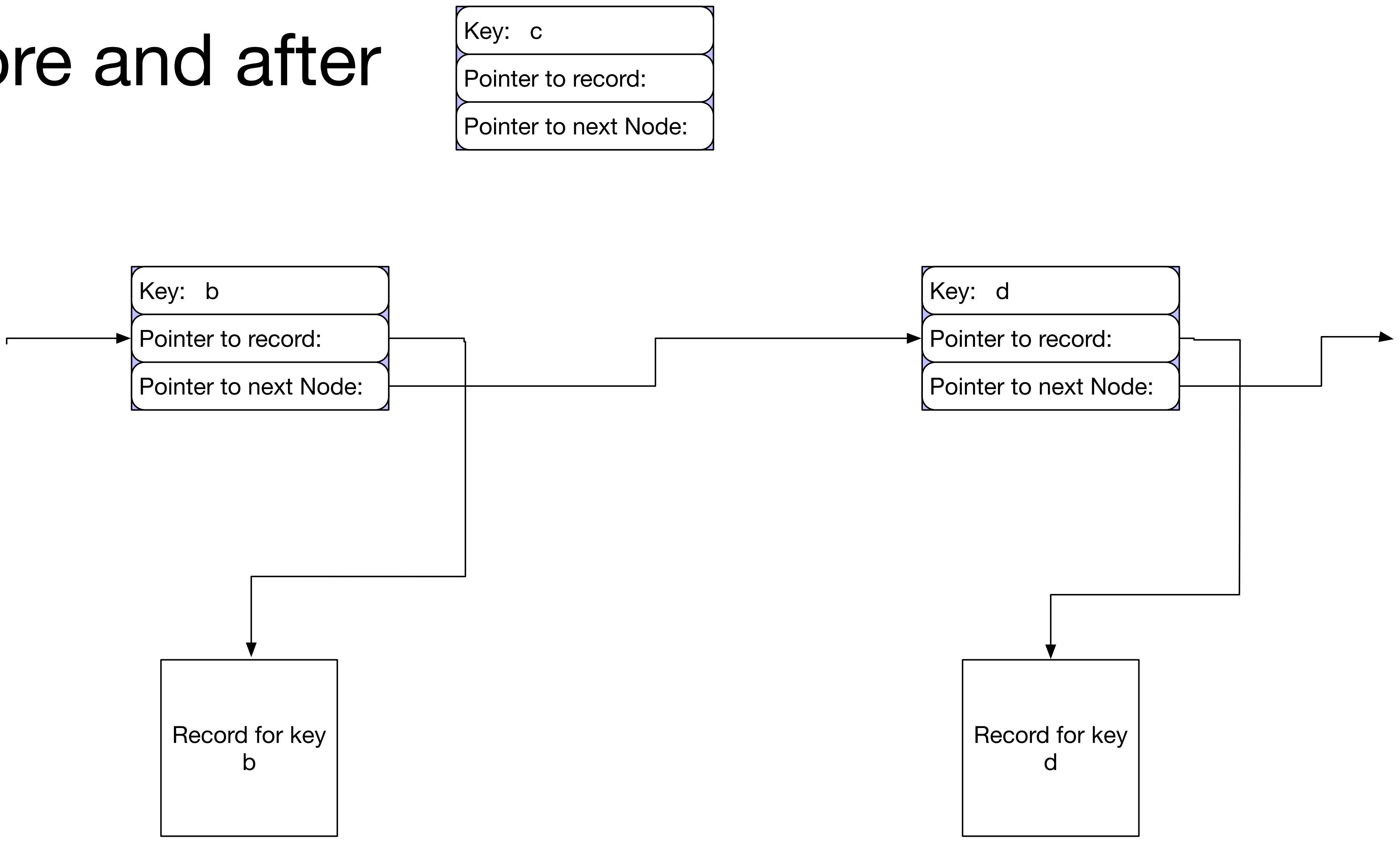
# Ordered Linked Lists

- How do you insert a record?
    - Create a new node with a key c



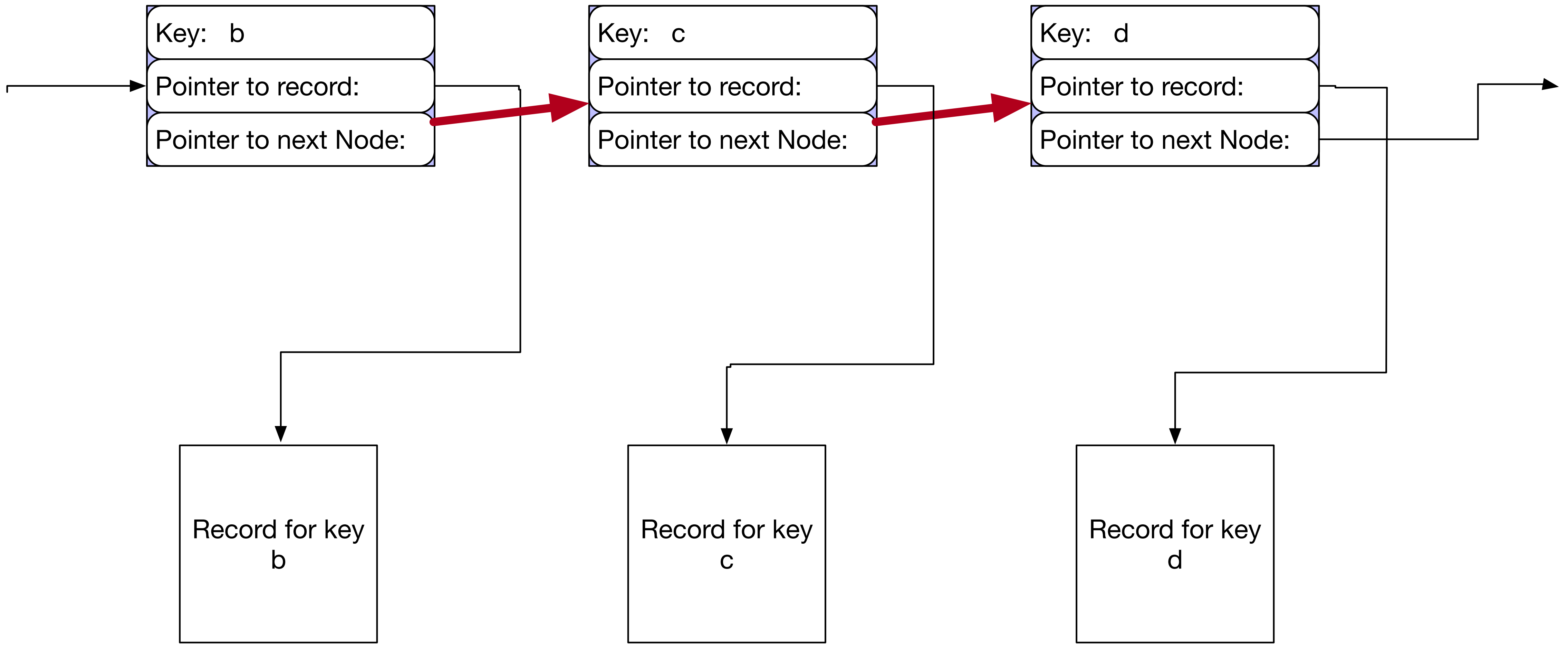
# Ordered Linked Lists

- How do you insert a record?
- Pretend that you look for the record
- Find the node before and after



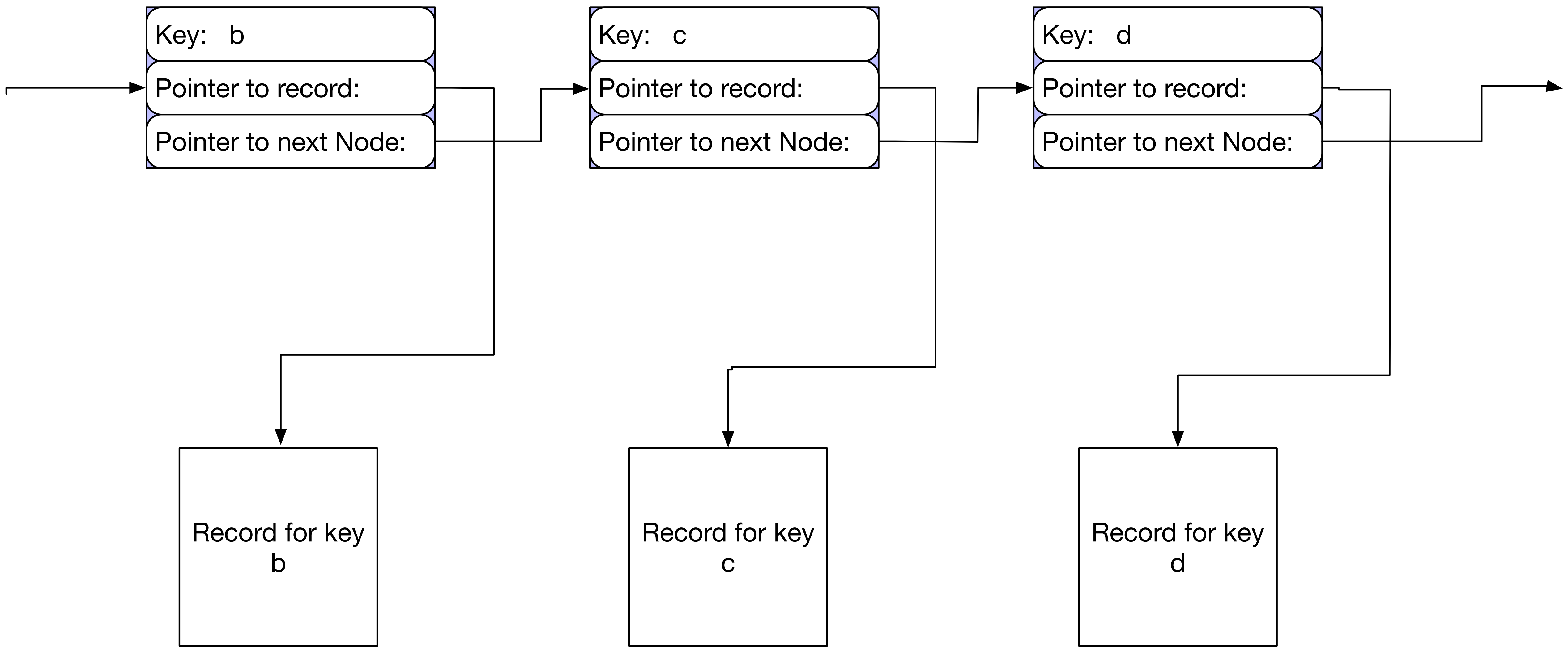
# Ordered Linked Lists

- How do you insert a record?
- Now set two pointers to connect the previous to the new and the new node to the following node.



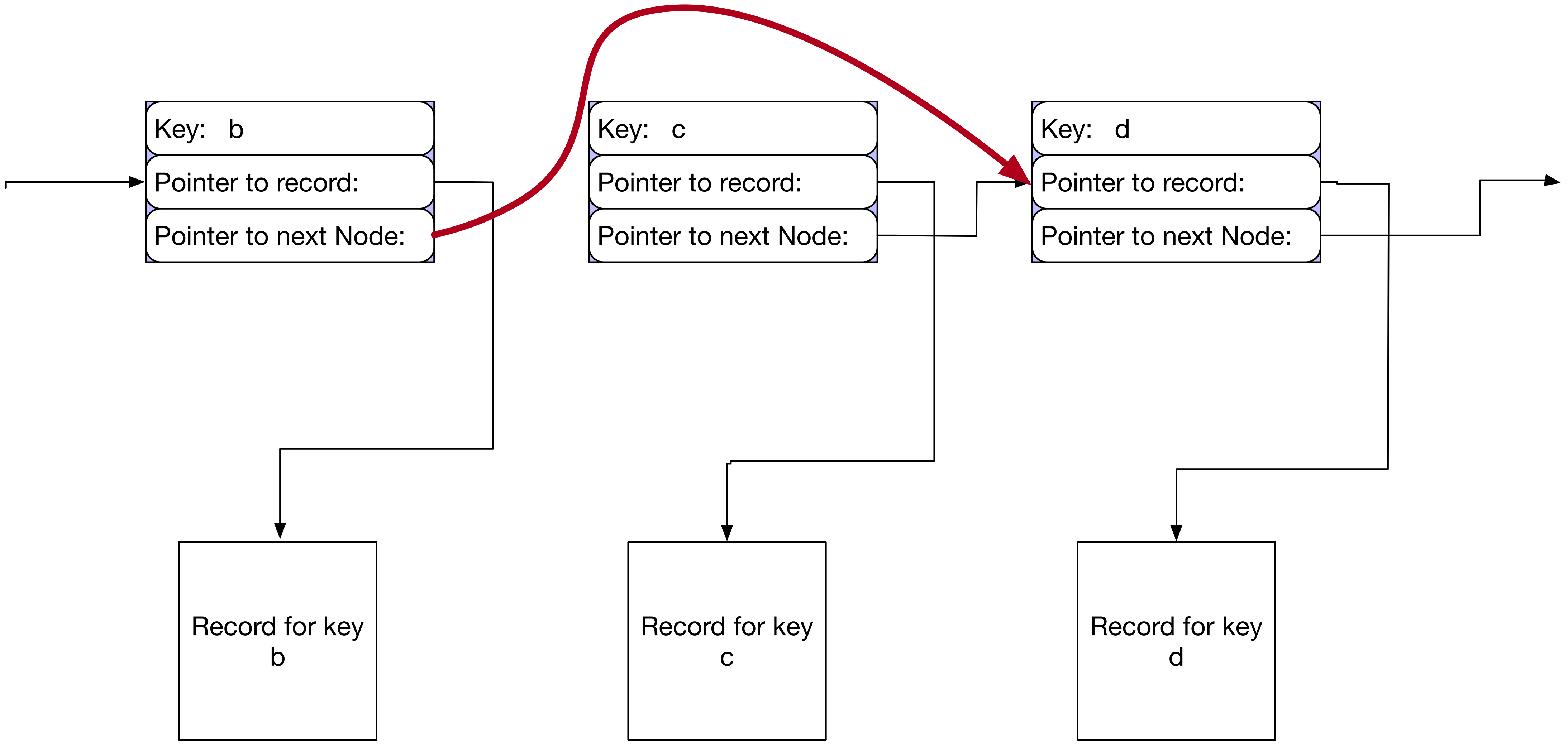
# Ordered Linked Lists

- How do you delete a record?
    - Find previous and subsequent node



# Ordered Linked Lists

- How do you delete a record?
- Change one pointer



# Ordered Linked Lists

- Analysis
    - Assume ordered list has *n* elements
    - Insert: Needs to insert after visiting on average *n*/2 elements, afterwards constant reconstruction work
    - Deletes: Needs to delete node after visiting on average *n*/2 nodes
    - Reads / Updates: Needs to find node after visiting on average *n*/2 nodes

# How to implement an Ordered Linked List

- Remarks for Python and Java programmers
    - Python and Java do not have explicit pointers
    - But an object is given by its address
    - Objects persists in memory until nobody has a reference (i.e. a pointer) to them

# How to implement an Ordered Linked List

- What do we need for a node:
    - A place for the key
    - A pointer to the record
    - A pointer to the next node

# How to implement an Ordered Linked List

```
class Node:
    def __init__(self, key  = None, nextN = None, data = None):
        self.key = key
        self.next = nextN
        self.data = data

    def __str__(self):
        return "Node: key={}, data={}, next={}".format(self.key, 
self.data, self.next)
```

# How to implement an Ordered Linked List

- An ordered linked list is given by a node with sentinel value minus infinity
- To find in an OLL:
  - Follow the next pointer until you hit a node with the key that you are looking for
  - Then follow the data link

# How to implement an Ordered Linked List

- Implementing look-up

```
def find(self, key):
        currentNode = self.head
        while currentNode and currentNode.key < key:
            currentNode = currentNode.next
        if currentNode and currentNode.key == key:
            return currentNode.key, currentNode.data
        else:
            return None
```

- Question:  Why do I know that at the end of the while loop, currentNode.key >= key?

# How to implement an Ordered Linked List

- Implementing insert

```
def insert(self, key, data):
        current = self.head
        while current.next and key > current.next.key:
            current = current.next
        if current.next and current.next.key == key:
            print('key error: insertion failed, {}'.format(key))
            return
        newNode = Node(key, current.next, data)
        current.next = newNode
        return
```

- Why do I know that current after the while loop is the node just to the left of the insertion point?

# How to implement an Ordered Linked List

- Loop Invariant:
    - key > current.next.key
- Only violated when I jump out of the while loop
- Therefore
    - current.key < key < current.next.key if current.next exists
    - current.key < key  otherwise

# How to implement an Ordered Linked List

- Implementing delete

```
def delete(self, key):
        current = self.head
        while current.next and key > current.next.key:
            current = current.next
        if key != current.next.key:
            return
        else:
            current.next = current.next.next
```

# Skip Lists

Thomas Schwarz, SJ

# Skip List Motivation

- Can you guess what this list is?

San Francisco
22nd Street
Bayshore
South San Francisco
San Bruno
Millbrae
Burlingame
San Mateo
Hayward Park
Hillsdale
Belmont
San Carlos
Redwood City
Menlo Park
Palo Alto
California Ave.
San Antonio
Mountain View
Sunnyvale
Lawrence
Santa Clara
College Park
San José
Tamien
Capitol
Blossom Hill
Morgan Hill
San Martin
Gilroy

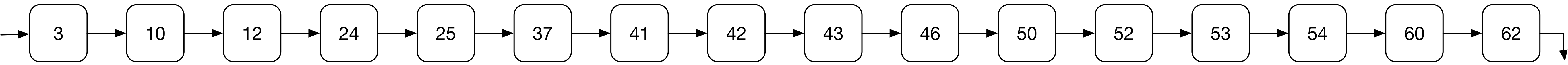
# Skip List Motivation

- Caltrain stations from San Francisco South
- Underlined stations are for the "Baby Bullet"
- You can take time of your trip if you use the baby bullet, and then switch to a local
- That is how skip lists improve on linked lists.

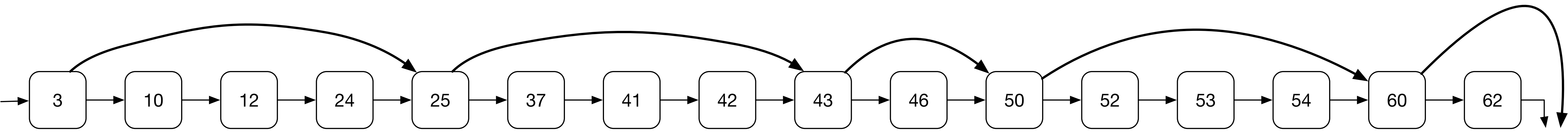
```
San Francisco
22nd Street
Bayshore
South San Francisco
San Bruno
Millbrae
Burlingame
San Mateo
Hayward Park
Hillsdale
Belmont
San Carlos
Redwood City
Menlo Park
Palo Alto
California Ave.
San Antonio
Mountain View
Sunnyvale
Lawrence
Santa Clara
College Park
San José
Tamien
Capitol
Blossom Hill
Morgan Hill
San Martin
Gilroy
```

# Ordered Linked Lists with a Baby Bullet

- One simple way to speed up look-up in linked lists is to have shortcuts between nodes.
    - An ordinary ordered linked list



- The same ordered linked lists with short-cuts



# Ordered Linked Lists with a Baby Bullet

- To find a record with key *c* or an insertion point for the record with key *c*:
    - Use the baby-bullet links until the node pointed to has a key value larger than *c* or does not exist
    - Then switch to the normal links

## Ordered Linked Lists with a Baby Bullet

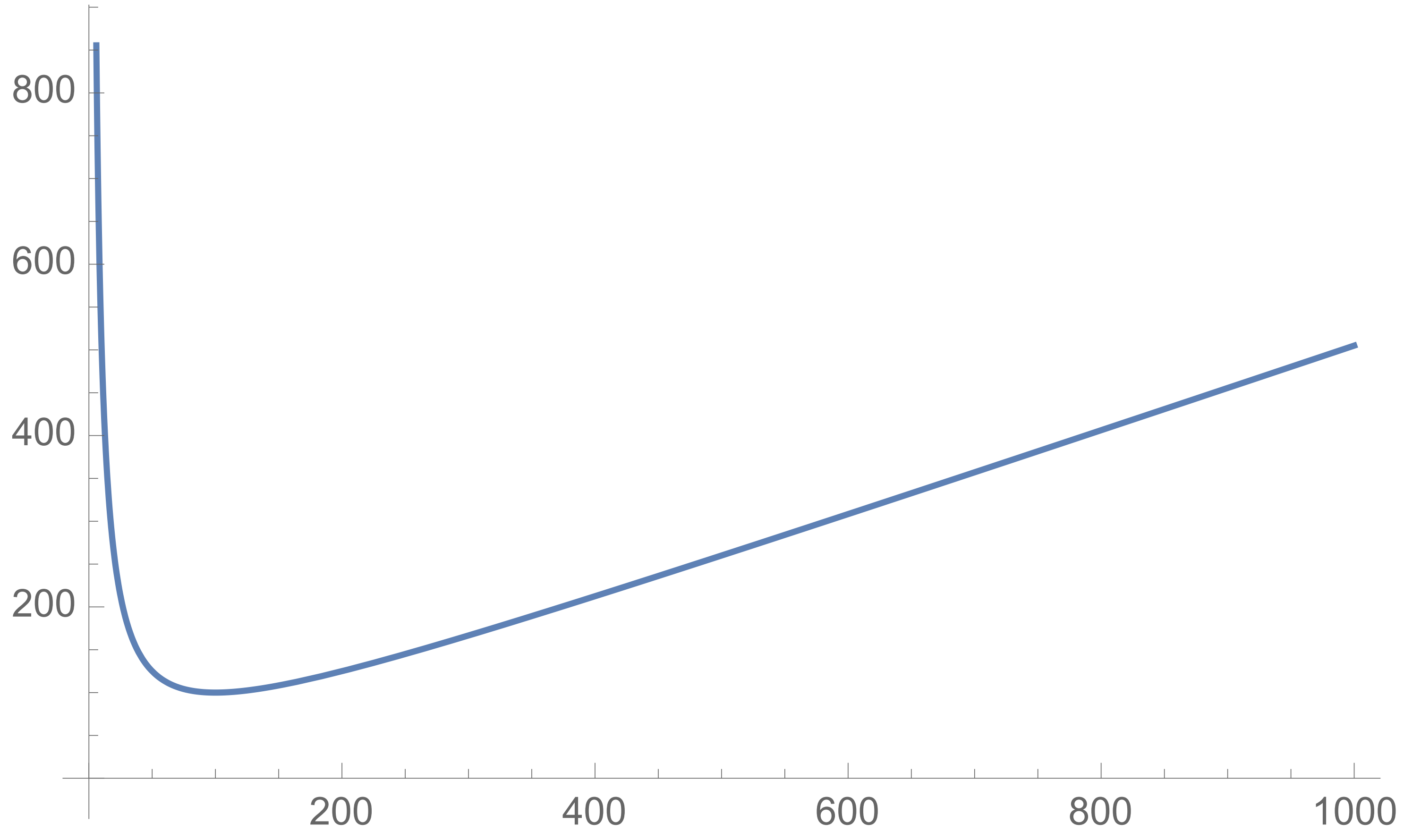
- How to maintain the Baby Bullet stations
  - Strategy 1:
    - Whenever the distance between two baby bullet stations is too large, we introduce a new baby bullet station in the middle
  - Strategy 2:
    - Whenever we insert a record, the corresponding node becomes a baby bullet train station with a given probability

# Ordered Linked Lists with a Baby Bullet

- Analysis
  - Assume *n* nodes per list, *m* nodes that are baby bullet nodes
  - Average distance between two baby bullet nodes is $n/m$
  - On average, will need $m/2$ baby bullet stations and then $(n/m)/2 = \frac{n}{2m}$ normal stations to find a record / insertion point.

# Ordered Linked Lists with a Baby Bullet

- Minimize $f(m,n) = \frac{m}{2} + \frac{n}{2m}$ with respect to $m$ for constant $n$:
- For $n = 10000$:

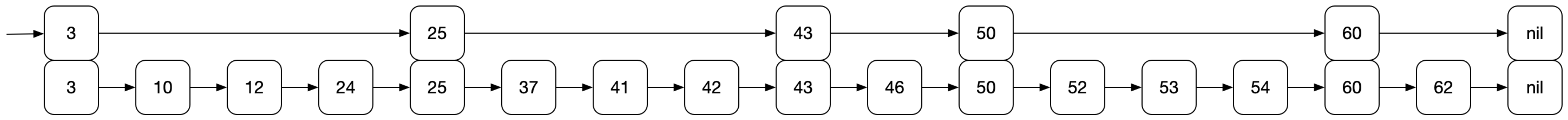


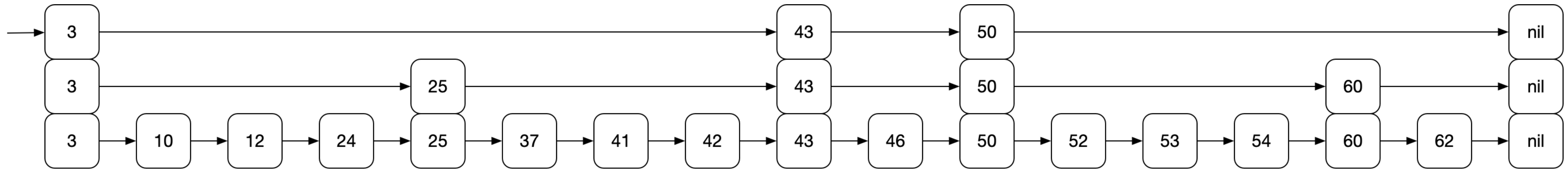
# Ordered Linked Lists with a Baby Bullet

- Calculate the derivative and set it equal to zero
  - $\frac{\delta f(m,n)}{\delta m} = \frac{1}{2} - \frac{n}{2m^2}$ is zero if
  - $m = \pm\sqrt{n}$
- This suggests that our two strategies will not work well for growing lists
- One possibility: Make a new node a baby bullet train station with a probability that slowly sinks in dependence on the number of elements inserted

# Puig's Skip List

- Skip List: Create more and more nodes at a higher level

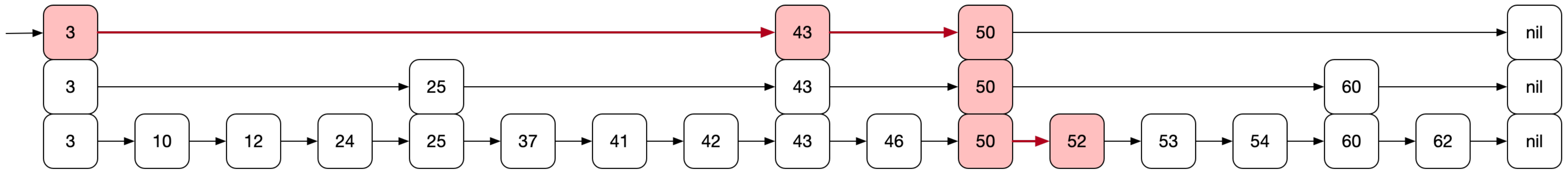




- For searches, inserts, deletes use the highest level, then if you overshoot, go down one level

# Puig's Skip List

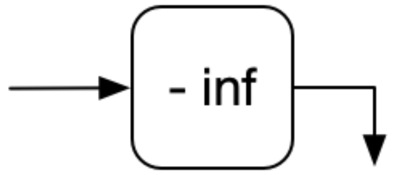
- Example:
  - Searching for 52



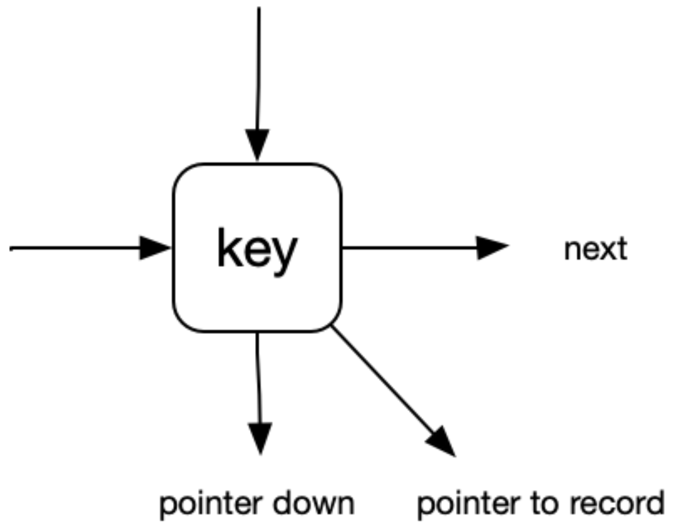
- start out at three level 2
- move to 43 level 2
- move to 50 level 2
- overshoot: move to 50 level 1
- overshoot: move to 50 level 0
- move to 52

# Puig's Skip List

- Creating a skip list
  - Start out with a start node with one level
  - With a sentinel value of - infinity



- Nodes have
  - key (assumed to be an integer)
  - pointer to next on the same level
  - pointer to down node (or nil if we are at level 0)
  - pointer to record if we are at level 0



## Puig's Skip List

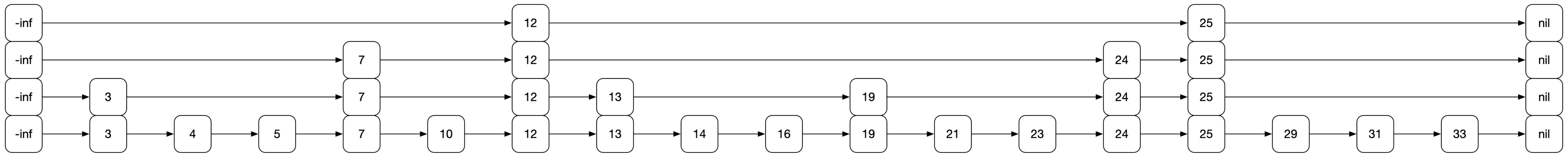
- Construction
  - Beginning node has to have the maximum level of any other nodes
  - Could have a last node with key infinity to finish or could have pointers having a null value

# Puig's Skip List

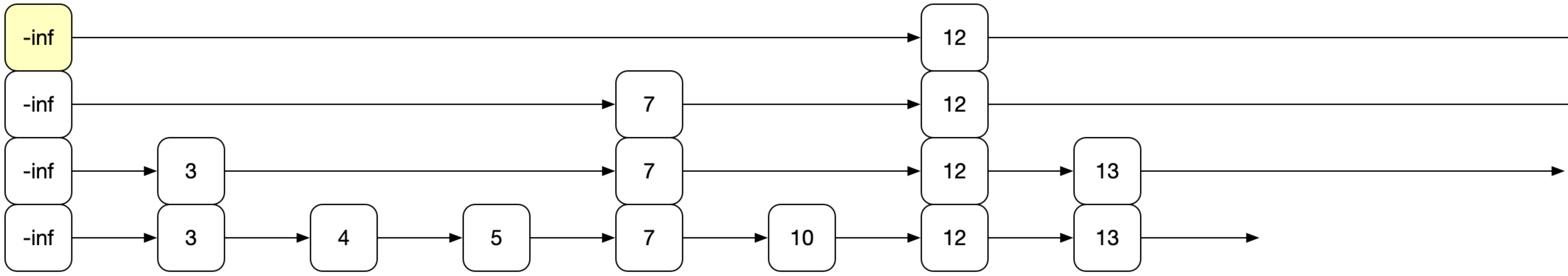
- Searching for a record with key *c*
  - Start in the highest level start node; set it to currentNode
    - Guaranteed to have level equal to the highest level node in the list
  - Follow the forward pointer
    - If forward pointer points to a key with key larger than *c* or the forward pointer is null:
      - Follow the downward pointer: currentNode = currentNode.down
      - If downward pointer is zero, then the record with key *c* does not exist
    - Otherwise:
      - Follow the forward pointer: currentNode = currentNode.next

# Puig's Skip List

- Example: Looking for node 23

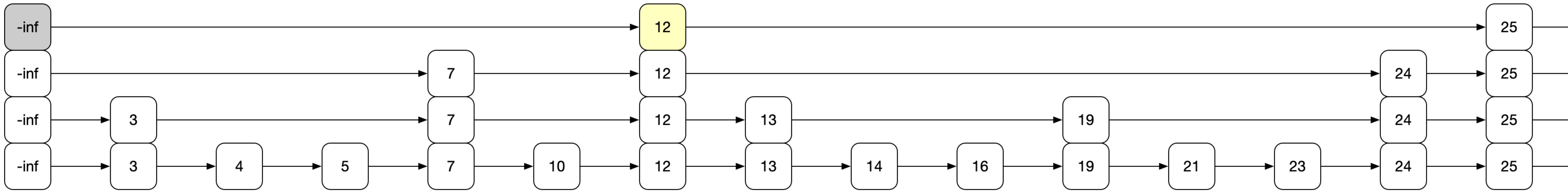


- Start out in the highest level start node
- Get the key of the next node at this level
    - curpoint.next.key is 12



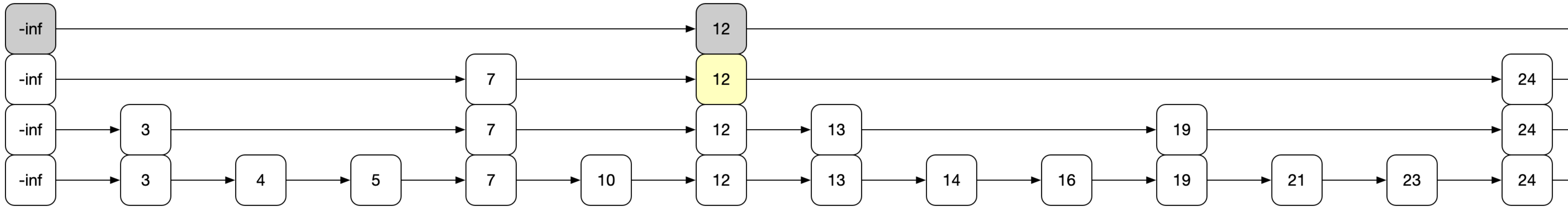
# Puig's Skip List

- Since 12 < 23, follow the next pointer



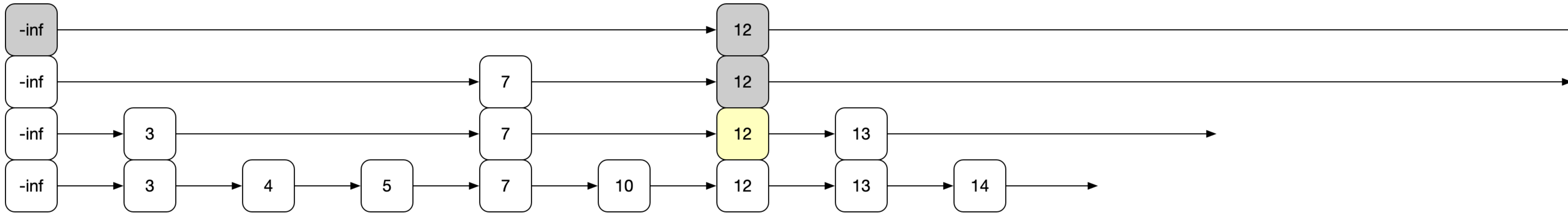
# Puig's Skip List

- The key in the next node is 25, which is larger than 23
- Go down



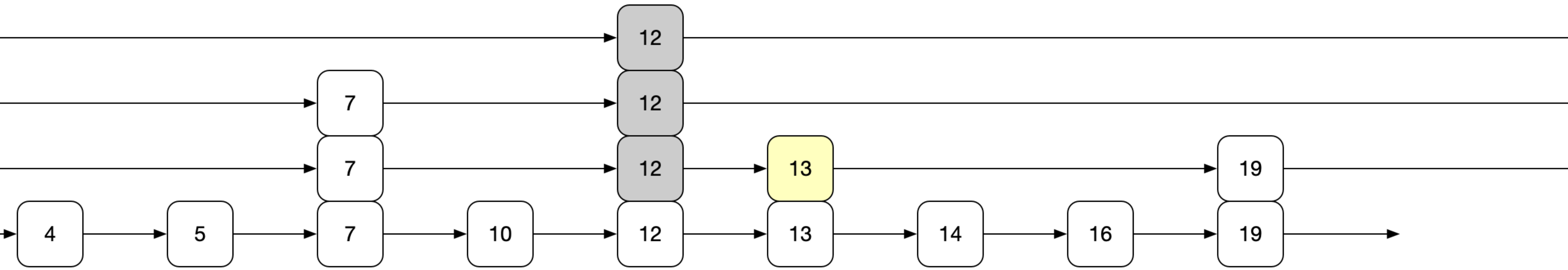
# Puig's Skip List

- The key in the next node at this level is 24, which is larger than 23
- Go down one more to level 1



# Puig's Skip List

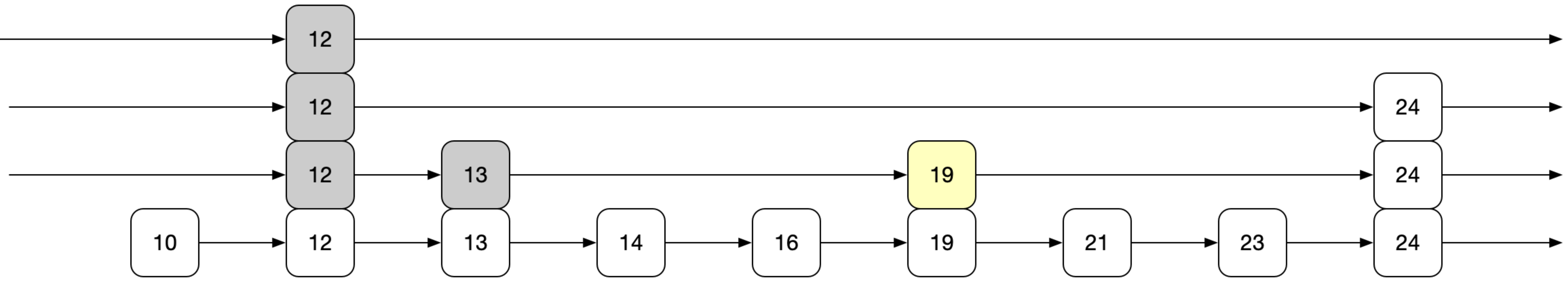
- The key in the next node is 13, so we follow the next link



- The current node now has key 13 and is at level 1
- Since the key in the next node is 19, which is < 23 we follow the next link

# Puig's Skip List

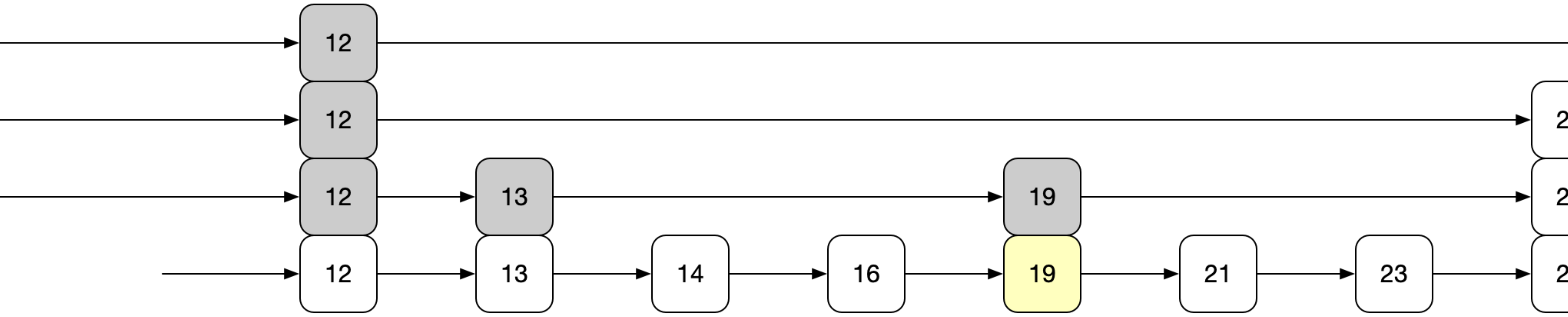
- Current node has key 19 and next node has key 24



- Therefore, we follow the downward pointer

# Puig's Skip List

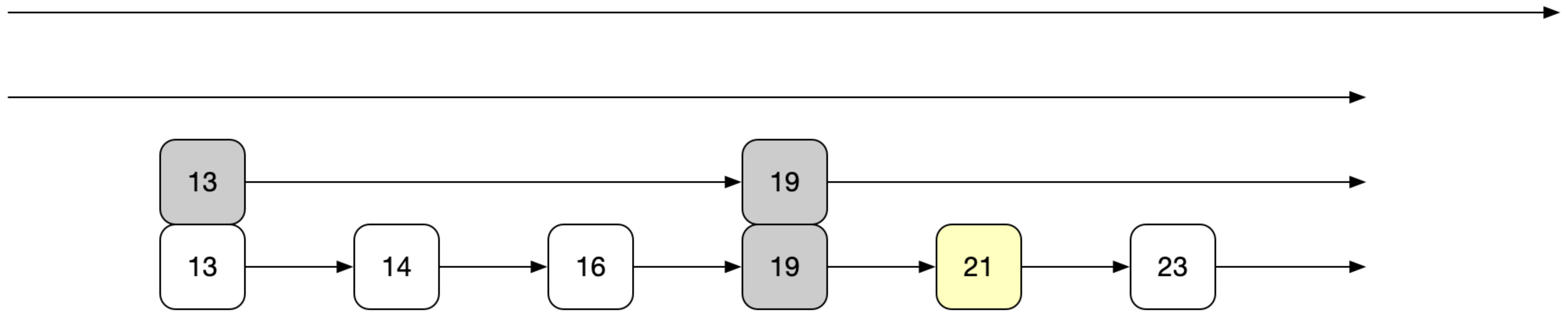
- Current node has level 0, next node has key 21



- Set current node to the next node

# Puig's Skip List

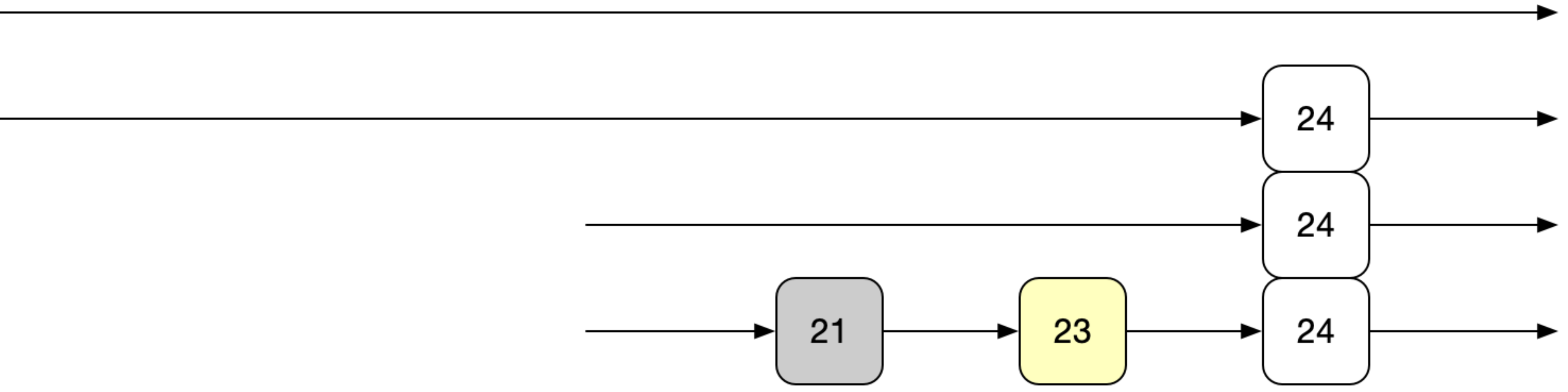
- Current node has key 21 and next node has key 23



- Go to it

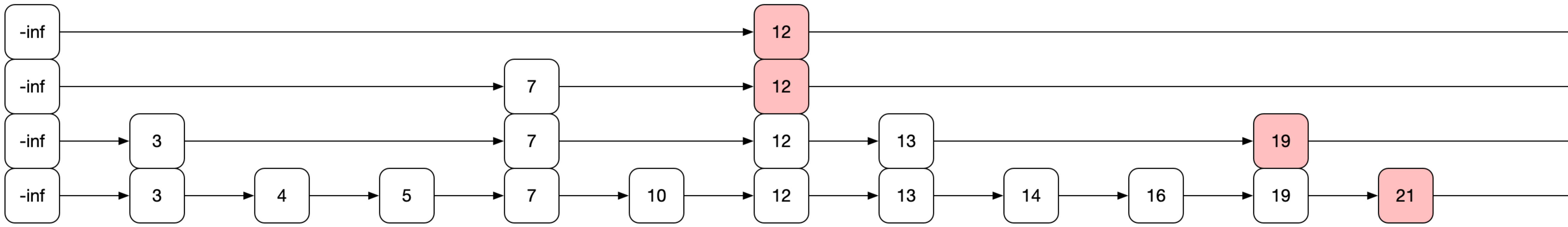
# Puig's Skip List

- Current node has the key we are looking for
- Follow the record link to retrieve the record



# Puig's Skip List

- To insert a node:
  - Do a search, but remember each last level node
  - Example: Inserting 22



# Puig's Skip List

- Node insertion:
  - Determine the level probabilistically
    - Use base probability *p*
    - With probability $p$ : Node goes up one level
    - With probability $p^2$: Node goes up two levels
    - With probability $p^3$: Node goes up three levels
    - etc.

# Puig's Skip List

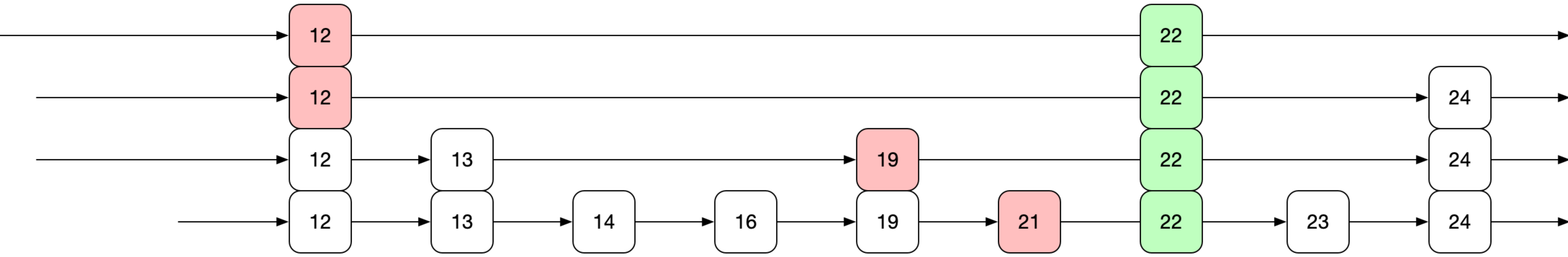
- Node Insertion:
  - In practice:  determine a maximum level maxLevel

```
import random

def level(maxLevel, p):
    level = 0
    while (level < maxLevel):
        if random.random() > p:  #stop with probability 1-p
            return level
        level += 1
    return level
```

# Puig's Skip List

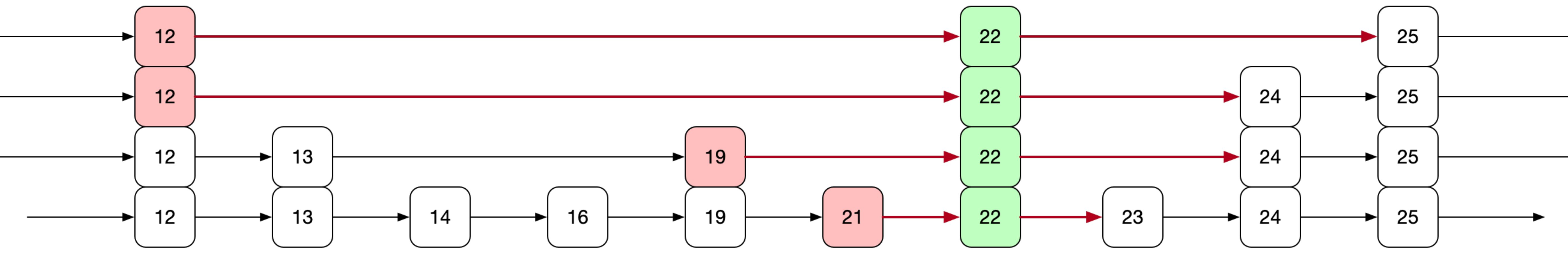
- At each level of the node, splice the node into the existing levels
- Assume we have a level 4 new node



- Need to set four next pointers in the new nodes
- Switch the predecessor pointers to the new nodes

# Puig's Skip List

- Final result:



# Puig's Skip List

- Deletion:
    - Works similar to ordered linked lists

# Puig's Skip List

- Analysis
  - To show: Searches (inserts, and deletes) in time $\Theta(\log n)$ for a list of $n$ elements
  - Can show: Probability that a search exceeds time $C\log(n)$ vanishes fast

# Puig's Skip List

- The expected number of nodes in each level is
    - $np^0 = n$ for level 0
    - $np^1 = pn$ for level 1
    - $np^2 = p^2n$ for level 2
    - etc.

# Puig's Skip List

- Define $L(n)$ to be the level (in dependence on $n$) where there are $\frac{1}{p}$ nodes.
  - Recall that $\frac{1}{p}$ is larger than one
  - From what we just have seen:
    - $np^{L(n)} = \frac{1}{p}$
  - which implies
    - $n = \frac{1}{p}^{L(n)+1}$ or $L(n) = \log_{\frac{1}{p}}(n) - 1$

# Puig's Skip List

- Analysis:
  - Trick: go backward from node
    - Although all nodes and levels are known, we act as if we discover them while backtracking the search path
  - Let $c(k)$ be the costs of going up $k$ levels in an infinite list
    - When we go towards the beginning, we either can move up (with probability $p$) or move right (with probability $1 - p$)

# Puig's Skip List

$$c(k) = p(1+\text{cost move up}) + (1-p)(1+\text{cost move left})$$

$$= p(1 + c(k-1)) + (1-p)(1 + c(k))$$

$$= p + (1-p) + pc(k-1) + (1-p)c(k)$$

$$= 1 + p \cdot c(k-1) + (1-p) \cdot c(k)$$

This means (by subtracting $(1-p) \cdot c(k)$ on both sides

$$p \cdot c(k) = 1 + p \cdot c(k-1)$$

or $c(k) = \dfrac{1}{p} + c(k-1)$.

This implies $c(k) = \dfrac{k}{p}$.

# Puig's Skip List

- Since we are in an infinite list, we cannot just set $k = \infty$.
- Let's set $k = L(n)$.
- At that level, there are $1/p$ nodes and $1/p$ leftward moves
- At level $L(n) + 1$, we expect one node
- At level $L(n) + 2$, we have on average $p$ nodes
- etc.
- In total, there are $\sum_{l=1}^{\infty} p^{l-1} = \frac{1}{1-p}$ nodes at and above $L(n)$

# Puig's Skip List

- Therefore:
  - $L(n)/p$ moves to get to level $L(n)$
  - Afterwards, on average $\frac{1}{1-p}$ moves to the top of the initial node
- For a total of

$$\frac{L(n)}{p} + \frac{1}{1-p} = \frac{\log_{\frac{1}{p}} n}{p} + + \frac{1}{1-p} = \Theta(\log(n))$$

# Puig's Skip List

- Practical considerations:
    - If we have an idea about the maximum length $N$ of a list, use a maximum level of $L(N) = \log_{\frac{1}{p}}(N)$
    - We do not need to keep nodes at different levels separate:
        - Just have nodes with $L(N)$ pointers at different levels