

CSCI 104

Skip Lists

Mark Redekopp

Sources / Reading

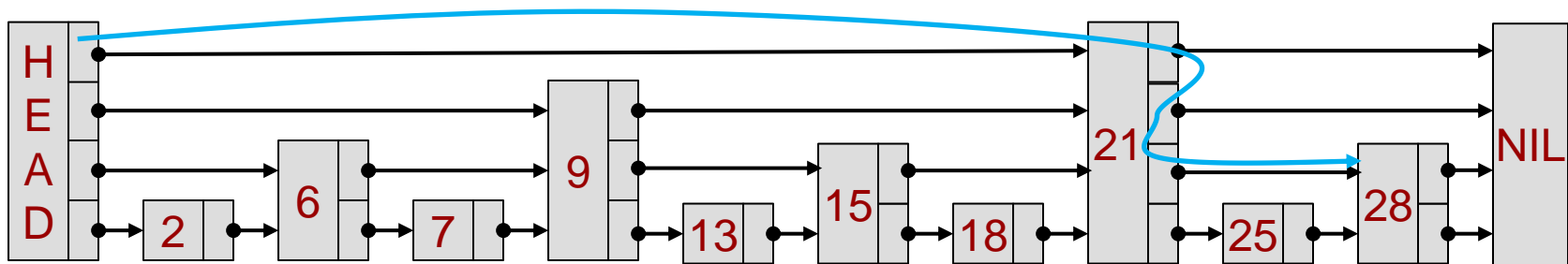
- Material for these slides was derived from the following sources
 - <http://courses.cs.vt.edu/cs2604/spring02/Projects/1/Pugh.Skiplists.pdf>
 - <http://www.cs.umd.edu/~meesh/420/Notes/MountNotes/lecture11-skiplist.pdf>

Skip List Intro

- Another map/set implementation (storing keys or key/value pairs)
 - Insert, Remove, Find
- Remember the story of Goldilocks and the Three Bears
 - **Father's porridge was too hot**
 - **Mother's porridge was too cold**
 - **Baby Bear's porridge was just right**
- Compare Set/Map implementations
 - **BST's were easy but could degenerate to $O(n)$ operations with an adversarial sequence of keys (too hot?)**
 - **Balanced BSTs guarantee $O(\log(n))$ operations but are more complex to implement and may require additional memory overhead (too cold?)**
 - **Skip lists are fairly simple to implement, fairly memory efficient, and offer "expected" $O(\log(n))$ operations (just right?)**
 - Skip lists are a probabilistic data structure so we expect $O(\log(n))$
 - Expectation of $\log(n)$ does not depend on keys but only random # generator

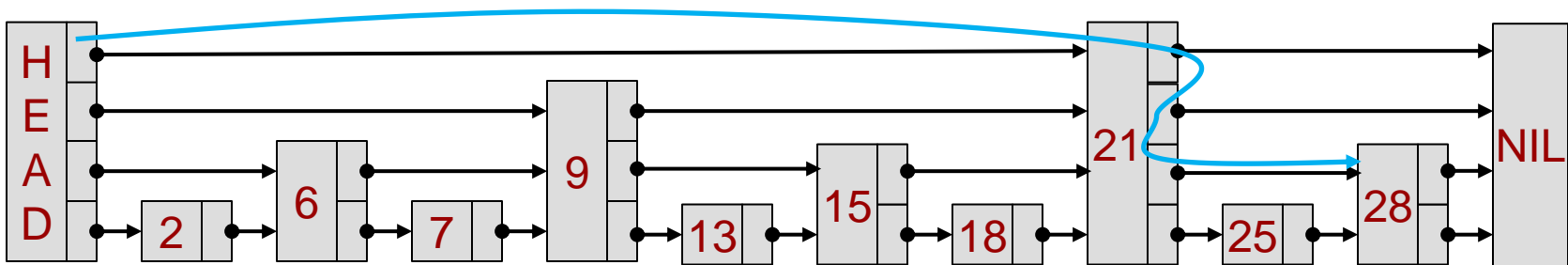
Skip List Visual

- Think of a skip list like a **sorted** linked list with shortcuts (wormholes?)
- Given the skip list below with the links (arrows) below what would be the fastest way to find if 28 is in the list?



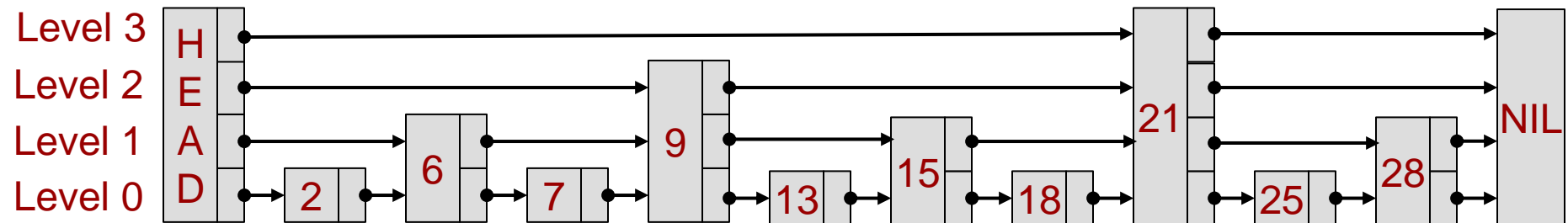
Skip List Visual

- Think of a skip list like a **sorted** linked list with shortcuts (wormholes?)
- Given the skip list below with the links (arrows) below what would be the fastest way to find if 28 is in the list?
 - Let p point to a node. Walk at level i while the desired search key is bigger than $p \rightarrow \text{next} \rightarrow \text{key}$, then descend to the level $i-1$ until you find the value or hit the NIL (end node)
 - NIL node is a special node whose stored key is BIGGER than any key we might expect (i.e. $\text{MAXKEY}+1$ / $+\text{infinity}$)



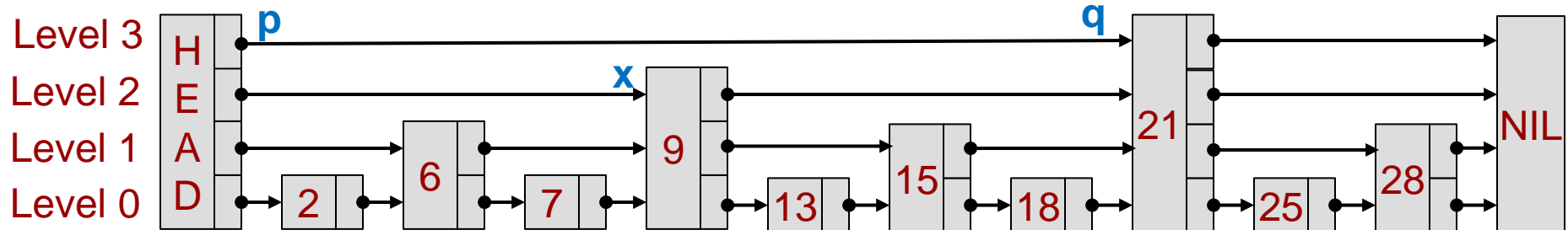
Perfect Skip List

- How did we form this special linked list?
 - We started with a normal linked list (level 0)
 - Then we took every other node in level 0 (2nd node from original list) and added them to level 1
 - Then we took every other node in level 1 (4th node from the original list) and raised it to level 2
 - Then we took every other node) in level 2 (8th node from the original list) and raised it to level 3
 - There will be $O(\log_2(n))$ levels (We would have only 1 node at level $\log_2(n)$)



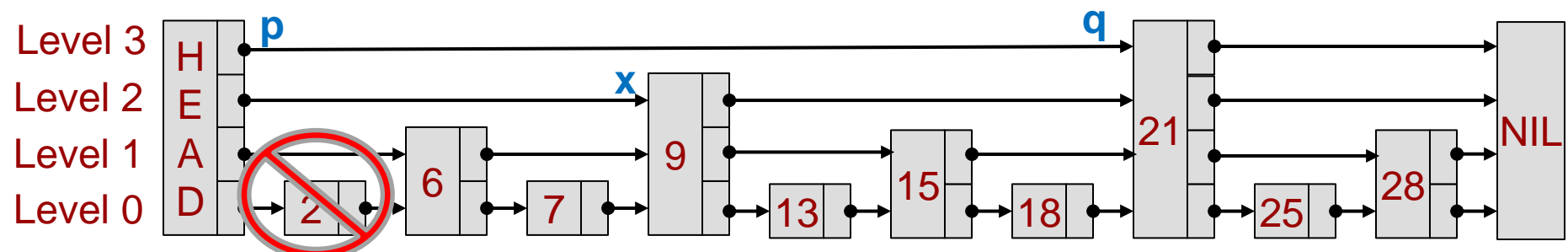
Search Time for Perfect Skip List

- How long would it take us to find an item or determine it is not present in the list
 - $O(\log(n))$
- Proof
 - 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 one other node in level i before descending
 - There are $O(\log(n))$ levels
 - So we visit at most $O(2 * \log(n))$ levels = $O(\log(n))$



The Problem w/ Perfect Skip Lists

- Remember in a perfect skip list
 - Every 2nd node is raised to level 1
 - Every 4th node is raised to level 2
 - ...
- What if I want to insert a new node or remove a node, how many nodes would need their levels adjusted to maintain the pattern described above?
 - In the worst case, all n-1 remaining nodes
 - Inserting/removing may require n-1 nodes to adjust



Quick Aside

- Imagine a game where if you flip a coin and it comes up heads you get \$1 and get to play again. If you get tails you stop.
- What's the chance you win at least
 - \$1
 - \$2
 - \$3
- $P(\$1)=1/2$, $P(\$2)=1/4$, $P(\$3)=1/8$

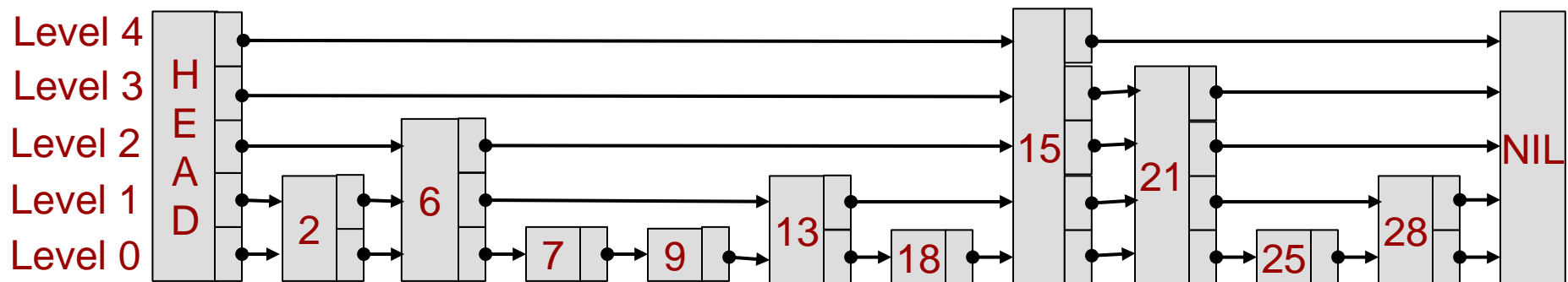


Randomized Skip Lists

- Rather than strictly enforcing every other node of level i be promoted to level $i+1$ we simply use probability to give an "expectation" that every other node is promoted
- Whenever a node is inserted we will promote it to the next level with probability p ($=1/2$ for now)...we'll keep promoting it while we get heads
- What's the chance we promote to level 1, 2, 3?
- Given n insertions, how many would you expect to be promoted to:
 - Level 1 = $n/2$, Level 2 = $n/4$, Level 3 = $n/8$

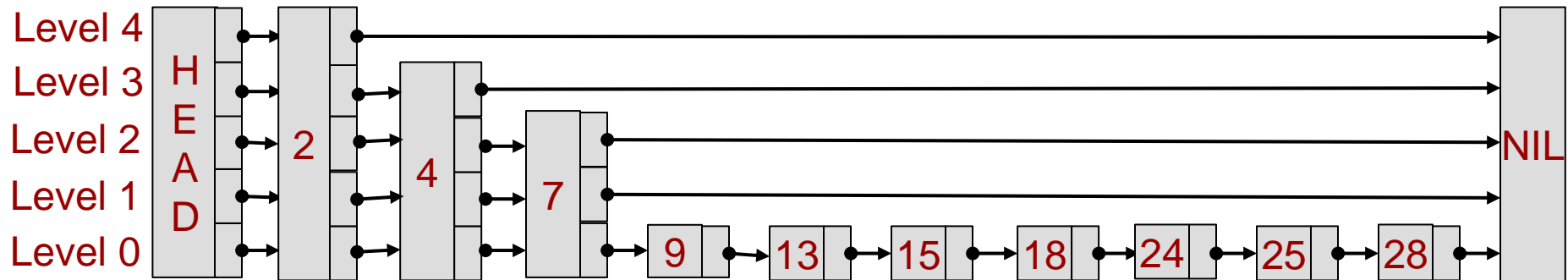
Randomized Skip List

- As nodes are inserted they are repeating trials of probability p (stopping when the first unsuccessful outcome occurs)
- This 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
- Note: This scheme introduces the chance of some very high levels
 - We will usually cap the number of levels at some MAXIMUM value
 - However the expected number of levels is still $\log_2(n)$



Worst Case

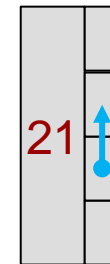
- What might a worst case skip list look like?
 - All the same height
 - Or just ascending or descending order of height
- These are all highly unlikely



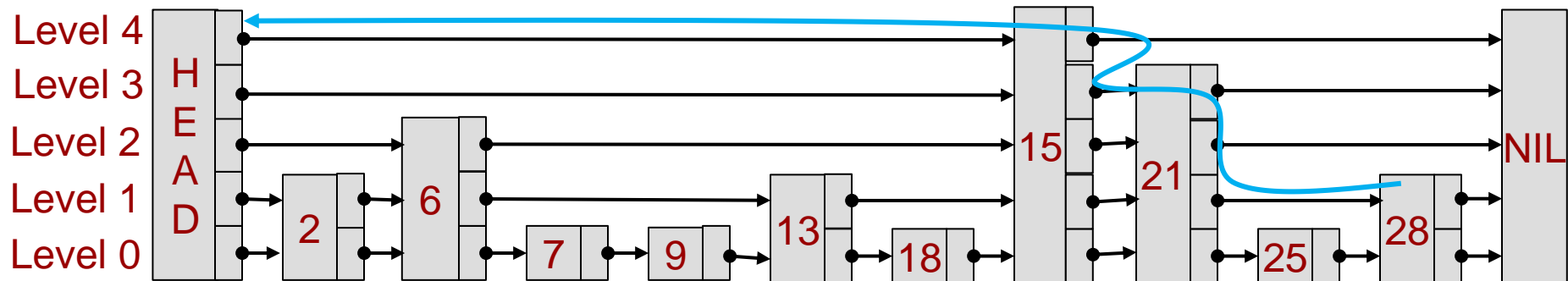
Analysis

- To analyze the search time with this randomized approach let's start at the node and walk backwards to the head node counting our expected number of steps
 - Recall 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

Option A:
If we can move up
we do



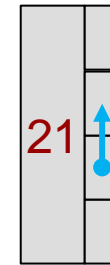
Option B: No
higher level, move
left



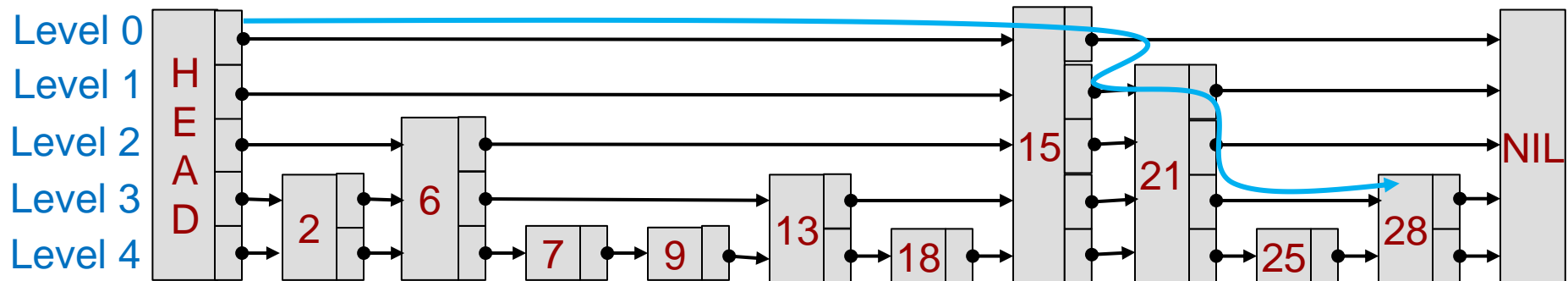
Analysis

- Probability of Option A: p
 - Recall we added each level independently with probability p
- Probability of Option B: $1-p$
- For this analysis let us define the top level at level 0 and the current level where we found our search node as level k (expected max $k = \log_2(n)$)

Option A:
If we can move up
we do



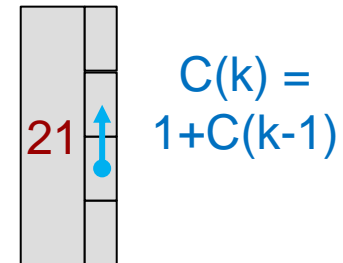
Option B: No
higher level, move
left



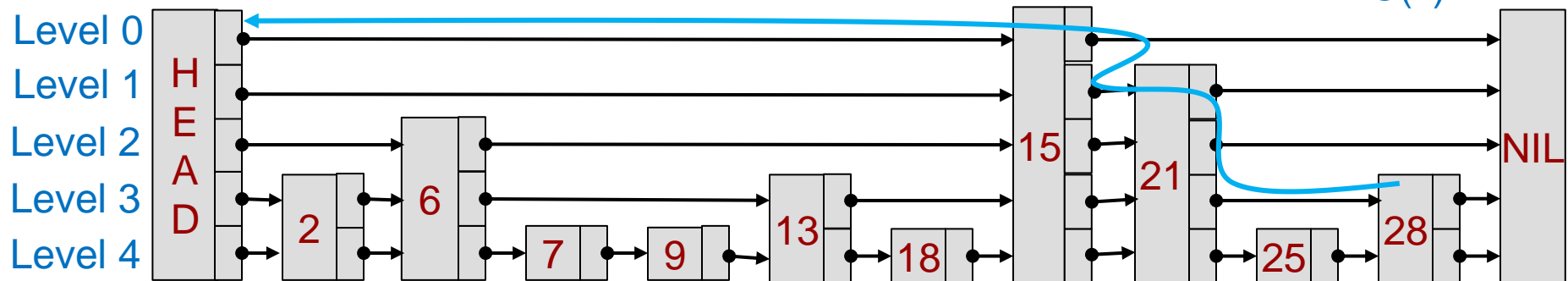
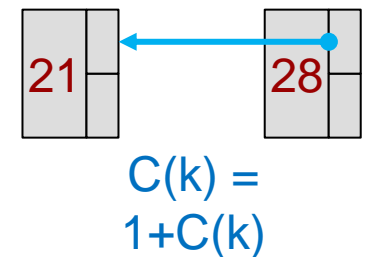
Analysis

- Define a recurrence relationship of the cost of walking back from level k to level 0
- Base case: $C(0) = O(1)$
 - Only expect 1 node + head node at level 0
- Recursive case: $C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$
 - $1+C(k)$ = Option B and its probability is $(1-p)$
 - $1+C(k-1)$ = Option A and its probability is p

Option A:
If we can move up
we do



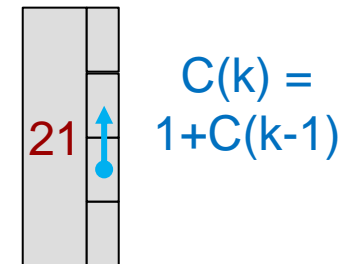
Option B: No
higher level, move
left



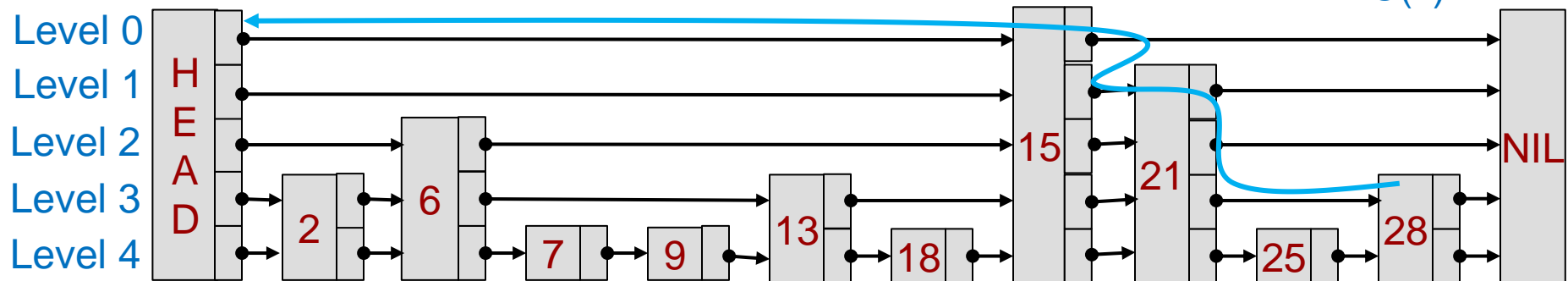
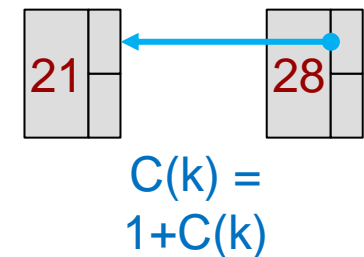
Analysis

- Solve $C(k) = (1-p)(1+C(k)) + p(1+C(k-1))$
 - $C(k) = (1-p) + (1-p)C(k) + p + pC(k-1)$
 - $pC(k) = 1 + pC(k-1)$
 - $C(k) = 1/p + C(k-1)$
 - $= 1/p + 1/p + C(k-2)$
 - $= 1/p + 1/p + 1/p + C(k-3)$
 - $= k/p$
 - $= \log_2(N) / p = O(\log_2(N))$

Option A:
If we can move up
we do



Option B: No
higher level, move
left



Node & Class Definition

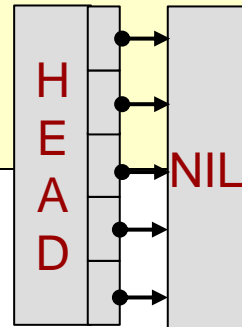
- Each node has an array of "forward" ("next") pointers
- Head's key doesn't matter as we'll never compare it
- End's forward pointers don't matter since its key value is +INF

```
template < class K, class V >
struct SkipNode{
    K key;
    V value;
    SkipNode** forward; //array of ptrs

    SkipNode(K& k, V& v, int level){
        key = k; value = v;
        forward = new SkipNode*[level+1];
    } };
```

```
template < class K, class V >
class SkipList{
    int maxLevel;    // data members
    SkipNode* head;

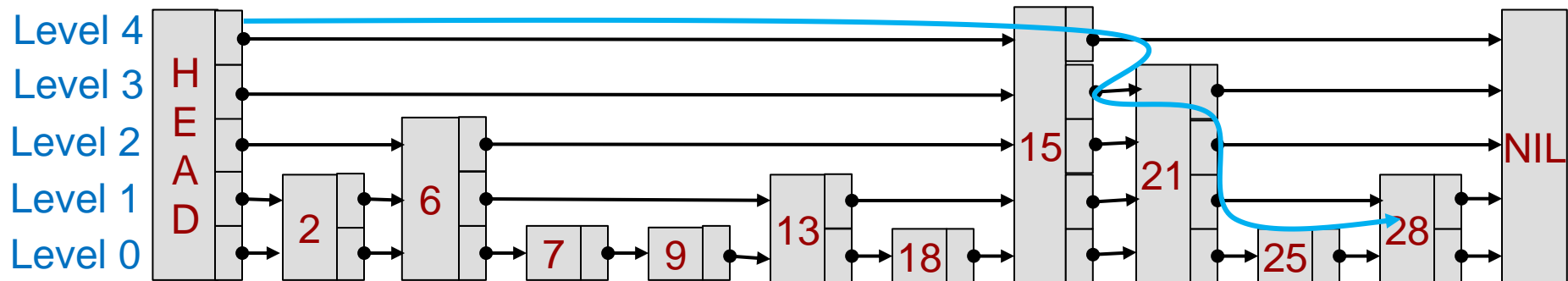
    SkipList(int max){
        maxLevel = max;
        head = new SkipNode(dummy,dummy,maxLevel);
        SkipNode* end =
            new SkipNode(INFINITY,dummy,maxLevel);
        for(int i=0; i < maxLevel; i++){
            header->forward[i] = end;
        }
    };
```



Search Pseudocode

- search(28) would stop the for loop with current pointing at **node 25**, then take **one more step**

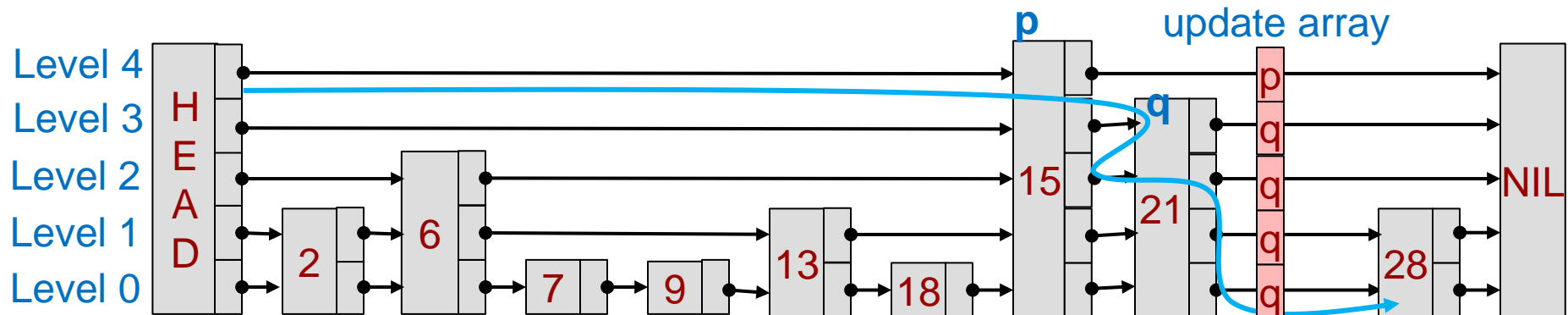
```
template < class K, class V >
SkipNode<K,V>* SkipList<K,V>::search(const Key& key){
    SkipNode<K,V>* current = head;
    for(int i=maxLevel; i >= 0; i--){
        while( current->forward[i]->key < key){
            current = current->forward[i];
        }
    }
    // will always stop on level 0 w/ current=node
    // just prior to the actual target node or End node
    current = current->forward[0];
    if(current->key == key) return current;
    else return NULL; // key is not in the list
}
```



Insert Pseudocode

- insert(25)
- As we walk we'll fill in an "update" array of the last nodes we walked through at each level since these will need to have their pointers updated

```
template < class K, class V >
void SkipList<K,V>::insert(const Key& key,
                           const Value& v){
    SkipNode<K,V>* current = head;
    vector<SkipNode<K,V>*> update(maxLevel+1);
    // perform typical search but fill in update array
    ...
    current = current->forward[0];
    if(current->key == key)
        { current->value = v; return; }
    else {
        int height = randomLevel();
        // Allocate new node, x
        for(int i=0; i < height; i++){
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
    }
}
```



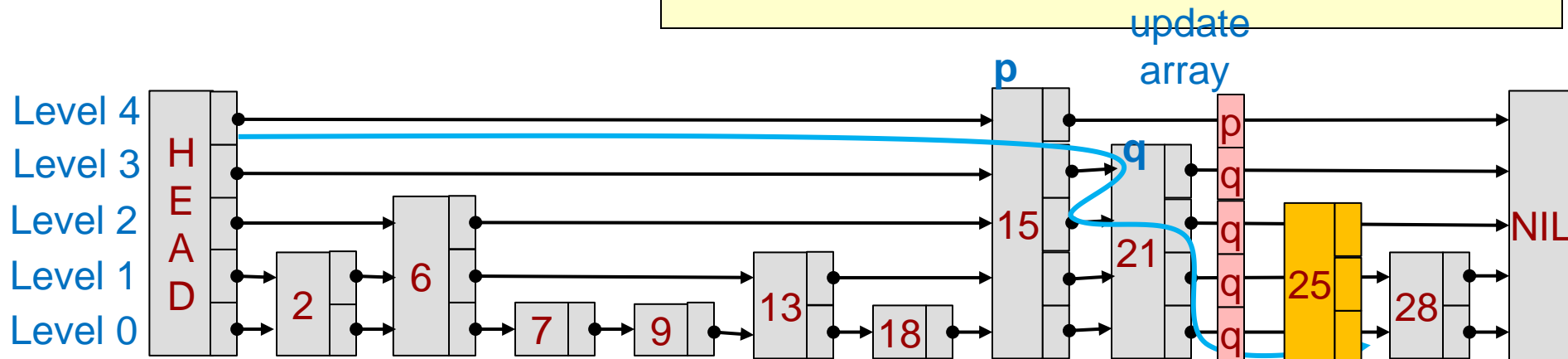
Insert Pseudocode

```
int SkipList<K,V>::randomLevel()
{
    int height = 1;
    // assume rand() returns double in range [0,1)
    while(rand() < p && height < maxLevel)
        height++;
    return height;
}
```

```
template<class K, class V>
SkipList<K,V>::insert(const Key& key,
                    const Value& v){
    V*& current = head;
    Node<K,V*>* update(maxLevel+1);
    // typical search but fill in update array

    while(current->forward[0]
           < key == key)
    {
        current->value = v; return;
    }
    else {
        int height = randomLevel();
        // Allocate new node, x
        for(int i=0; i < height; i++){
            x->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = x;
        }
    }
}
```

- randomLevel returns a height $> h$ with probability $(1/p^h)$



Summary

- Skip lists are a randomized data structure
- Provide "expected" $O(\log(n))$ insert, remove, and search
- Compared to the complexity of the code for structures like an RB-Tree they are fairly easy to implement
- In practice they perform quite well even compared to more complicated structures like balanced BSTs