Lecture 6

- Randomized data structures
- Random number generation
- Skip lists: ideas and implementation
- Skip list time costs

Reading:

"Skip Lists: A Probabilistic Alternative to Balanced Trees" (author William Pugh, available online);

Weiss, Chapter 10 section 4

Deterministic and probabilistic data structures

- Most typical data structures are deterministic data structures:
  - starting with an empty structure and a sequence of insert, find, and delete operations with specific key values, you will always get exactly the same resulting structure
- In the last few years, there has been a lot of interest in probabilistic, or randomized, algorithms and data structures:
  - in a probabilistic data structure, the same sequence of operations will usually not result in the same structure; it depends on the sequence of random bits generated during the operations
- It turns out that these probabilistic data structures can work about as well as the best deterministic ones, but can be much easier to implement
- Skip lists and randomized search trees are examples of randomized data structures
- We will look at skip lists in a moment
- But first, we will briefly discuss random number generation

Random number generation

- Random numbers are useful in many applications: cryptography, software testing, games, probabilistic data structures, etc.
- A truly random number would be truly unpredictable
- However, ordinary programs are deterministic, and therefore their outputs are predictable, if you know the inputs, and the algorithm!
- So, you can't write an ordinary program to generate truly random numbers; you generate *pseudo*random numbers
  - (you could connect your computer to a nonalgorithmic device and collect truly random bits from it; see /dev/random on Unix systems, or various web services)
  - A typical pseudorandom number generator (RNG) is initialized with a seed; if the seed is truly random, the RNG will have as much randomness as there is in the seed
- Though the generated numbers aren't truly random, they should appear quite random... what does that mean?

Tests for randomness

- A good pseudorandom number generator (RNG) will generate a sequence of numbers that will pass various statistical tests of randomness... the numbers will appear to be random, and can be used in many applications as if they were truly random

- Some good properties of a random sequence of numbers:
  - Each bit in a number should have a 50% probability of changing (1->0 or 0->1) when going from one number to the next in the sequence
  - The sequence of numbers should not repeat at all, or if the numbers do cycle, the cycle should be very (very!) long

- How to write a program that generates good pseudorandom numbers? One common approach is the *linear congruential* RNG

Linear congruential RNG's

- A typical pseudorandom number generator works like this:
  - The RNG is initialized by setting the value of a "seed"
    - The sequence of numbers generated is determined by the value of the seed; if you want the same sequence again, start with the same seed
    - If the seed has 64 bits, and those bits are picked truly randomly (e.g. by flipping a fair coin) then the RNG sequence has only 64 bits of "true randomness"
  - To generate the next pseudorandom number, the RNG updates the value of the seed, and returns it (or some part of it):

```
seed = F(seed);
return partOf(seed);
```

- The function `F` can take various forms. If it has the form

```
F(x)  { return  (a * x  +  c)  %  m ; }
```

- ... for some constants `a`, `c`, `m`, then this is a "linear congruential" RNG: it computes a linear (or affine) function of the seed, with the result taken congruent with respect to (i.e., modulo) `m`

  (Interesting to compare this to some hash functions used for strings...)

Linear congruential generators, cont'd

- How to choose the constants `a`, `c`, `m` for the generator?
- Not so easy to do! Many values for these constants produce RNG's whose sequences aren't very random at all
- The following values work well, and are used in some C/C++ standard library implementations:

```
a = 1103515245
c = 12345
m = 1<<31
```

- The generator in the java.util.Random class uses the following values. This RNG's low-order bits are not very random; they have a short cycle. So, computation is done with 64-bit longs, and the low order 16 bits are not exposed to the user:

```
a = 25214903917
c = 11
m = 1<<48
```

- Other fancier approaches are needed for highly-secure cryptography, etc.

Pseudorandom numbers in C++

- The `rand()` and `srand(int)` functions are defined in the C standard library, for use in C and C++ programs
- The `rand()` function returns pseudorandom numbers in the range `[0,RAND_MAX]`
  - `#include <cstdlib>` to define the const int `RAND_MAX`
- `srand(x)` sets the seed for `rand()`'s generator
- If `rand()` is called in your program before any call to `srand(int)`, it first calls `srand(1)`
- Call `srand(int)` at any time to reset the seed
  - call `srand(1)` to restart the default sequence
  - `#include <ctime>`, and call `srand(time(0))` to get a seed that depends on the current system clock time, but only to 1 second resolution

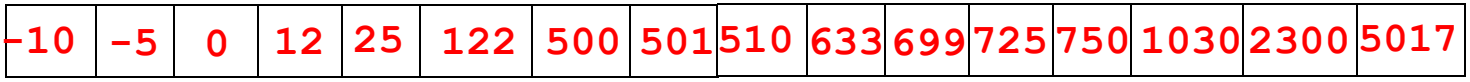
Toward skip lists

- Skip lists are a data structure that uses randomization in its construction

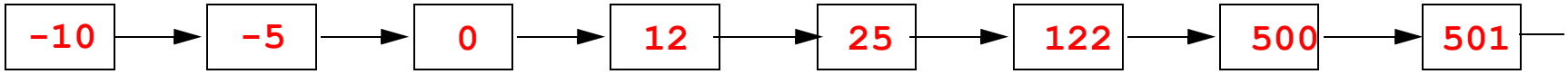
- One way to understand the idea of skip lists...
  - start with ordinary linked lists
  - then think about how to make search in a linked list more like binary search in an array

Fast searching in lists

- As you know, binary search in a sorted array is very fast: O(log N) worst-case
  - the direct-access property of arrays lets you compare to the "middle" element of the array in constant time; and then to the "middle" element of the appropriate remaining half of the array in constant time; etc.
  - each comparison eliminates (about) half of the remaining possibilities



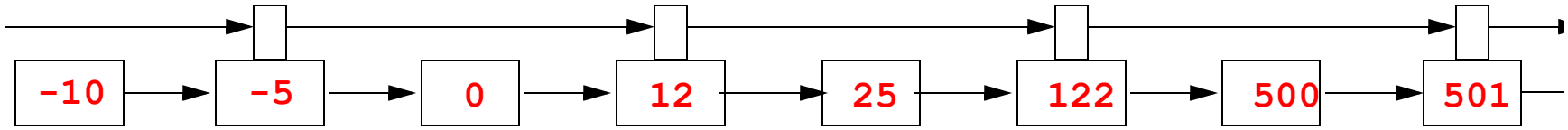
- However, search in a linked list is slow, even if the list is sorted: O(N) average case
  - the sequential-access property of linked lists forces a linear search



- How to make this faster?

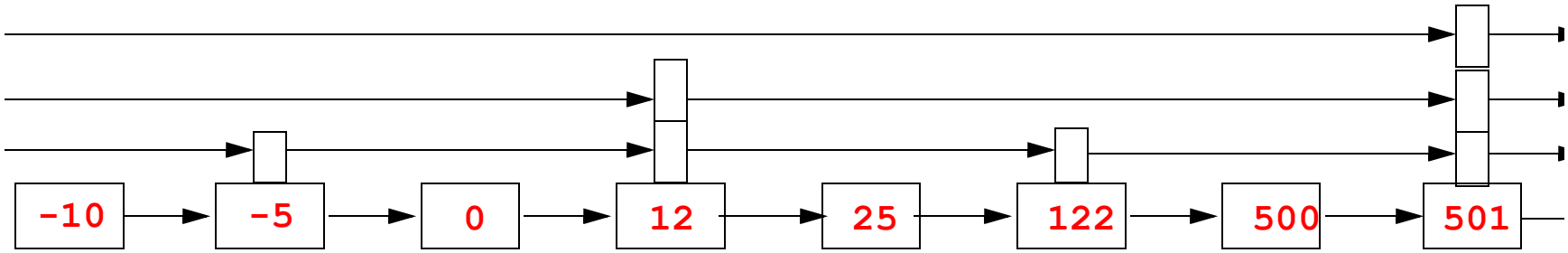
Fast searching in lists, cont'd

- Suppose every other list node, besides having a pointer to the next node in the list, also has a pointer two nodes ahead:



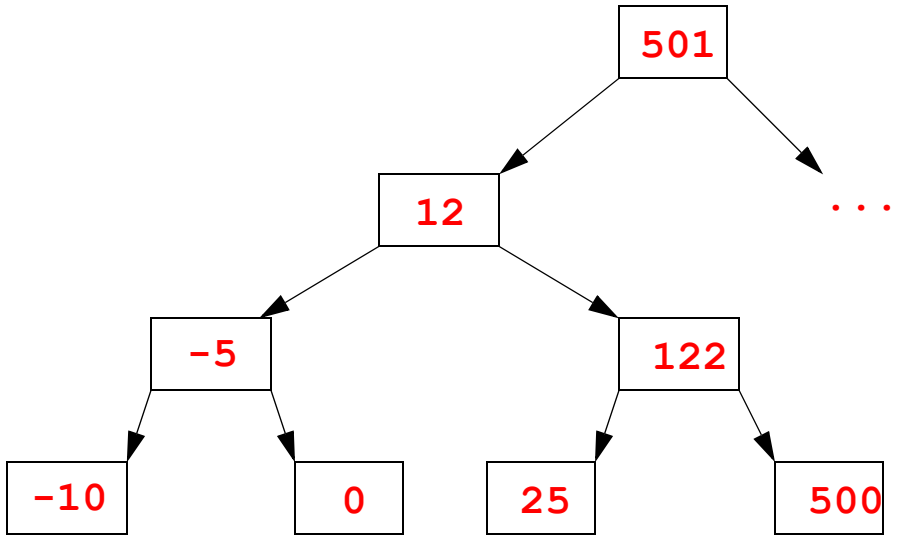
- ... then search can be about twice as fast as before:
  - Start following the "level 2" pointers, until it leads to a node with a key larger than the one you are looking for; then drop down to the "level 1" pointers
- If every fourth node also has a pointer four nodes ahead, search can be 4 times as fast:
  - Start following the "level 3" pointers, until it leads to a node with a key larger than the one you are looking for; then drop down to the "level 2" pointers, etc.
- If you fill this out, so that for J = 0,...,logN, $1/2^J$ of the nodes in the list have a pointer 2^J nodes ahead, you get a structure as shown on the next page
- ... with worst-case search cost O(log N)

Pointers for fast searching in a linked list



“Fast lists”, binary search and BST’s

- The added pointers in that enhanced “fast list” permits binary search in a linked list
- Note that a well-balanced binary search tree has the same property: each comparison eliminates about half of the remaining possibilities
- In fact, searching in that example structure will visit nodes in the same sequence as in this binary search tree:



Skip lists

- The enhanced linked list structure shown permits fast O(logN) search, worst case
- However, to preserve that perfect nice structure as an invariant when doing an insert or delete operation would require O(N) steps worst-case, because every node may need its pointers changed
- Pugh [1990] showed that in practice you really don't need to preserve a perfectly regular structure when doing insert and delete operations:
  - Instead, you can use a random number generator during insert operations, and create a randomized structure, and still almost certainly get very good performance
- The resulting structure is called a *skip list*
- Skip lists are relatively easy to implement, and have expected time cost O(logN) for search, insert, and delete
- They have worst case time cost O(N), but this worst case is extremely unlikely to occur if you use a good random number generator

Skip lists: some definitions

- In a skip list, nodes in the list contain different numbers of *forward pointers*
- The forward pointers in a node are stored in an array in that node, and they point to nodes farther down the skip list (or are null)
- The *level* of a node is defined to be the length of its array of forward pointers
- The skip list has associated with it a probability p, $0<p<1$. This determines the probability of the level of a new node
  - When creating a new node, flip a coin that has probability p of heads
  - Count the number of heads before the first tails, plus one: that is the level of the new node
  - Common choices for p are 1/2 and 1/4:
    - With $p = 1/2$, the probability of a new node being level 1 is 1/2; of being level 2 is 1/4; of being level 3 is 1/8; etc.
    - With $p = 1/4$, the probability of a new node being level 1 is 3/4; of being level 2 is 3/16; of being level 3 is 3/64; etc.

Skip list node level distribution: $p = 1/2$

- Suppose you have a completely filled binary tree with 2^K- 1 nodes
  - How many nodes are at the highest level (i.e. the root)? ____
    - What fraction is that of the total? ___
  - How many nodes are at the next highest level (i.e. children of the root)? ____
    - What fraction is that of the total? ___

  .....
  - How many nodes are at the lowest level (i.e. the leaves)? ___
    - What fraction is that of the total? ___

- Using 1-based counting of levels going up from the leaves, the fraction of nodes at level L from the leaves is $2^{(K-L)}/(2^K - 1)$, which is approximately $1/2^L$

- The idea behind a skip list with p = 1/2 is to approximate that distribution of nodes at different levels, in the hope of getting performance similar to a completely filled binary search tree

Skip list node level distribution: $p = 1/4$

- Suppose you have a completely filled 4-ary tree with $(4^K - 1)/3$ nodes
  - How many nodes are at the highest level (i.e. the root)? ____
    - What fraction is that of the total? ___
  - How many nodes are at the next highest level (i.e. children of the root)? ____
    - What fraction is that of the total? ___

  .....
  - How many nodes are at the lowest level (i.e. the leaves)? ___
    - What fraction is that of the total? ___

- Using 1-based counting of levels up from the leaves, the fraction of nodes at level L from the leaves is $4^{(K-L)}/(4^K - 1)/3$, which is approximately $3/(4^L)$

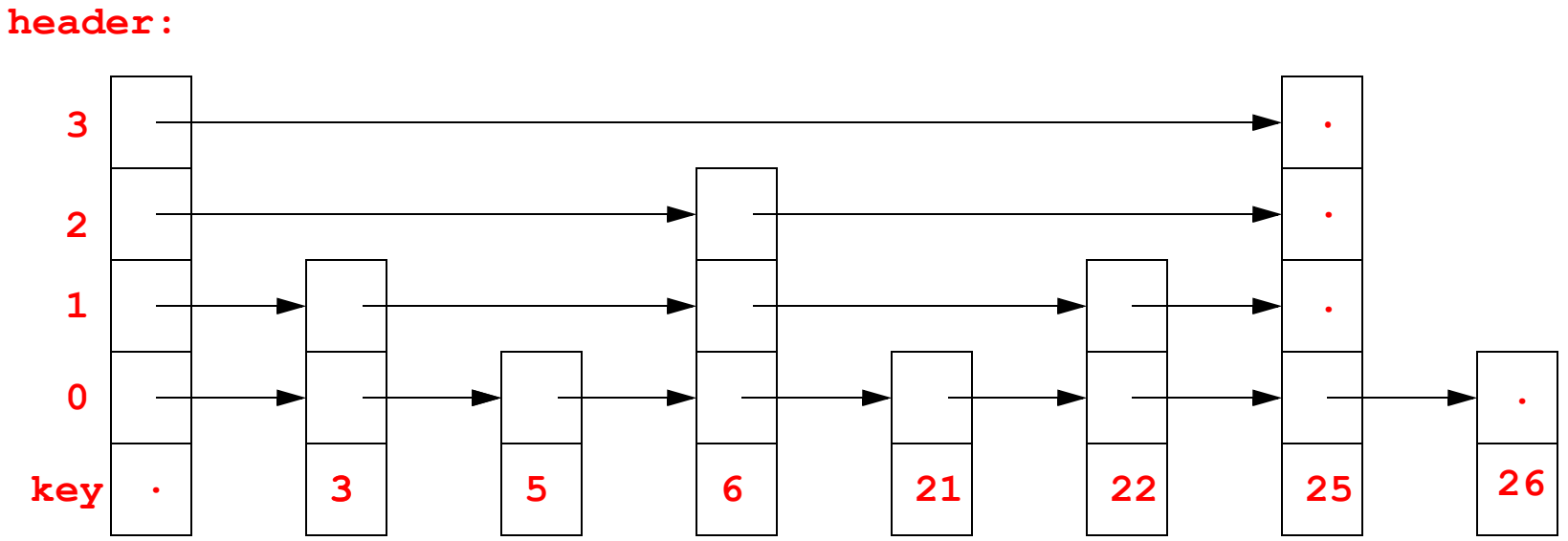
- The idea behind a skip list with p = 1/4 is to approximate that distribution of nodes at different levels, in the hope of getting performance similar to a completely filled 4-ary search tree

Skip lists: some definitions, cont'd

- Every node has at least one pointer (index 0 in that node's pointer array), and at most MAXLEVEL pointers
- MAXLEVEL should be chosen to be approximately $\log_{1/p}$ of the maximum number of items that will be stored in the skip list
  - if many more than that many items are stored in the skip list, performance will deterioriate somewhat
  - if many fewer than that many items are stored, you are wasting some space storing pointers that don't really help performance
- The index 0 forward pointer in a node points to the next node in the list (or null if none)
- The index *j* pointer points to the next node in the list of level greater than *j* (that is, the next node that contains a pointer with index *j*), or null if none
- A skip list has a header node that contains MAXLEVEL pointers, all initially null

Skip list: example

- Here is a skip list containing keys 3, 5, 6, 21, 22, 25, 26 with ordinary integer ordering
- Levels randomly generated for nodes containing those keys are 2,1,3,1,2,4,1
- In this example, MAXLEVEL == 4



- Note that the sequence of keys in the skip list is determined by the value of the keys only; the levels of the nodes are determined by the random number generator only

Insert in skip lists

- When inserting a key in a skip list, a new node is created to hold the key
- The sequence of nodes in a skip list is always maintained in key-sorted order, so the *postion* of the new node in the list is determined by the key values
- However, the *level* of the new node is selected randomly and independently of the value of the key:
  - the randomly selected level is a number in the range [1,MAXLEVEL]
  - the probability that the level will be 1 is (1-*p*)
  - the probability of a node that has a level *i* pointer NOT having a level *i*+1 pointer is (1 - *p*)
  - ... unless *i* == MAXLEVEL (because no nodes have pointers at level greater than MAXLEVEL)
- The level of a node is determined by a random number generator when the node is created, and, once determined, it is never changed (of course, the node may be deleted later)

Skip list implementation

- Skip list nodes need to hold a key, and an array or vector of forward pointers
- The keys need to be comparable to each other
- The length of the pointer vector is equal to the “level” of the node

```
template <typename T>
class SkipNode {

   friend class SkipList; // SkipList class can get access to
                          // all SkipNode members

private:

   T* key;  // pointer to the key in this node
   std::vector<SkipNode<T>*> forward; // vector of forward pointers

   SkipNode(T* key, int level) :
      key(key), forward(level,(SkipNode<T>*)0) {}
};
```

The STL std::vector container

- `std::vector` is a very useful, high performance container class
- You can think of a vector as a 'safe', resizeable array

- Some vector features we will make use of:

  - A constructor that takes two arguments (there are also other constructors...):
    - an unsigned int, saying how many elements the vector has
    - a value that will be used to initialize those elements of the vector

  - `operator[]` overloaded so a vector can be indexed like a native array

  - member function `at(unsigned int i)` which acts like the indexing operator[], but which does runtime index bounds checking

Skip list implementation, cont'd

- The skip list will need a vector of MAXLEVEL forward pointers. This can be implemented using a dummy header SkipNode with null data

```
template <typename T>
class SkipList {

private:
   SkipNode<T> header;
   const int MAXLEVEL;
   float prob;

public:
   SkipList(int maxlevel, float p) :
      MAXLEVEL(maxlevel), header((T*)0, maxlevel)
     {  prob = p < 1.0 && p > 0.0 ? p : 0.5; }

   void insert(T* key);  // will define this later...
};
```

Random level generation

- Use a random number generator to decide the level of a new node to be inserted in the skip list
- Flip a coin that has probability *p* of being heads: the number of heads in succession up to the first tails, plus one, will be the randomly selected level
- So, probability of a node being level 1 is 1-*p*, probability of being level 2 is *p(1-p)*, probability of being level 3 is $p^2(1-p)$, etc.
- (Need to make sure we don't exceed MAXLEVEL)

```
private:
   /** Return a random double in range [0,1) */
   double drand() {
      return rand() / (RAND_MAX + 1.0);
   }

   int randomLevel() {
      int level;
      // note the empty loop body!
      for(level = 1; drand() < prob && level < MAXLEVEL; level++) ;
      return level;
   }
```

Find in skip lists

- To find key *k*:

  - 0. Start at list header, and consider it to be the current node *x*. Find the highest index pointer in the header that is not null; let this index be *i*.
  - 1. While the pointer in node *x* indexed *i* is not null and points to a node containing a key smaller than *k*, set that node pointed to to be the current node *x* (i.e. follow the pointer).
  - 2. If the pointer in node *x* indexed *i* points to a node containing a key equal to *k*, you have found the key *k* in the skip list and you are done.
  - 3. If the pointer in node *x* indexed *i* points to a node containing a key larger than *k*, decrement *i* (i.e., keep *x* as the current node but "drop down" to the pointer with next smaller index).
  - 4. If now $i < 0$ (i.e., you have run out of pointers in the current node), the key *k* is not in the skip list and you are done. Else, go to 1

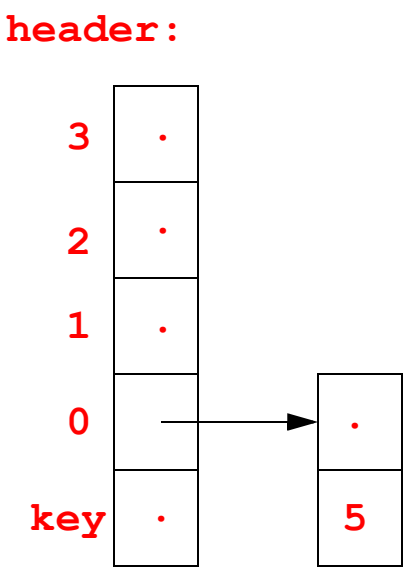
Insert in skip list

- Basic idea:
  - Use the skip list search algorithm to find the place in the skip list to insert the new key (recall that keys are kept in sorted order in the list)
  - While searching, use a local array of "update" pointers to keep track of previous nodes at each level, so their pointers can be easily updated when you splice in the new node
  - Create a new node with a random level to hold the new key, and splice it in

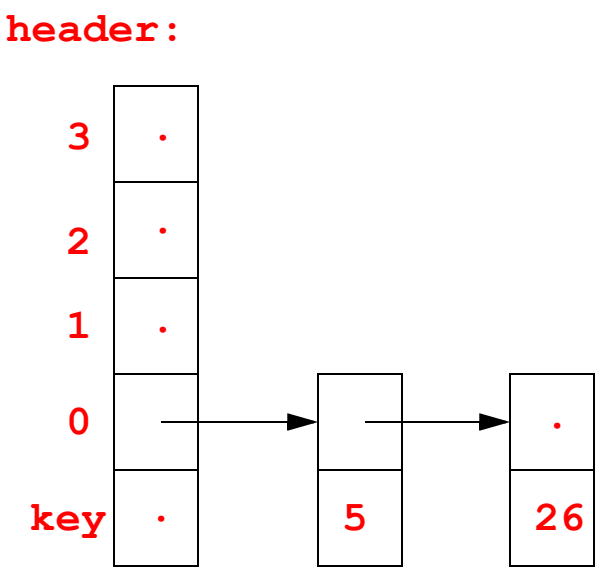
Inserting in a skip list: example

- Starting with an empty skip list with MAXLEVEL 4, insert these keys, with these "randomly generated" levels
  - 5, with level 1
  - 26, with level 1
  - 25, with level 4
  - 6, with level 3
  - 21, with level 1
  - 3, with level 2
  - 22, with level 2

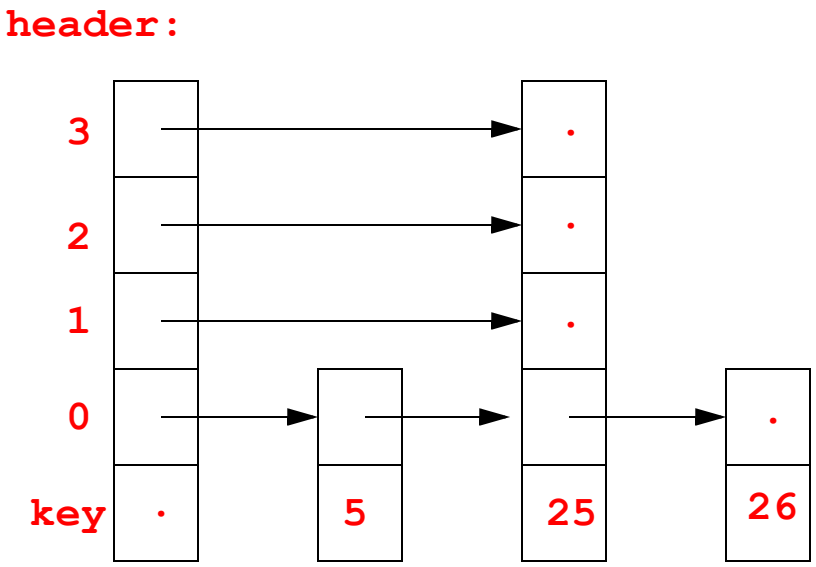
Insert 5 with level 1



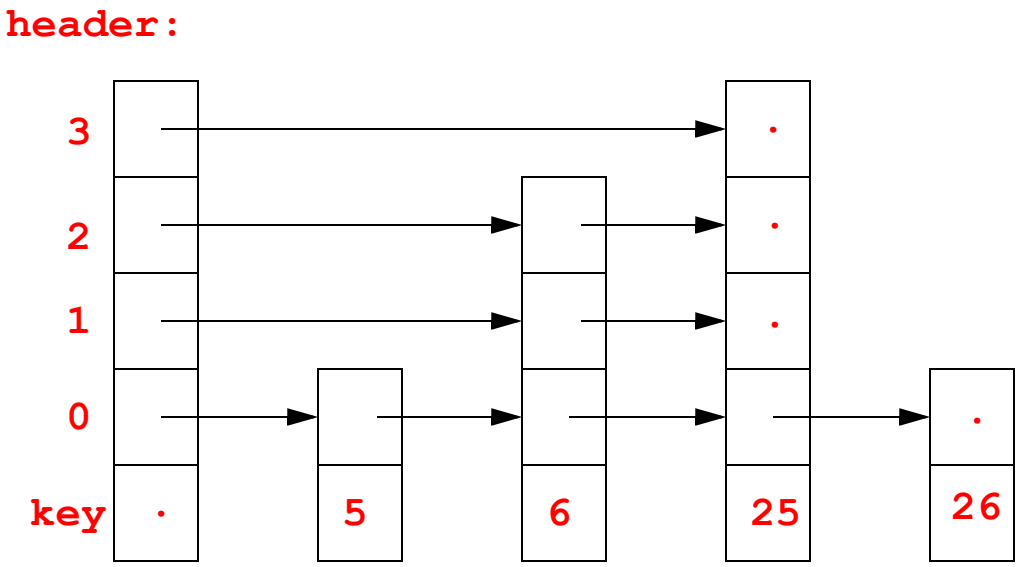
Insert 26 with level 1



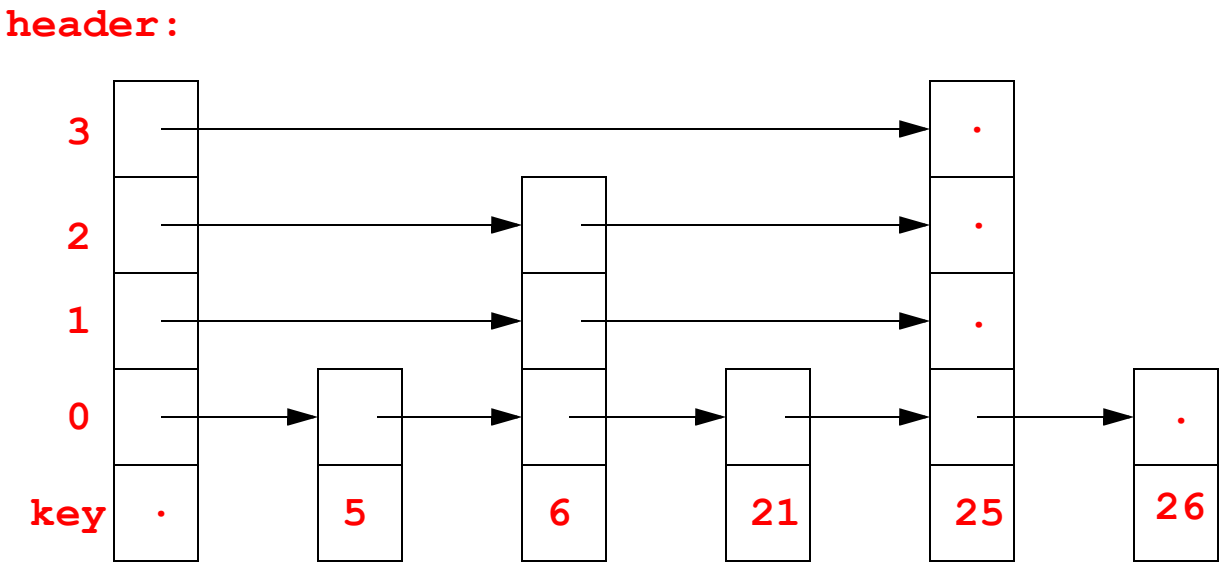
Insert 25 with level 4



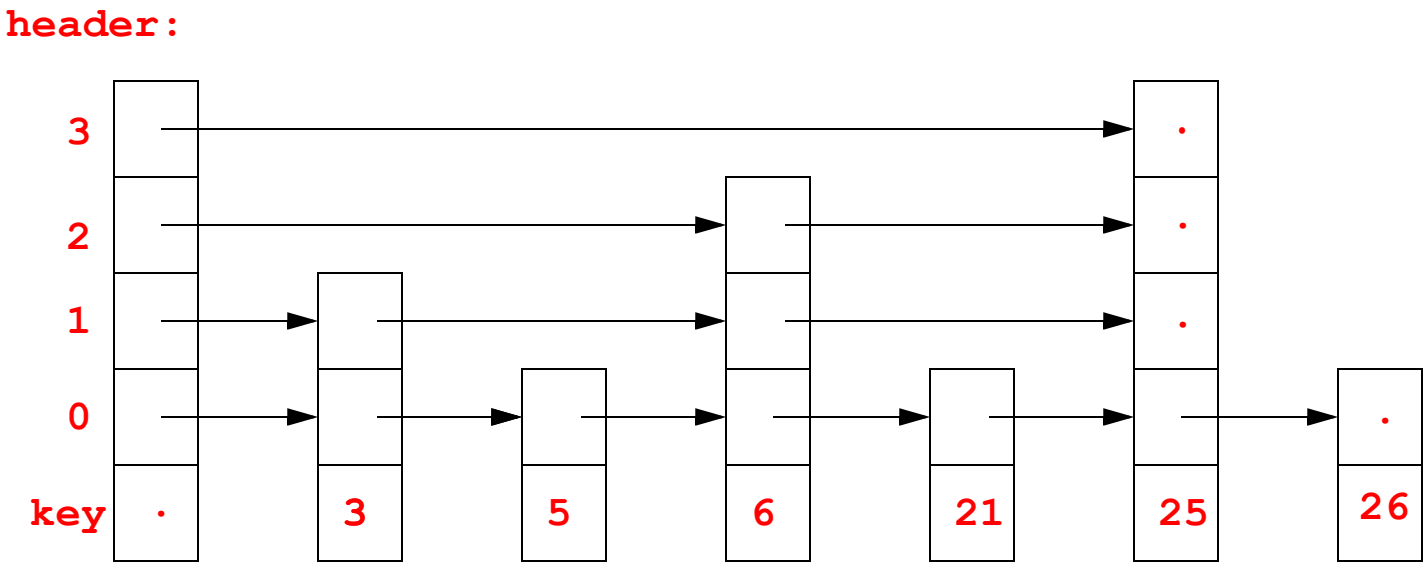
Insert 6 with level 3



Insert 21 with level 1



Insert 3 with level 2



Inserting 22 with level 2

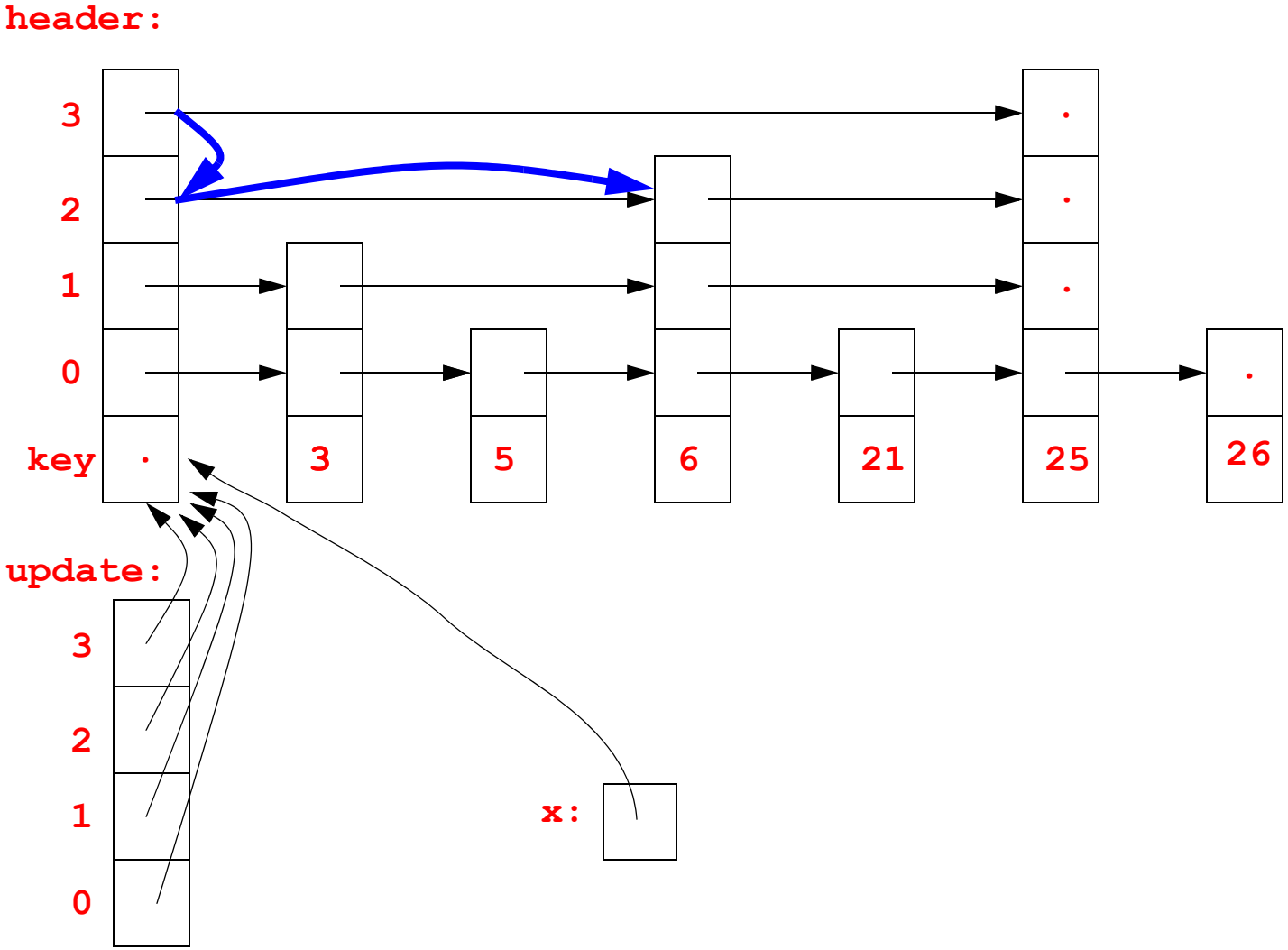
- Let's look at the example of inserting this next key in more detail

- The insert algorithm maintains two local variables (besides the skip list header):
  - `x`, a pointer which points to a node whose forward pointers point to nodes whose key we are currently comparing to the key we want to insert
    - this lets us quickly compare keys, and follow forward pointers
  - `update`, a vector of node pointers which point to nodes whose forward pointers may need to be updated to point to the newly inserted node, if the new node is inserted in the list just before the node `x` points to
    - this lets us quickly update all the pointers necessary to splice in the new node

- 6 keys have been inserted, and see how key 22 is inserted in a new node with level 2

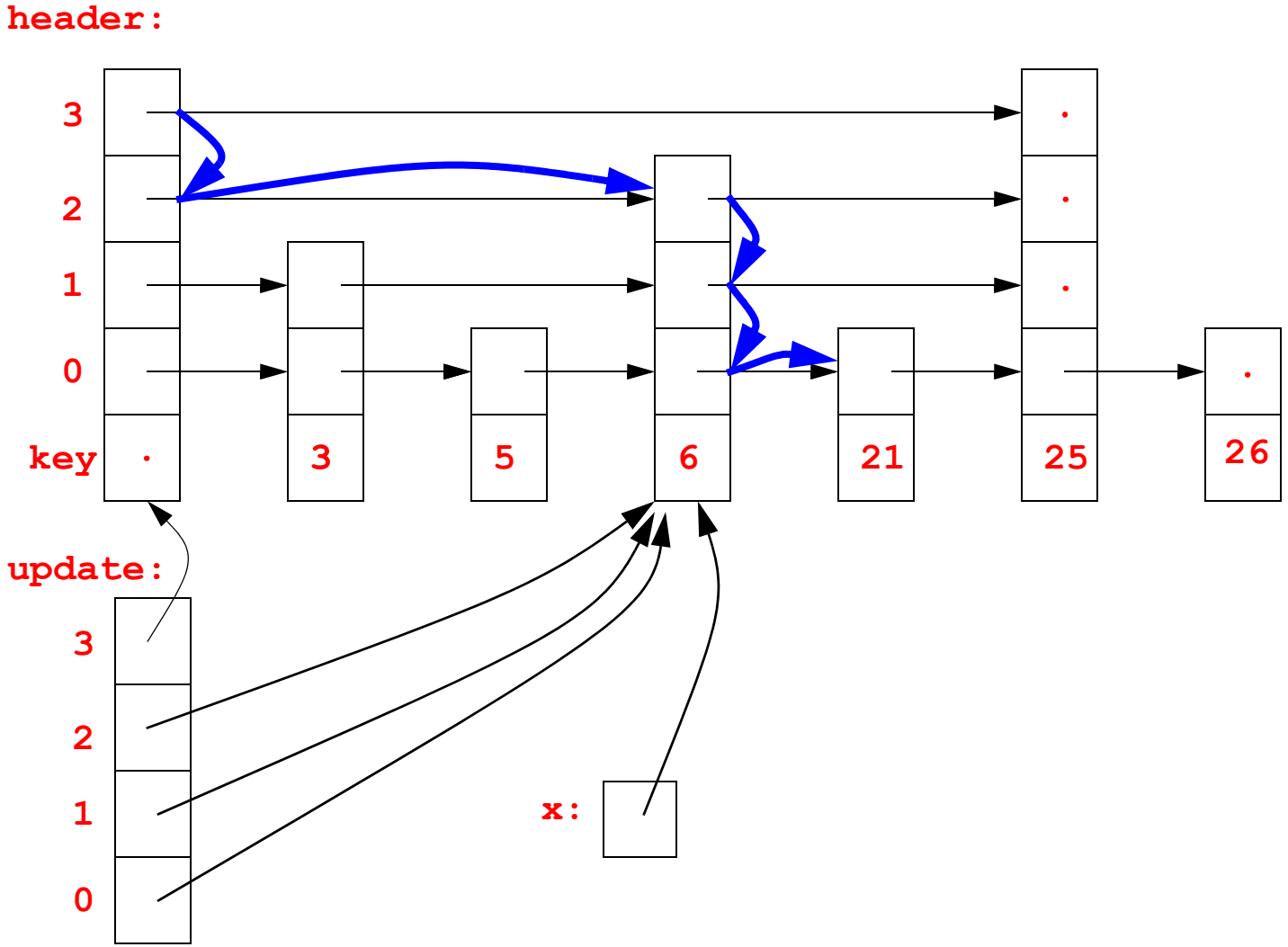
Inserting 22 with level 2: frame 1

- Follow top-level pointer: 25 > 22, so drop down and follow pointer: 6 < 22, so update



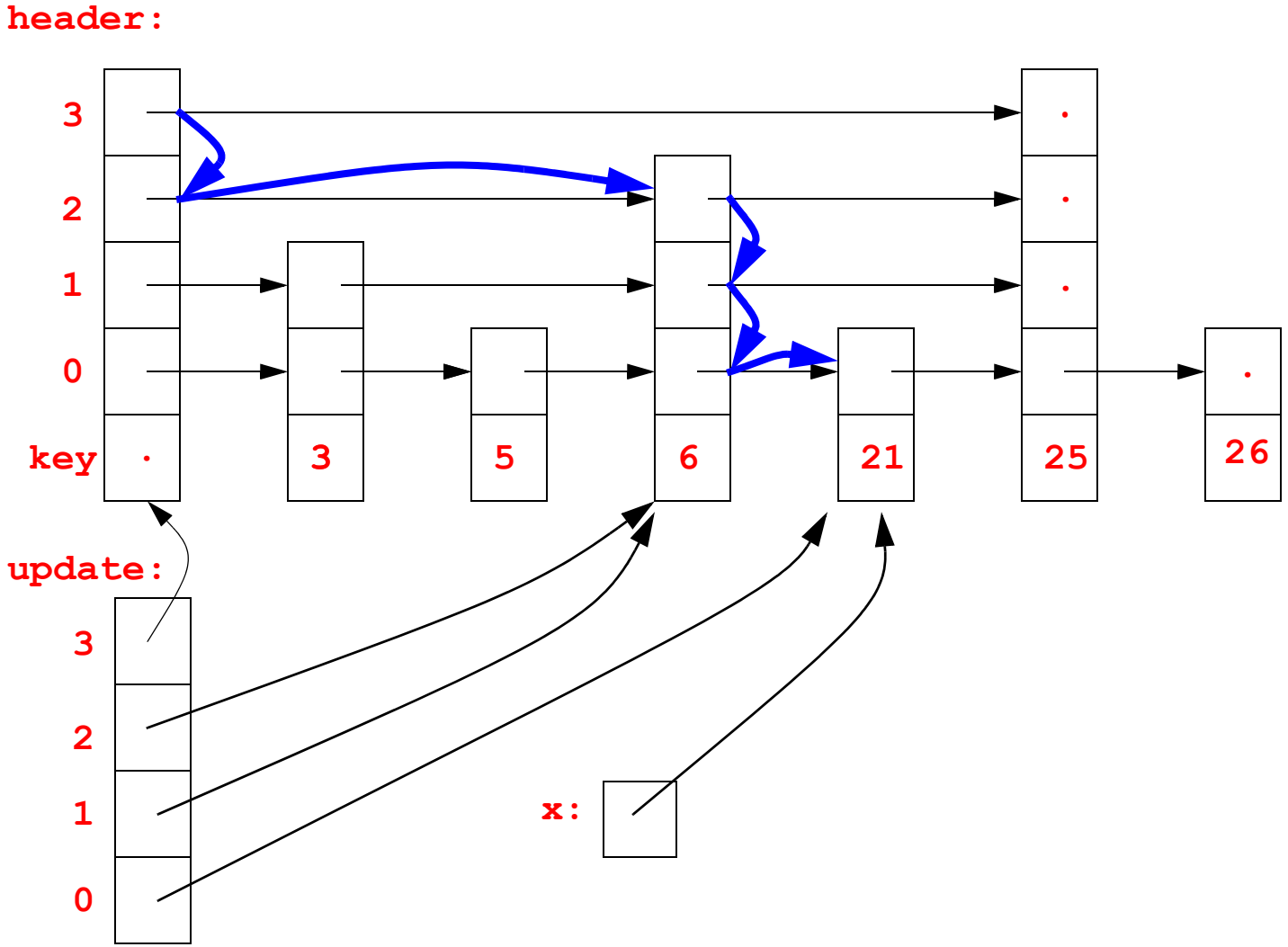
Inserting 22 with level 2: frame 2

- Follow pointer: 25 > 22, so drop down twice and follow pointer: 21 < 22, so update



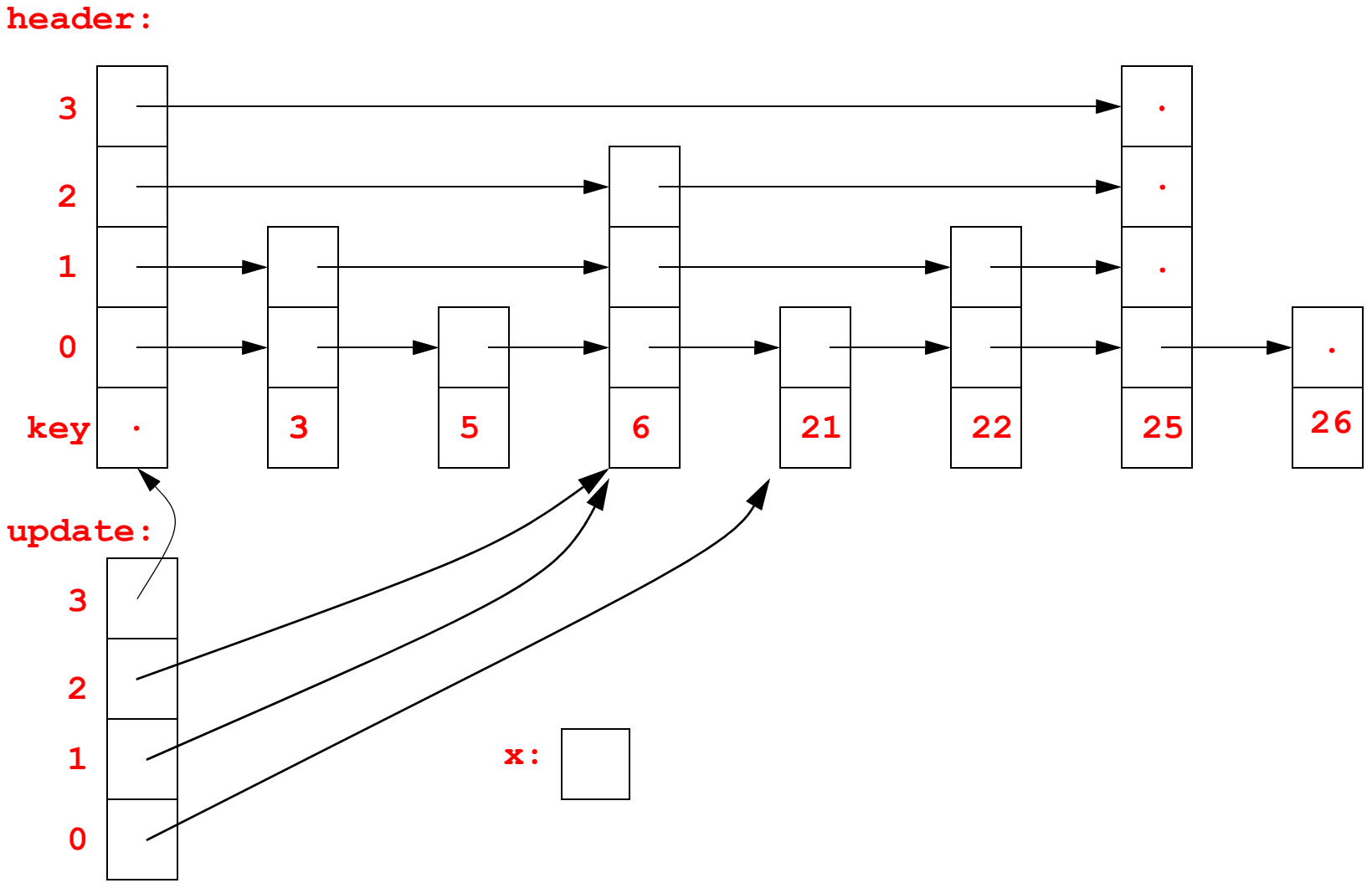
Inserting 22 with level 2: frame 3

- Follow pointer: 25 > 22 and we are at lowest level, so we have found insert point

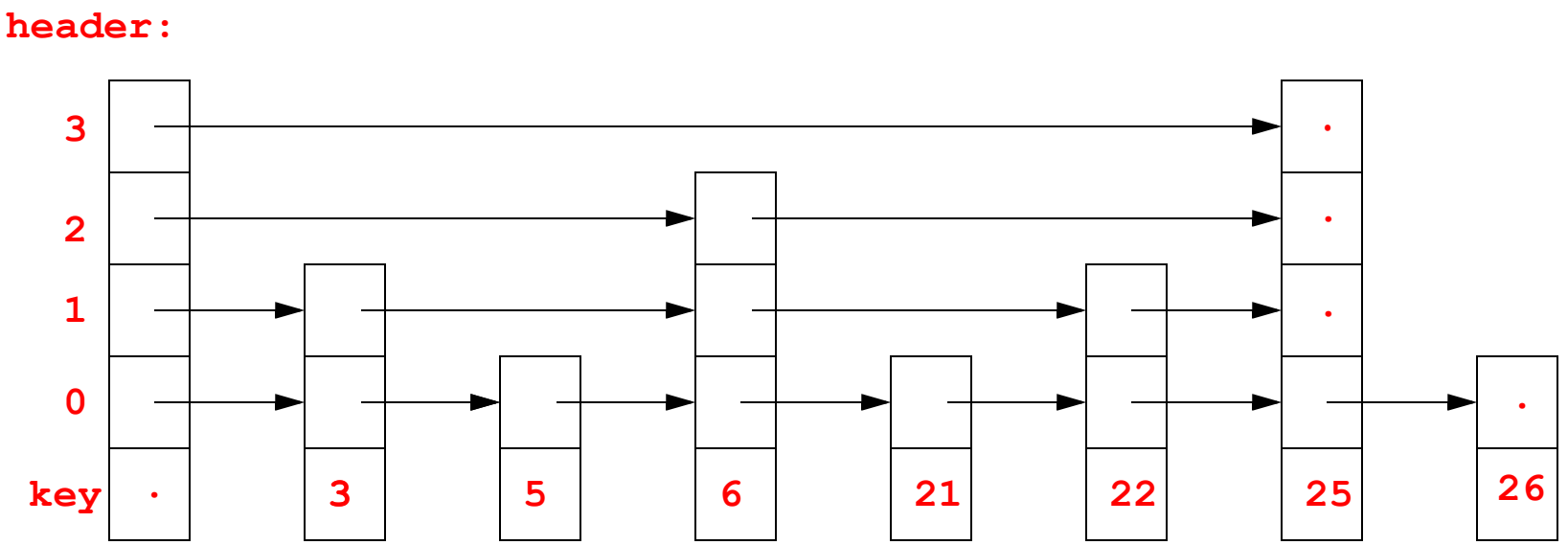


Inserting 22 with level 2: frame 4

- The update array shows which nodes' forward pointers are involved in the insert



Here is the skip list after all keys have been inserted:



- Note that the ordering of the keys in the skip list is determined by the value of the keys only; the levels of the nodes are determined by the random number generator only

Insert in skip lists: code

```
template <typename T>
void SkipList<T>::insert(T* key) {
   std::vector<SkipNode<T*>> update(MAXLEVEL,0); // "update" vector
   SkipNode<T>* x = &header;  // start at dummy header node
   for(int i=MAXLEVEL-1; i>=0; i--) { // search for insert posn.
      while( x->forward.at(i) &&
             *(x->forward.at(i).key) < *key ) {
         x = x->forward.at(i);  // follow pointer with index i
      }
      update[i] = x;  // keep track of end of chain at index i
   }
   int newLevel = randomLevel();  // generate level of the new node
   x = new SkipNode<T>(key, newLevel); // create new node

   // splice into list
   for (int i=0; i<newLevel; i++) {
      x->forward[i] = update[i]->forward[i]; // who x points to
      update[i]->forward[i] = x;  // who points to x
   }
}
```

Delete in skip lists

- Basic idea, similar to insert:
  - Use the skip list search algorithm to find the node to delete from the list
  - The `update` array is important here too: it keeps track of nodes whose pointers may need to change to "unsplice" the to-be-deleted node

Properties of skip lists

- It is possible for a skip list to be badly "unbalanced", so that searching in it is like searching in a linked list
  - for example, this will happen if the levels of the nodes in the list are nonincreasing (or nondecreasing) as you move from the beginning of the list to the end
- So, it is possible that the time cost of operations in a skip list will be $O(N)$
- However, Pugh shows that the expected number of comparisons to do a find in a skip list with N nodes using probability p for generating levels is

$$E[comparisons] \leq \frac{\log_{1/p} N}{p} + \frac{1}{1-p}$$

- (For $p = 1/2$, this implies that the average number of comparisons should be no more than about $2 \log_2 N$, which can be compared to about $1.386 \log_2 N$ for a randomized search tree)
- And so the expected time costs for insert and delete are also $O(\log N)$

Properties of skip lists, cont'd

- The expected time costs are like average time costs, averaged over many constructions of a skip list with the same N keys, but different randomly chosen node levels
- An interesting question is: For a single skip list with N keys, how likely is it that its height is much greater than log N?
- Pugh analyzed this question and, though he did not publish a formula to compute it, he shows graphs which indicate that the answer is:
  - it is possible, but it can be considered extremely unlikely

- Overall, skip lists are comparable to randomized search trees in
  - time costs of the basic operations
  - memory cost
  - implementation complexity

Next time

- Red-black trees
- Red-black tree properties
- Insert in red-black trees: rotations and recolorings

Readings: Weiss, Ch. 12 section 2