Skip Lists

Sridhar Alagar

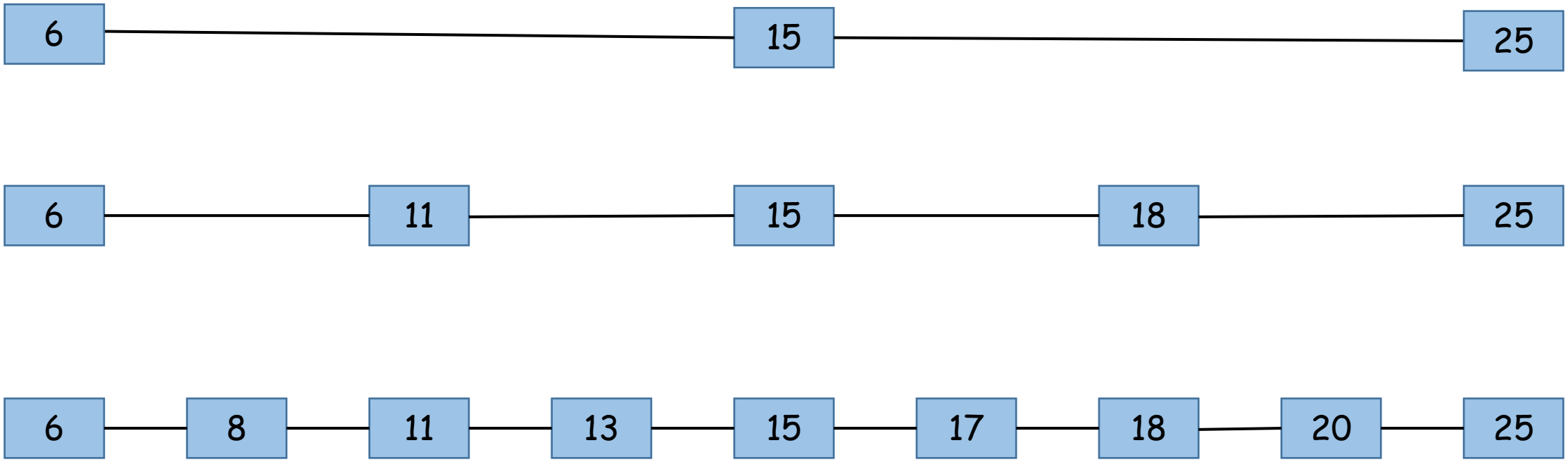
Motivation

Operations	Linked List (sorted)
delete	O(n). But O(1) if reference to node available
insert	O(1)
search	O(n)

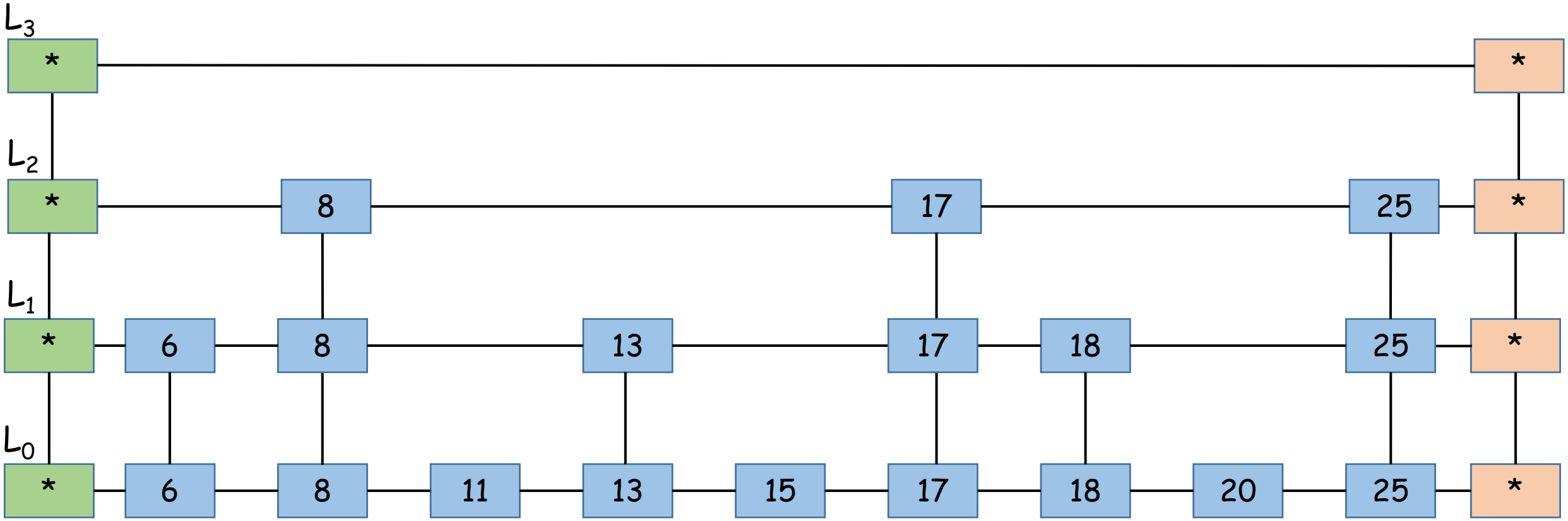
Linked List is a simpler structure compared to BST

Can we search in O(log n) time in Linked List?

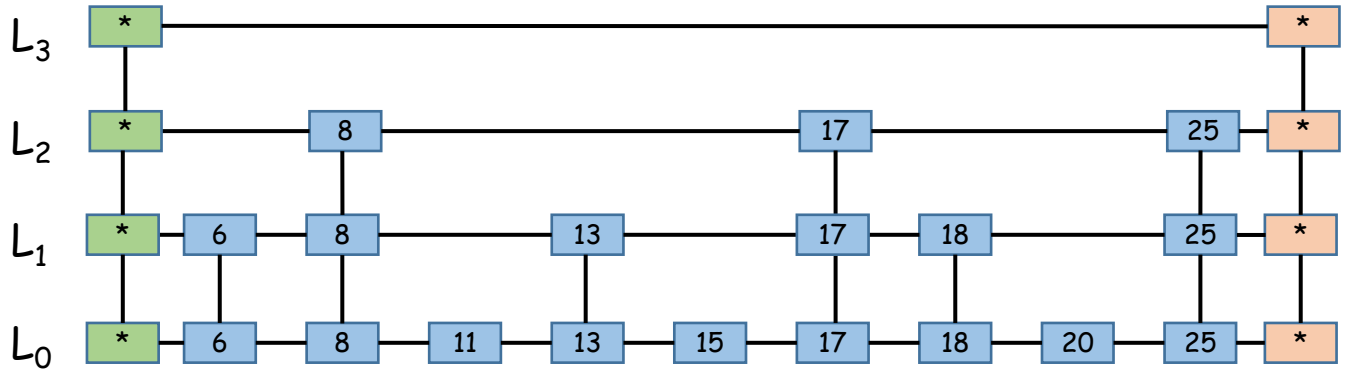
Mimic Binary Search in Linked List?



Skip List



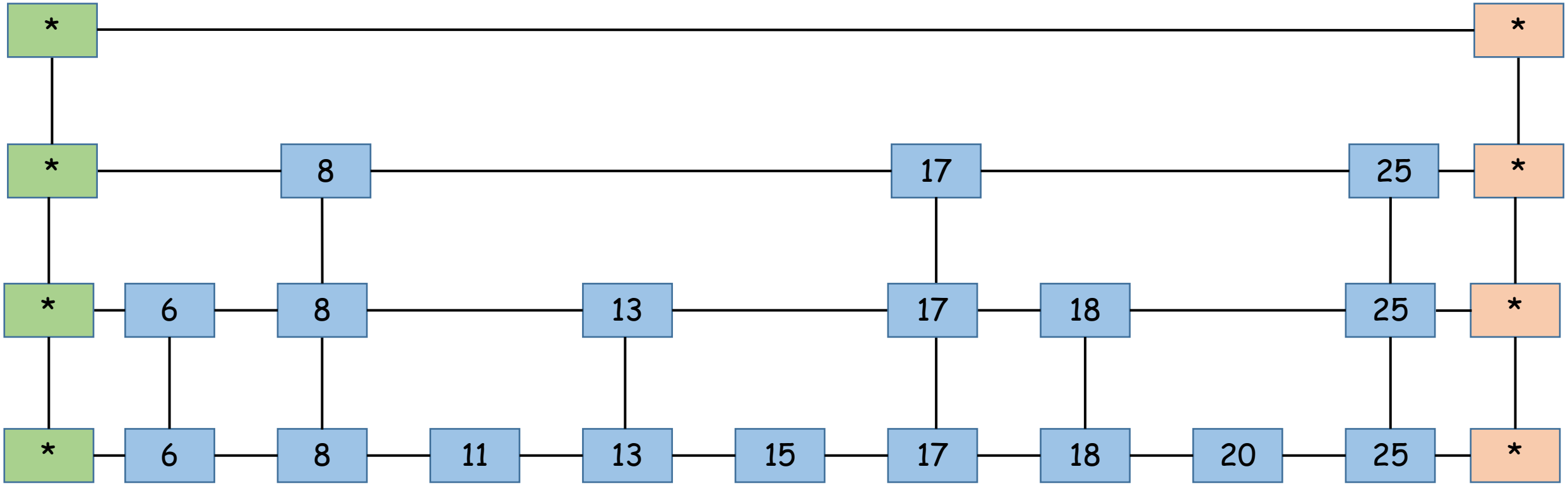
Skip List



- A skip list is a series of lists L_0 to L_h
- Each list has two special nodes head and tail
- L_{i+1} is a subset of L_i
- L_0 has all the elements
- L_h contains only special nodes
- Same elements present in L_{i+1} and L_i, have references to each other
- Each list is sorted (non-decreasing order)
- Height of a node is the number of lists in which it appears
  - Height chosen randomly

Skip List – search(15)

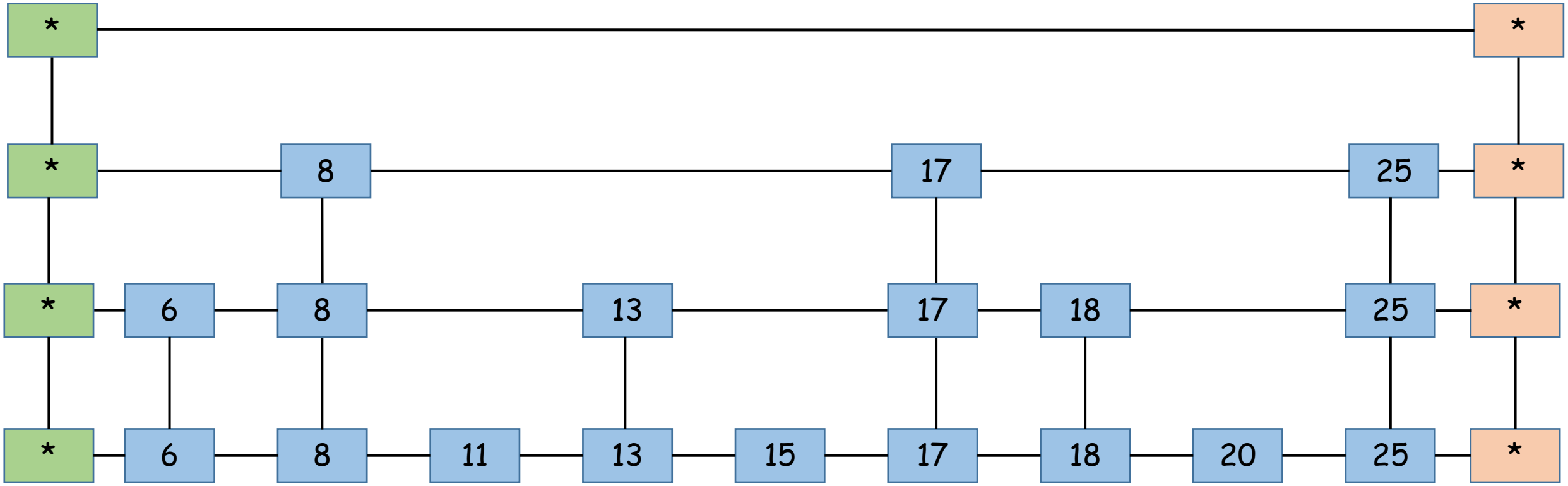
Series of moves to right and down



Expected length of the search path is $O(\log n)$

Skip List - search(12)

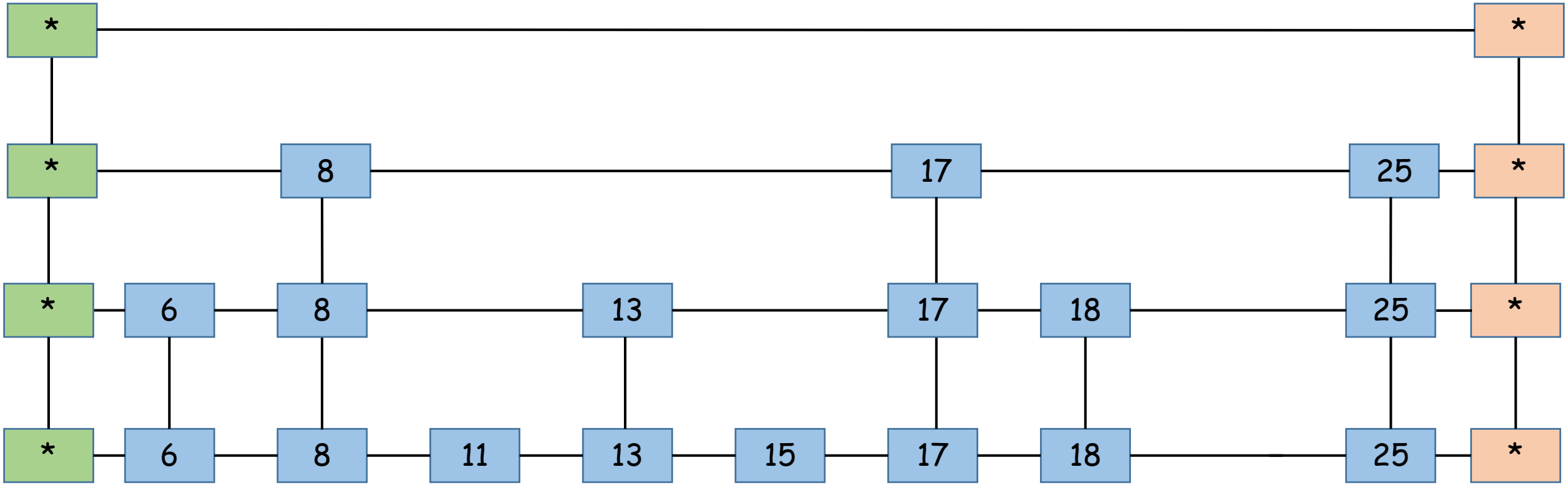
Series of moves to right and down



Expected length of the search path is O(log n)

Skip List – add(19)

Search for 19, and insert where it fails



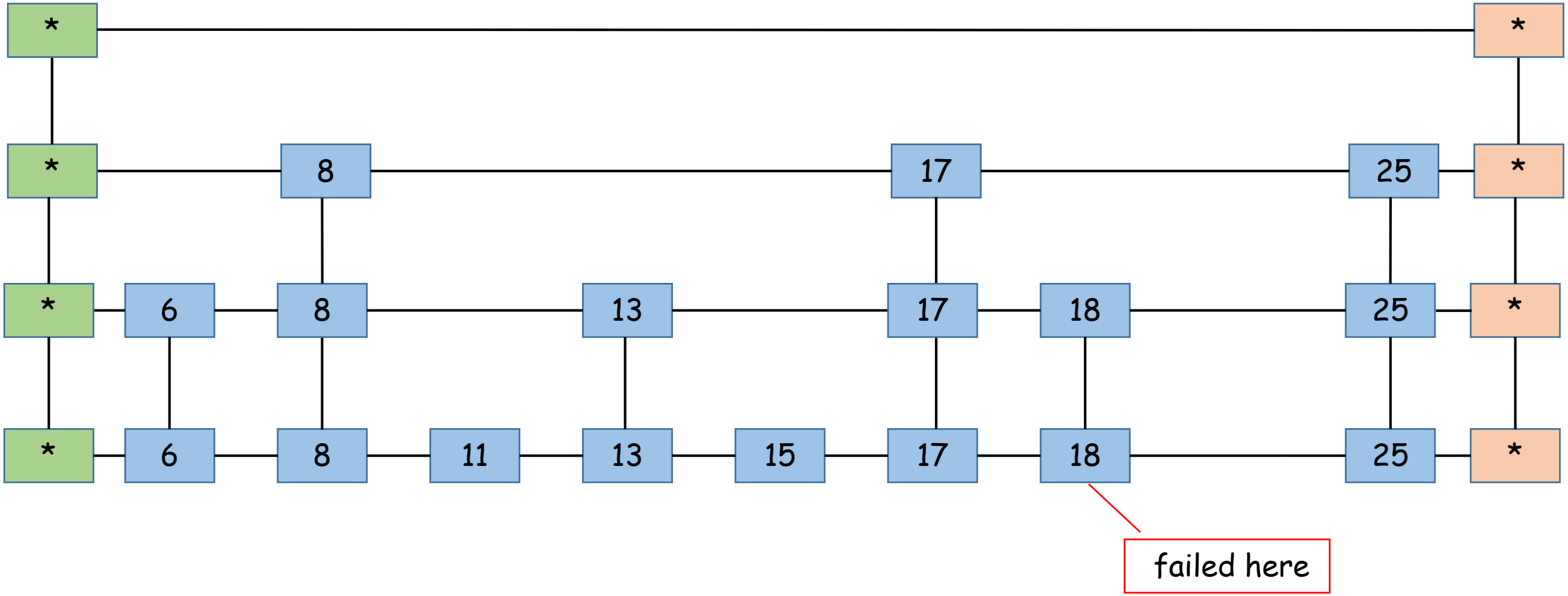
Skip List – add(19)

How to insert?

In how many lists?

Where in each list?

Search for 19, and insert where it fails



Skip List – add(19)

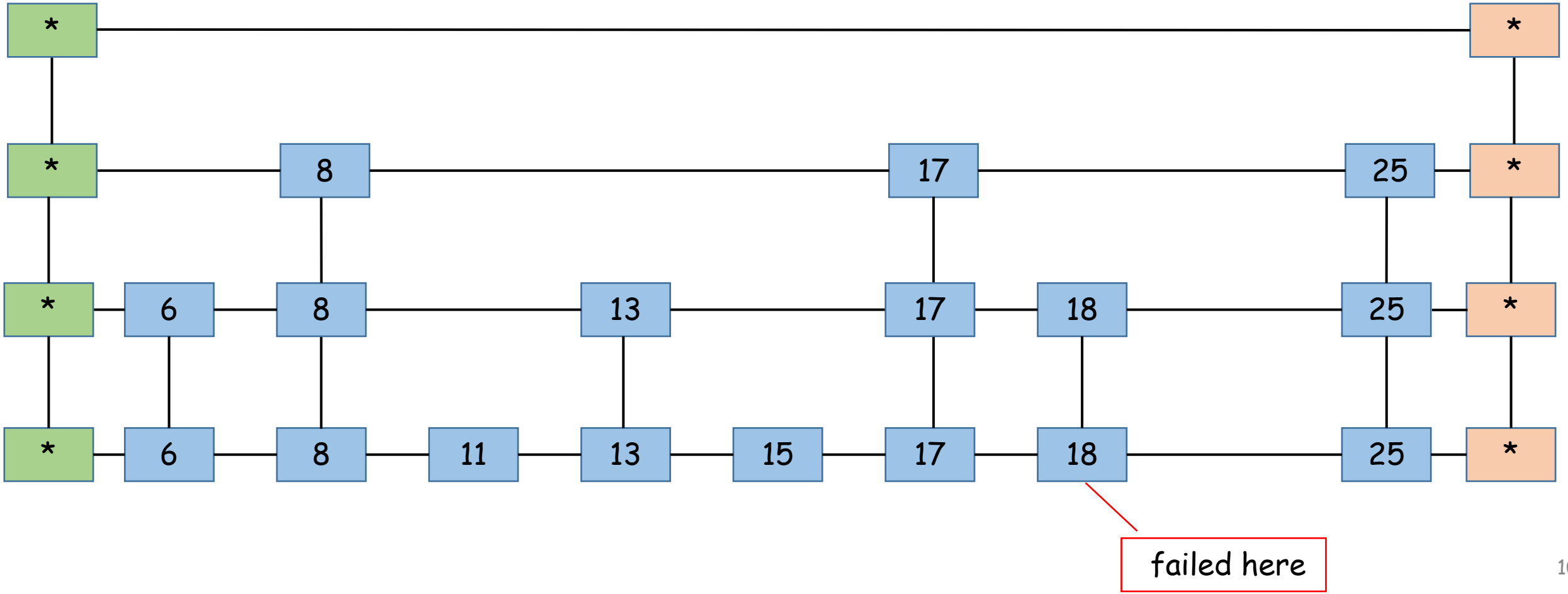
Search for 19, and insert where it fails

In how many lists?

determine height randomly

Where in each list?

traceback the route and find last entry visited at each level



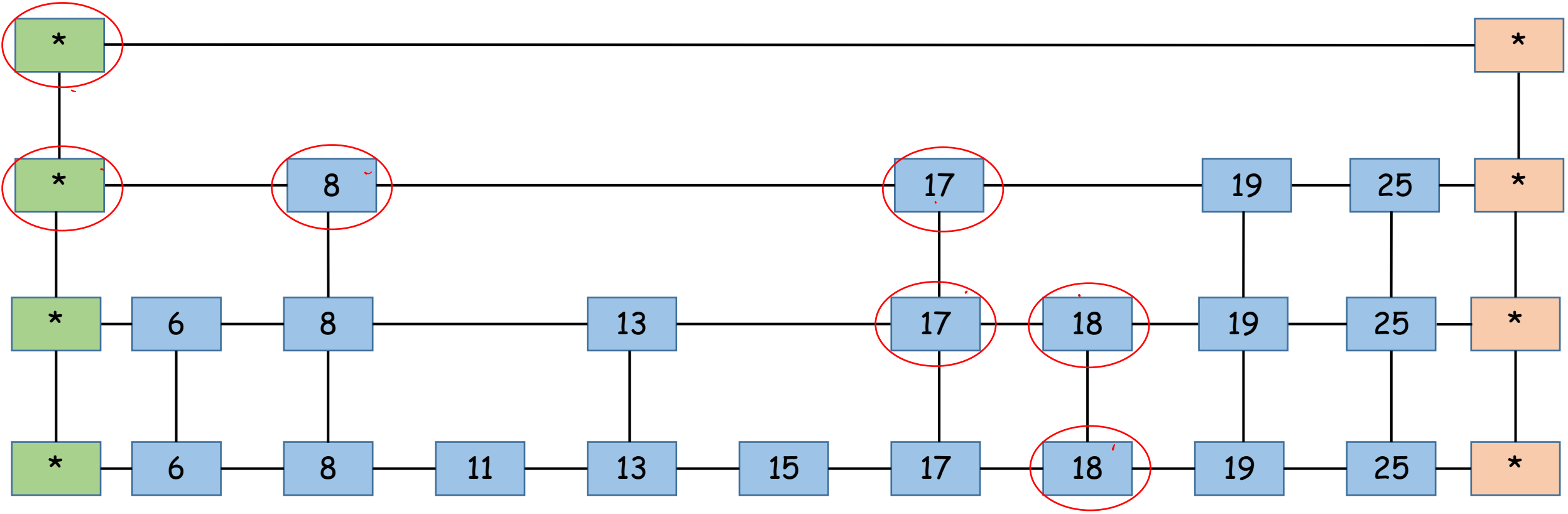
Skip List – add(19)

In how many lists?

determine height randomly

Where in each list?

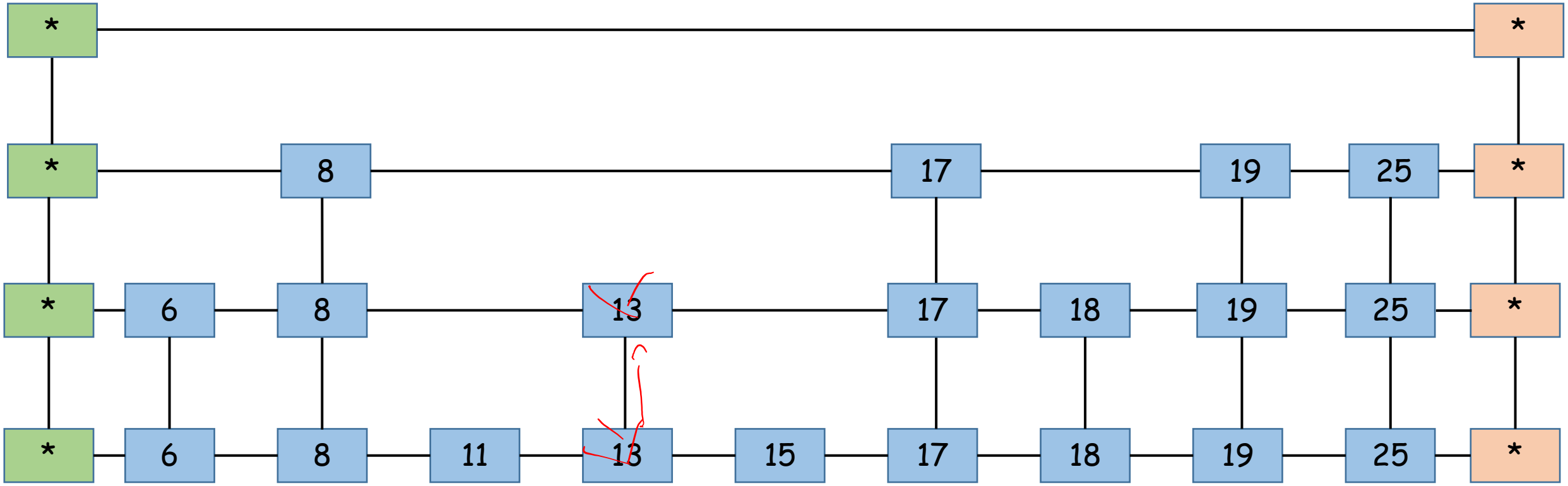
traceback the route and find last entry visited at each level



Expected RT is O(log n)

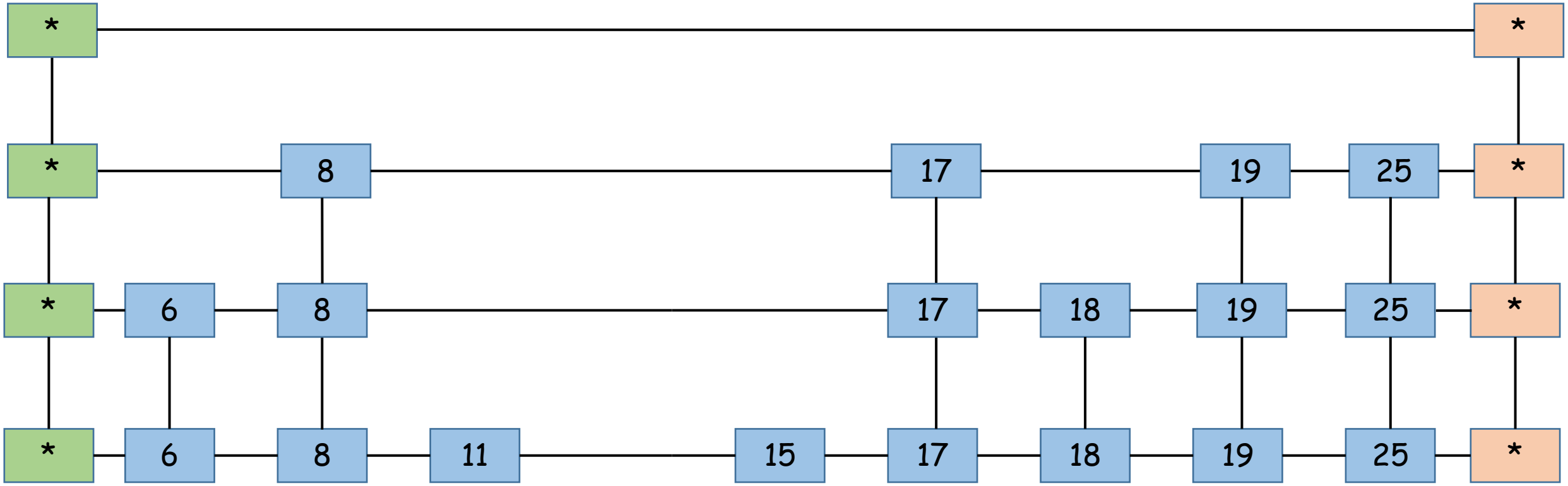
Skip List - remove(13)

Search for 13 and remove



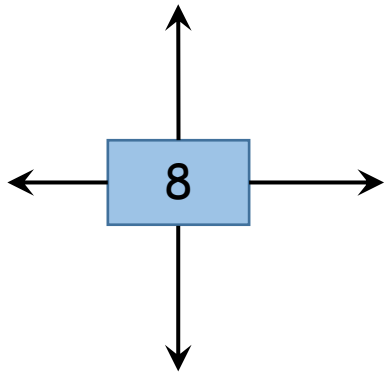
Skip List – remove(13)

Search for 13 and remove



Expected RT is O(log n)

Structure of node

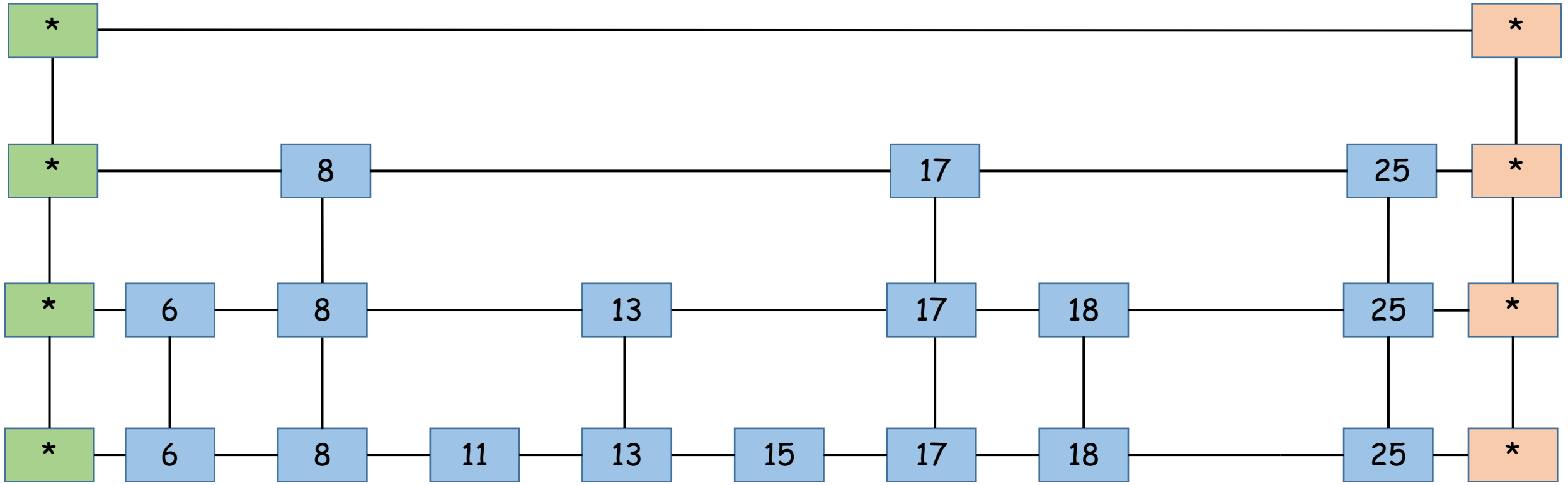


- Can we be space efficient?
- Can we avoid trace back?

Skip List – add(19)

Do not want to trace back

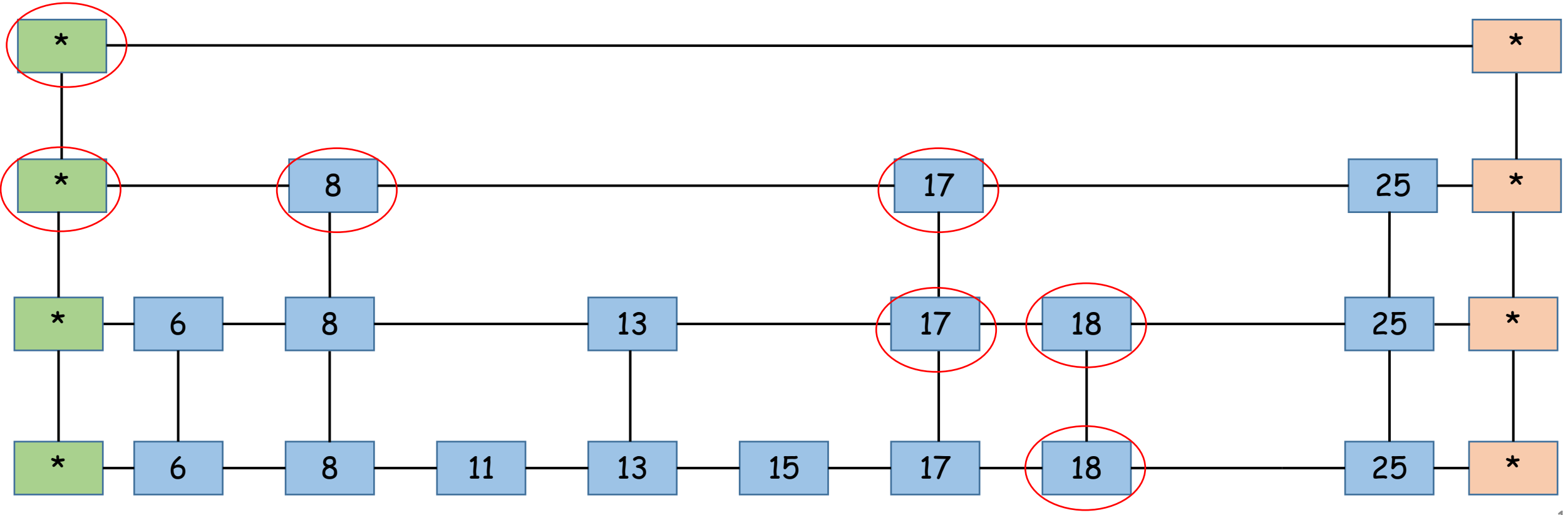
Remember the path



Skip List – add(19)

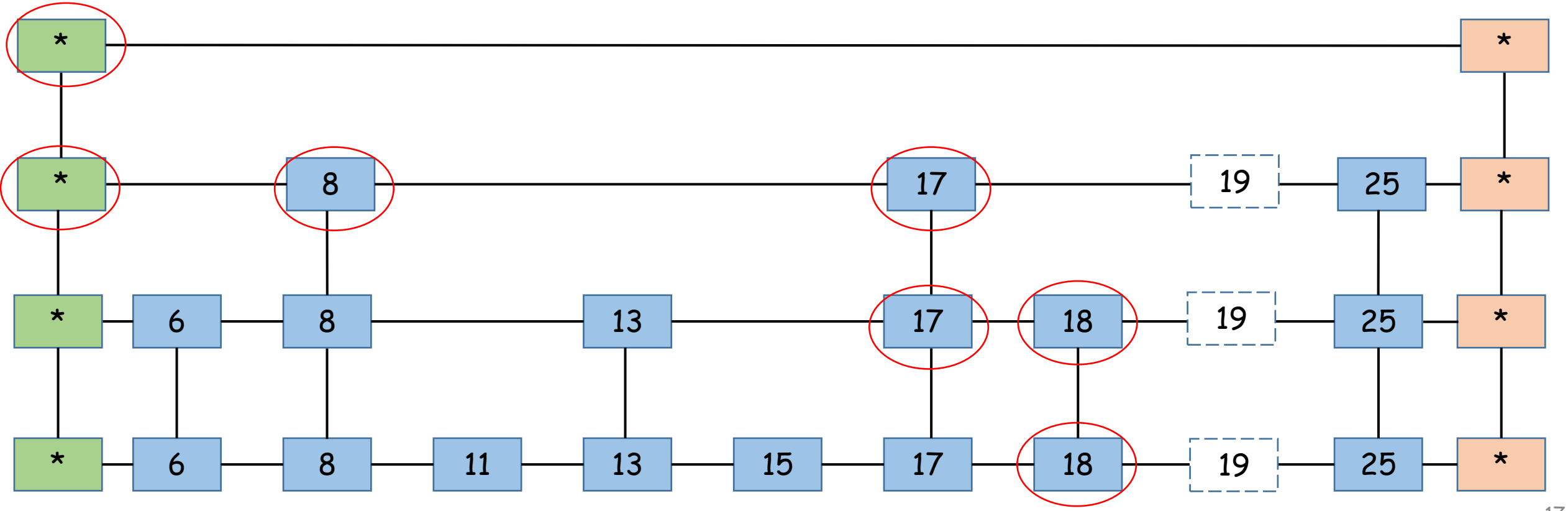
Do not want to trace back

Remember the path



Skip List – add(19)

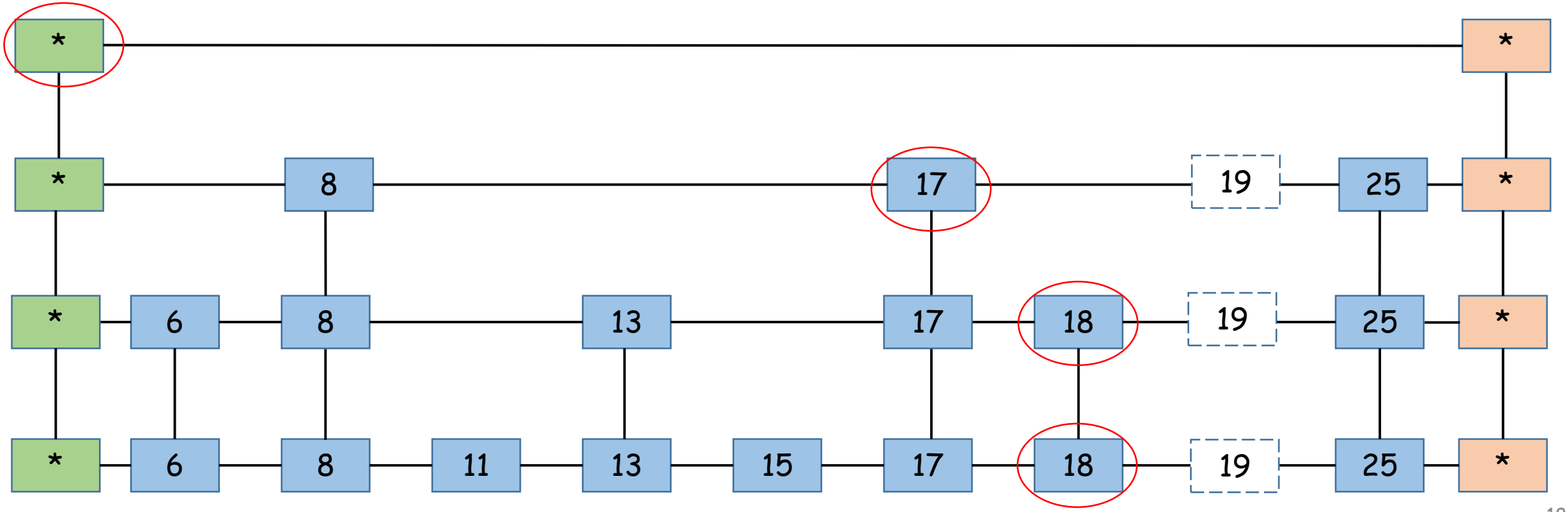
Do we need to remember the entire path?



Skip List – put(19)

Do we need to remember the path?

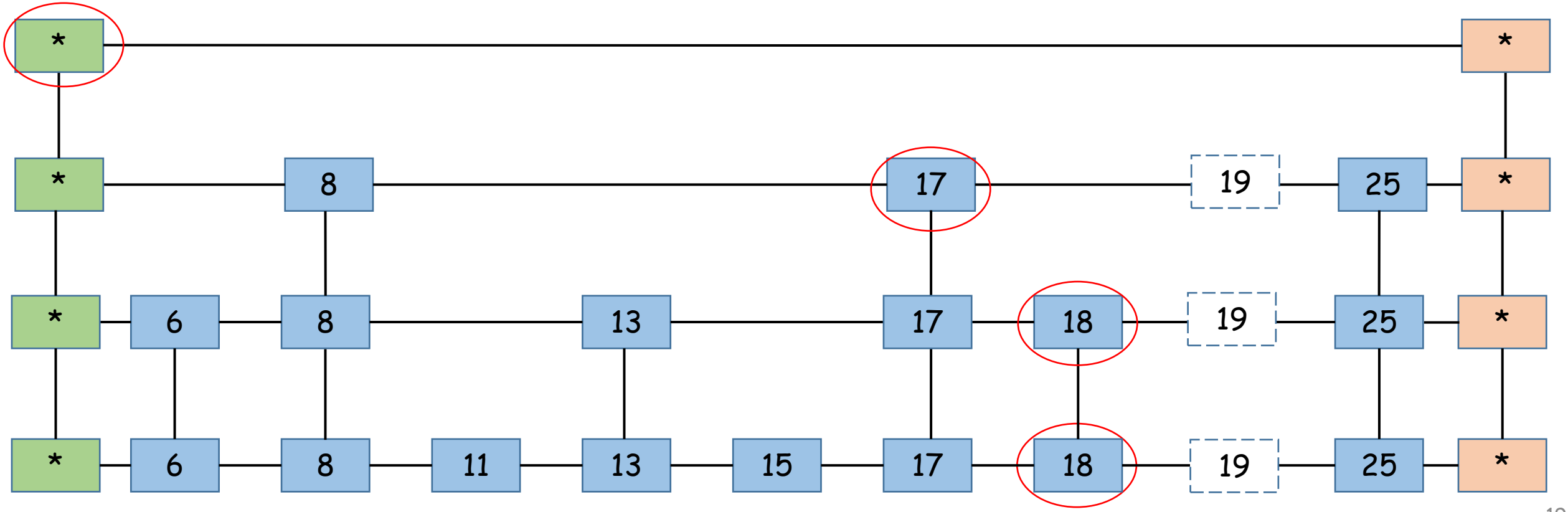
Remember only the last node visited at every level



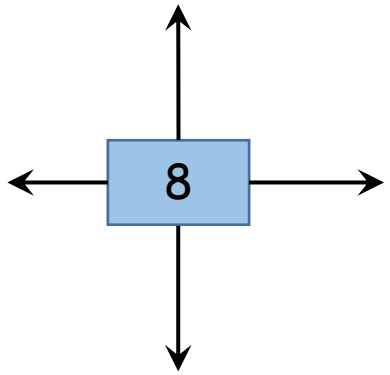
Skip List – put(19)

Data structure to remember the last node visited at every level?

Array of references to nodes

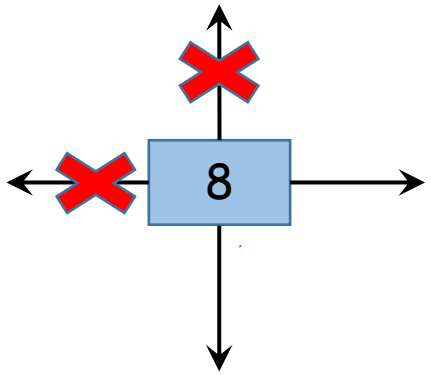


Structure of node



- Can we be space efficient?
- Can we avoid tracing back?

Structure of node

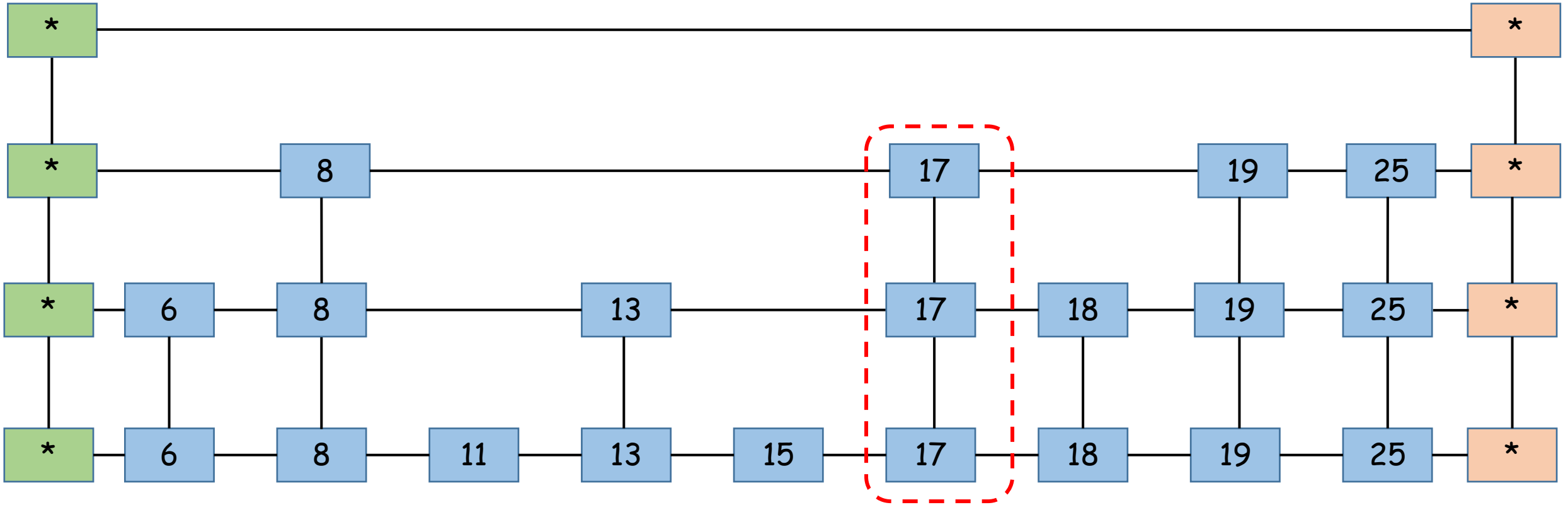


- Can we be space efficient?
- ~~Can we avoid tracing back?~~

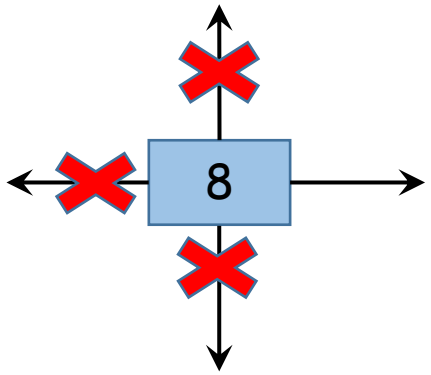
Skip List

Array of references to next node with level # as index

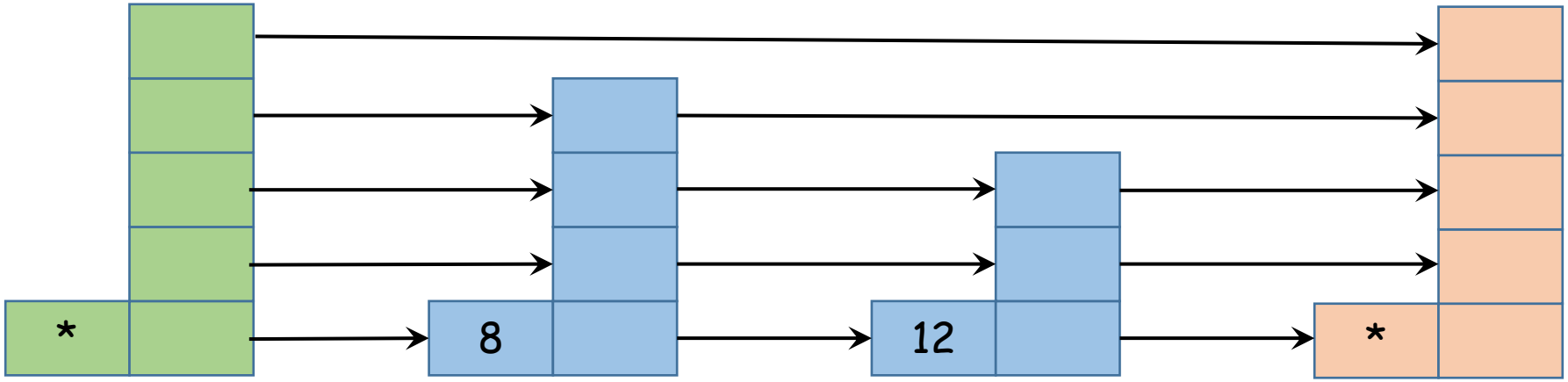
Step down the level by decrementing the index



Structure of node

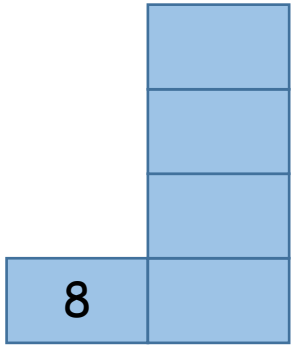


- ~~Can we be space efficient?~~
- ~~Can we avoid tracing back?~~



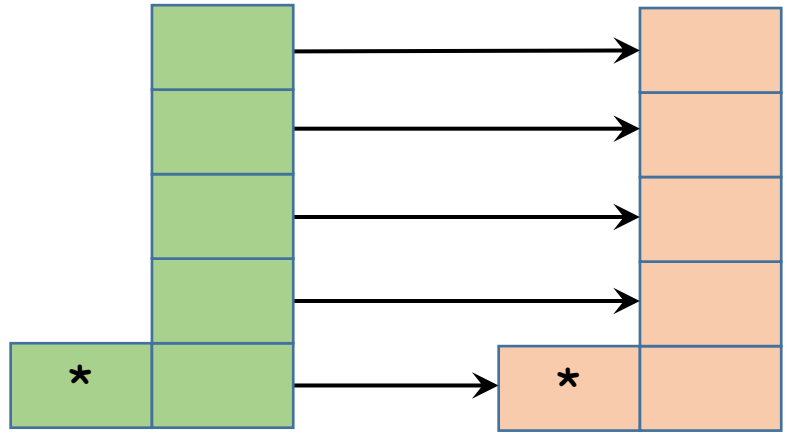
Entry Class

```
Class Entry <T> {
      T element;
      Entry[] next;
      Entry(T x, int levels){
            element = x;
            next = new Entry[levels];
      }

      // returns the # of levels this entry is in
      int level() {return next.length()}
}
```



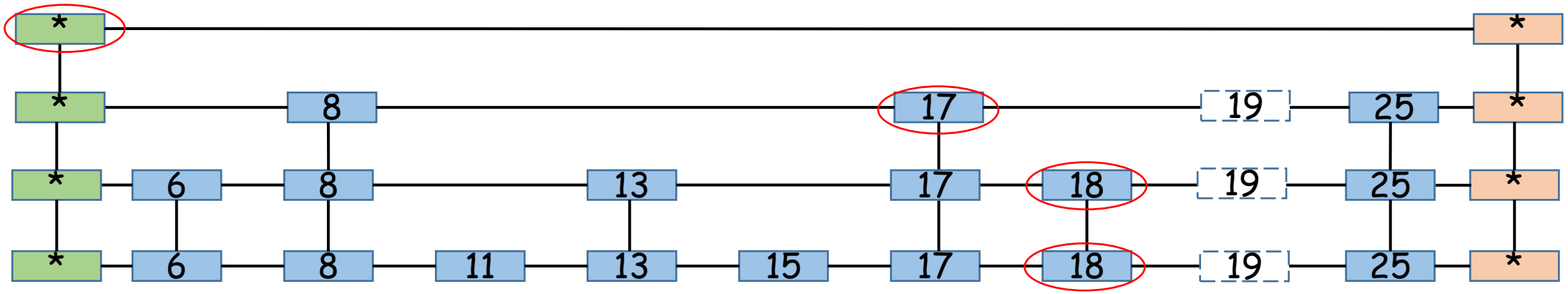
SkipList Class

```
Class SkipList <T> {
      Entry head, tail;
      int size;
      static final int maxLevel = 33;
      SkipList(){
            head = new Entry(null, maxLevel);
            tail = new Entry(null, maxLevel);
            //initialize head.next[]
      }

      // methods for add, remove, contains

}
```



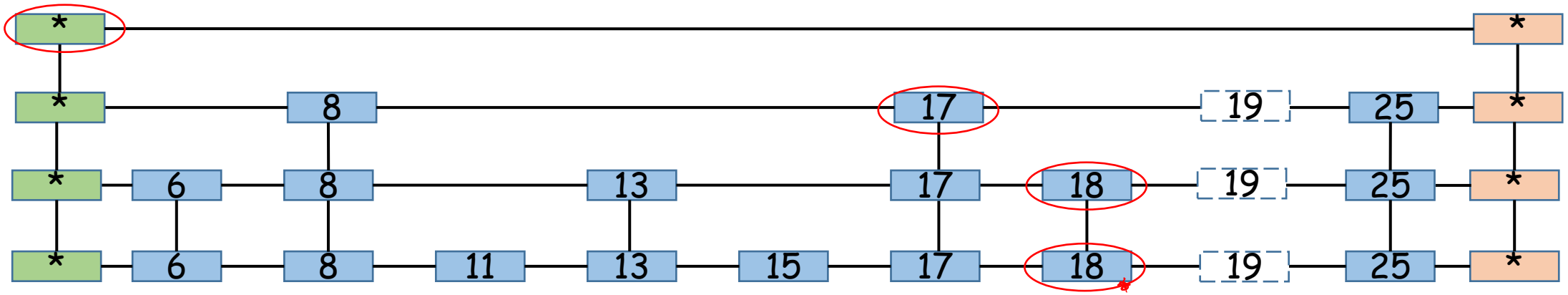
Helper Method - findPred(x)

```
findPred(x){
      p = head;
      for i = p.level() - 1 downto 0 do
       while(p.next[i].element < x)//watch NPE due to null in tail
            p = p.next[i];
       pred[i] = p; //remember last node visited at each level
      }
}
```



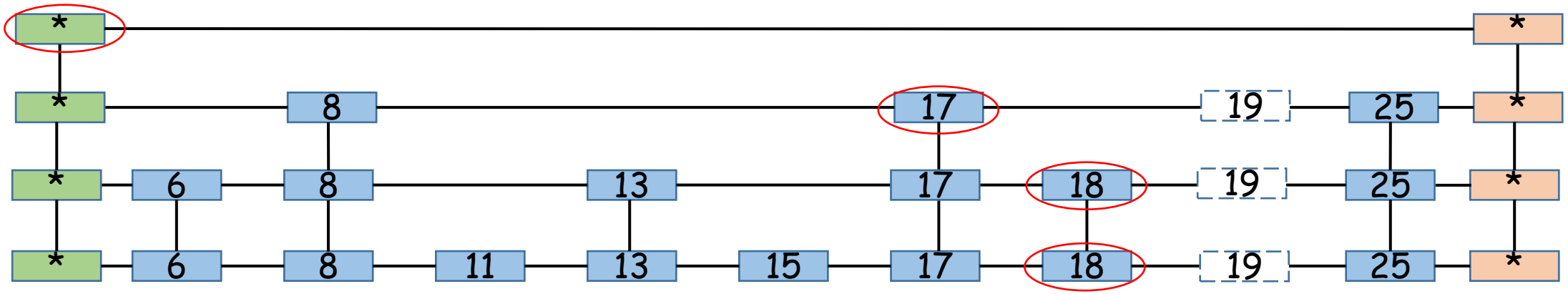
contains(x)

```
contains(x){
        findPred(x);
        return pred[0].next[0].element == x
}
```



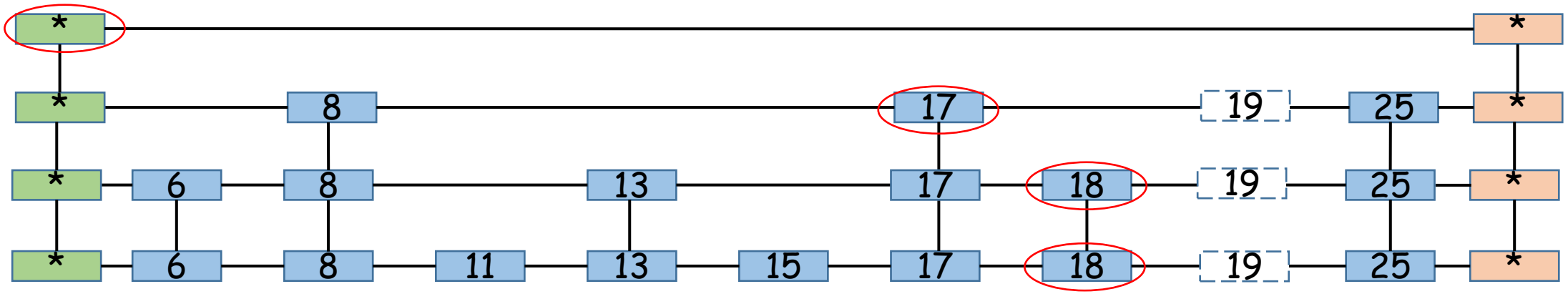
add(x)

```
add(x){
      if contains(x) then return false; // no duplicates
      lvl = chooseLevel(); // size of next[] in Entry for x
      entry = Entry(x, lvl)
      for i = 0 to lvl - 1 do
            entry.next[i] = pred[i].next[i];
            pred[i].next[i] = entry;
      size++
}
```



remove(x)

```
remove(x){
      if !contains(x) then return null; // not in list
      entry = pred[0].next[0];
      lvl = entry.level()
      for i = 0 to lvl - 1 do
            pred[i].next[i] = entry.next[i];
      size--
}
```



chooseLevel()

```
chooseLevel(){
      r = new Random()
      lvl = 1 + Integer.numberOfTrailingZeros(r.nextInt())
      return min(lvl, maxLevel - 1); // to control height
}
```

get(i): return the element at index i

```
get(i){
      if i > size - 1 then raise exception
      p = head;
      for k = 0 to i do
            p = p.next[0];
      return p.element;
}
```

RT: O(i)
Can we do better?