Data Structure:
Disjoint Set
Skipped list

Yongjun Park

Hanyang University

Disjoint sets

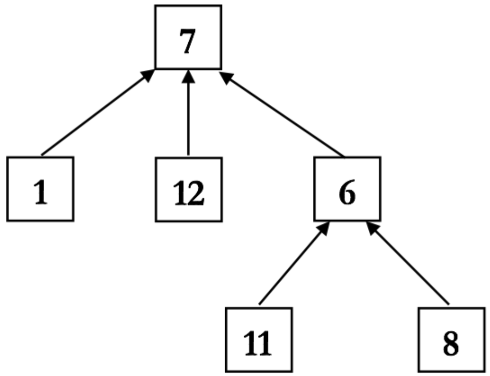
- We assume that the sets being represented are pairwise disjoint.
- If S_i and S_j are two sets and i != j, then there is no element that is in both S_i and S_j
- Basic operations needed for Disjoint Set
  - union
  - find

$S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$

$S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$

Union-Find in disjoint sets

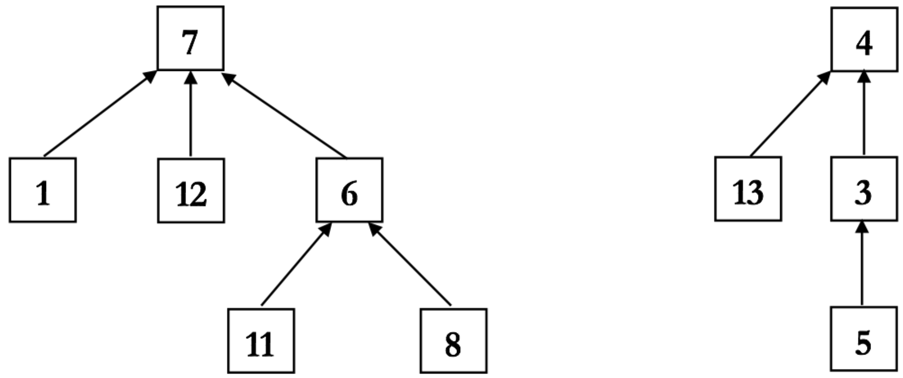
- maintain elements of S in a forest of inverted trees
  - pointers in the tree are directed towards the root.
  - the root of a tree has a NULL parent pointer
  - two elements are in the same set iff they are in the same tree.



$S_1 = \{1, 6, 7, 8, 11, 12\}$

Union-Find in disjoint sets

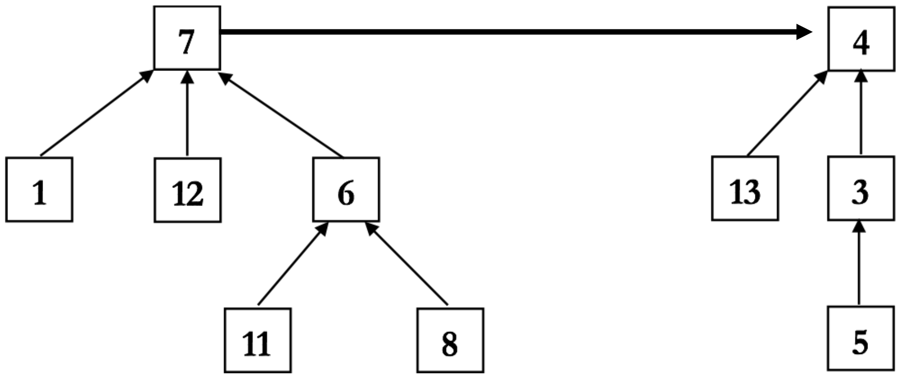
- Find(S, i)
  - find the node containing i
  - follow the parent links up to the root.
  - return the root node as the "name" of the set.



$S_1 = \{1, 6, 7, 8, 11, 12\}$ $S_2 = \{4, 3, 5, 13\}$

Union-Find in disjoint sets

- Union(i, j)



$S_1 = \{1, 6, 7, 8, 11, 12\}$ $S_2 = \{4, 3, 5, 13\}$

$S_1 \cup S_2 = \{1, 6, 7, 8, 11, 12, 4, 3, 5, 13\}$

Union-Find in disjoint sets

- Init(S): set all parent to 0

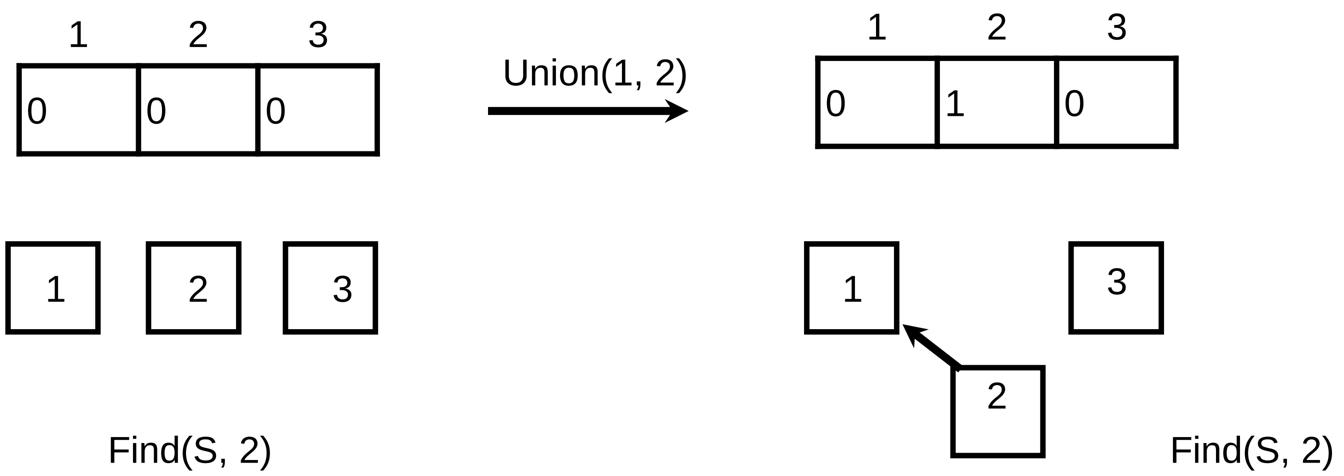
  ex) when S = {1, 2, 3}, [1] = {1}, [2] = {2}, [3] = {3}

- Union(S, t) link the root of one tree into the root of the other tree

  ex) Union(1, 2): [1] = {1, 2}, [3] = {3}

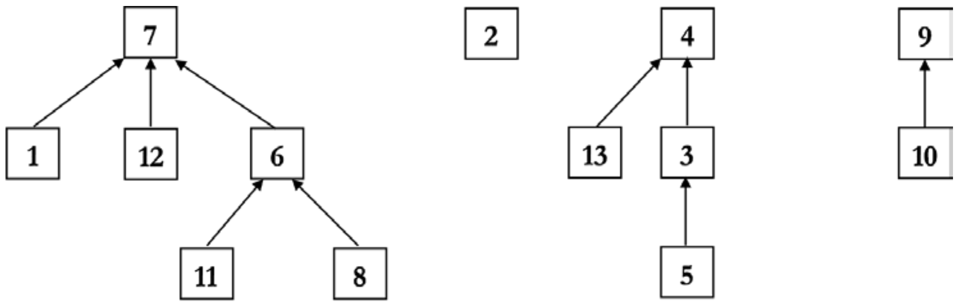
- Find(S, i): follow the parent link

ex) Find(S, 1) = 1, Find(S, 2) = 1, Find(S, 1) = Find(S, 2)



Union-Find in disjoint sets

S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
the current partition: {1, 6, 7, 8, 11, 12}, {2}, {3, 4, 5, 13}, {9, 10}



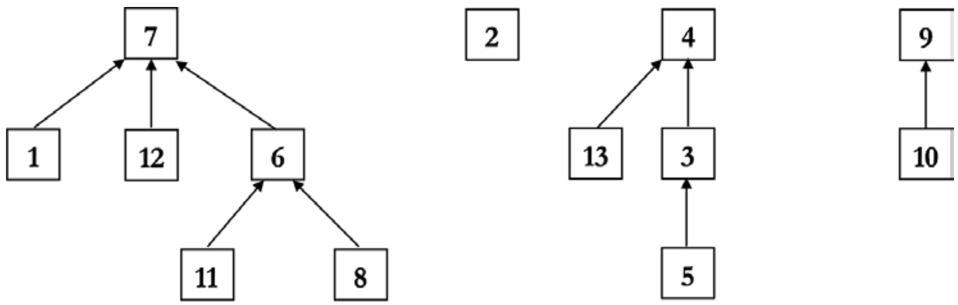
7	0	4	0	3	7	0	6	0	9	6	7	4
1	2	3	4	5	6	7	8	9	10	11	12	13

- there is no order how the tree should be structured
- the element is an index, not a *key*
- for each element, the array S[1..n] stores the index of the parent in the tree
- index of 0 means a null pointer

Union-Find in disjoint sets

S = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}

the current partition: {1, 6, 7, 8, 11, 12}, {2}, {3, 4, 5, 13}, {9, 10}



7	0	4	0	3	7	0	6	0	9	6	7	4
1	2	3	4	5	6	7	8	9	10	11	12	13

- What is the time complexity for union?

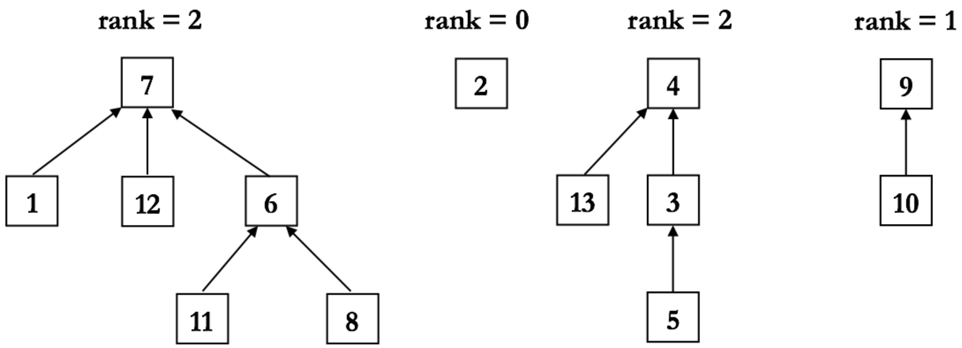
- Union({2}, {9, 10})

If we link {9, 10} into {2}, the resulting height is 2

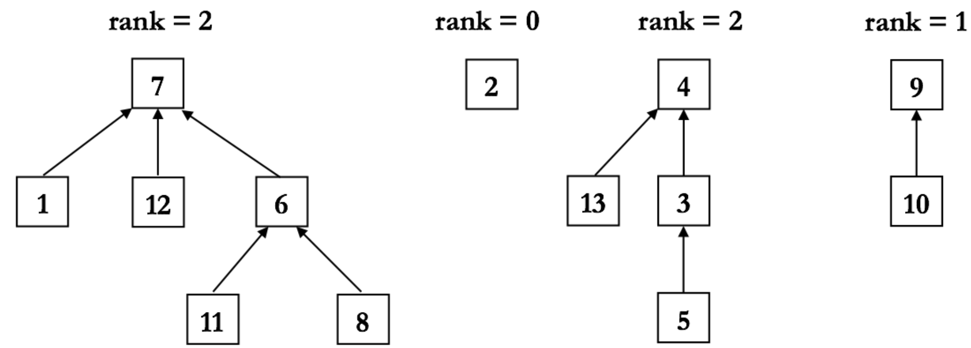
If we link {2} into {9, 10}, the resulting height is 1

- How can we improve the simple union?

Union-Find in disjoint sets ADT



Union-Find in disjoint sets ADT



Array representation

7	0	4	-2	3	7	-2	6	-1	9	6	7	4
1	2	3	4	5	6	7	8	9	10	11	12	13

7	0	4	0	3	7	0	6	0	9	6	7	4
1	2	3	4	5	6	7	8	9	10	11	12	13

- to perform smart Union, each tree includes an extra information called *rank*, which is the height of the tree.
- link the tree with smaller rank to the tree with larger rank.
- where do we store the rank?
  - We only need to maintain the rank for the root nodes
  - One clever way is to store the negative of rank at the root node
    - if S[i] is strictly positive because it is a parent pointer (represented using array index).
    - Otherwise, i is a root and -S[i] is the rank of the tree.

Union-Find in disjoint sets ADT

S = {1, 2, 3, 4}

Perform the following operation in order

union(1, 2), union(2,3), union(3,4)

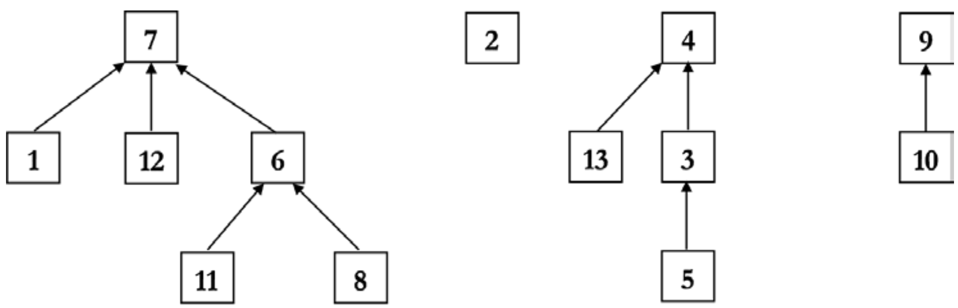
Union-Find in disjoint sets ADT

```
Disj_Sets S[n];
void Init(Disj_Sets *S)
{
        for (i = 1; i <= n; i++)  S[i] = 0;
}
```

```
Set_Type Find1(Elt_Type x, Disj_Sets *S)
{
        while (S[x] > 0)
                        x = S[x];
        return x;
}
```

Union-Find in disjoint sets ADT

```
void Union(Disj_Sets *S, Set_Type r1, Set_Type r2)
{
        if (S[r2] < S[r1])
                        S[r1] = r2;            /* if |S[r2]| > |S[r1]|,   add r1 to r2 */
        else
        {
              if (S[r2] == S[r1])
                                                              S[r1]--;
              S[r2] = r1;            /* add r2 to r1 */
        }
}
```



Analysis of running time

Init( ) takes $O(n)$, but is done once.
Union( ) takes a constant time, $O(1)$.
Find( ) takes worst-case time proportional to the height of tree.

Theorem Using the Union( ) and Find1( ) procedure, any tree containing m ele ments has height at most log m.

This follows by proving the following Lemma.
Lemma Using the Union( ) and Find1( ) procedure, any tree with height h has at least 2^h elements.

If we prove the lemma, the following theorem holds.
Theorem After initialization, any sequence of n Unions and n Finds can be perf ormed in time $O(n \log n)$.

Analysis of running time

Proof: Induction on the number of Unions performed to build the tree.

- Basis: No unions. Tree with 1 element of height 0. $2^0 = 1$
- Induction Step: Suppose the theorem is true for all trees built with fewer than k unions, and we want to prove the lemma for a tree T built with exactly k union operations. Such a tree is formed by unioning two trees T_1 and T_2, of heights h_1 and h_2 and size n_1 and n_2, respectively. Since these trees were formed with fewer than k unions $n_1 \geq 2^{h1}$ and $n_2 \geq 2^{h2}$.

Assume that T_2 was made a child of T_1 (that is, $h_2 \leq h_1$)

– if $h_2 < h_1$ then $h = h_1$, and

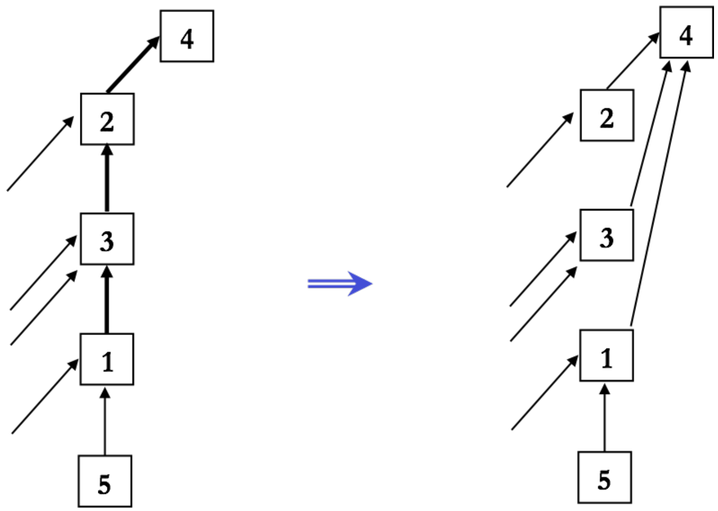
$$n = n_1 + n_2 \geq 2^{h1} + 2^{h2} \geq 2^{h1} = 2^h$$

– if $h_2 = h_1$ then $h = h_2 + 1 = h_1 + 1$, and

$$n = n_1 + n_2 \geq 2^{h1} + 2^{h2} \geq 2^{h-1} + 2^{h-1} = 2^h$$

path compression

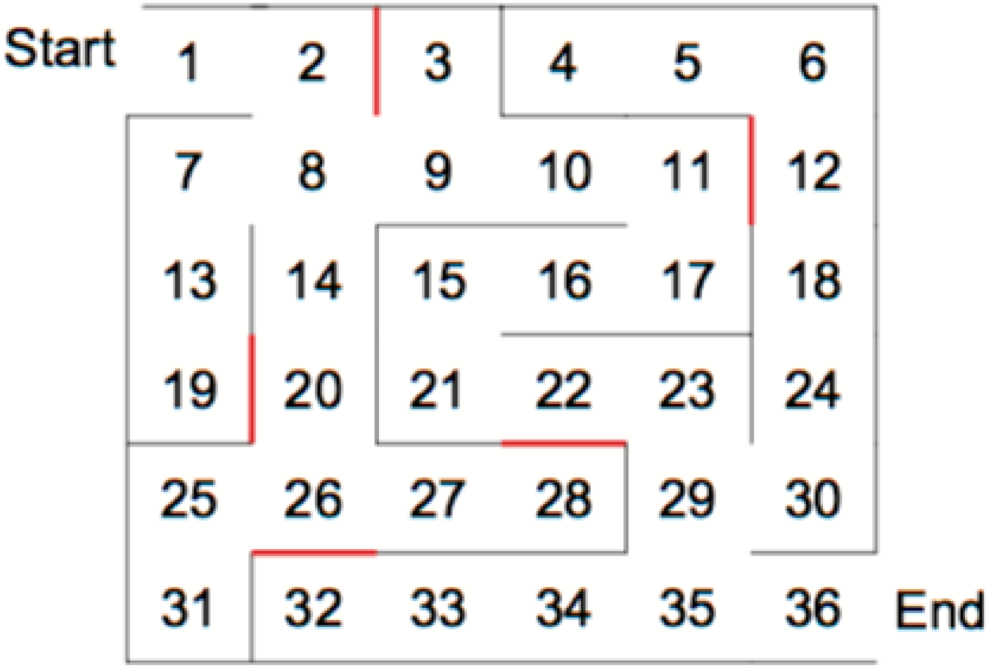
- a simple heuristic to improve running time significantly
  (ALMOST gets rid of the log *n* factor in the running time *O*(*n* log *n*))
- If we compress the paths on each Find(), subsequent Find() will go much faster.
- "Compress the path" means that when we find the root we set all parent pointers of the node on our find path to the root.




```
Set_Type Find2(Elt_Type x, Disj_Sets S)
{
        if (S[x] <= 0)
                        return x;
        else
             return (S[x] = Find2(S[x], S));
}
```

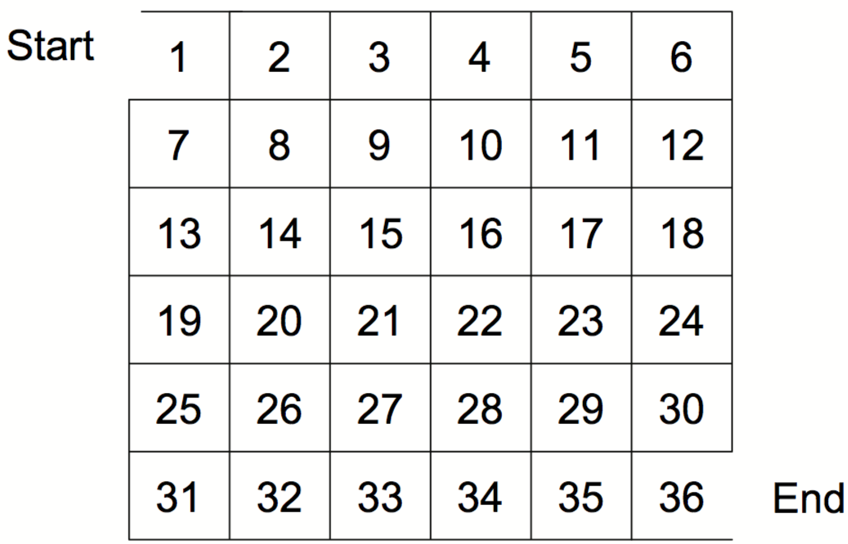
- running time of Find2() is still proportional to the height of the tree
- each time you spend lots of time in Find2(), you make the tree flatter, thus making subsequent Find2() faster.

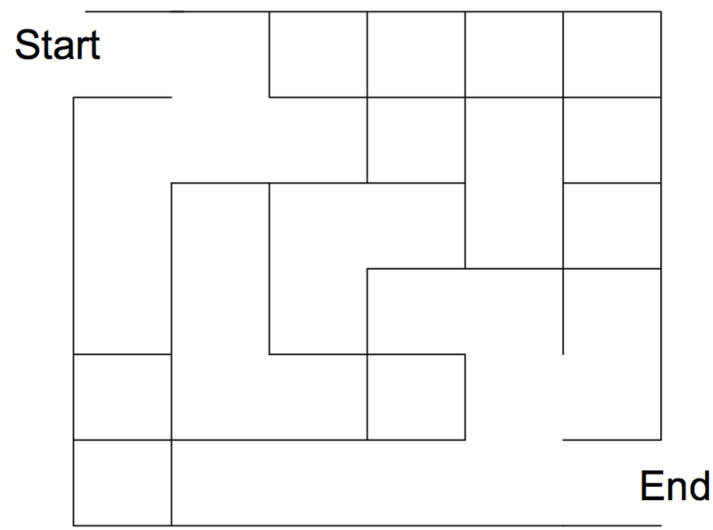
disjoint sets ADT



disjoint sets ADT

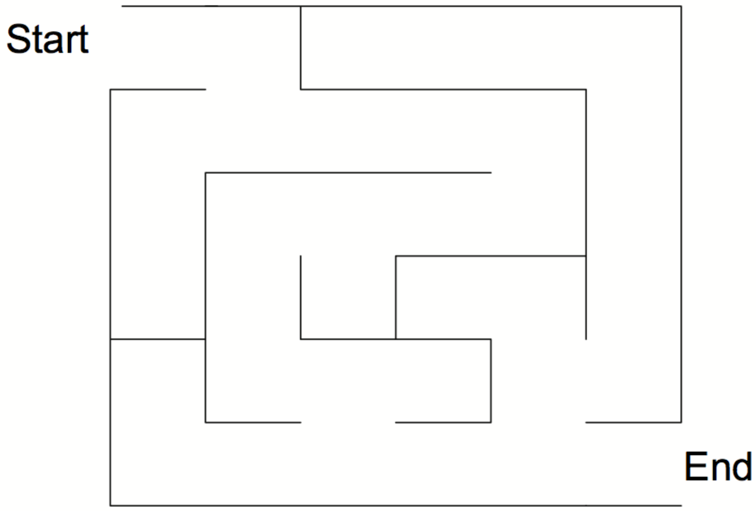
Idea: build a random maze by erasing edges.

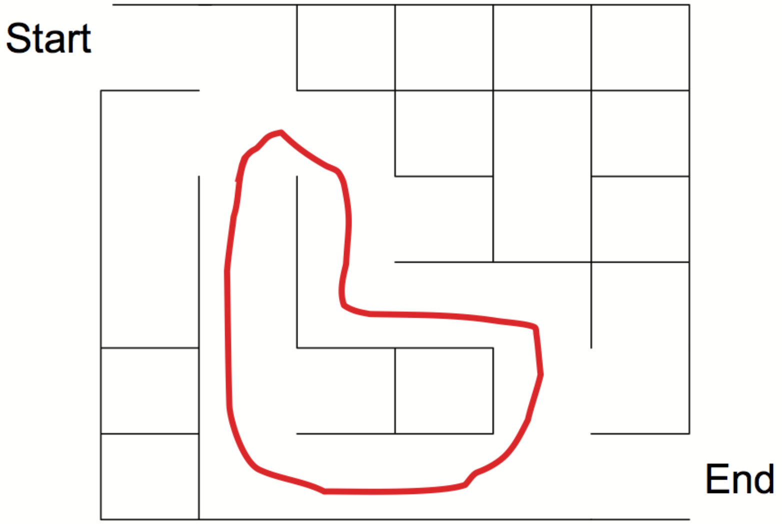




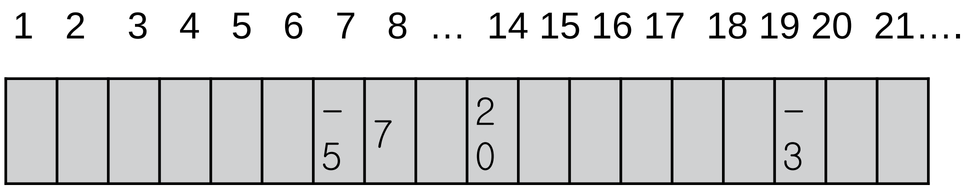
disjoint sets ADT

Idea: build a random maze by erasing edges.





Union-Find in disjoint sets ADT



{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,30,32
33,34,35,36}

Union(8, 14)

Find(8) = 7, Find(14) = 20

{1,2,7,8,9,13,19, 14, 20, 26, 27}

skip lists

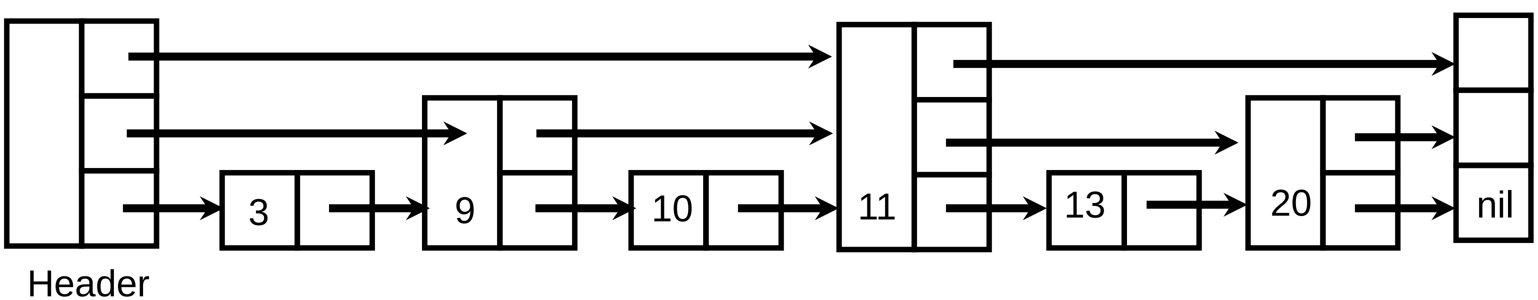
- linked lists do
  - insertion and deletion in O(1), but find in O(n)
  - not store sorted lists
- how can we make linked lists better?

- store in sorted linked lists?
- a randomized data structure: it uses the random number generator

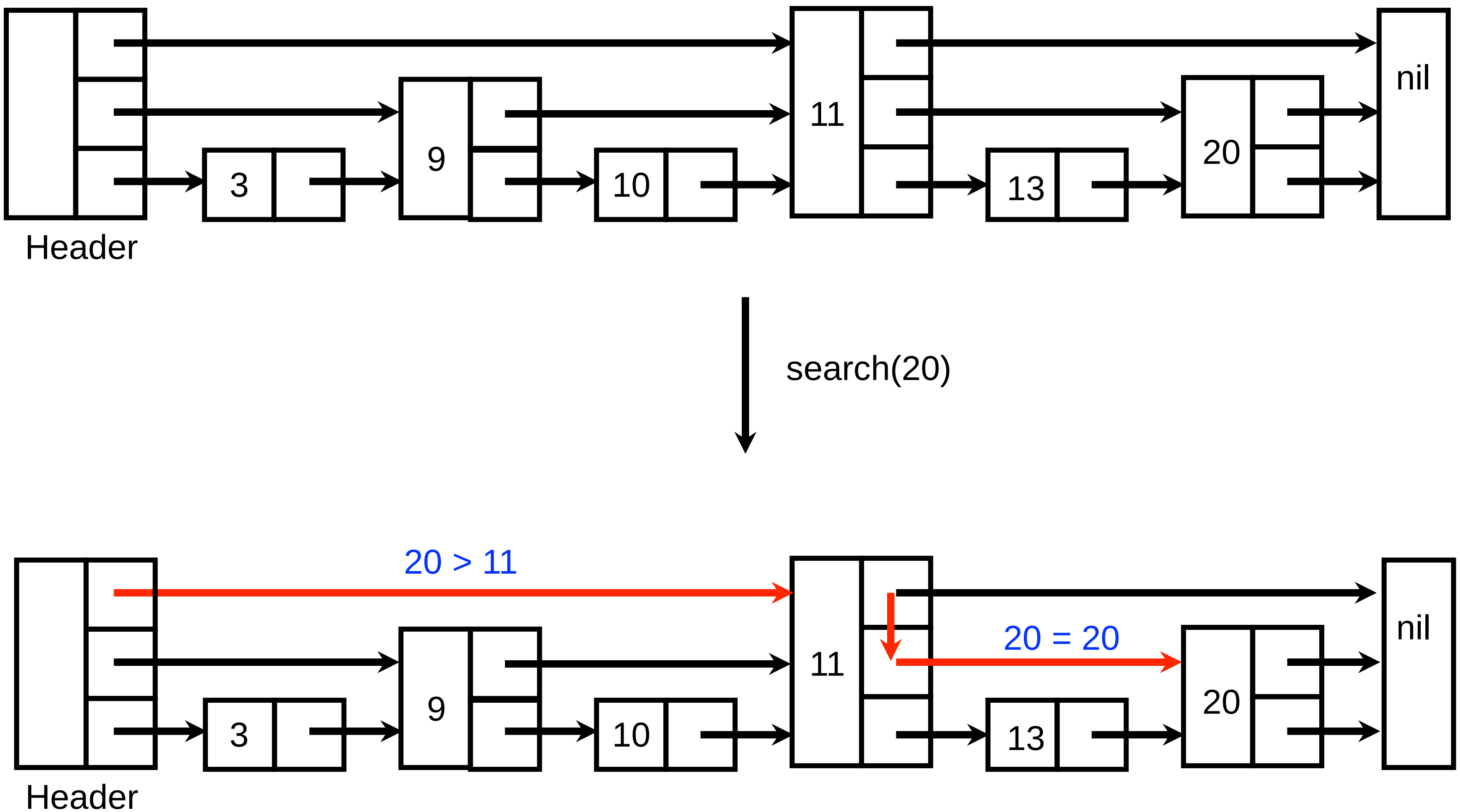
- skip lists
  - use hierarchy of sorted linked lists
  - skip over lots of items to find an element
  - expected search time is O(log n) with high probability

perfect skip lists

- nodes are of variable size, including 1 and O(log n) pointers
- search(k)
  - if k = next_key, done
  - if k < next_key, go down a level
  - if k > next_key, go right
    - In the worst case,
  - we have to go through all log n levels
  - at each level, we visit at most 2 nodes: $O(\log n)$



perfect skip lists: search



How about search(14)?

randomized skip lists

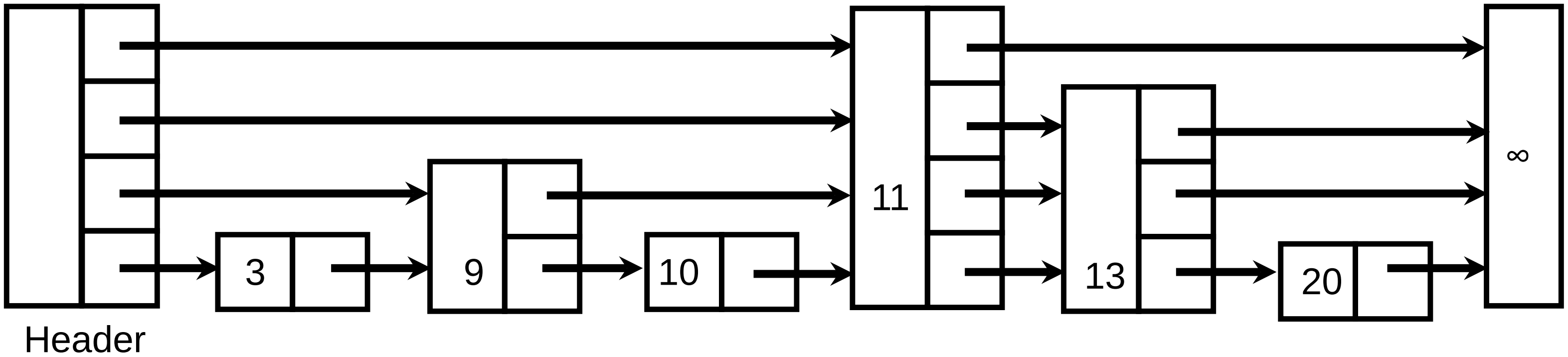
- perfect skip lists need to rearrange the entire list after insertion and deletion
- to insert or delete x,
  - search for x in the skip list
- find the position $p_0, p_1, \ldots, p_i$ of the items that has the largest key less than x in each level 0, 1, . . . , I
  - The maximum level (the size of header node) should be log *n* when *n* is the maximum number of nodes allowed

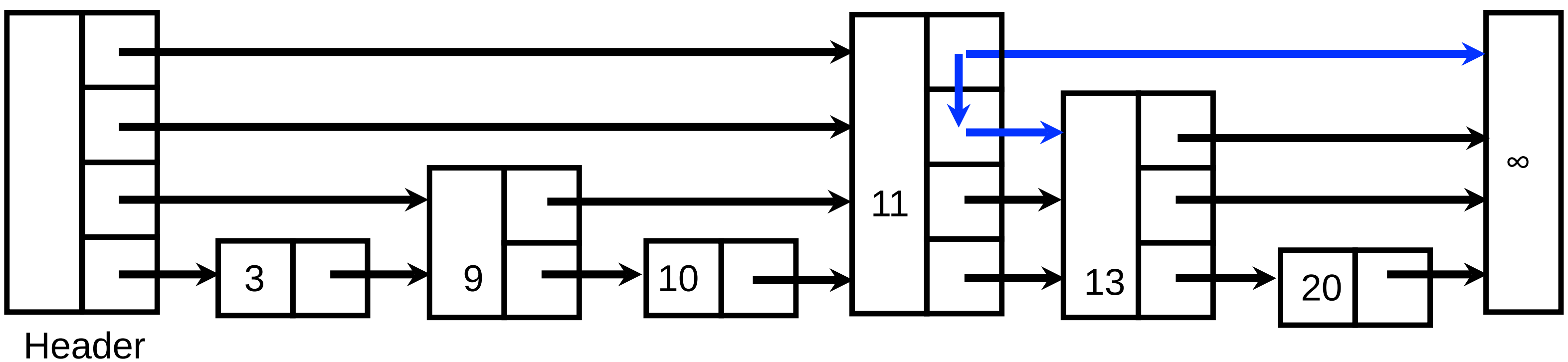
```
struct skip_node {
        element_type    element;
        int     level;
        struct skip_node     **forward;
} *s;

s = (skip_node*)malloc( sizeof(struct skip_node) );
s->forward = (skip_node**)malloc( sizeof(skip_node *)*(level+1) );
```

randomized skip lists: search

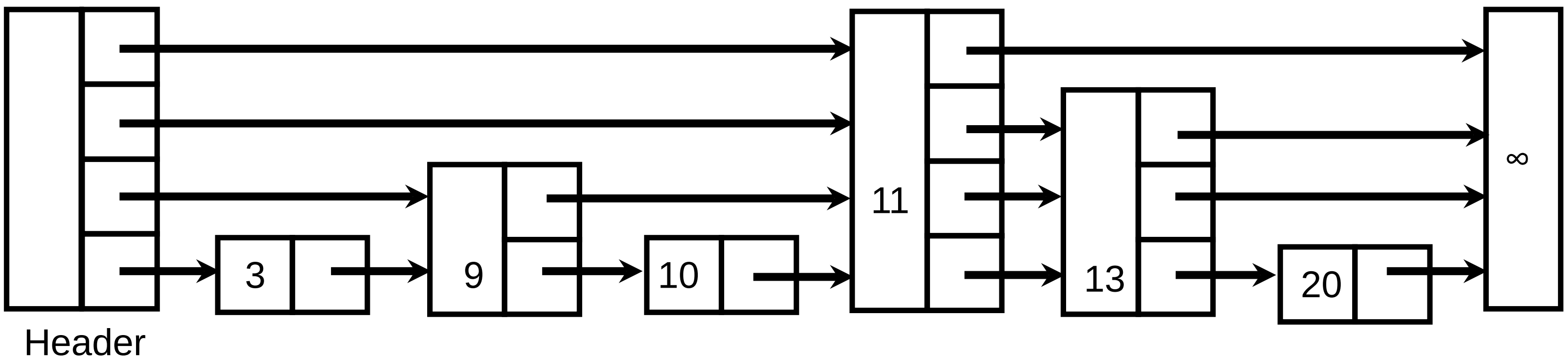
search(13)

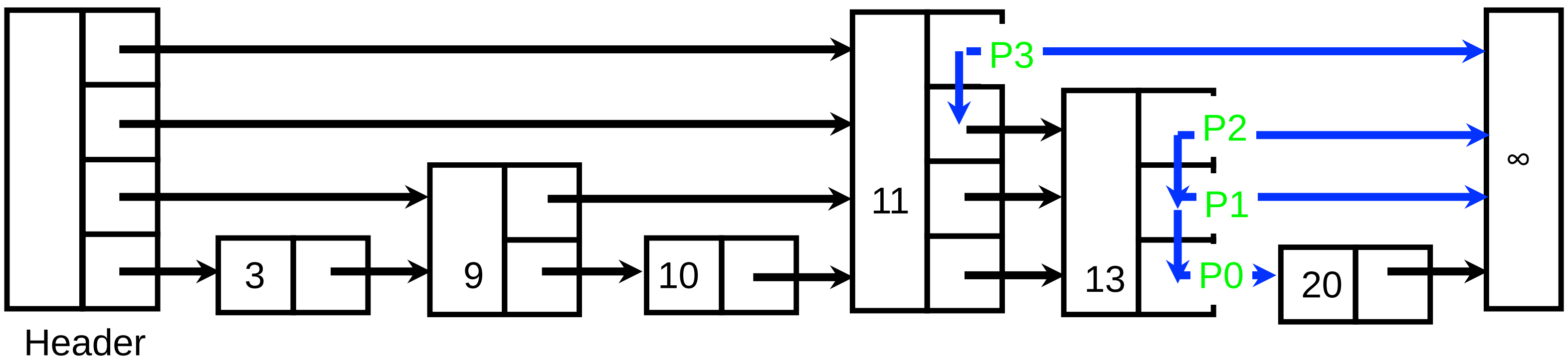




randomized skip lists: insert

insert(14)

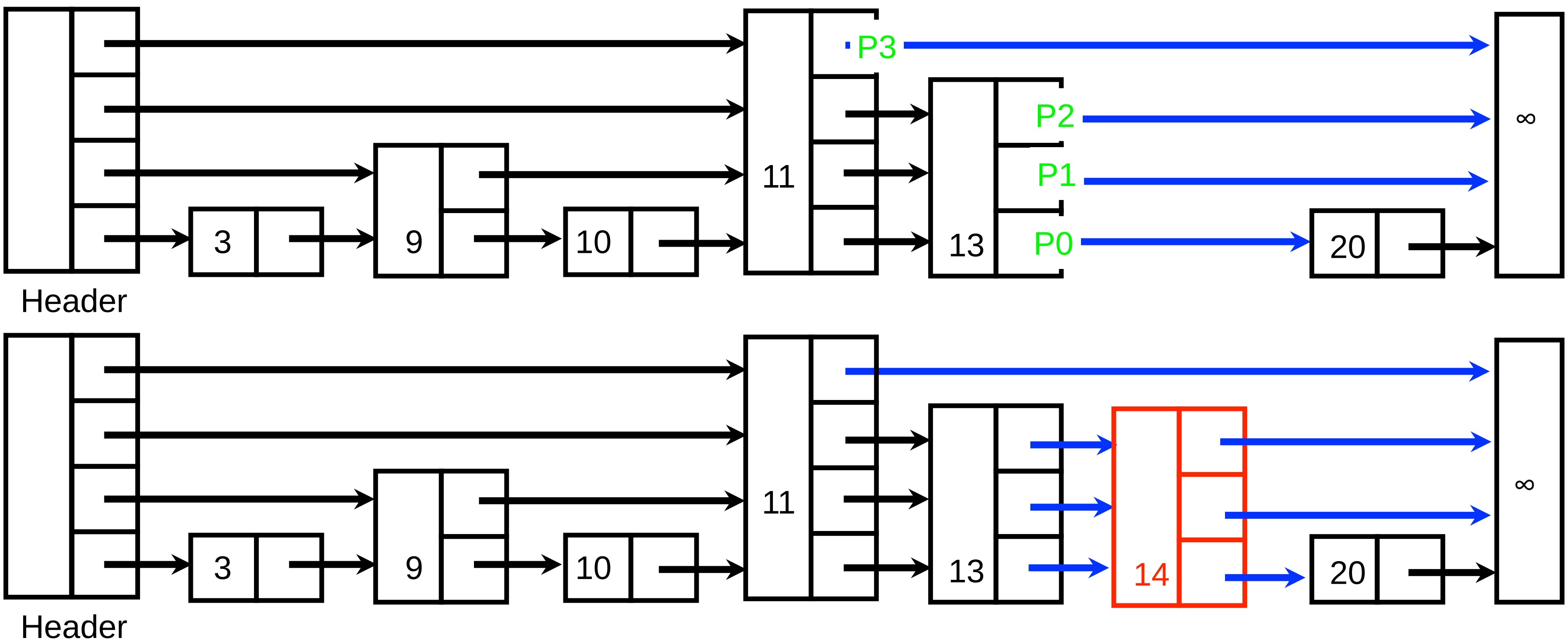




found the place for insertion!

randomized skip lists: insert

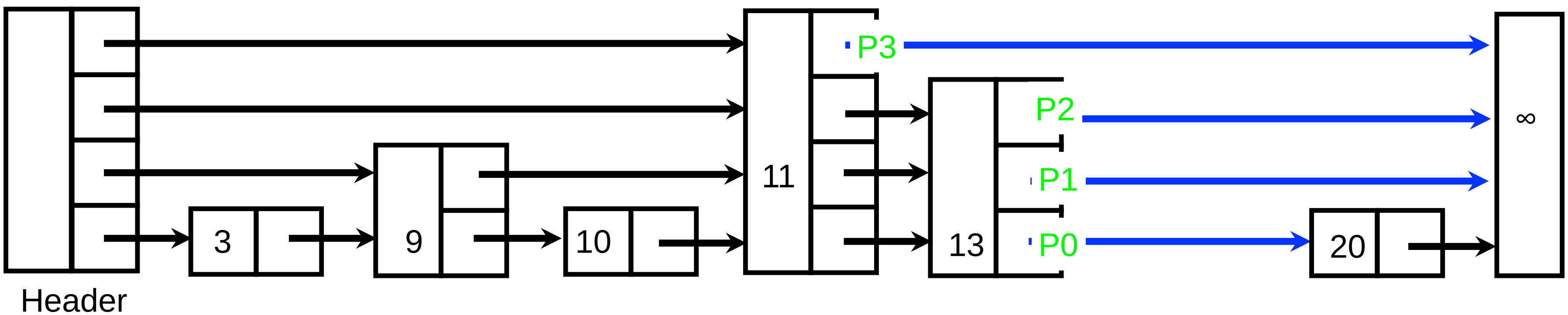
insert(14) at level 2

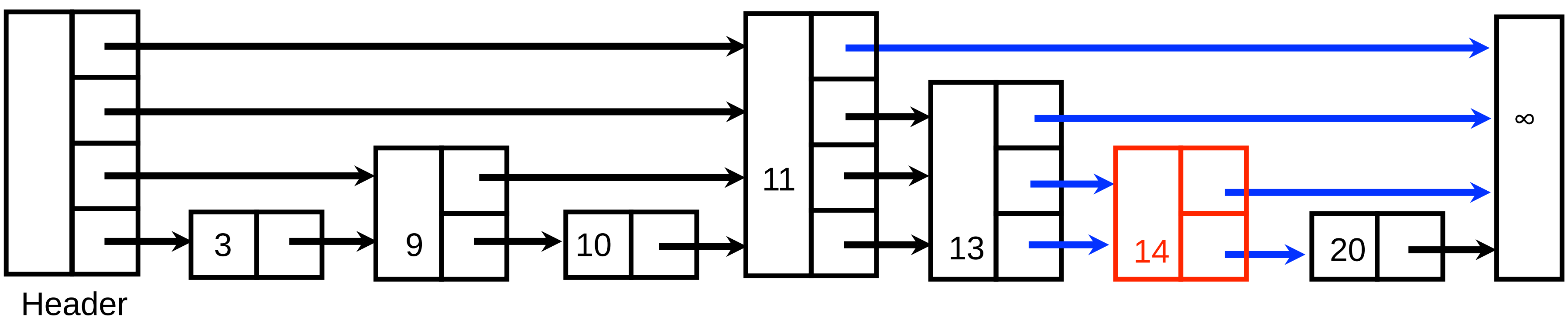



```
search(x);                          # find x
level = 0;                          # insert node in level 0
while (FLIP()  ==  "heads")
        level ++;                   # move the level of the new node up
```

randomized skip lists: insert

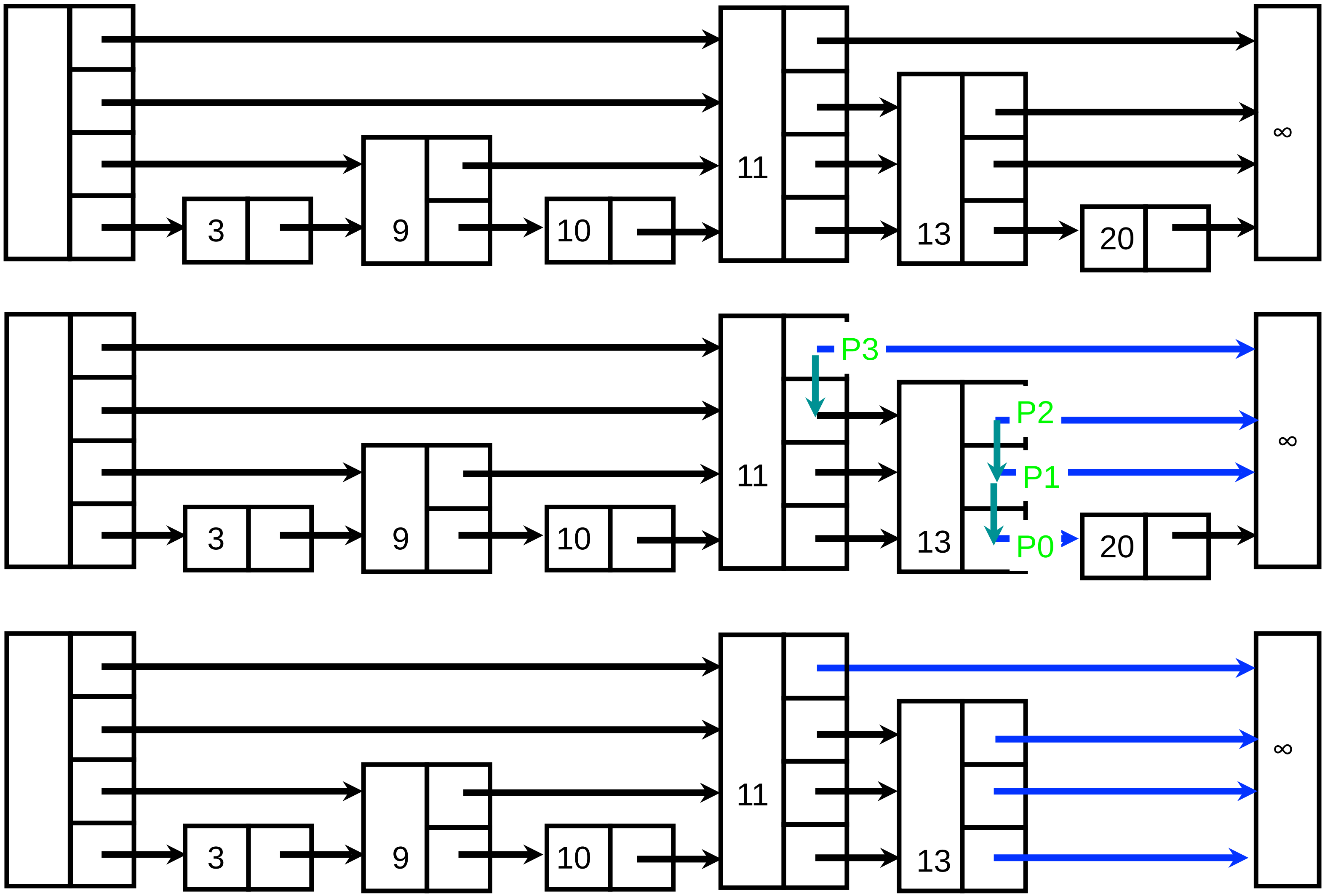
insert(14) at level 1





randomized skip lists: delete

delete(20)



randomized skip lists: delete

delete(9)