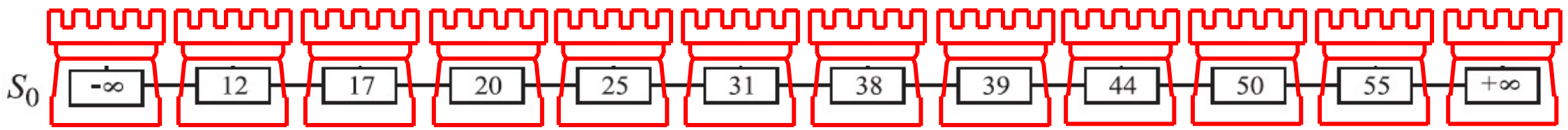
SKIP LISTS

SKIP LISTS

- As mentioned earlier, when implementing an ordered map using a **search table** :
  - Finding an entry whose key = k takes O(log n) time (since it uses binary search)
  - Inserting or erasing an entry takes O(n) time (since we must shift other entries)
- However, if we implement an ordered map using a "**skip list**", then:
  - Finding an entry whose key = k takes O(log n) time **on average**
  - Inserting or erasing an entry takes O(log n) time **on average**
- Here, the notion of **average time complexity** depends on the use of a **random-number generator** in the implementation of the insertions, to help decide where to place the new entry.
- The running time is averaged over **all possible outcomes** of the random numbers used when inserting entries.

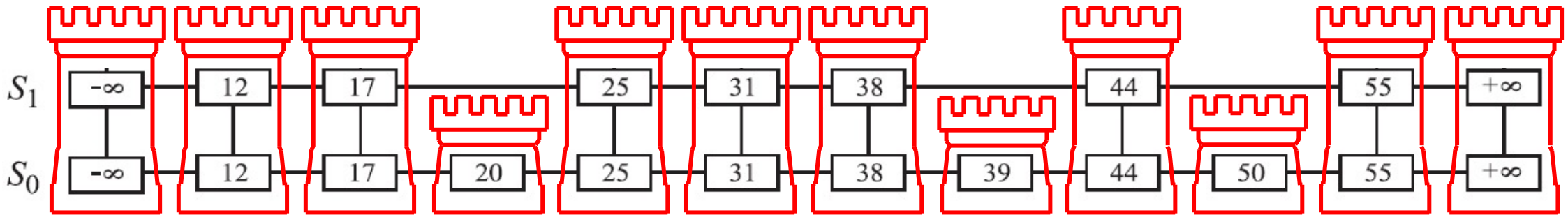
A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list



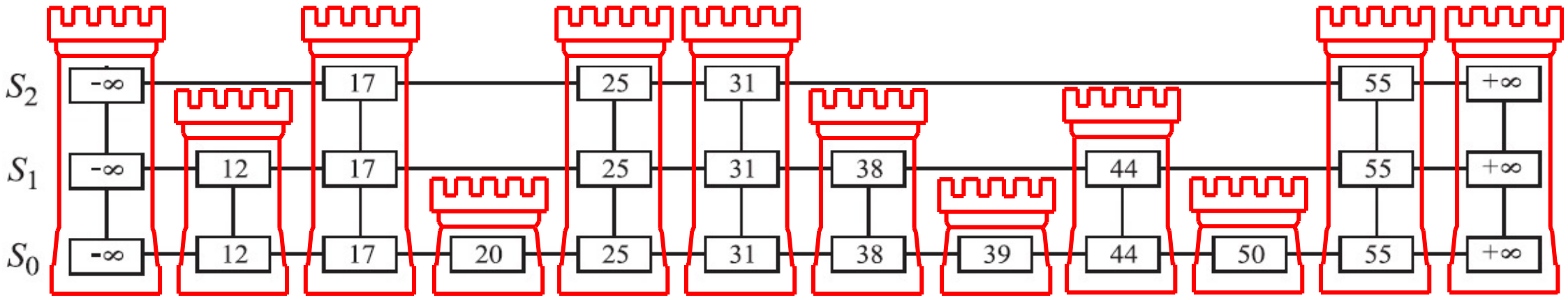
A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list
- At each level, each entry has a **50/50 chance of building an additional level!**
  - Each tower is implemented as a doubly-linked list



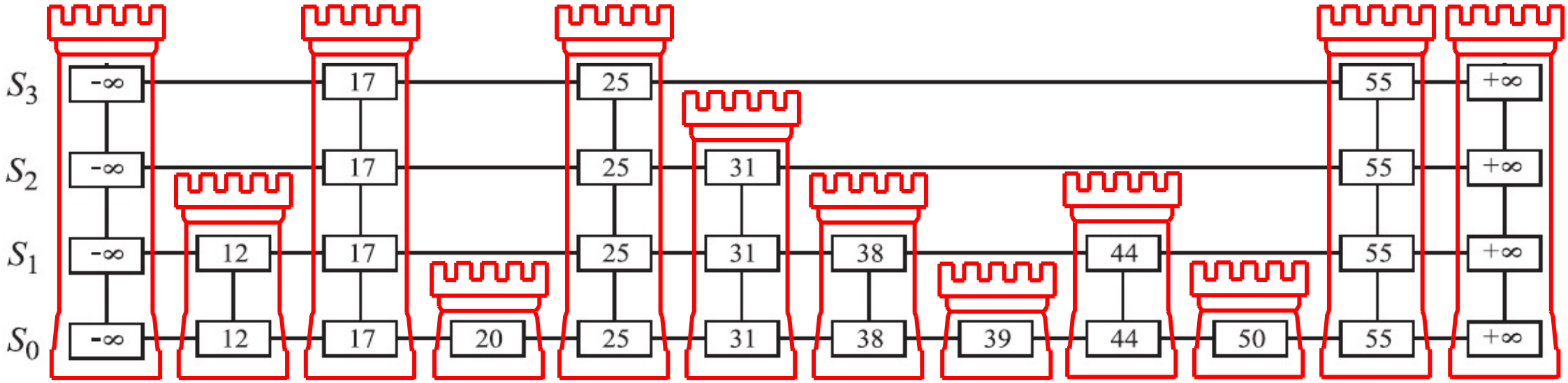
A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list
- At each level, each entry has a **50/50 chance of building an additional level!**
  - Each tower is implemented as a doubly-linked list



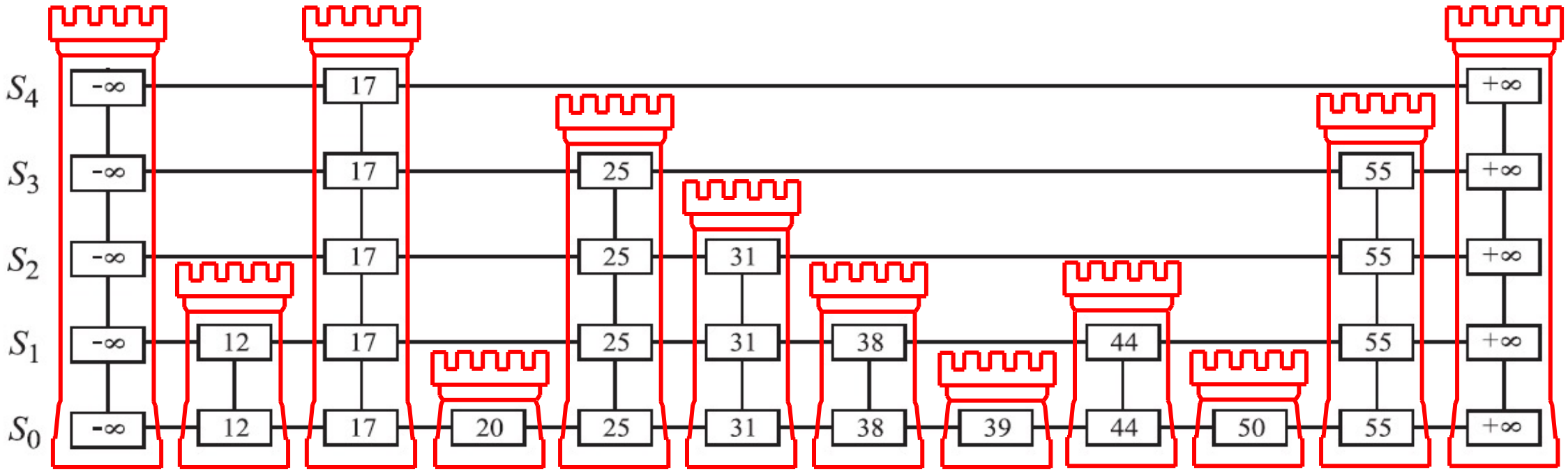
A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list
- At each level, each entry has a **50/50 chance of building an additional level!**
  - Each tower is implemented as a doubly-linked list



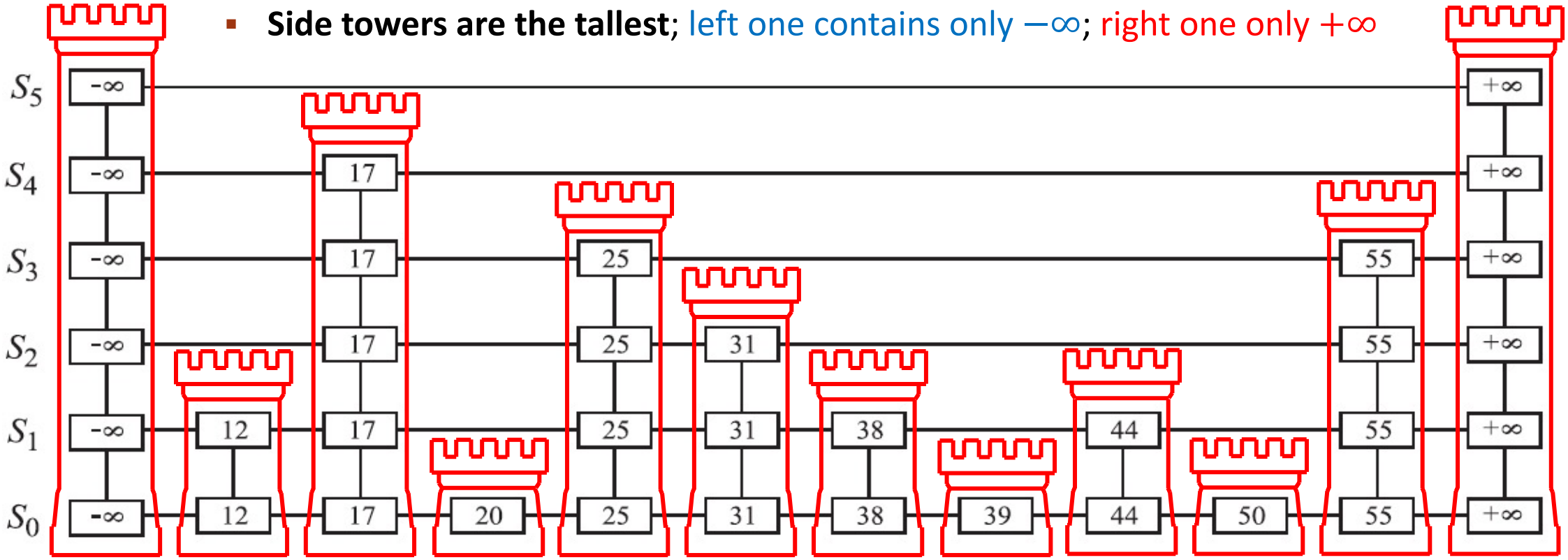
A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list
- At each level, each entry has a **50/50 chance of building an additional level!**
  - Each tower is implemented as a doubly-linked list



A skip list can be arranged into levels and **towers**:

- The **bottom level**, S_0, contains all entries **sorted** (plus the $-\infty$ and $+\infty$)
  - For simplicity, **this illustration only shows the keys** of the entries
  - Each level is implemented as a doubly-linked list
- At each level, each entry has a **50/50 chance of building an additional level!**
  - Each tower is implemented as a doubly-linked list
- **Side towers are the tallest**; left one contains only $-\infty$; right one only $+\infty$



SKIP LIST – TRAVERSAL

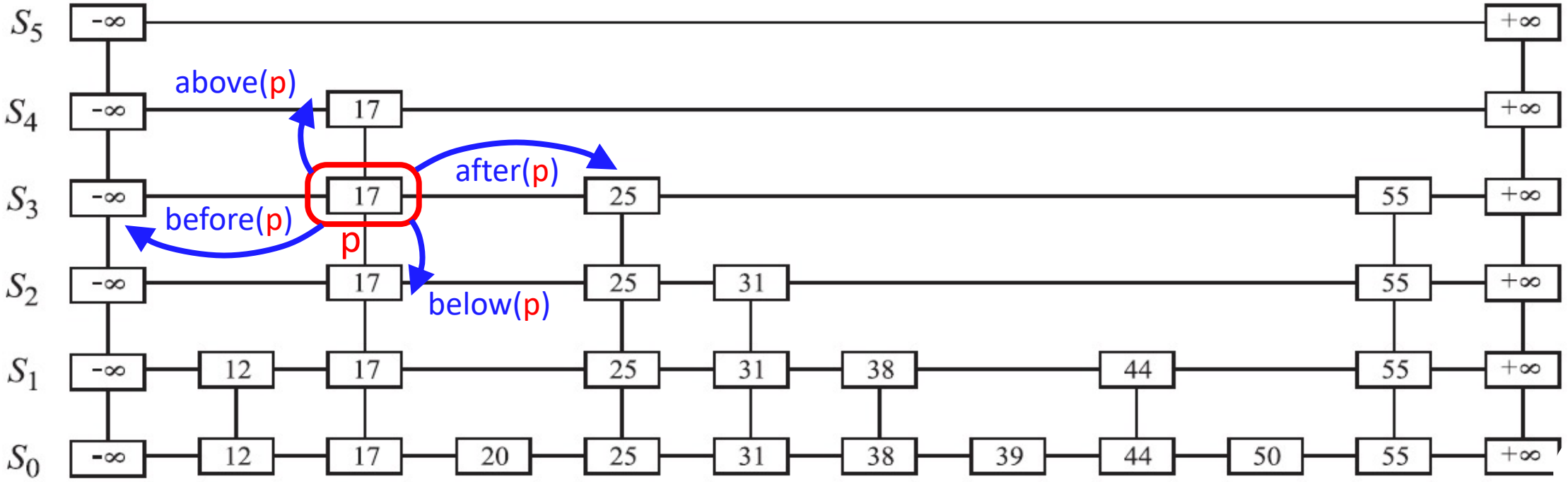
A skip list can be traversed via **positions**, p, using the following operations:

after(*p*): Return the position following *p* on the same level.

before(*p*): Return the position preceding *p* on the same level.

below(*p*): Return the position below *p* in the same tower.

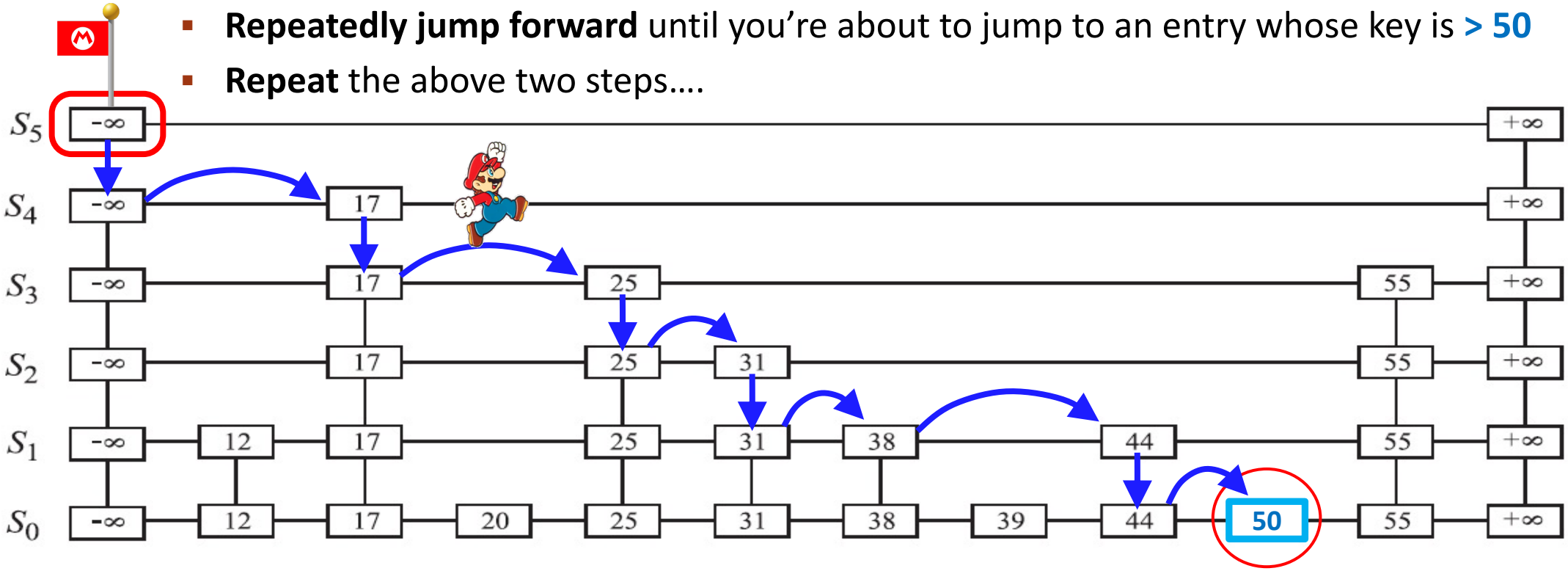
above(*p*): Return the position above *p* in the same tower.



SKIP LIST – SEARCH

Here is an example of how to **search** a skip list, S, for the entry whose key **= 50**:

- Set position p to be at the **top of the left tower**:
- **Jump once downward** one step by calling p = S.below(p); unless you've reached the bottom, in which case the search terminates.
- **Repeatedly jump forward** until you're about to jump to an entry whose key is **> 50**
- **Repeat** the above two steps….



SKIP LIST – SEARCH

Algorithm SkipSearch(*k*):

Input: A search key *k*

Output: Position *p* in the bottom list S_0 such that the entry at *p* has the largest key ***less than or equal*** to *k*

p ← *s*

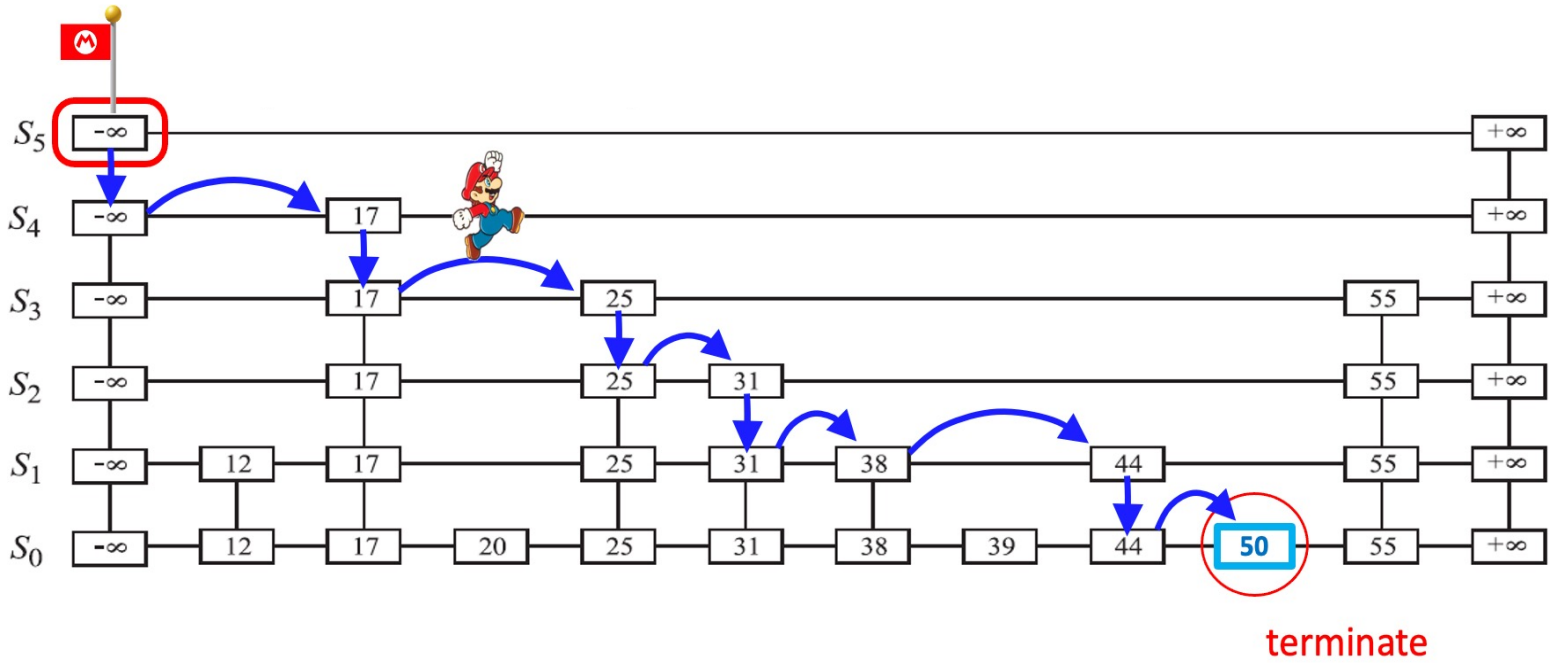
while below(*p*) ≠ *null* **do**

p ← below(*p*) {drop down}

while *k* ≥ after(*p*).key() **do**

p ← after(*p*) {scan forward}

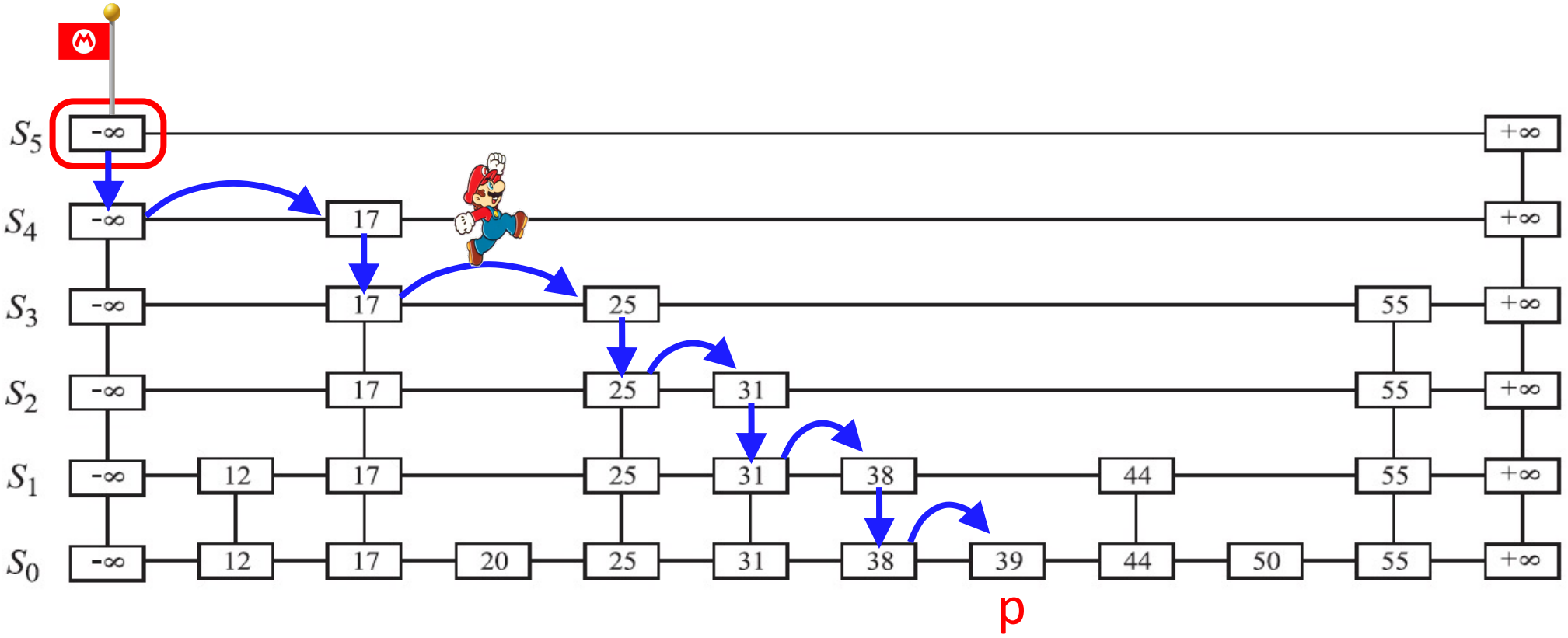
return *p*



SKIP LIST – INSERTION

Here is an example of how to **insert** an entry with key = **42**:

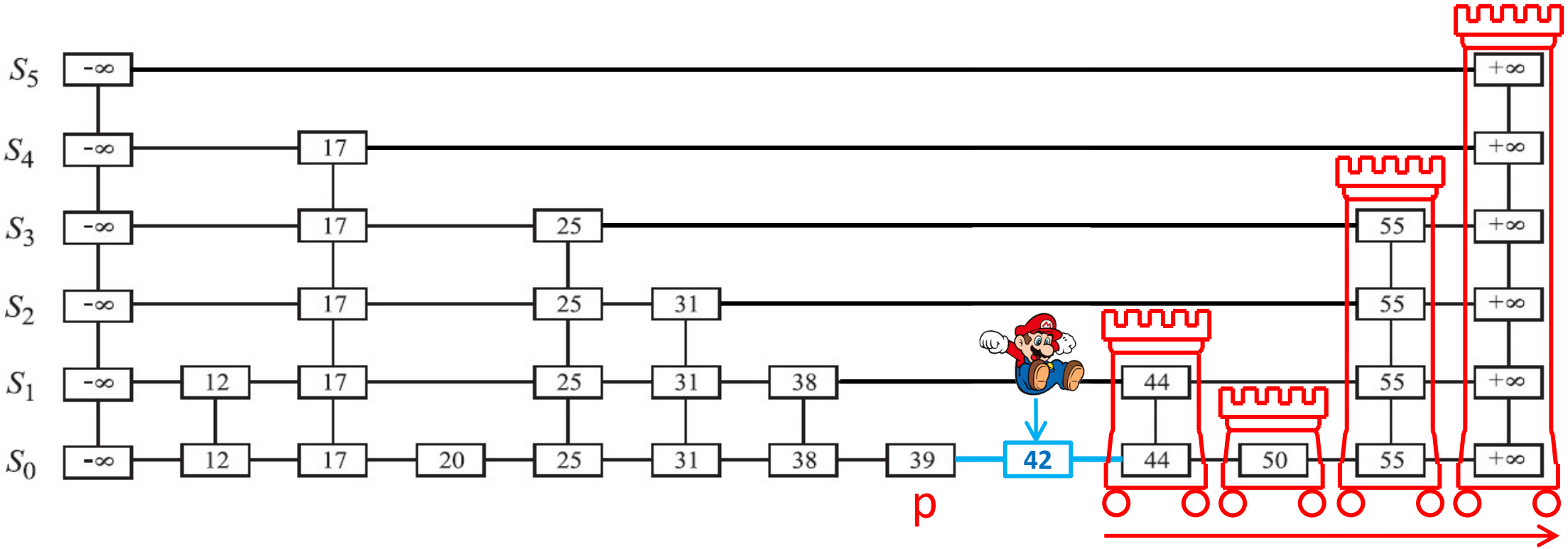
- **Search** for the largest key that is ≤ **42**, and set p to be its position



SKIP LIST – INSERTION

Here is an example of how to **insert** an entry with key = **42**:

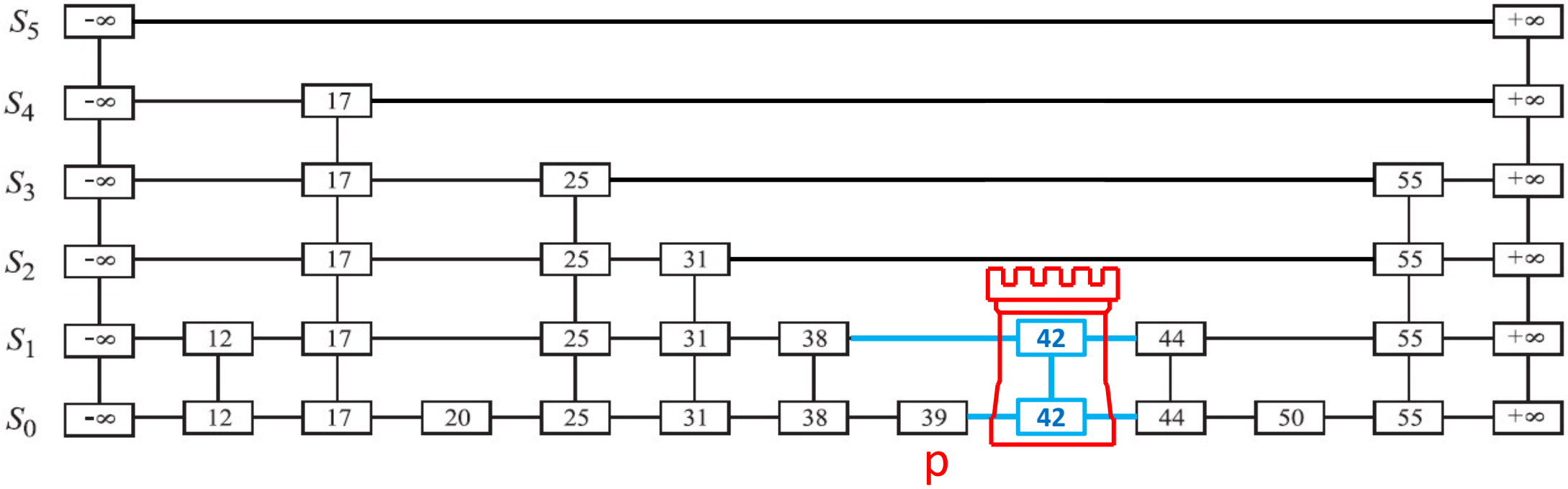
- **Search** for the largest key that is ≤ **42**, and set p to be its position
- **Insert** the entry right after p in the same level



SKIP LIST – INSERTION

Here is an example of how to **insert** an entry with key = **42**:

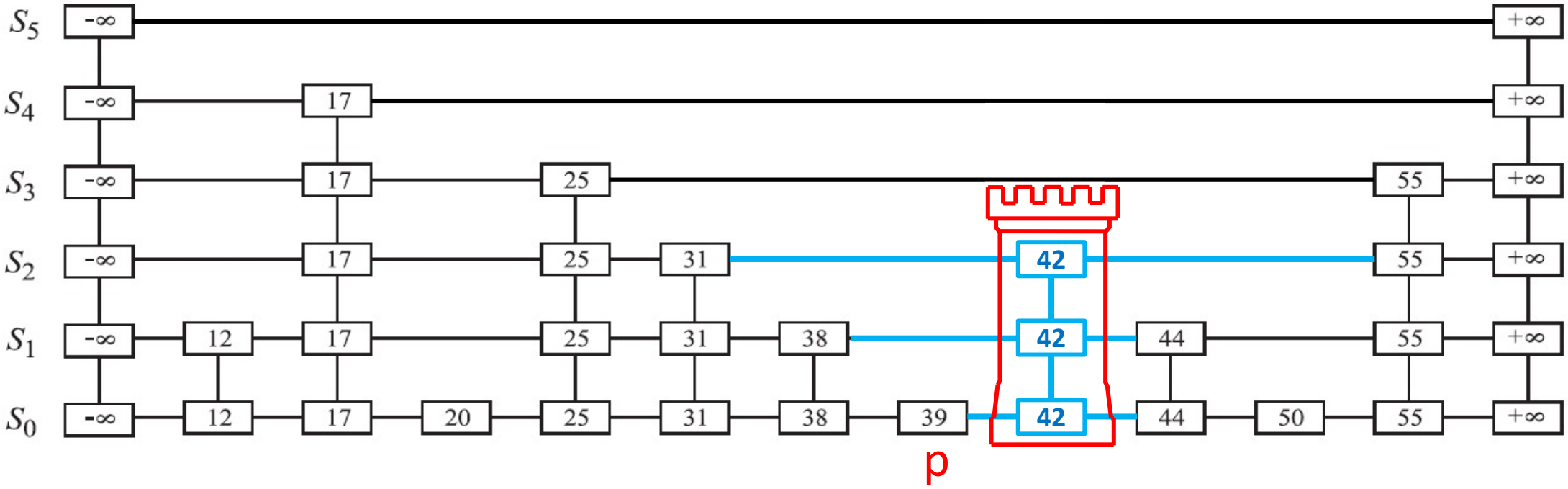
- **Search** for the largest key that is ≤ **42**, and set p to be its position
- **Insert** the entry right after p in the same level
- **Build a tower** at p by repeatedly flip a coin to decide whether to go one more level up!



SKIP LIST – INSERTION

Here is an example of how to **insert** an entry with key = **42**:

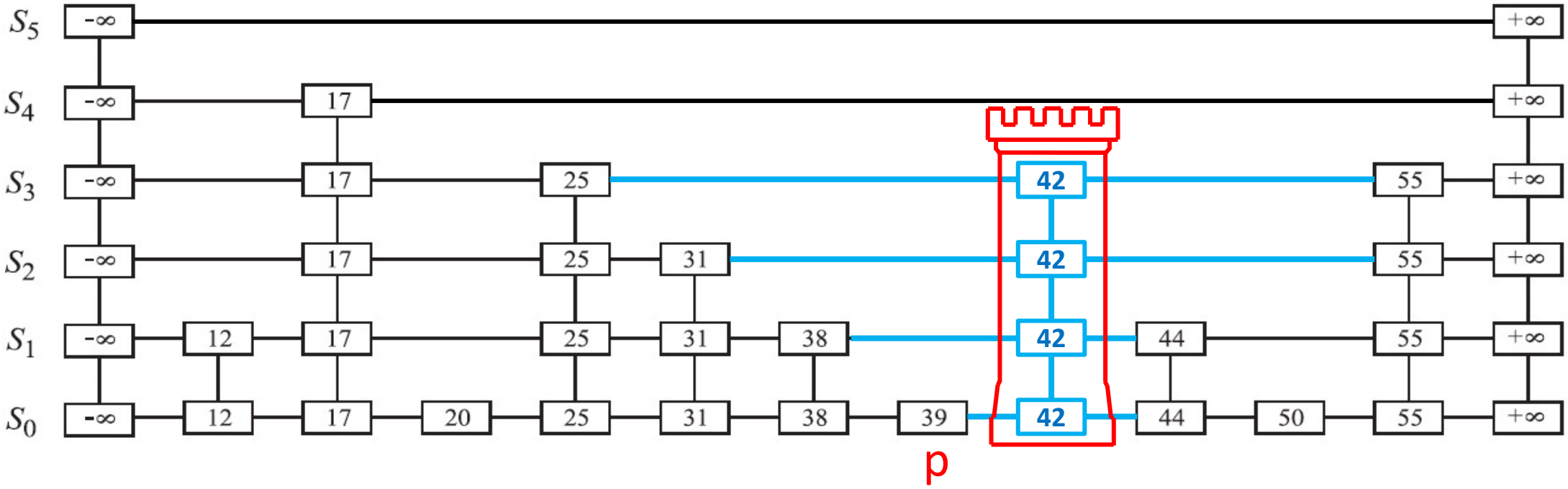
- **Search** for the largest key that is ≤ **42**, and set p to be its position
- **Insert** the entry right after p in the same level
- **Build a tower** at p by repeatedly flip a coin to decide whether to go one more level up!



SKIP LIST – INSERTION

Here is an example of how to **insert** an entry with key = **42**:

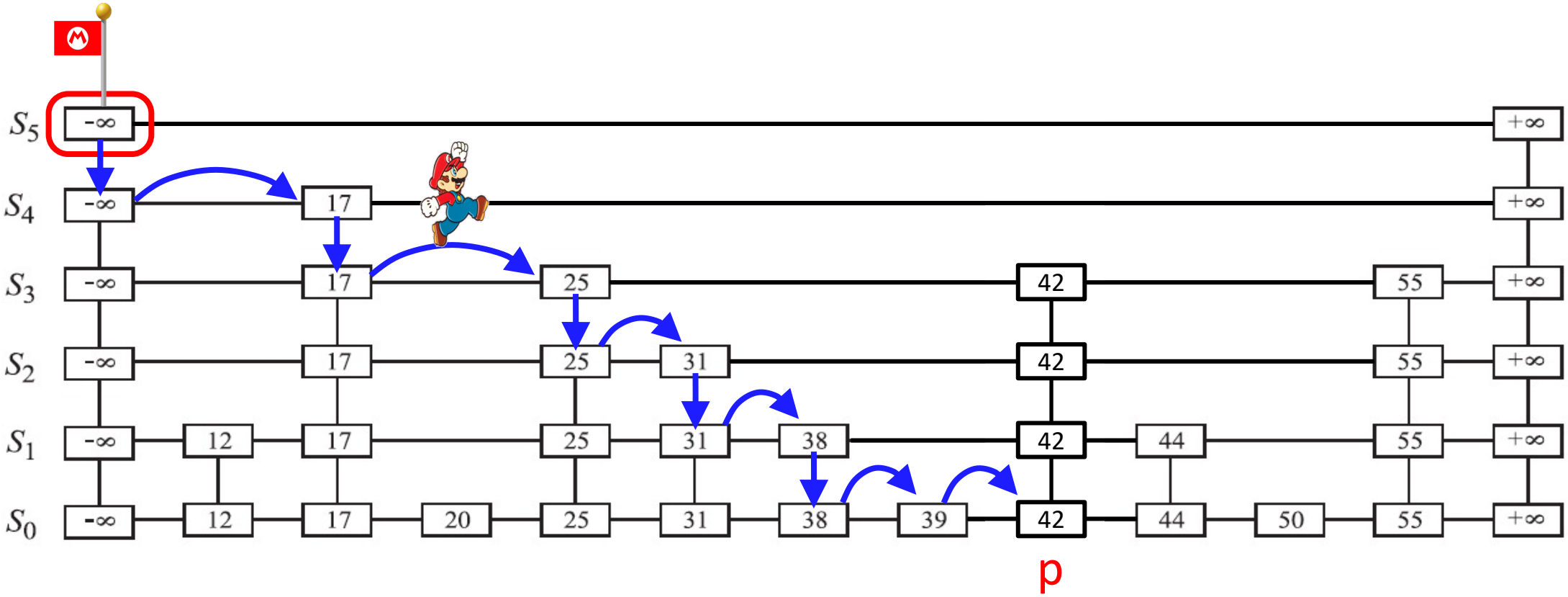
- **Search** for the largest key that is ≤ **42**, and set p to be its position
- **Insert** the entry right after p in the same level
- **Build a tower** at p by repeatedly flip a coin to decide whether to go one more level up!



SKIP LIST – REMOVAL

Here is an example of how to **remove** an entry with key = **42**:

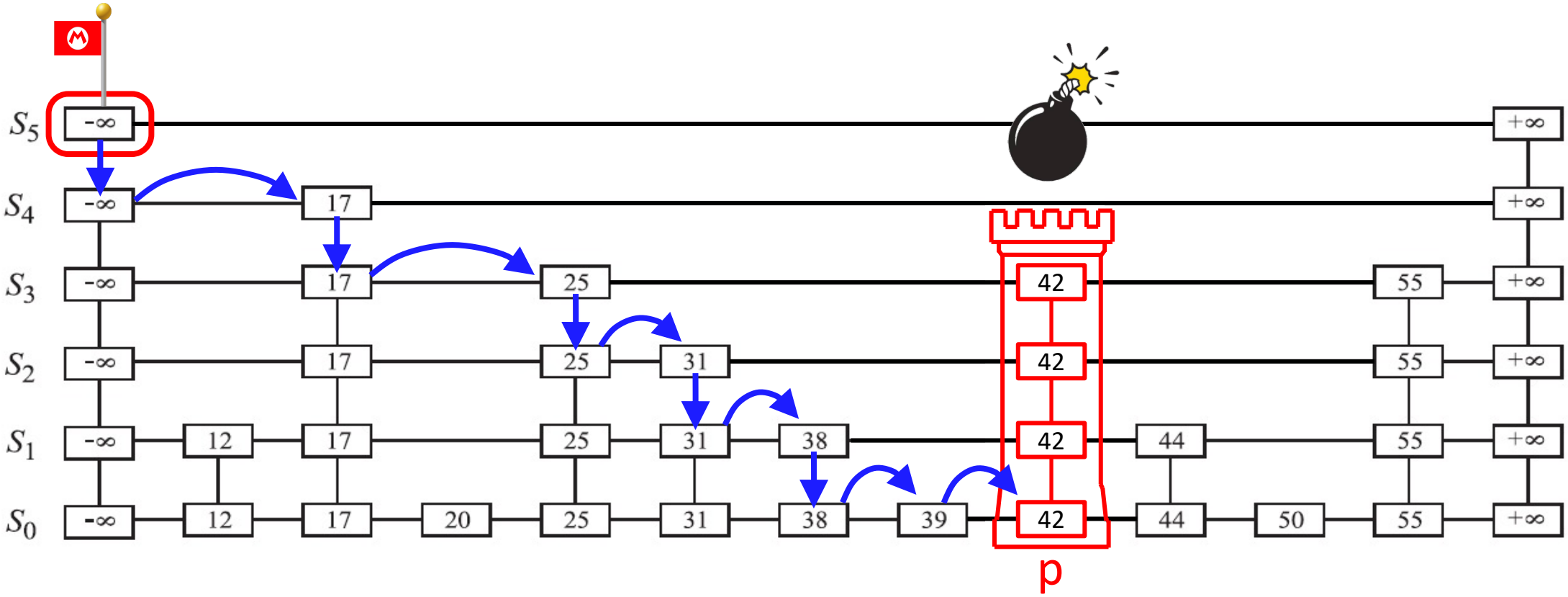
- **Search** for the entry whose key = **42**



SKIP LIST – REMOVAL

Here is an example of how to **remove** an entry with key = **42**:

- **Search** for the entry whose key = **42**
- **Remove** the entire tower at p



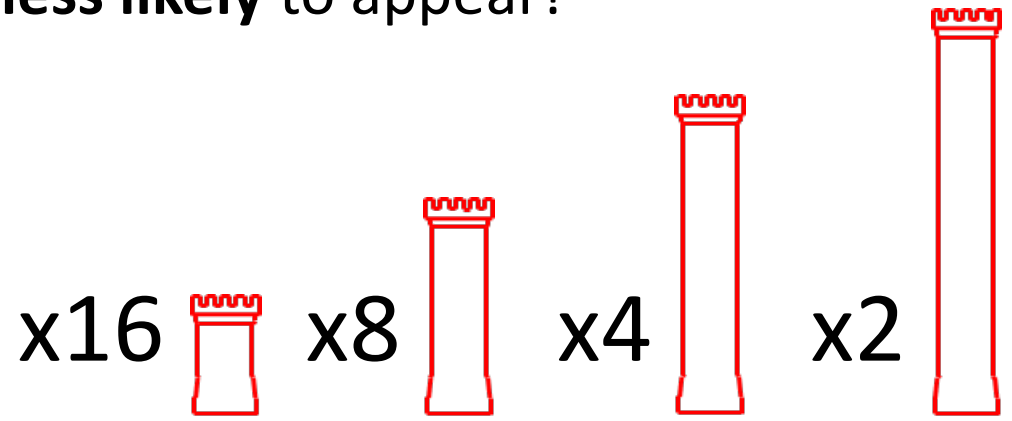
SKIP LISTS – SEARCH ANALYSIS

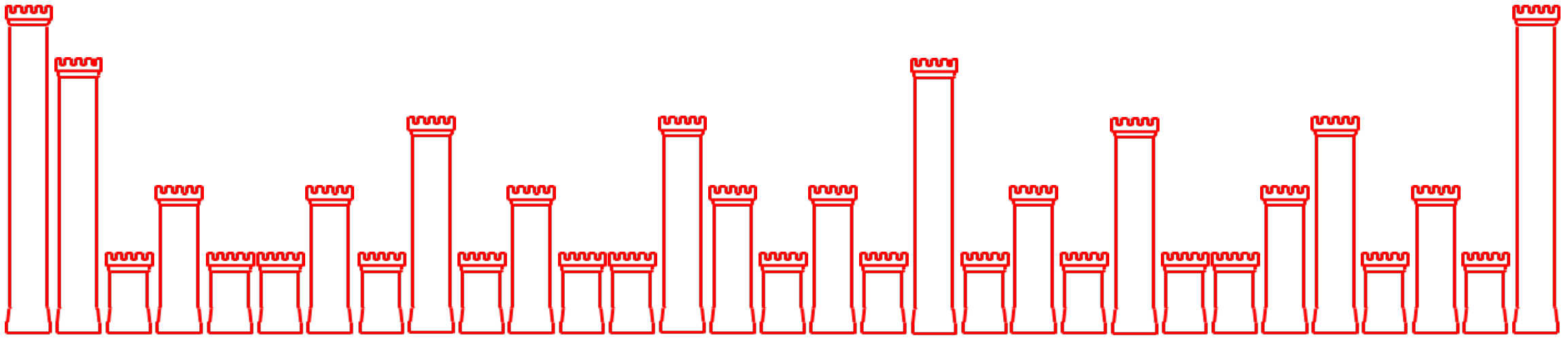
- Remember, searching for an entry whose key = k takes O(log n) with **binary search**, because with each step we slash half of the number of entries to consider

- But why would it also take O(log n) to search for an entry in a **skip list**? Where is the part in which we discard lots of entries to consider?

SKIP LISTS – SEARCH ANALYSIS

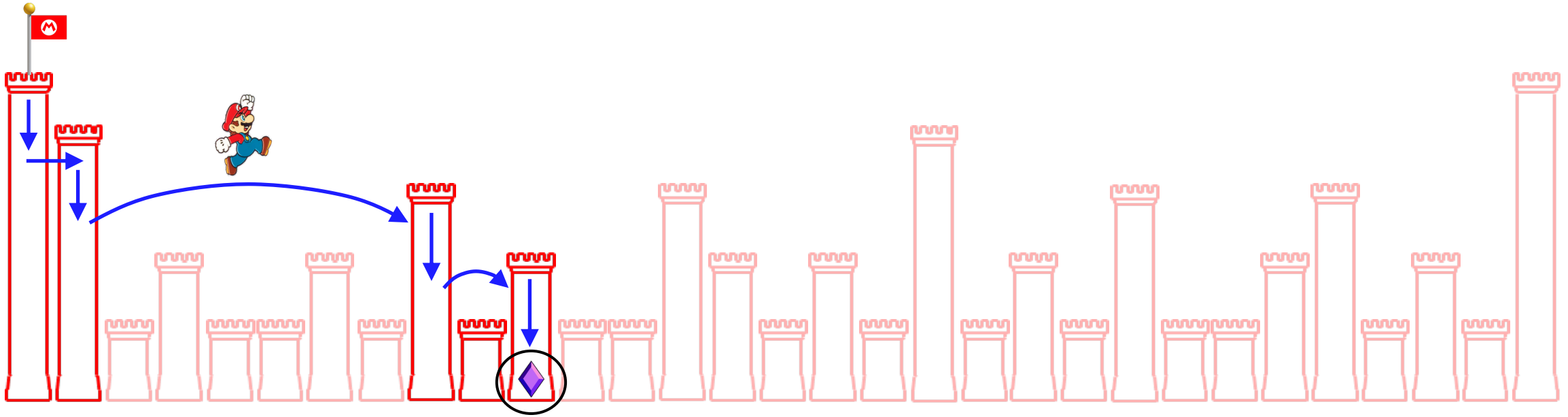
- A slightly higher tower is **twice less likely** to appear!





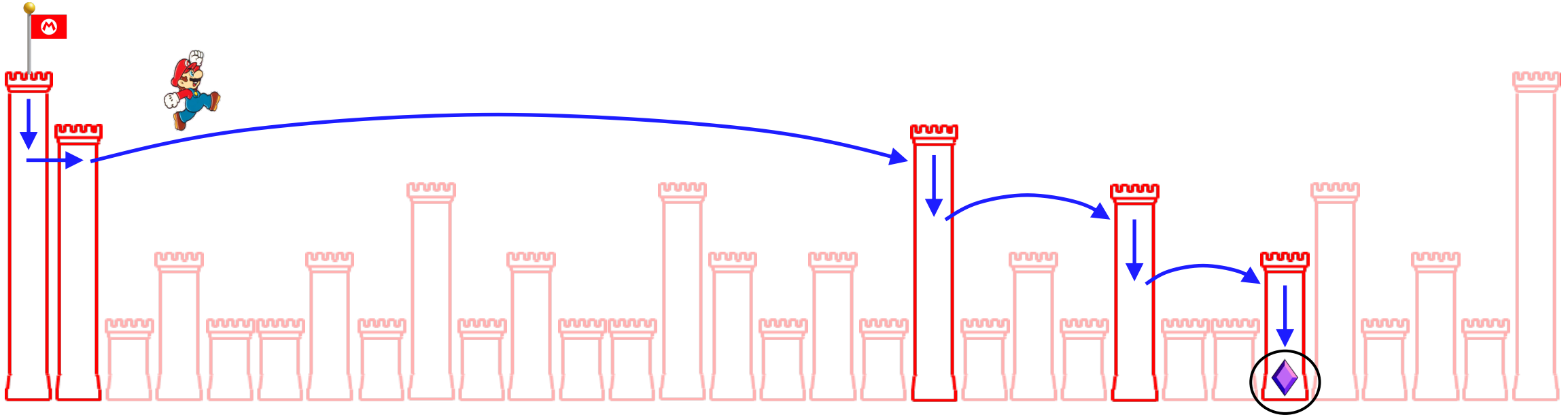
SKIP LISTS – SEARCH ANALYSIS

- A slightly higher tower is **twice less likely** to appear!
- Let us denote **the entry we're searching for** as: 
- If  is **close** to the start, we'll find it quickly and disregard everything after it!



SKIP LISTS – SEARCH ANALYSIS

- A slightly higher tower is **twice less likely** to appear!
- Let us denote **the entry we're searching for** as: 
- If  is **close** to the start, we'll find it quickly and disregard everything after it!
- If it is far from the start, we'll make large jumps across higher towers!



SKIP LISTS – SEARCH ANALYSIS

- A slightly higher tower is **twice less likely** to appear!
- Let us denote **the entry we're searching for** as: 
- If  is **close** to the start, we'll find it quickly and disregard everything after it!
- If it is far from the start, we'll make large jumps across higher towers!
- Actually, wherever  is, we will always skip most towers!

SKIP LISTS – BOUNDING THE HEIGHT

What is the probability that the height of a tower reaches h?

- If you flip a coin **once**, the probability of getting a head is: $\frac{1}{2}$

Possible outcomes:

- Outcome 1: 
- Outcome 2: 

SKIP LISTS – BOUNDING THE HEIGHT

What is the probability that the height of a tower reaches h?

- If you flip a coin **once**, the probability of getting a head is: $\frac{1}{2}$
- If you flip it **twice**, the probability of getting two consecutive heads is: $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$

Possible outcomes:

- Outcome 1: 
- Outcome 2: 
- Outcome 3: 
- Outcome 4: 

SKIP LISTS – BOUNDING THE HEIGHT

What is the probability that the height of a tower reaches h?

- If you flip a coin **once**, the probability of getting a head is: $\frac{1}{2}$
- If you flip it **twice**, the probability of getting two consecutive heads is: $\frac{1}{2} \times \frac{1}{2}$
- If you flip it **three** times, the probability of three consecutive heads is: $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$

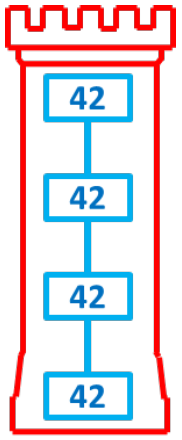
Possible outcomes:

- Outcome 1: 
- Outcome 2: 
- Outcome 3: 
- Outcome 4: 
- Outcome 5: 
- Outcome 6: 
- Outcome 7: 
- Outcome 8: 

SKIP LISTS – BOUNDING THE HEIGHT

What is the probability that the height of a tower reaches h?

- If you flip a coin **once**, the probability of getting a head is: $\frac{1}{2}$
- If you flip it **twice**, the probability of getting two consecutive heads is: $\frac{1}{2} \times \frac{1}{2}$
- If you flip it **three** times, the probability of three consecutive heads is: $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$
- If you flip it i times, the probability of i consecutive heads is: $\frac{1}{2^i}$
- Thus, for any given tower, the probability that its height reaches i is : $\frac{1}{2^i}$



SKIP LISTS – BOUNDING THE HEIGHT

- We found that, for any tower, the probability that its height reaches i is $\frac{1}{2^i}$
- Also, from Probability Theory, we know that the **probability that any one of** n **different events occurs** is **at most** the **sum** of the probabilities that each occurs.
- Hence, the **probability that at least one of** n **different towers reaches** a height of i is:

$$p_i \leq \frac{n}{2^i}$$

- Thus, the probability that the height of the entire skip list, h, reaches, say, $3 \log n$ is:

$$p_{3 \log n} \leq \frac{n}{2^{3 \log n}} = \frac{n}{n^3} = \frac{1}{n^2}$$

- For example, if n=10,000, the probability that h reaches $3 \log n$ is just 1 in 100,000,000
- Therefore, given a constant c, the probability that $\mathrm{h} \geq \mathrm{c} \log n$ is at most $\frac{1}{n^{c-1}}$
- Thus, with high probability, the height h of the skip list is $O(\log n)$

HASH TABLE VS. SEARCH TABLE VS. SKIP LIST

Method	Hash table	Search Table	Skip List
find	$O(1)$ expected $O(n)$ worst case	$O(\log n)$	$O(\log n)$ expected
insert	$O(1)$	$O(n)$	$O(\log n)$ expected
erase	$O(1)$ expected $O(n)$ worst case	$O(n)$	$O(\log n)$ expected

DICTIONARIES

DICTIONARY

- A dictionary is similar to a map, except that multiple entries can have the same key.

  size(): Return the **number of entries** in *D*.

  empty(): Return **true if *D* is empty** and false otherwise.

  begin(): Return an iterator to the **first entry of *D***.

  end(): Return an iterator to a **position just beyond the end of *D***.

  insert(*k*,*v*): **Insert entry (k,v) into *D***, returning an iterator referring to the newly created entry.

  erase(*p*): **Remove from *D* the entry referenced by iterator *p***; ERROR if *p* points to the end sentinel.

  erase(*k*): **Remove from *D* an arbitrary entry whose key = *k***; ERROR if *D* has no such entry.

  find(*k*): Return **an iterator to an entry whose key = k**; unless no such entry exists in D, then return the special iterator end.

  findAll(*k*): Return a pair of iterators (*b*,*e*), such that **all the entries whose key = *k*** lie in the range from *b* up to, but not including, *e*.

DICTIONARY – EXAMPLE

Example:

- In the **Output** column, we use the notation:

$$p_i : [(k, v)]$$

to mean that the operation returns an iterator p_i that refers to the entry (k, v)

Operation	*Output*	*Dictionary*
empty()	**true**	$\emptyset$
insert$(5,A)$	$p_1 : [(5,A)]$	$\{(5,A)\}$
insert$(7,B)$	$p_2 : [(7,B)]$	$\{(5,A),(7,B)\}$
insert$(2,C)$	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C)\}$
insert$(8,D)$	$p_4 : [(8,D)]$	$\{(5,A),(7,B),(2,C),(8,D)\}$
insert$(2,E)$	$p_5 : [(2,E)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(7)	$p_2 : [(7,B)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(4)	end	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
find(2)	$p_3 : [(2,C)]$	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
findAll(2)	(p_3, p_4)	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
size()	5	$\{(5,A),(7,B),(2,C),(2,E),(8,D)\}$
erase(5)	–	$\{(7,B),(2,C),(2,E),(8,D)\}$
erase(p_3)	–	$\{(7,B),(2,E),(8,D)\}$
find(2)	$p_5 : [(2,E)]$	$\{(7,B),(2,E),(8,D)\}$

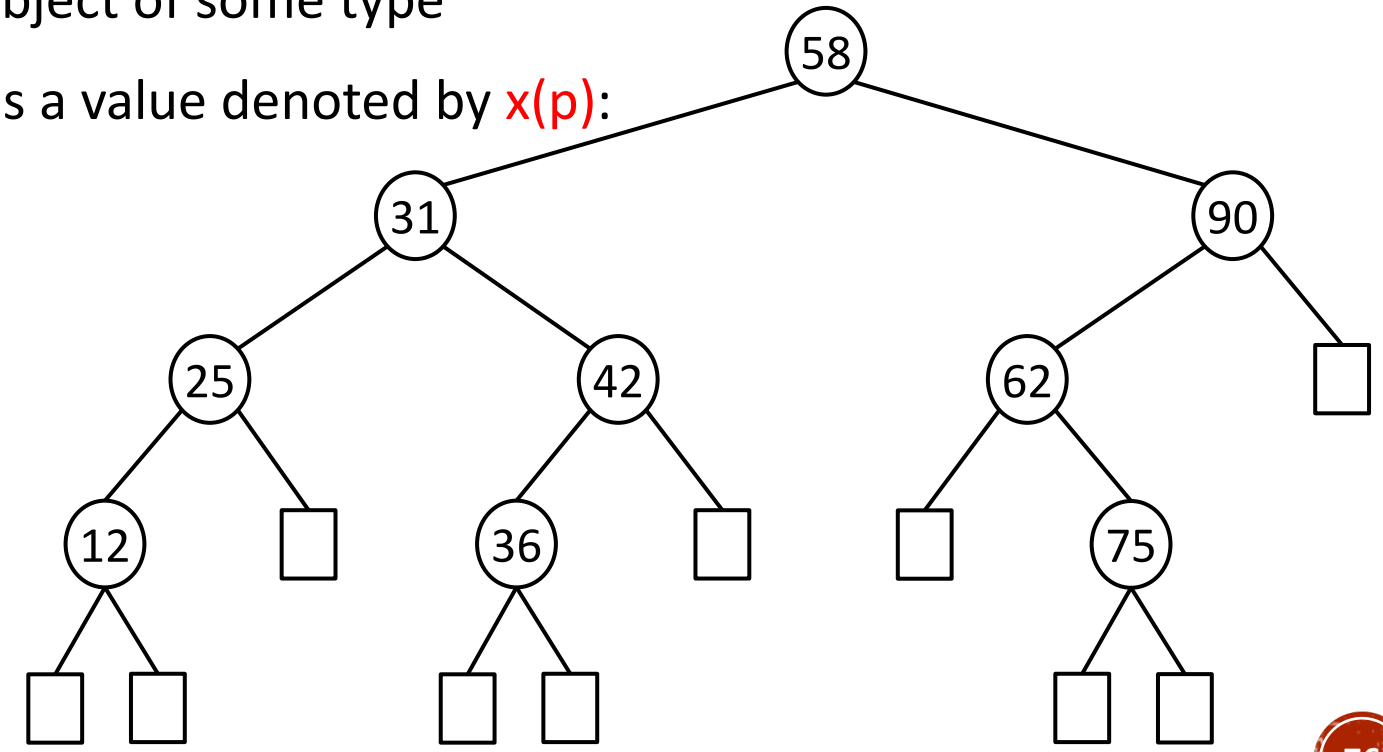
CHAPTER 10:
SEARCH TREES

BINARY SEARCH TREES

BINARY SEARCH TREE

An excellent data structure for storing the entries of a map or a dictionary is a binary search tree, which is defined as a **proper binary tree** where:

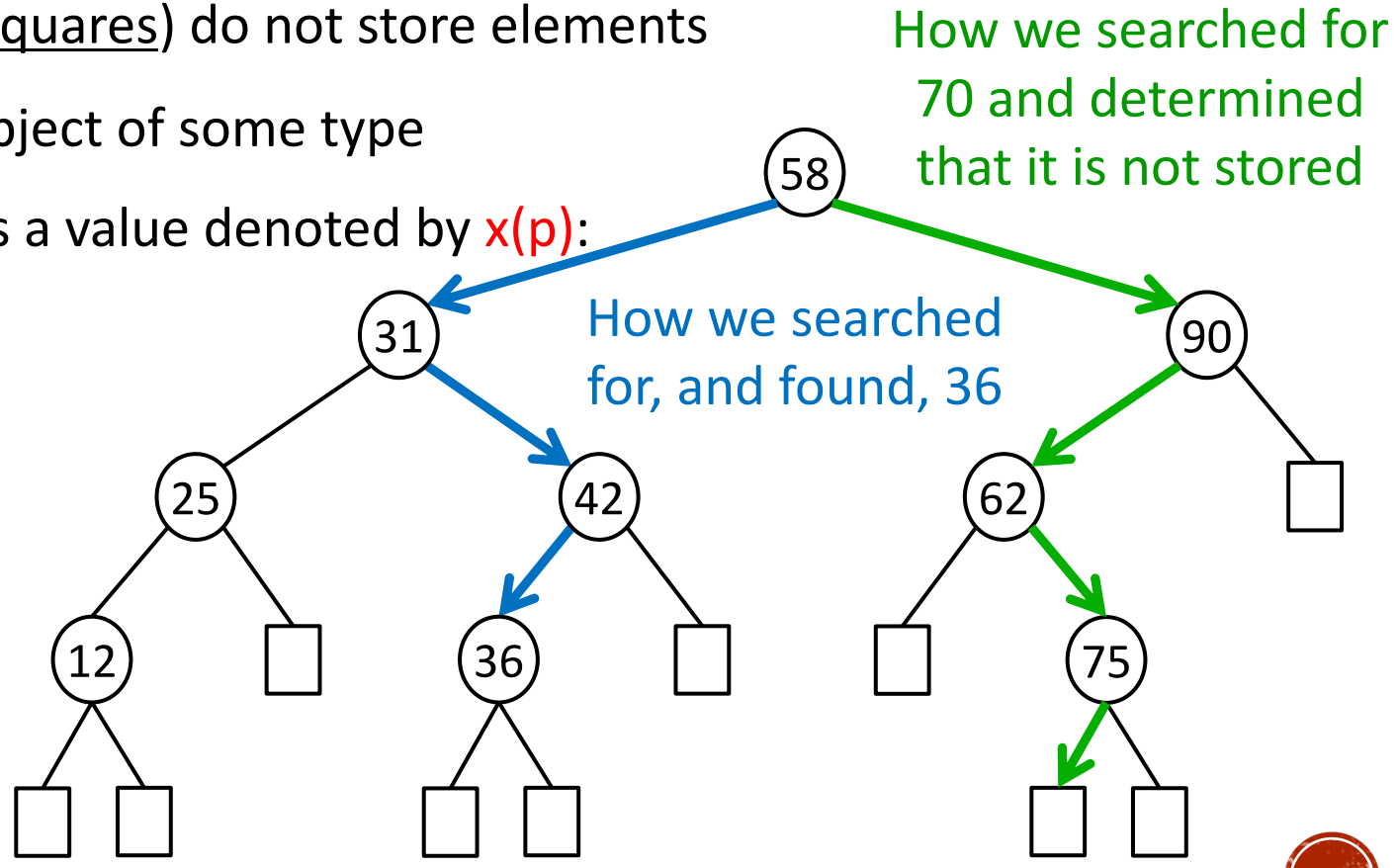
- **External nodes** (represented as squares) do not store elements
- Each **internal node** p stores an object of some type
- For each **internal node** p, there is a value denoted by x(p):
  - Any object stored in the **left subtree** of p has such a value that is $\leq$ x(p)
  - Any object stored in the **right subtree** of p has such a value that is $\geq$ x(p)



BINARY SEARCH TREE

An excellent data structure for storing the entries of a map or a dictionary is a binary search tree, which is defined as a **proper binary tree** where:

- **External nodes** (represented as squares) do not store elements
- Each **internal node** p stores an object of some type
- For each **internal node** p, there is a value denoted by x(p):
  - Any object stored in the **left subtree** of p has such a value that is $\leq$ x(p)
  - Any object stored in the **right subtree** of p has such a value that is $\geq$ x(p)



BINARY SEARCH TREE

Here is a **recursive algorithm**:

- TreeSearch(k, v): searches for a key k in the subtree rooted at node v
- To search for k in the entire tree, T, simply call TreeSearch(k, T.root())

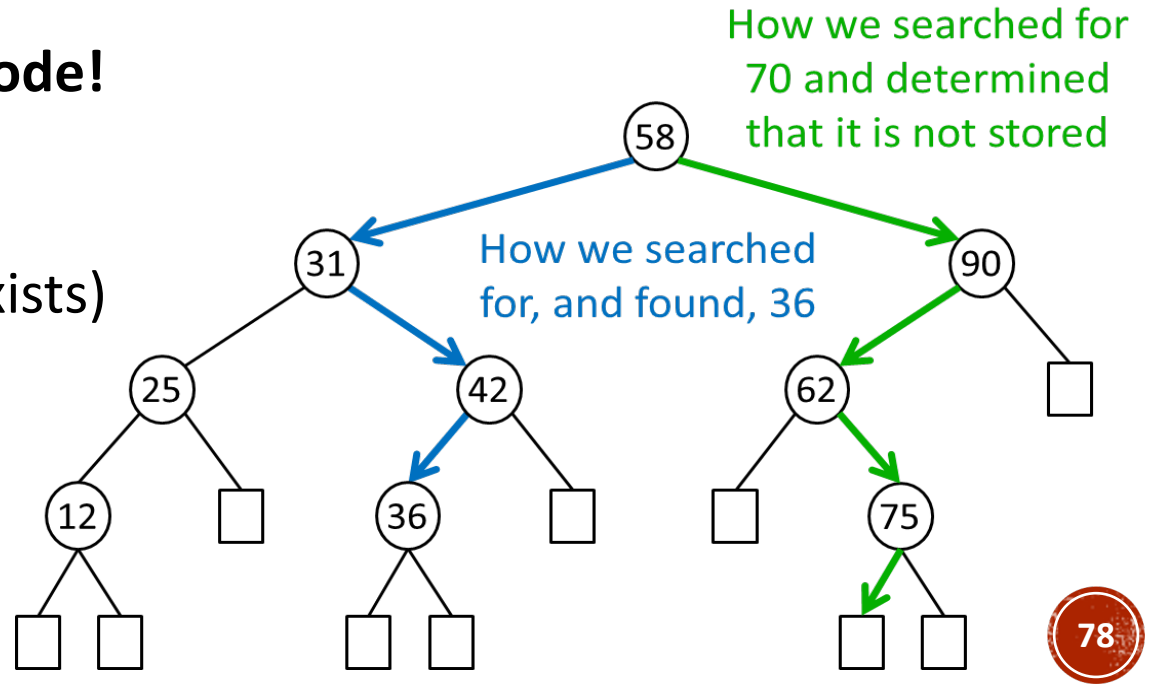
Note that this algorithm always returns a node!

Specifically, it return one of the following:

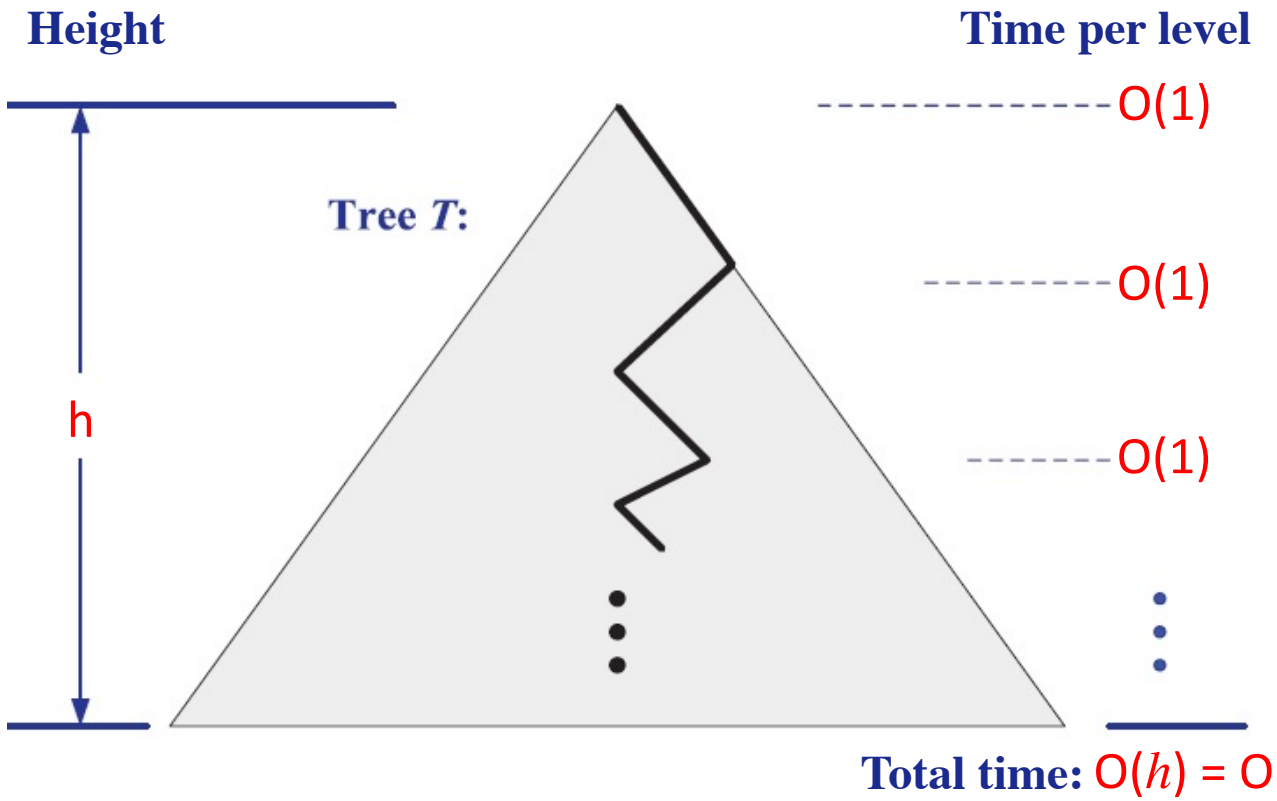
1. A nodes whose key = k (if such a node exists)
2. An external node where it would be suitable to insert a node whose key = k

This second output will be handy later on when implementing "insert"

```
Algorithm TreeSearch( k, v ):
    if T.isExternal(v) then
        return v {return external node}
    if k < v.key() then
        return TreeSearch( k, T.left(v) )
    else if k > v.key() then
        return TreeSearch( k, T.right(v) )
    return v  {v.key() must be equal to k}
```

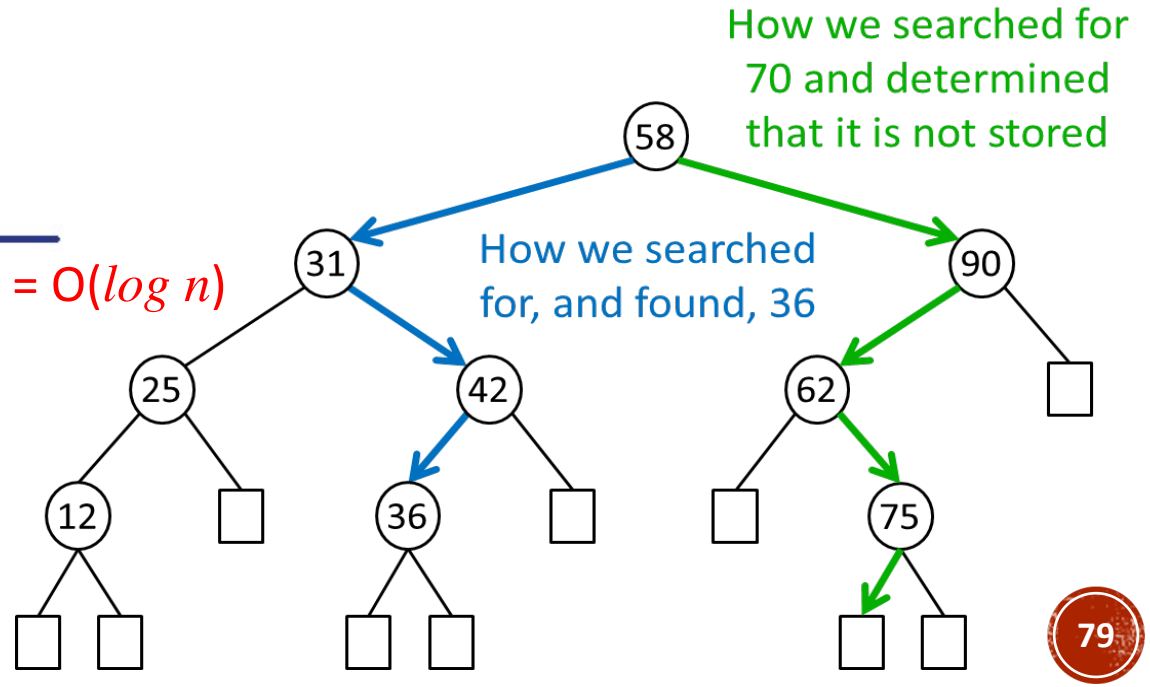


What is the complexity of this algorithm?



Total time: O(h) = O($log\ n$)

```
Algorithm TreeSearch( k, v ):
    if T.isExternal(v) then
        return v {return external node}
    if k < v.key() then
        return TreeSearch( k, T.left(v) )
    else if k > v.key() then
        return TreeSearch( k, T.right(v) )
    return v  {v.key() must be equal to k}
```



Since we spend O(1) time per node encountered in the search, the function find(k) on a map runs in O(h) time, where h is the height of the binary search tree