COMP2123
Data structures and Algorithms

Lecture 12: Randomized Algorithms
[GT 19.1 and 19.6]

André van Renssen
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Reasons for using randomization:

- Sampling data from a large population or dataset
- Avoid pathological worst-case examples
- Avoid predictable outcomes
- Allow for simpler algorithms

Randomized algorithms

Randomized algorithms are algorithms where the behaviour doesn't depend solely on the input. It also depends (in part) on random choices or the values of a number of random bits.

Today:

- Generating random permutations
- Skip lists

Generating random permutations

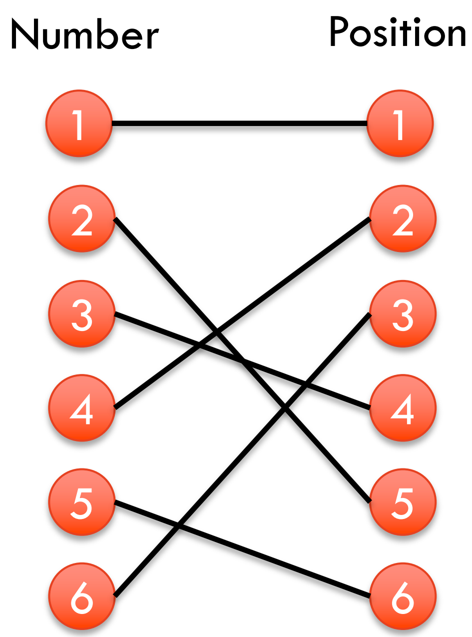
Input: An integer n.

Output: A permutation of {1, …, n} chosen uniformly at random, i.e., every permutation has the same probability of being generated.

Example:

n = 6

<1,4,6,3,2,5>



Generating random permutations

What are random permutations used for?

- Many algorithms whose input is an array perform better in practice after randomly permuting the input (for example, QuickSort).

- Can be used to sample k elements without knowing k in advance by picking the next element in the permuted order when needed.

- Can be used to assign scarce resources.

稀缺资源

- Can be a building block for more complex randomized algorithms.

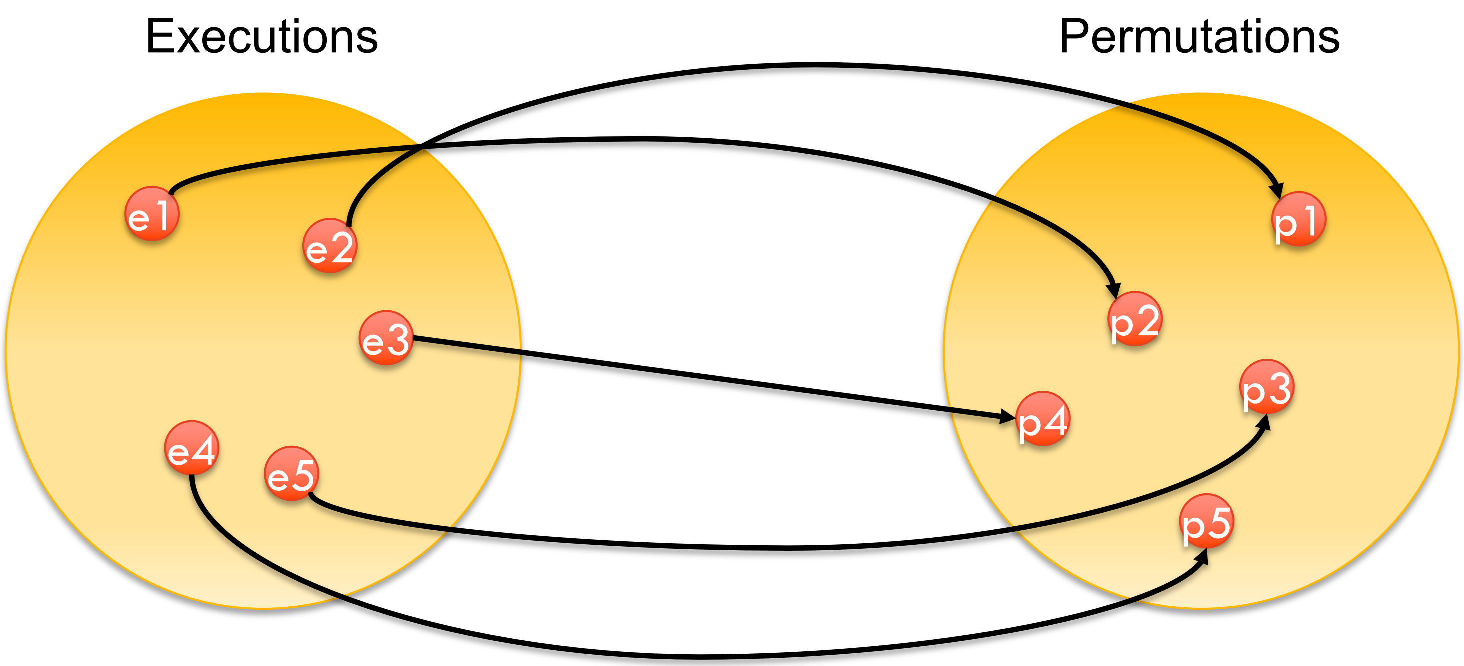
First (incorrect) attempt

```
def permute(A):

  # permute A in place
  n ← length of array A

  for i in {0, ..., n-1} do
    # swap A[i] with random position
    j ← random number in {0, ..., n-1}
    A[i], A[j] ← A[j], A[i]

  return A
```

Note that since j is picked at random, different executions lead to different outcomes

So, why is this incorrect?

For all permutations to be equally likely, we want that every permutation is generated by the same number of possible executions.



First (incorrect) attempt: Analysis

Number of executions:

$$\underbrace{n * n * n * \ldots * n}_{\text{n times}} = n^n$$

Number of permutations:

$$1 * 2 * 3 * \ldots * n = n!$$

n^n isn't divisible by $n!$

Example:

n = 3

$n^n = 27$

$n! = 6$

27 isn't a multiple of 6, so some permutations are more likely than others.

```
def permute(A):

  # permute A in place
  n ← length of array A

  for i in {0, ..., n-1} do
    # swap A[i] with random position
    j ← random number in {0, ..., n-1}
    A[i], A[j] ← A[j], A[i]

  return A
```

Second attempt

```
def FisherYates(A):

  # permute A in place
  n ← length of array A

  for i in {0, ..., n-1} do
    # swap A[i] with random position
    j ← random number in {i, ..., n-1}
    A[i], A[j] ← A[j], A[i]

  return A
```

Note that since j is picked at random, different executions lead to different outcomes

Second attempt: Analysis

Number of executions:
$1 * 2 * 3 * \ldots * n = n!$

Number of permutations:
$1 * 2 * 3 * \ldots * n = n!$

Observation: Every execution leads to a different permutation.

```
def FisherYates(A):

  # permute A in place
  n ← length of array A

  for i in {0, ..., n-1} do
    # swap A[i] with random position
    j ← random number in {i, ..., n-1}
    A[i], A[j] ← A[j], A[i]

  return A
```

Example: To generate <3,2,4,1> starting from <1,2,3,4>
<1,2,3,4> → <3,2,1,4>, i=0 and j=2
<3,2,1,4> → <3,2,1,4>, i=1 and j=1
<3,2,1,4> → <3,2,4,1>, i=2 and j=3
<3,2,4,1> → <3,2,4,1>, i=3 and j=3

Second attempt: Analysis

Theorem:

The Fisher-Yates algorithm generates a permutation uniformly at random.

Proof:

- Every execution of the algorithm happens with probability $1/n!$.
- Each execution generates a different permutation.
- Hence, the probability that a specific permutation is generated is $1/n!$, for all possible permutations of $\langle 1, 2, \ldots, n \rangle$.

Skip lists

Another way to implement a Map.

So, why are we looking at another different way of doing this?

- Relatively simple data structure that's built in a randomized way
- No need for rebalancing like in AVL trees
- Still has O(log n) expected worst-case time (this is NOT average case)

Applications:

- Various database systems use it
- Concurrent/parallel computing environments

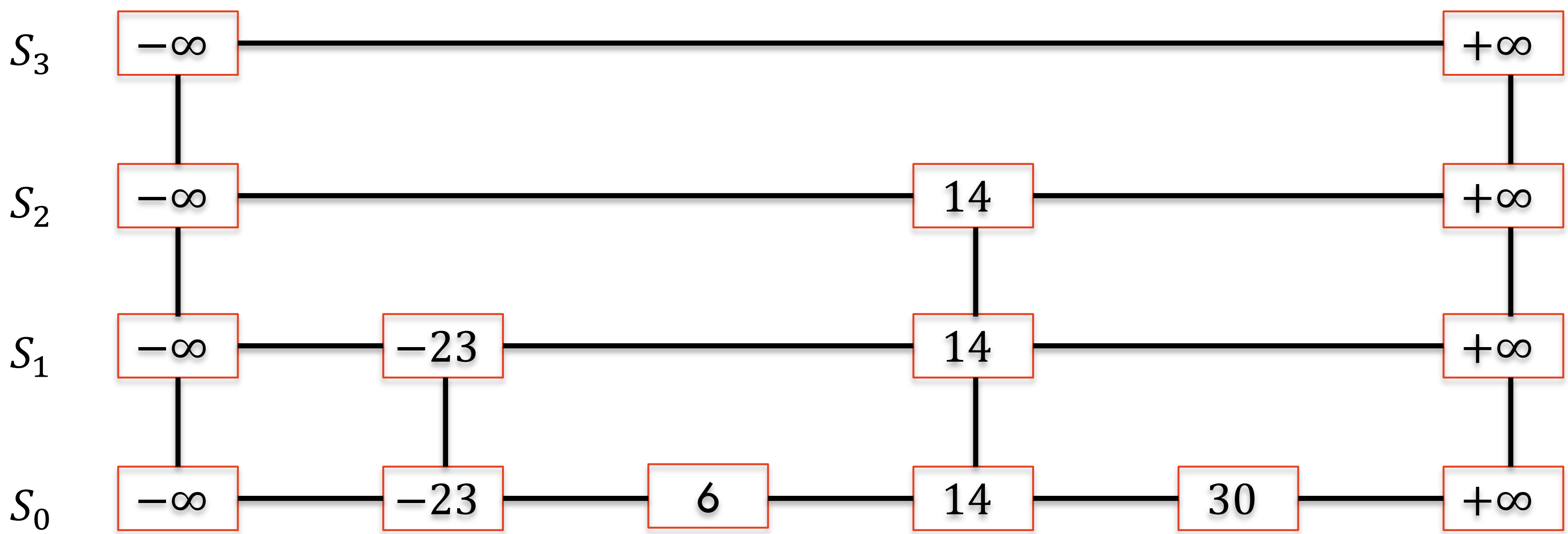
The Map ADT (recap)

- get(k): if the map M has an entry with key k, return its associated value
- put(k, v): if key k is not in M, then insert (k, v) into the map M; else, replace the existing value associated to k with v
- remove(k): if the map M has an entry with key k, remove it
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterable collection of the values in M

Skip lists

Leveled structure, where every level is a subset of the one below it.

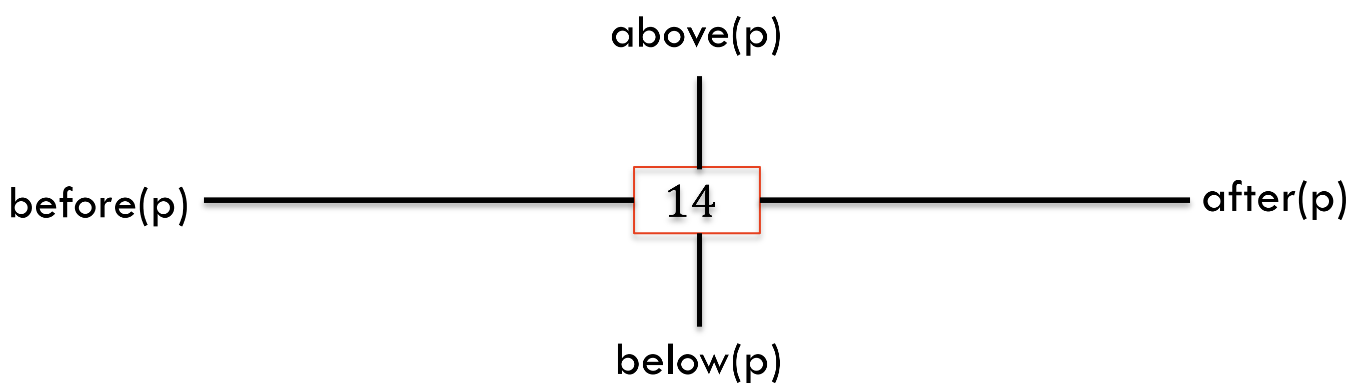
Next level's elements determined by coin flips.



Skip lists

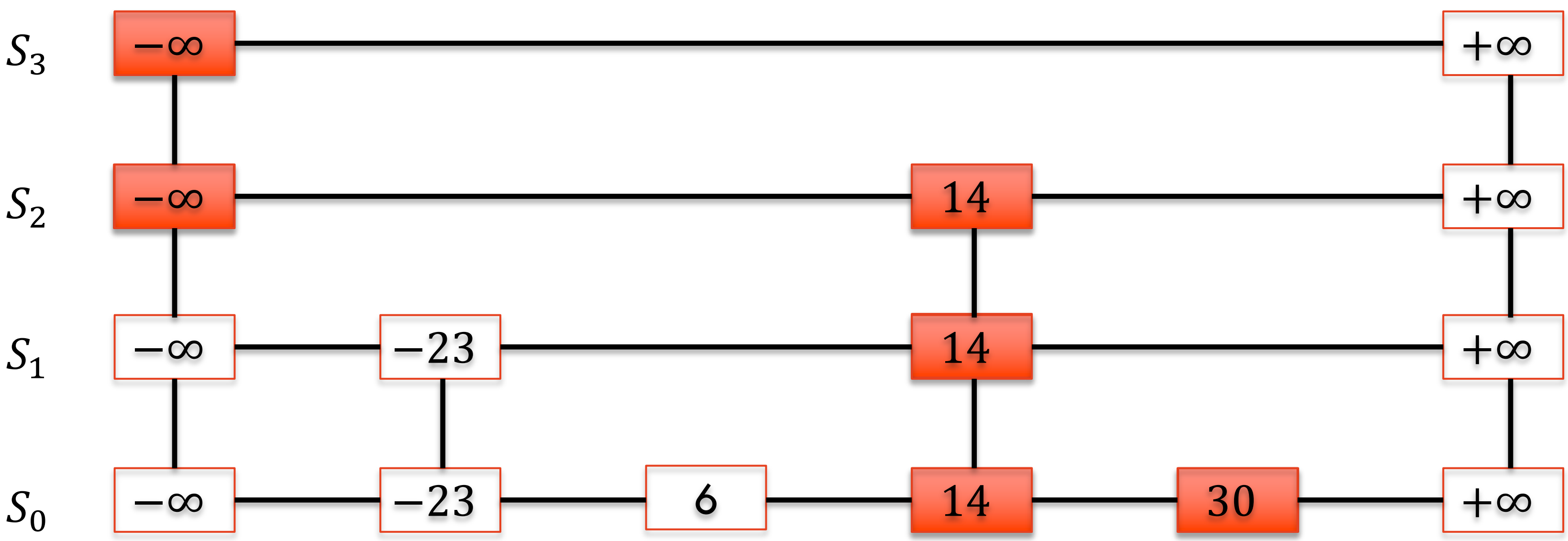
A node p has pointer to:

- after(p): Node following p on same level.
- before(p): Node preceding p in the same level.
- above(p): Node above p in the same tower.
- below(p): Node below p in the same tower.



Search

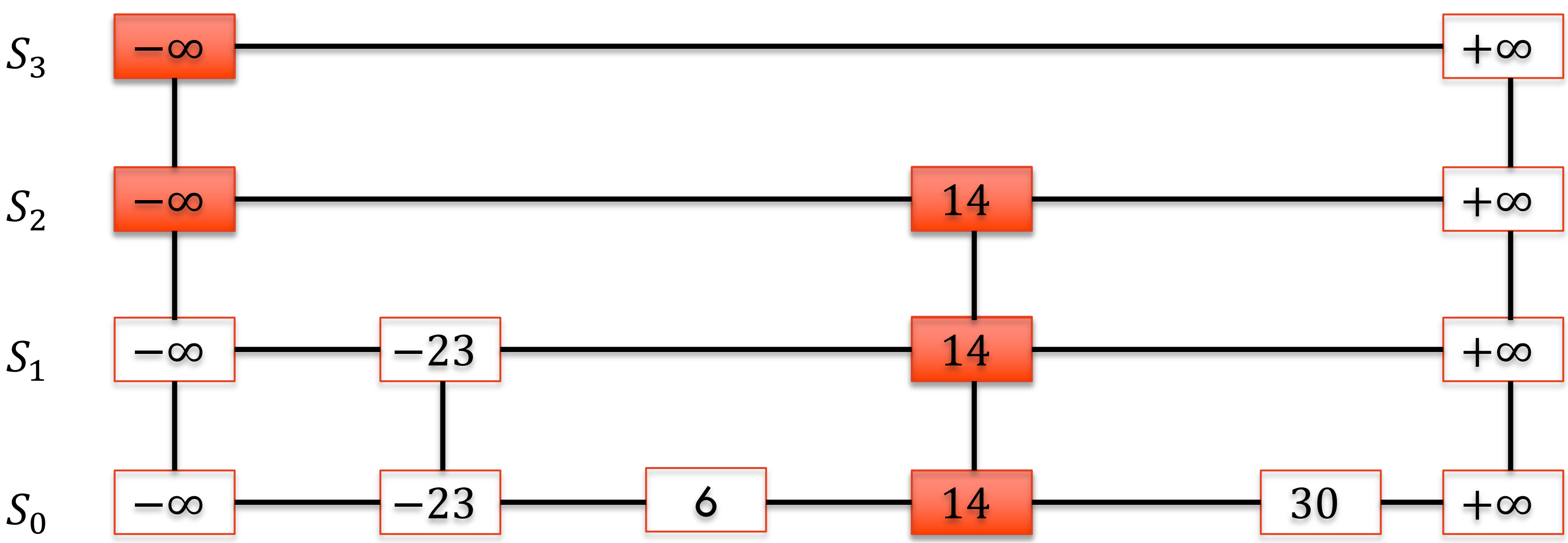
```
def search(p,k):
  while below(p) ≠ null do
    p ← below(p)
    while key(after(p)) ≤ k do
      p ← after(p)
  return p
```



Example: search(topleft node, 30)

Insertion

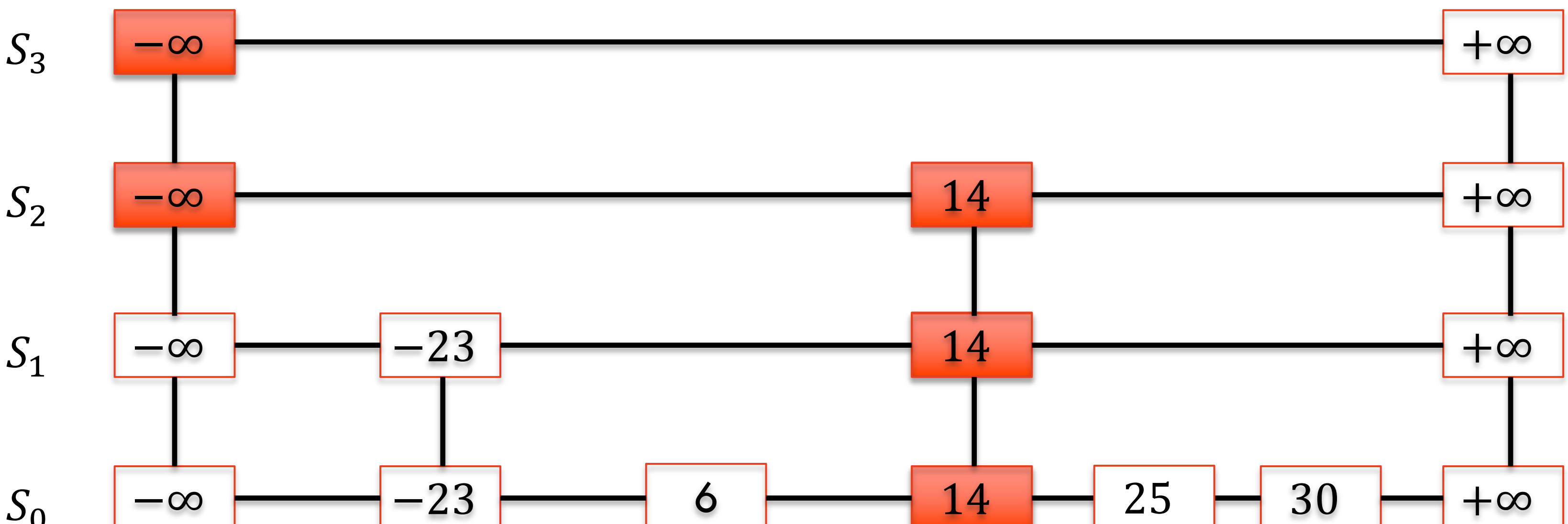
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Insertion

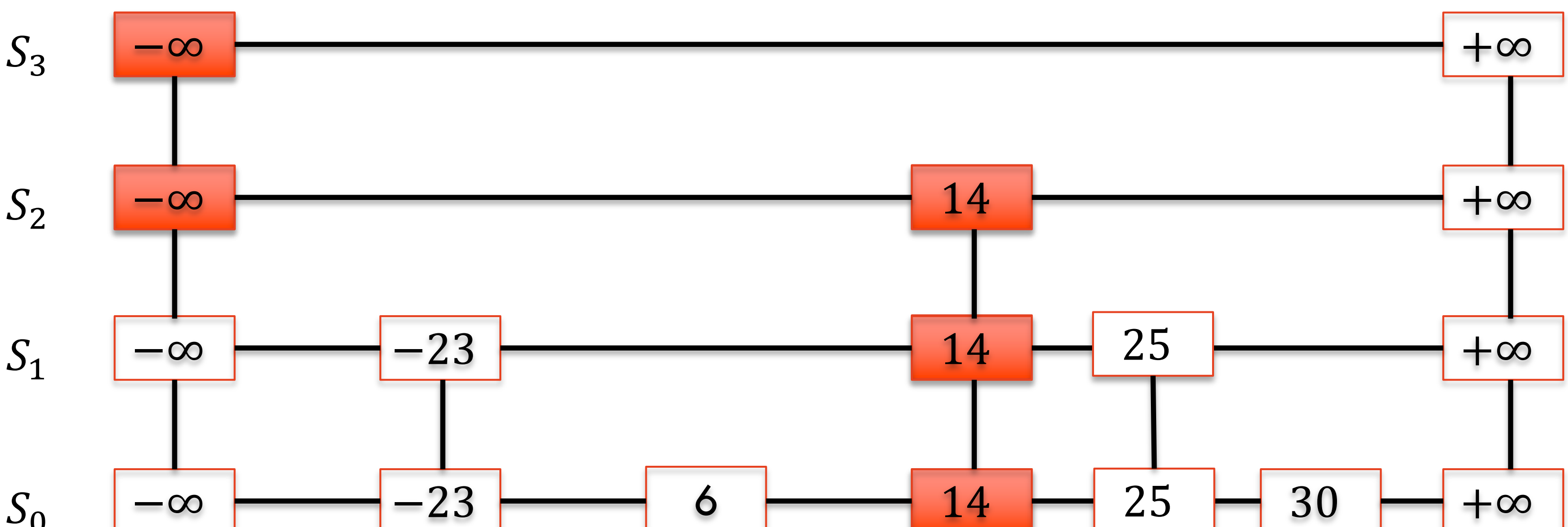
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Insertion

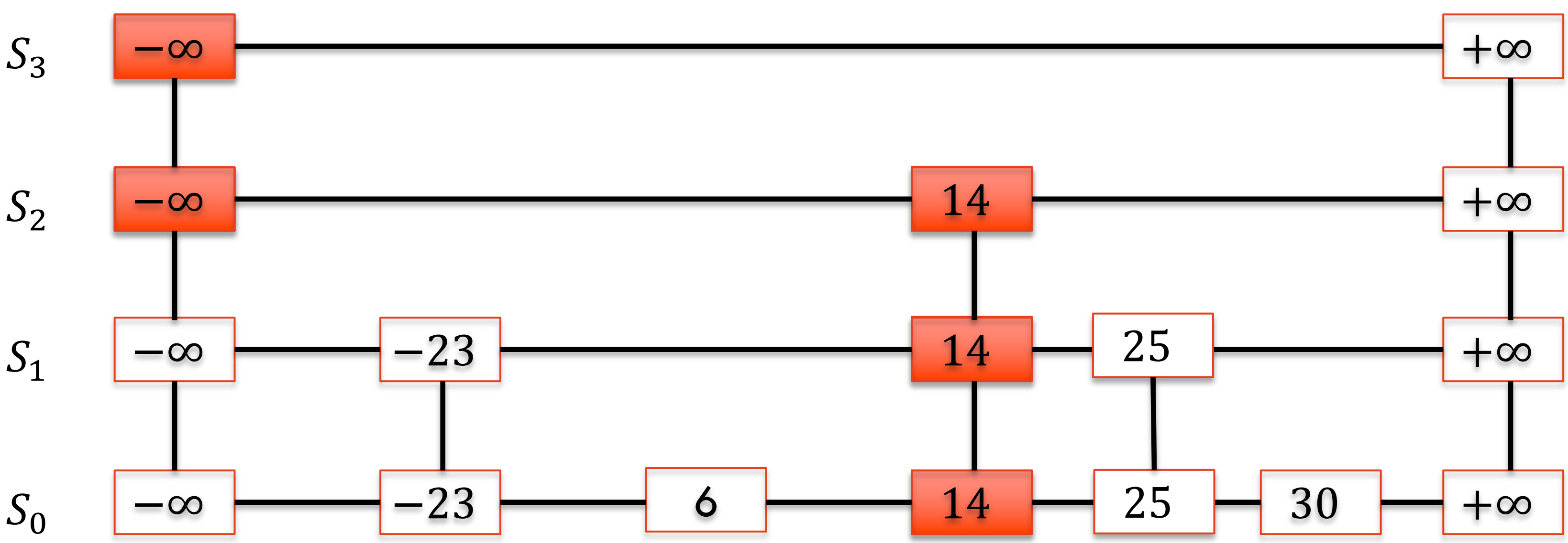
```
def insert(p,k):
  p ← search(p,k)
  q ← insertAfterAbove(p,null,k)
  while coin flip is heads do
    while above(p) = null do
      p ← before(p)
    p ← above(p)
    q ← insertAfterAbove(p,q,k)
```



Example: insert(topleft node, 25)

Removal

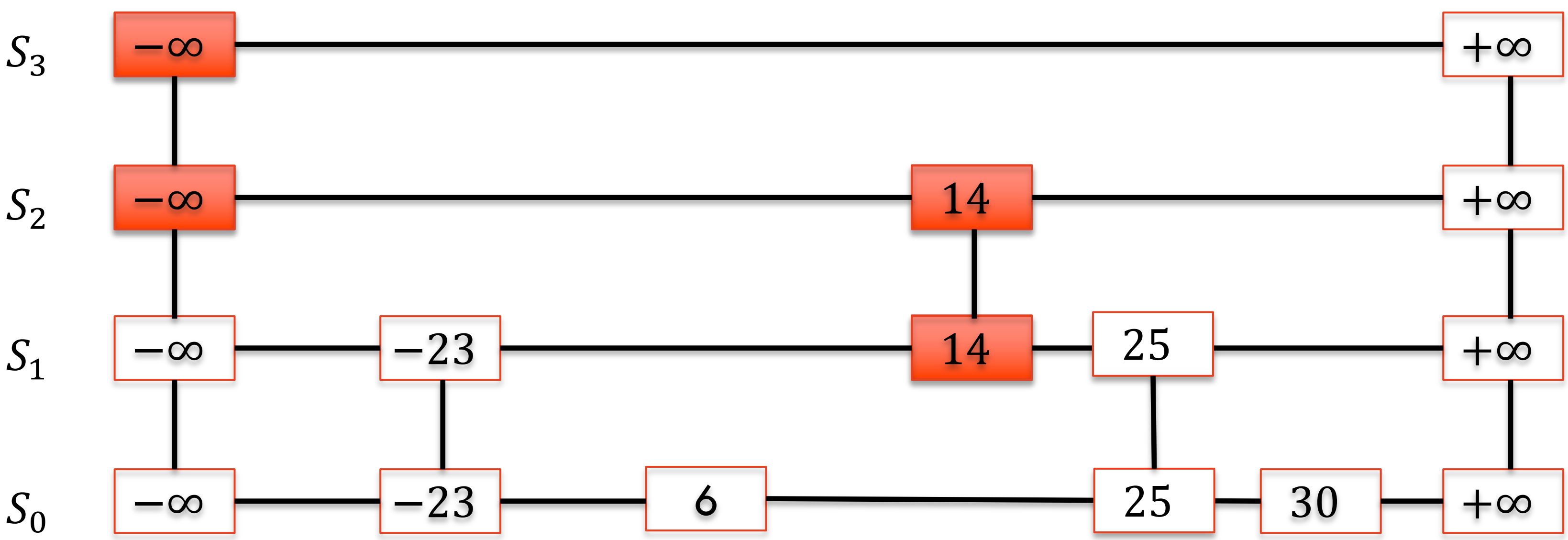
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

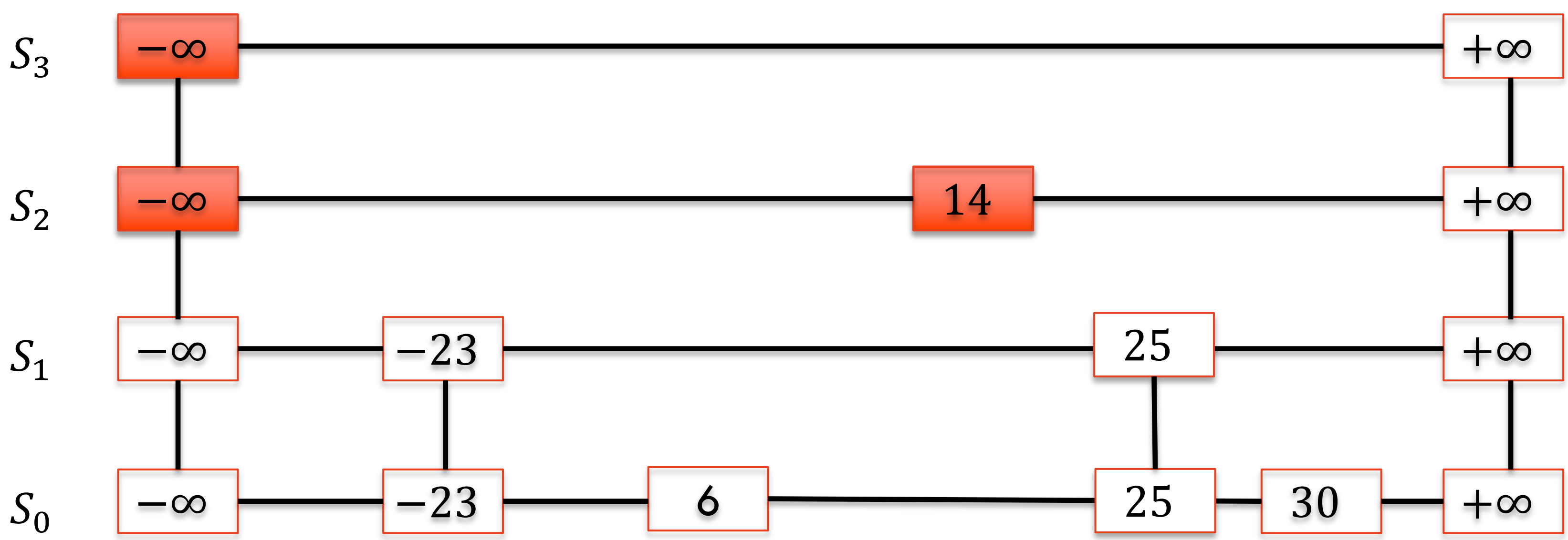
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

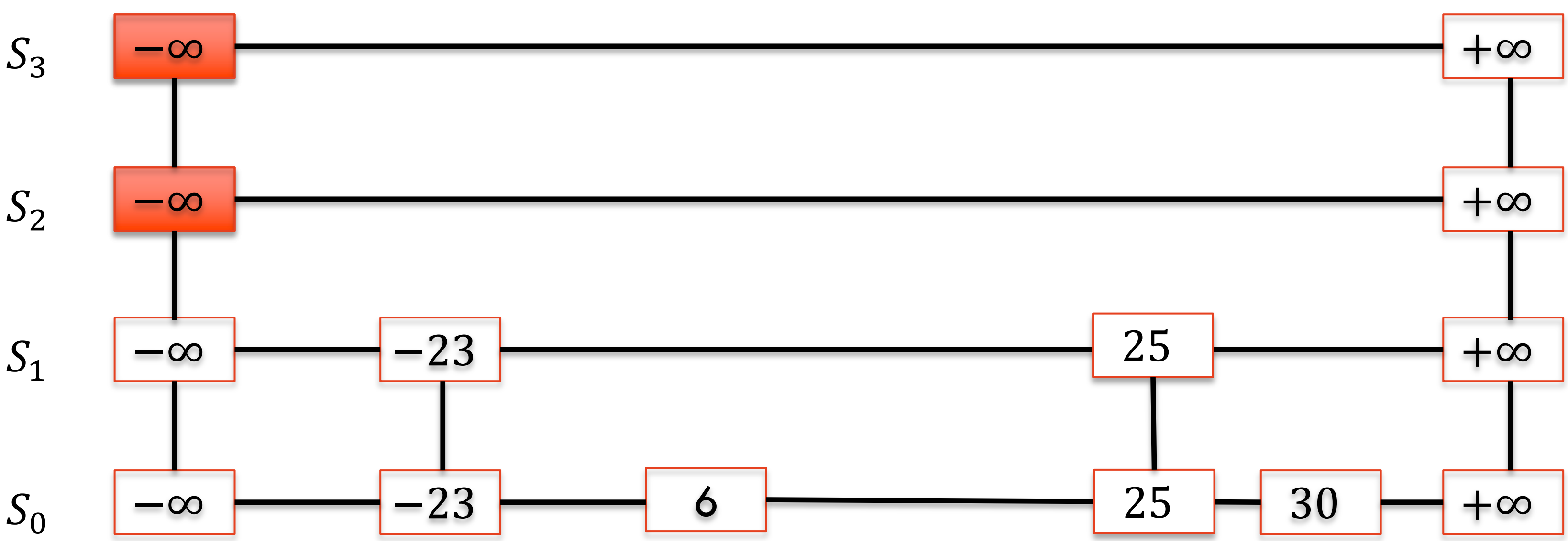
```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
   p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Removal

```
def remove(p,k):
  p ← search(p,k)
  if key(p) ≠ k then return null
  repeat
    remove p
    p ← above(p)
  until above(p) = null
```



Example: remove(topleft node, 14)

Skip lists: Top layer

Keep a pointer to the topleft node.

Choices for the top layer:

- Keep at a fixed level, say $\max\{10, 3[\log(n)]\}$
  - Insertion needs to take this into account

- Variable level
  - Continue insertion until coin comes up tails
  - No modification required
  - Probability that this gives more than O(log n) levels is very low

Skip lists: Analysis

Theorem:

The expected height of a skip list is O(log n).

Proof:

- The probability that an element is present at height i is $1/2^i$.
  - I.e., the probability that the coin comes up heads i times.
- The probability that level i has at least one item is at most $n/2^i$.
- The probability that skip list has height h is probability that level h

has at least one element.

- So, probability that skip list has height larger than c log n is at most

$$\frac{n}{2^{c\log n}} = \frac{n}{n^c} = \frac{1}{n^{c-1}}$$

- So, probability that skip list has height O(log n) is at least

$$1 - \frac{1}{n^{c-1}}$$

Skip lists: Search Analysis

Theorem:

The expected search time of a skip list is O(log n).

Proof:

- Searching consists of horizontal and vertical steps.
- There are h vertical steps, so O(log n) with high probability.
- To have a horizontal step on level i, the next node can't be on level i+1.
- The probability of this is 1/2.
- This means that the expected number of horizontal steps per level is 2.
- So we expect to spend O(1) time per level.
- Expected search time: O(log n) time with high probability.

Insertion and deletion take expected O(log n) time using similar analysis.

Skip lists: Space Analysis

Theorem:

The expected space used by a skip list is O(n).

Proof:

- Space per node: O(1)
- Expected number of nodes at level i is $n/2^i$.
- Thus expected number of nodes is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

Skip lists: Summary

Expected space: O(n)

Expected search/insert/delete time: O(log n)

Works very well in practice and doesn't require any complicated rebalancing operations.