Skip Lists

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2020

PhoneDirectory is now Map

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We implemented a PhoneDirectory in four ways:

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We implemented a PhoneDirectory in four ways:
  - unsorted array,

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We implemented a PhoneDirectory in four ways:
  - unsorted array,
  - sorted (ordered by key) array,

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We implemented a PhoneDirectory in four ways:
  - unsorted array,
  - sorted (ordered by key) array,
  - unsorted linked list,

PhoneDirectory is now Map

- From the BST (Binary Search Tree) lab, we now know that the formal name for a PhoneDirectory is a Map.
- *name* is now *key* and *number* is *value*.
- Let's have a look at the Map interface.
  - **V put(K key, V value)** is like **addOrChangeEntry**.
  - **V get(K key)** is like **lookupEntry**.
  - **V remove(Object Key)** is like **removeEntry**.
- We implemented a PhoneDirectory in four ways:
  - unsorted array,
  - sorted (ordered by key) array,
  - unsorted linked list,
  - sorted linked list.

Running Times

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the “gold coin” idea, actually called *divide and conquer*.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the “gold coin” idea, actually called *divide and conquer*.
- Everything else is $\mathrm{O}(n)$.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the “gold coin” idea, actually called *divide and conquer*.
- Everything else is $\mathrm{O}(n)$.
- BST can do everything in $\mathrm{O}(\log n)$, but only if the input is random.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the "gold coin" idea, actually called *divide and conquer*.
- Everything else is $\mathrm{O}(n)$.
- BST can do everything in $\mathrm{O}(\log n)$, but only if the input is random.
- BST is $\mathrm{O}(n)$ for everything if the input is sorted.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the "gold coin" idea, actually called *divide and conquer*.
- Everything else is $\mathrm{O}(n)$.
- BST can do everything in $\mathrm{O}(\log n)$, but only if the input is random.
- BST is $\mathrm{O}(n)$ for everything if the input is sorted.
- You will learn how to fix this in CSC317 using a version of BST called the *Red-black Tree*.

Running Times

- The sorted array allows us to implement get (used to be lookupEntry) in $\mathrm{O}(\log n)$ time using *binary search*.
- Binary search is an application of the "gold coin" idea, actually called *divide and conquer*.
- Everything else is $\mathrm{O}(n)$.
- BST can do everything in $\mathrm{O}(\log n)$, but only if the input is random.
- BST is $\mathrm{O}(n)$ for everything if the input is sorted.
- You will learn how to fix this in CSC317 using a version of BST called the *Red-black Tree*.
- That's what java.util.TreeMap uses.

This week's application

This week's application

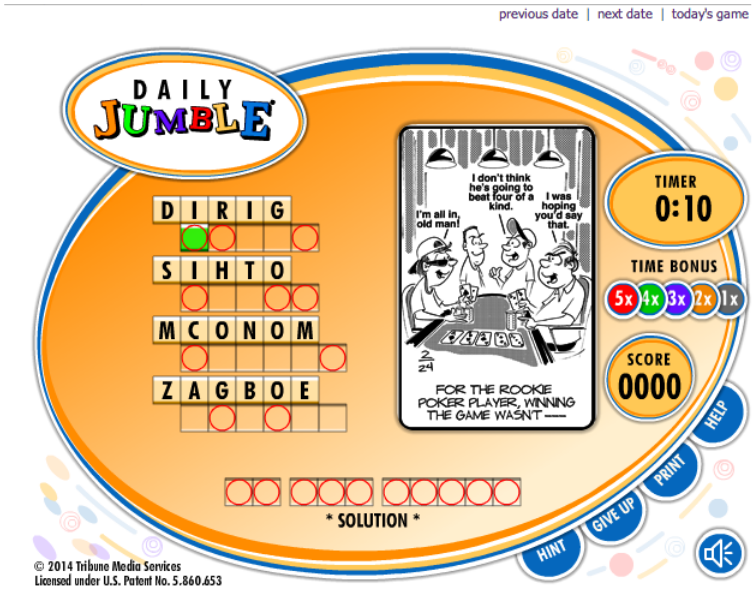
- We need a nice application for our Map.

This week's application

- We need a nice application for our Map.
  - Yet another game!

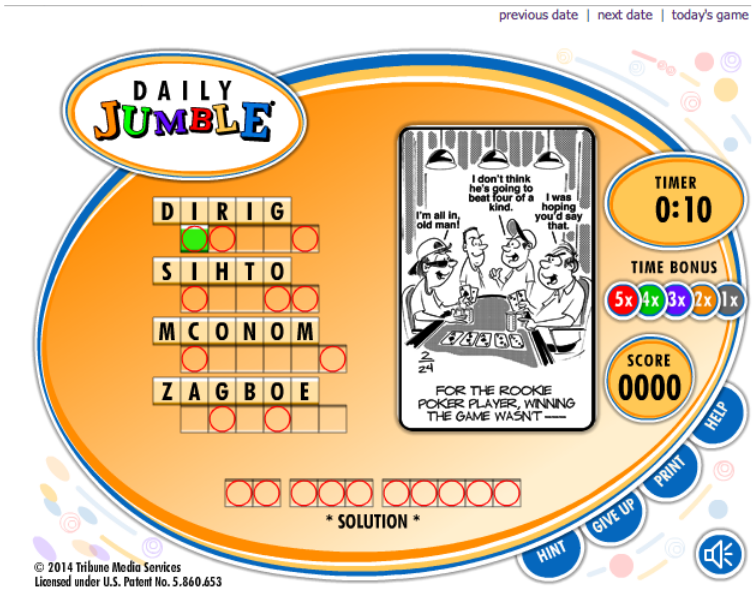
This week's application

- We need a nice application for our Map.
  - Yet another game!
  - Daily Jumble



This week's application

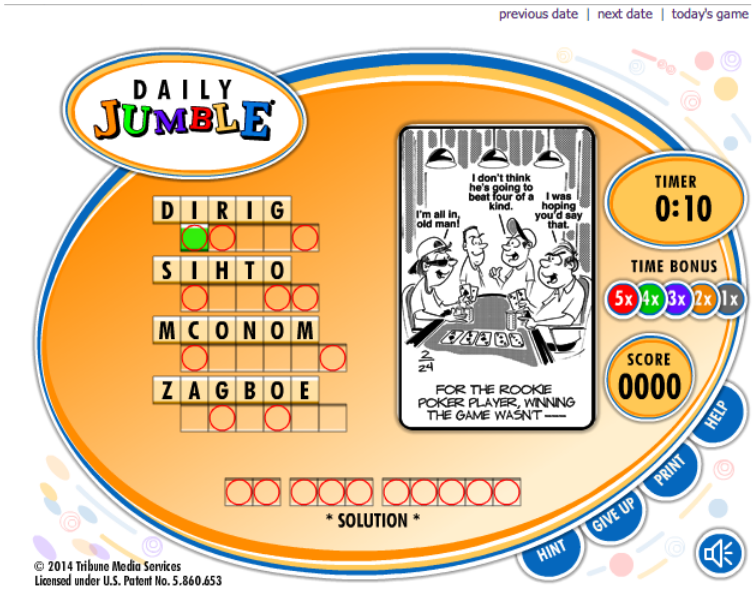
- We need a nice application for our Map.
  - Yet another game!
  - Daily Jumble



- Need to unscramble words.

This week's application

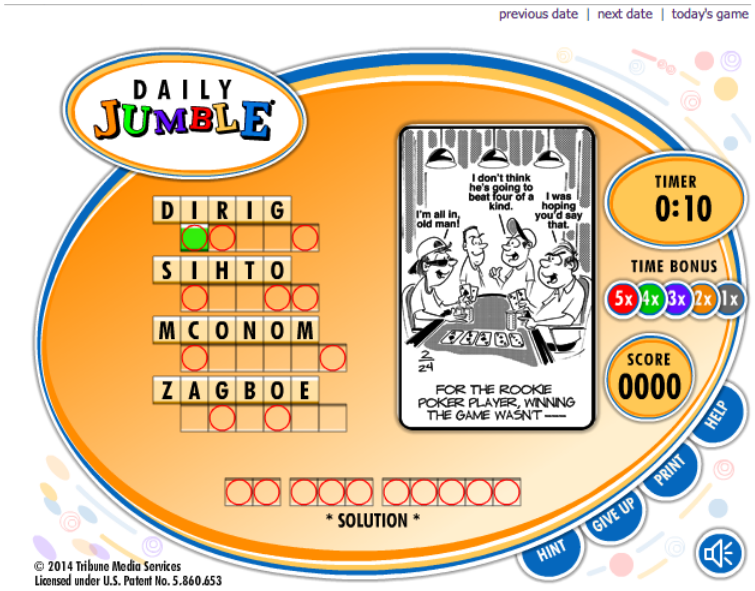
- We need a nice application for our Map.
  - Yet another game!
  - Daily Jumble



- Need to unscramble words.
  - Puzzle has "rtpocmue"?

This week's application

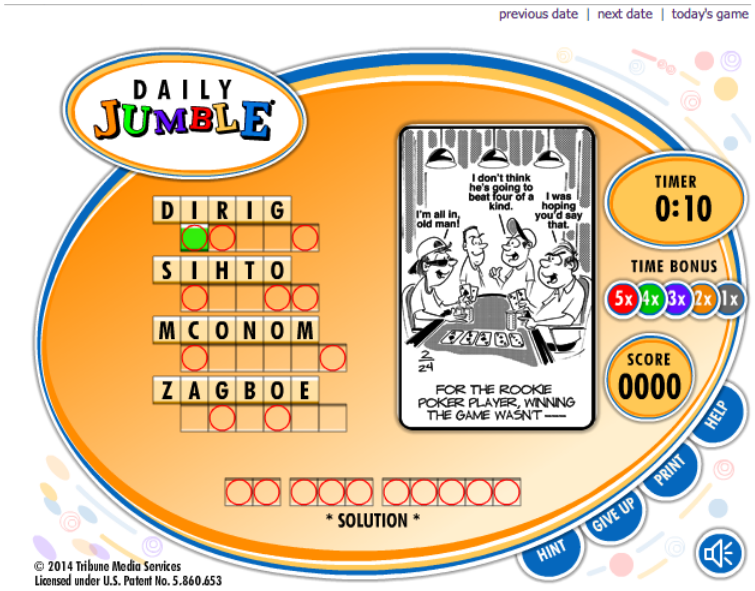
- We need a nice application for our Map.
  - Yet another game!
  - Daily Jumble



- Need to unscramble words.
  - Puzzle has "rtpocmue"?
  - Unscrambled is "computer".

This week's application

- We need a nice application for our Map.
  - Yet another game!
  - Daily Jumble



- Need to unscramble words.
  - Puzzle has "rtpocmue"?
  - Unscrambled is "computer".
  - How can a Map help us to do that?

Slow Way

Slow Way

- We have a dictionary file.

Slow Way

- We have a dictionary file.
  - Read it in.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?
  - Lookup might be $\mathrm{O}(\log n)$ time, good.

Slow Way

- We have a dictionary file.
  - Read it in.
  - Try every possible ordering of “rtpocmue”.
  - Look up each one in the dictionary.
- What is the running time?
  - Lookup might be $O(\log n)$ time, good.
  - But the number of orderings is 8! = 40,320, bad!.

Using a Map

Using a Map

- Let's use a Map.

Using a Map

- Let's use a Map.
  - The value will be "computer".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble “rtpmceuo”:

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble “rtpmceuo”:
  - Alphabetize it to “cemoprtu”.
  - Look it up in the map: “computer”.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".
  - So the value will be "read" because it is last.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".
  - So the value will be "read" because it is last.
  - Solution is to use **List<String>** as the value type.

Using a Map

- Let's use a Map.
  - The value will be "computer".
  - What will be the key?
  - How about the letters in alphabetical order?
  - That is "cemoprtu".
- To get ready:
  - Read each word from the dictionary file,
  - Put it into the Map.
  - The key will be the letters of the word in alphabetical order.
  - The value will be the word.
- To solve a scramble "rtpmceuo":
  - Alphabetize it to "cemoprtu".
  - Look it up in the map: "computer".
- Does anyone see a problem?
  - The words "dare", "dear", and "read"
  - will all be stored under the key "ader".
  - So the value will be "read" because it is last.
  - Solution is to use **List<String>** as the value type.
  - But we won't do that this time.

Implementation using our Map implementations

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $O(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry *i* in the list takes *i* steps.
  - An array can do it in one step (**theArray[i]**)

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $\mathrm{O}(n)$

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $\mathrm{O}(n)$
  - The linked list lets us add quickly once we get there,

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $\mathrm{O}(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.

Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $O(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,

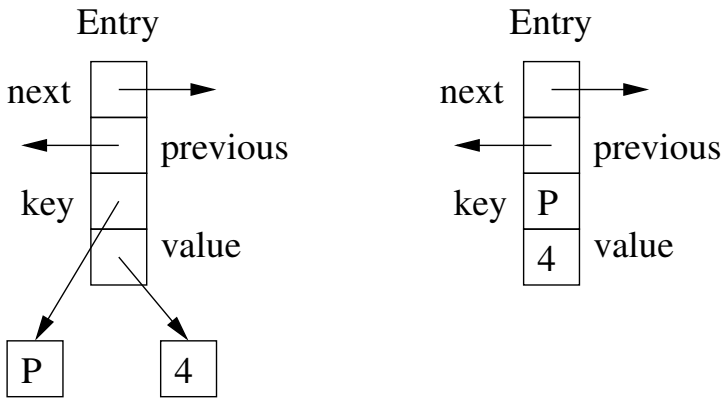
Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $\mathrm{O}(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $\mathrm{O}(n)$
  - If we add n words, that's $\mathrm{O}(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $\mathrm{O}(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,
  - but still much slower than a balanced tree (TreeMap).

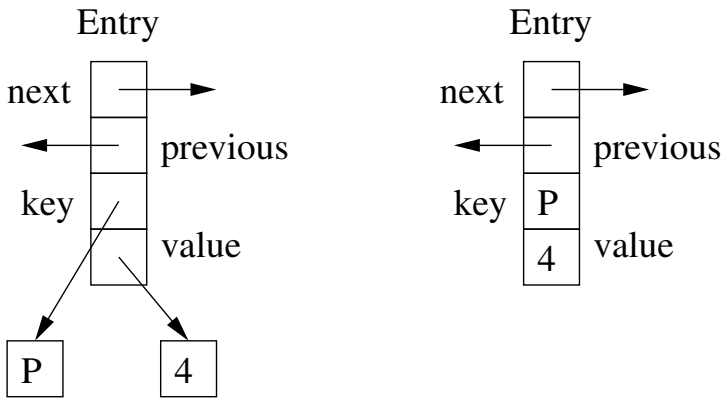
Implementation using our Map implementations

- I will run the Jumble solver for you using the different implementations of Map.
  - The lookup might be as fast as $O(\log n)$ (for SortedPD) but that's not the problem.
  - The problem is adding all the words in the dictionary.
  - We never got add faster than $O(n)$
  - If we add n words, that's $O(n^2)$,
  - which is pretty slow.
- It's even slower for a bigger dictionary, as I will show you.
  - To get to entry i in the list takes i steps.
  - An array can do it in one step (**theArray[i]**)
  - but then adding at the head will cost $O(n)$
  - The linked list lets us add quickly once we get there,
  - but it takes a while to get there.
  - BST is faster,
  - but still much slower than a balanced tree (TreeMap).
- We need a faster way.

Linked Lists

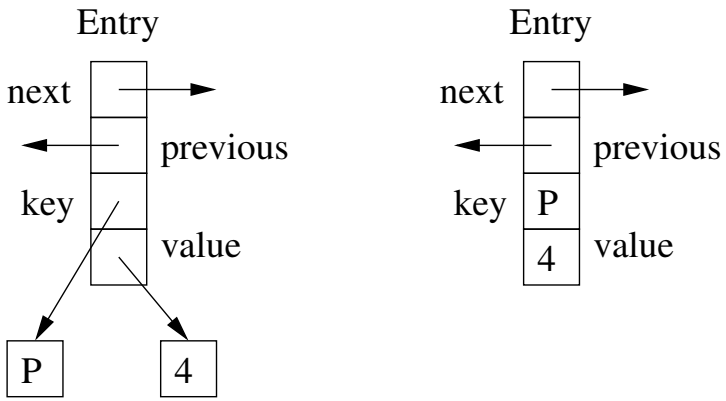


Linked Lists



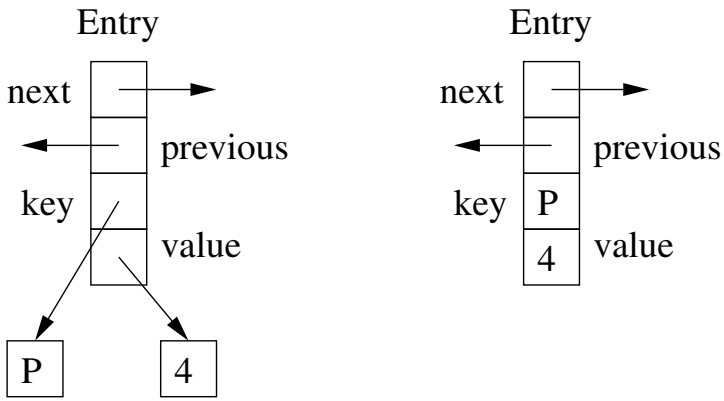
- A linked list uses an Entry class

Linked Lists



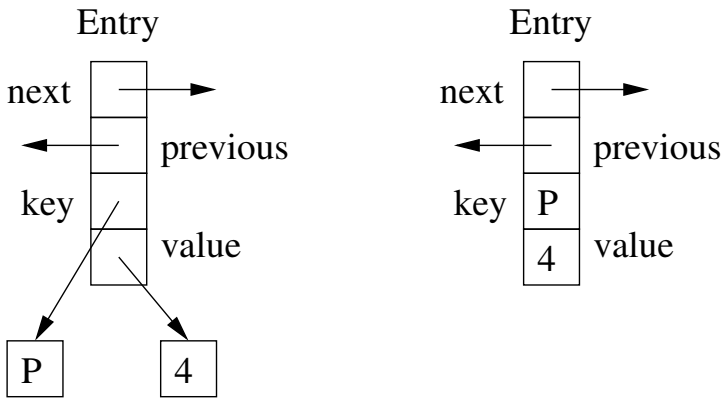
- A linked list uses an Entry class
- with next, previous, key, and value.

Linked Lists



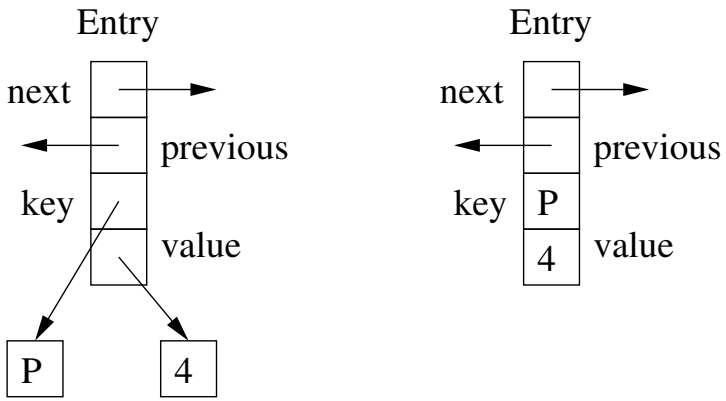
- A linked list uses an Entry class
- with next, previous, key, and value.
- They are all pointers (arrows) to other objects,

Linked Lists



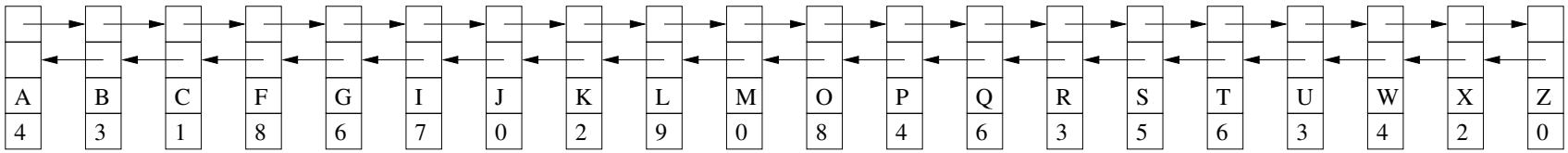
- A linked list uses an Entry class
- with next, previous, key, and value.
- They are all pointers (arrows) to other objects,
- but we usually just write the key and value in the boxes.

Linked Lists

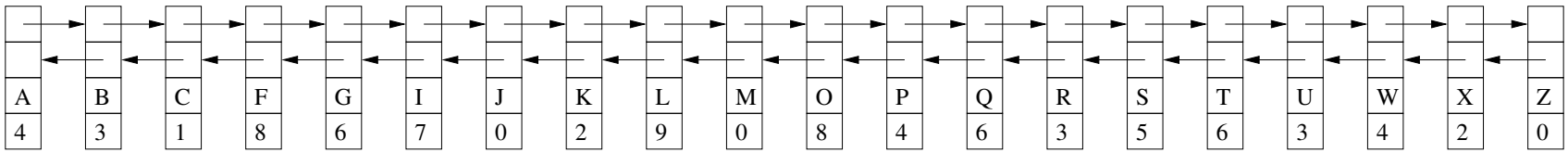


- A linked list uses an Entry class
- with next, previous, key, and value.
- They are all pointers (arrows) to other objects,
- but we usually just write the key and value in the boxes.
- For convenience of drawing the diagrams, I have put next and previous before key and value.

Map using Sorted Linked List

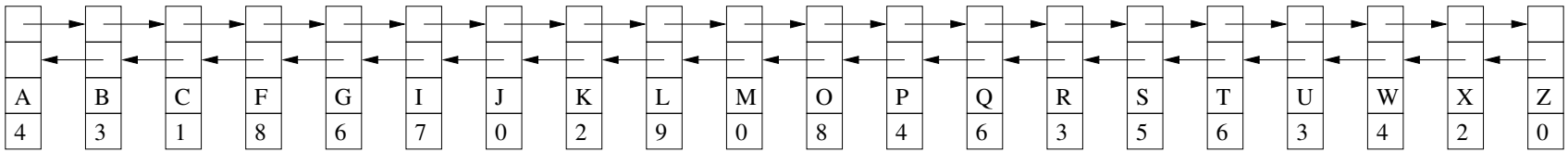


Map using Sorted Linked List



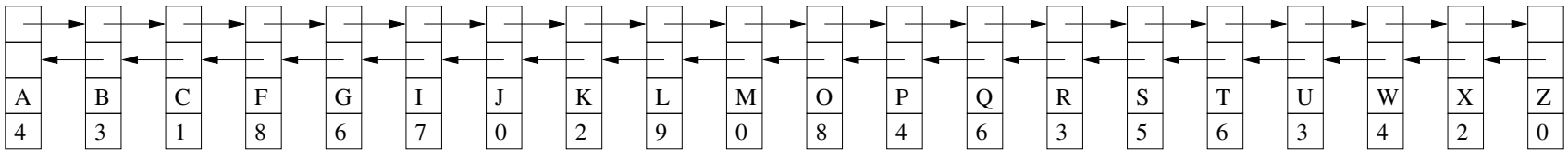
- Linked lists allow us to add or remove an Entry in $O(1)$ time,

Map using Sorted Linked List



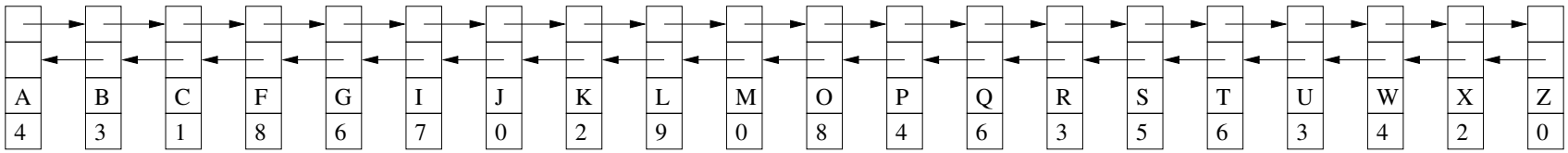
- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time

Map using Sorted Linked List



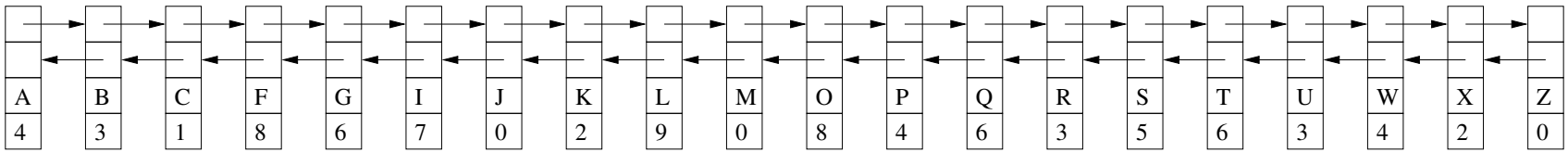
- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time
  - because it may have to move many of the elements forward or back.

Map using Sorted Linked List



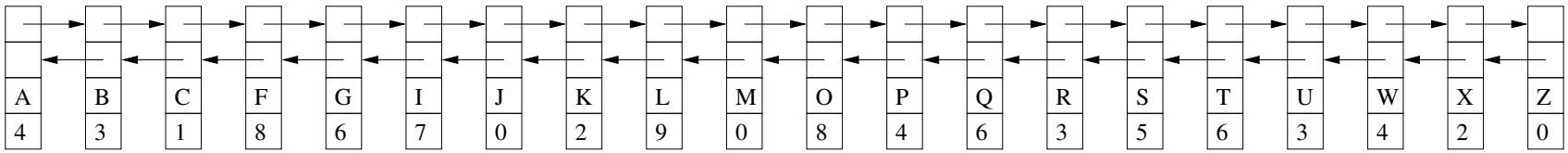
- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time
  - because it may have to move many of the elements forward or back.
- Unfortunately, it takes $O(n)$ time to get to an Entry in a linked list.

Map using Sorted Linked List



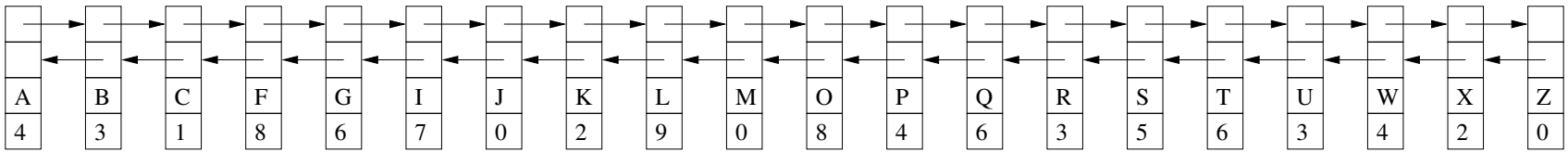
- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time
  - because it may have to move many of the elements forward or back.
- Unfortunately, it takes $O(n)$ time to get to an Entry in a linked list.
  - Binary search would actually be $O(n \log n)$ time on a linked list.

Map using Sorted Linked List



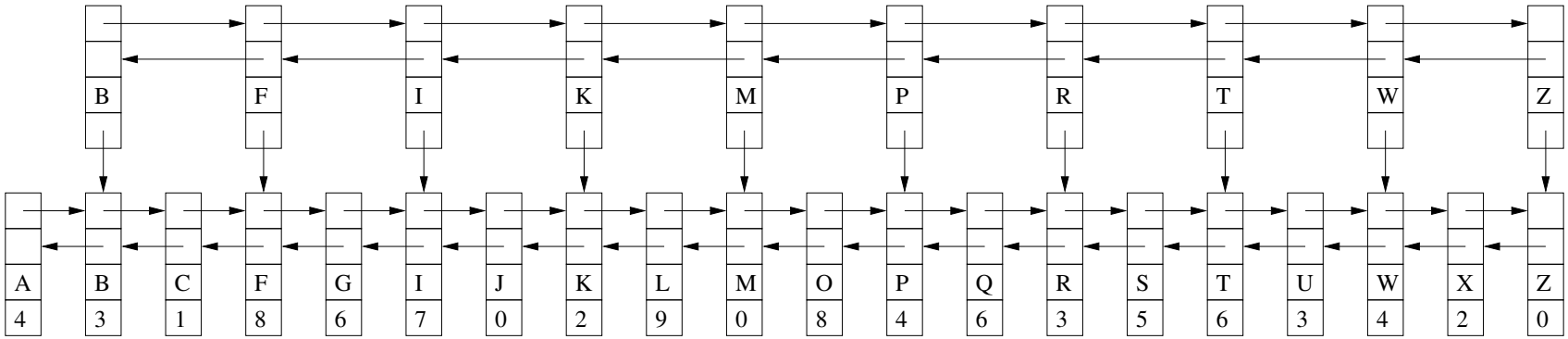
- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time
  - because it may have to move many of the elements forward or back.
- Unfortunately, it takes $O(n)$ time to get to an Entry in a linked list.
  - Binary search would actually be $O(n \log n)$ time on a linked list.
  - It would do $O(\log n)$ gets, each in $O(n)$ time.

Map using Sorted Linked List

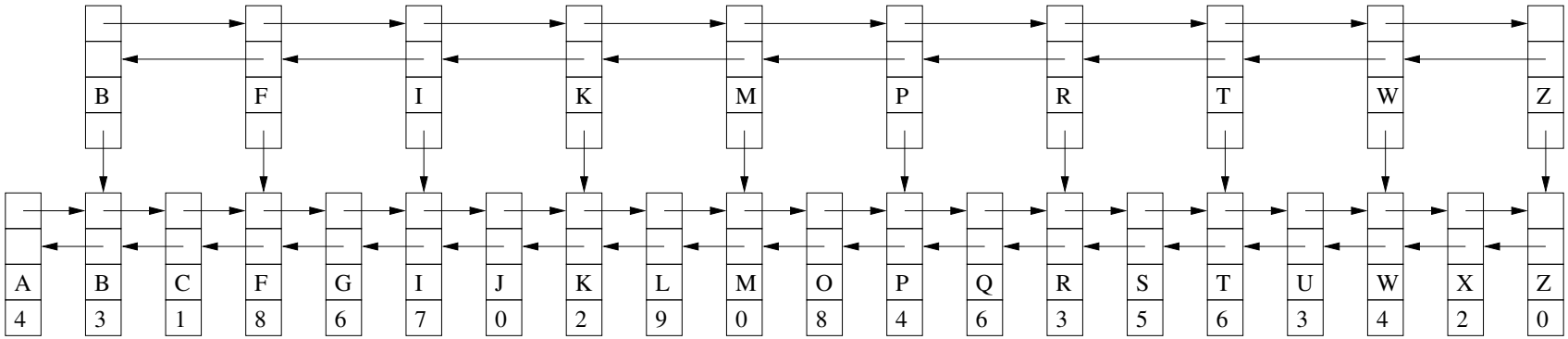


- Linked lists allow us to add or remove an Entry in $O(1)$ time,
  - unlike an array which requires $O(n)$ time
  - because it may have to move many of the elements forward or back.
- Unfortunately, it takes $O(n)$ time to get to an Entry in a linked list.
  - Binary search would actually be $O(n \log n)$ time on a linked list.
  - It would do $O(\log n)$ gets, each in $O(n)$ time.
- So how can we apply the “gold coin” idea?

Skipping Keys

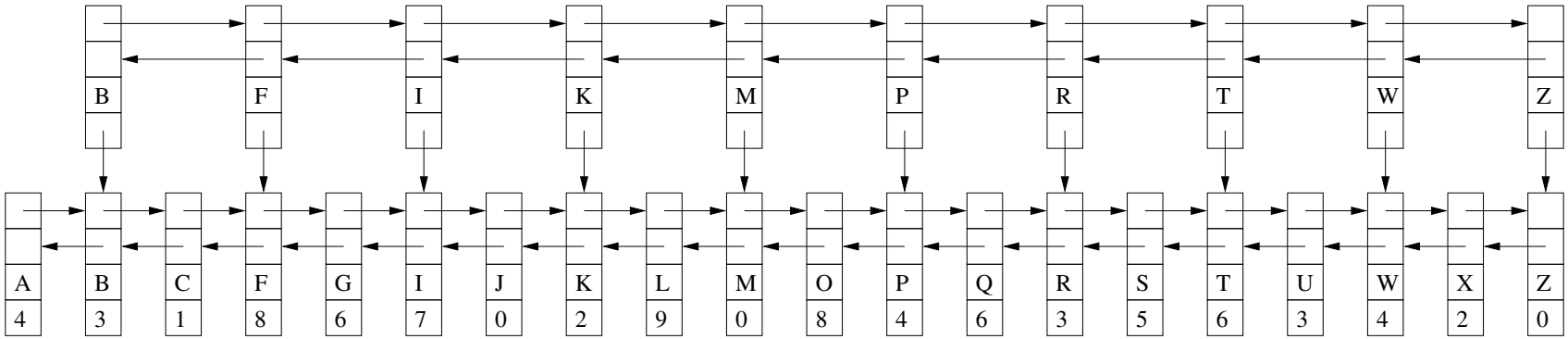


Skipping Keys



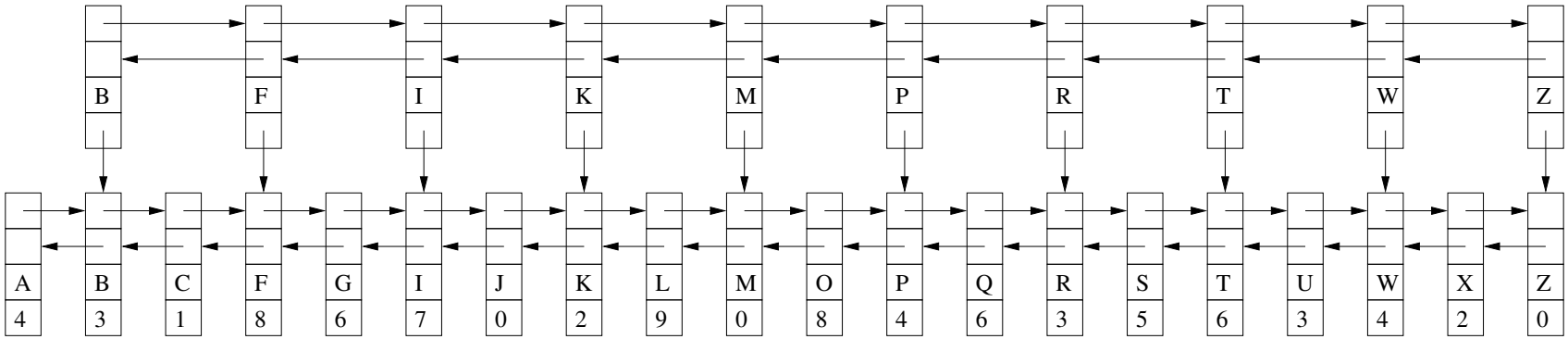
- What if we create a *second* linked list that skips every other key?

Skipping Keys



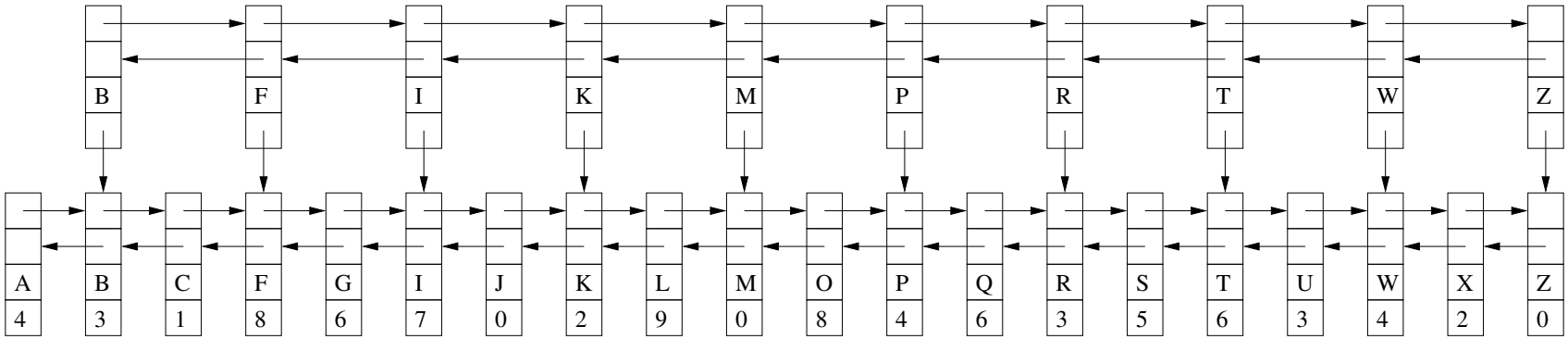
- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.

Skipping Keys



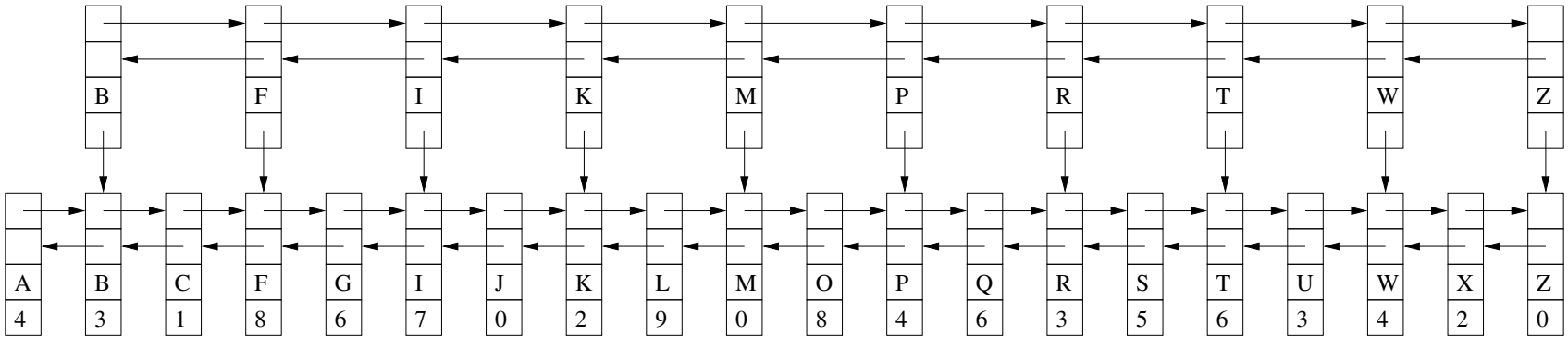
- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.
- We could get as close as possible to a key in the second list,

Skipping Keys



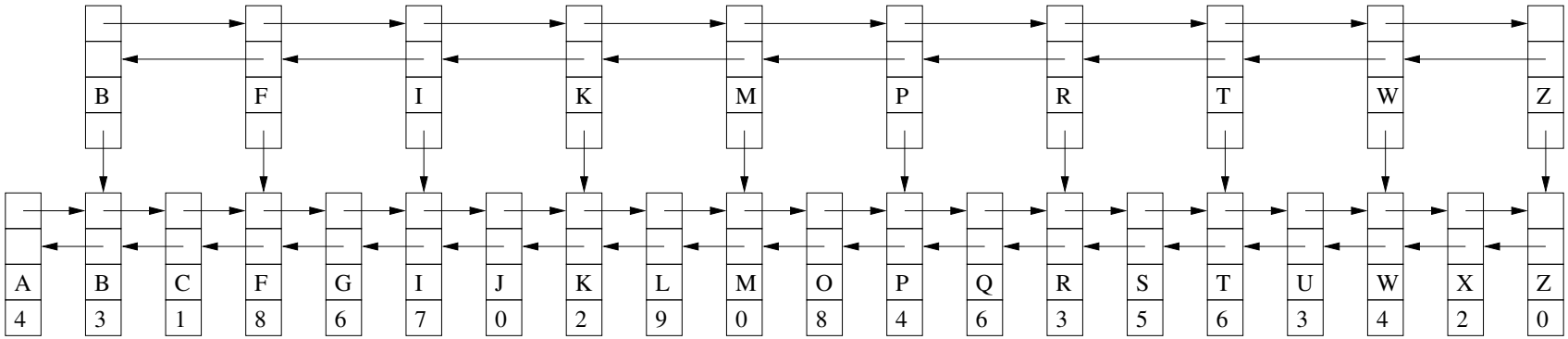
- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.
- We could get as close as possible to a key in the second list,
- and then go at most *one* step in the first list to get to the key.

Skipping Keys



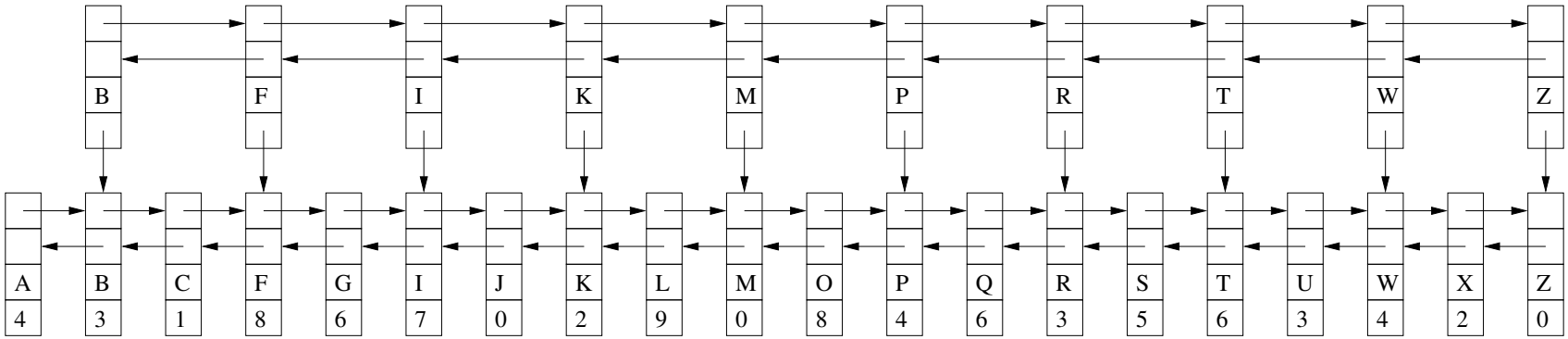
- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.
- We could get as close as possible to a key in the second list,
- and then go at most *one* step in the first list to get to the key.
- That reduces the number of steps from n to $n/2 + 1$.

Skipping Keys



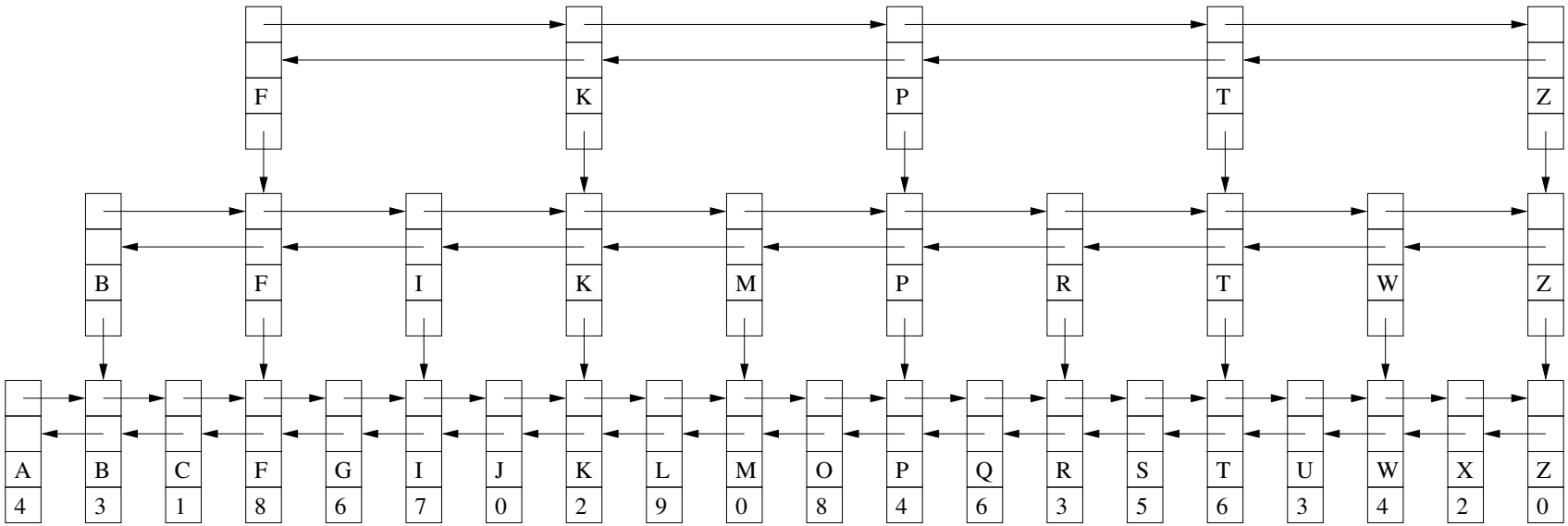
- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.
- We could get as close as possible to a key in the second list,
- and then go at most *one* step in the first list to get to the key.
- That reduces the number of steps from n to $n/2 + 1$.
- But that is still $\mathrm{O}(n)$.

Skipping Keys

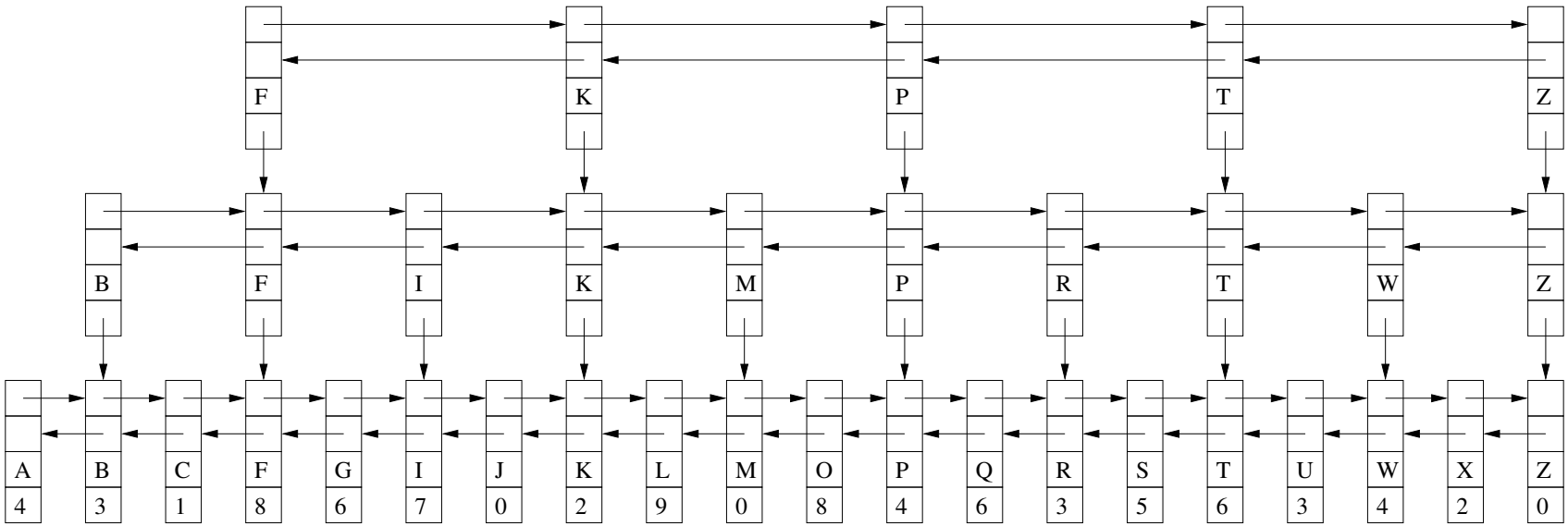


- What if we create a *second* linked list that skips every other key?
- The value would be a pointer to the Entry with that key in the first list.
- We could get as close as possible to a key in the second list,
- and then go at most *one* step in the first list to get to the key.
- That reduces the number of steps from n to $n/2 + 1$.
- But that is still $\mathrm{O}(n)$.
- Too bad. What should we do?

Do it again!

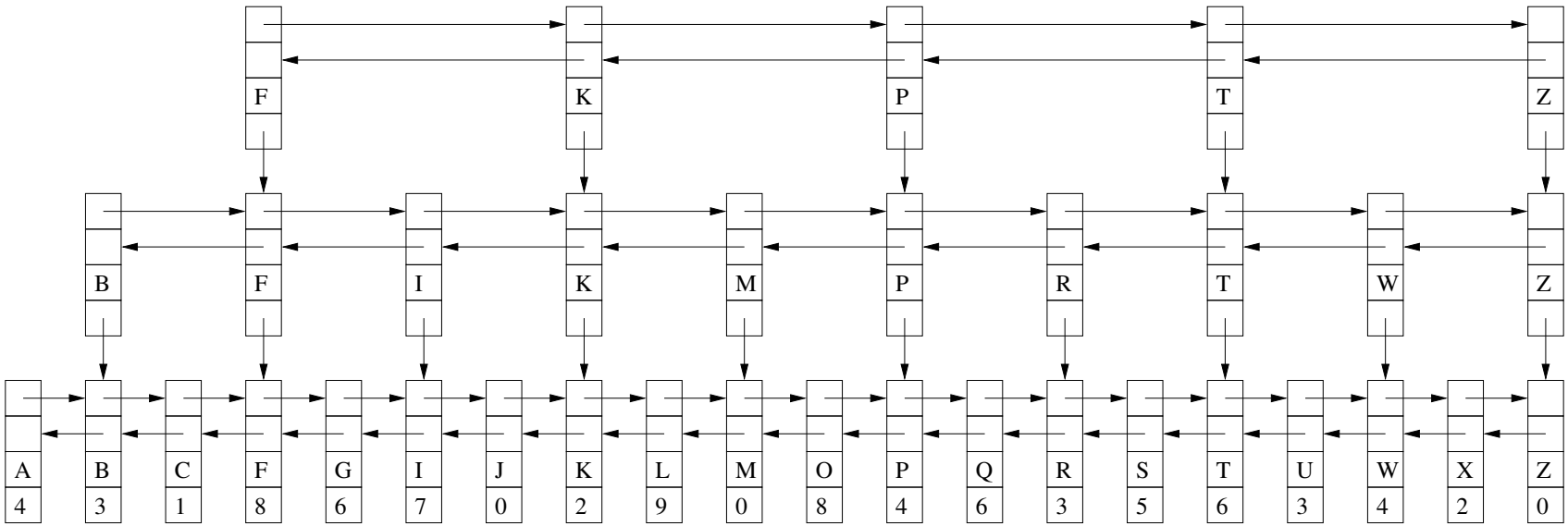

F K P T Z
B F I K M P R T W Z
A B C F G I J K L M O P Q R S T U W X Z
4 3 1 8 6 7 0 2 9 0 8 4 6 3 5 6 3 4 2 0

Do it again!



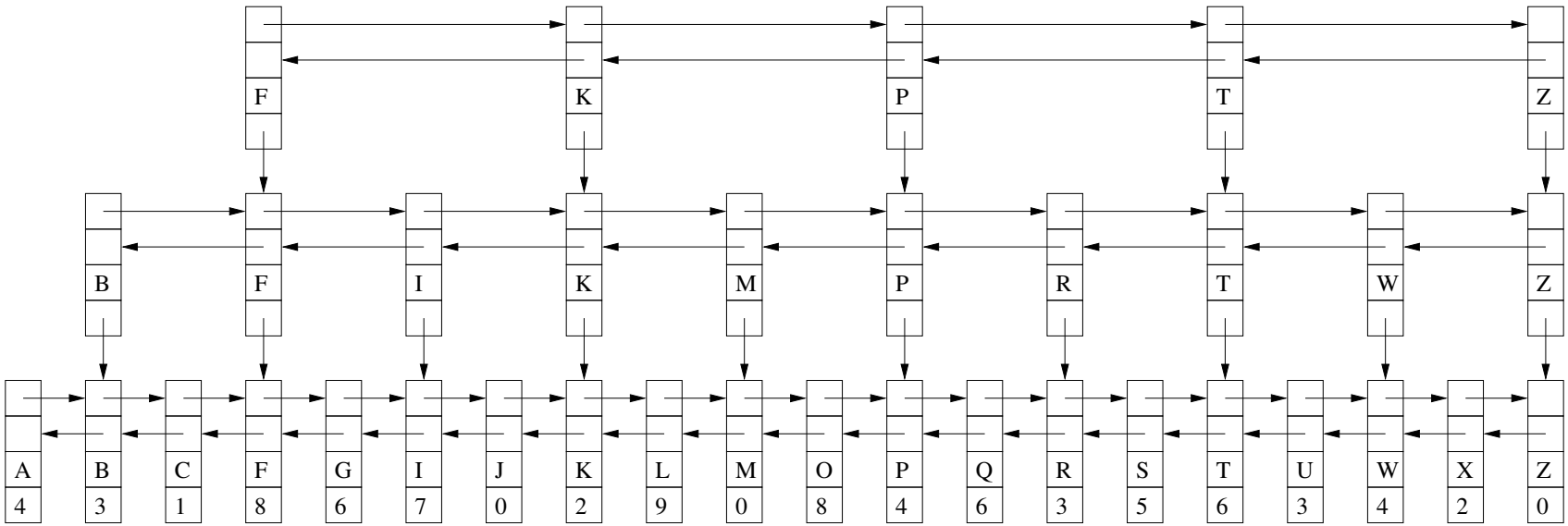
- OK, add a *third* list that skips every other key in the second list.

Do it again!



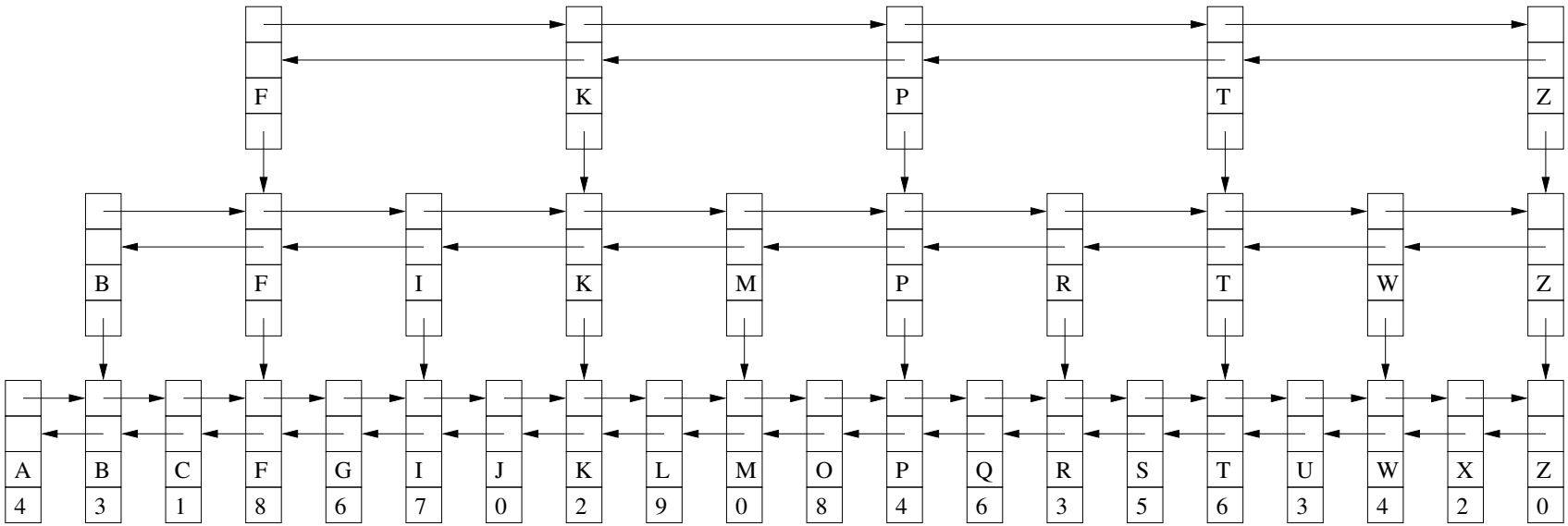
- OK, add a *third* list that skips every other key in the second list.
- So we will only have to go at most *one* step in the second list.

Do it again!



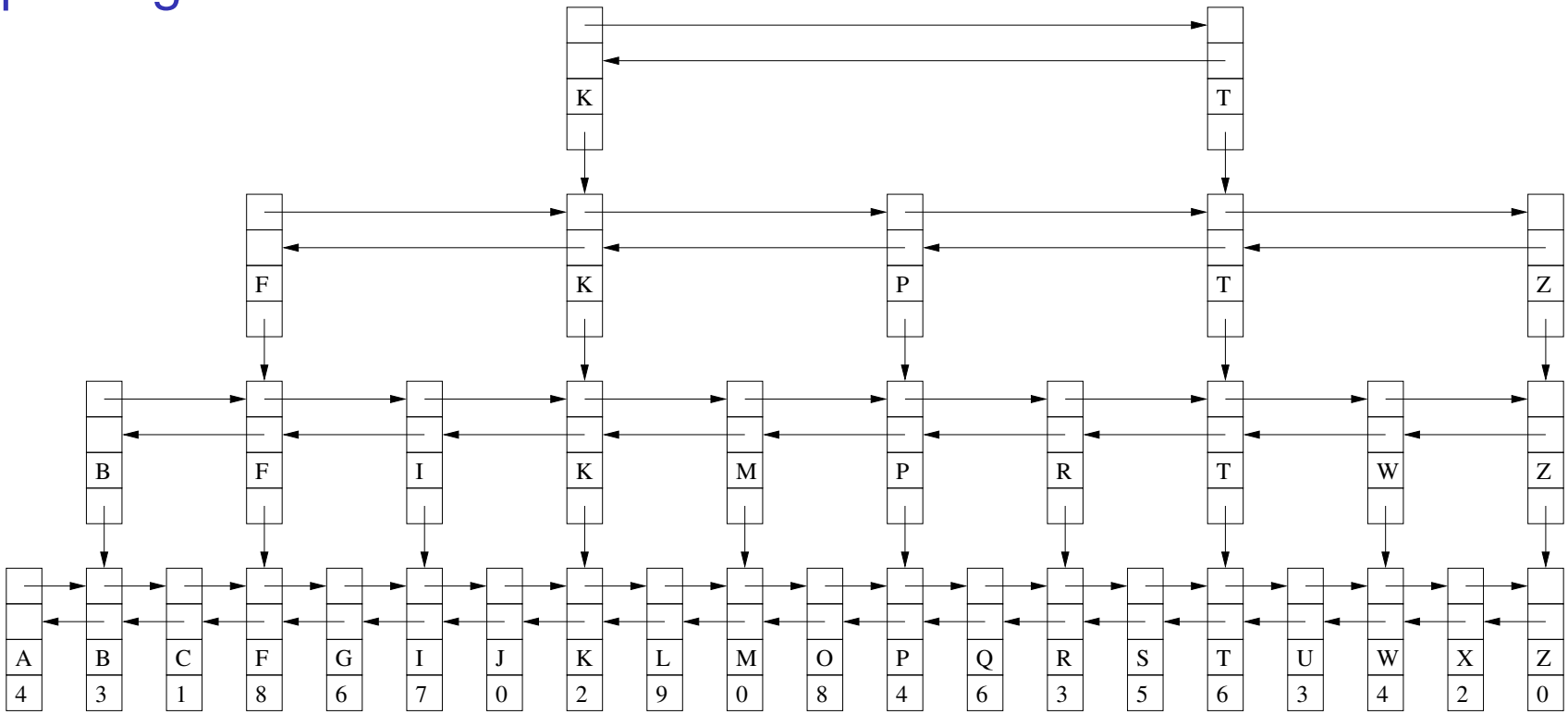
- OK, add a *third* list that skips every other key in the second list.
- So we will only have to go at most *one* step in the second list.
- And as before at most one step in the first list.

Do it again!

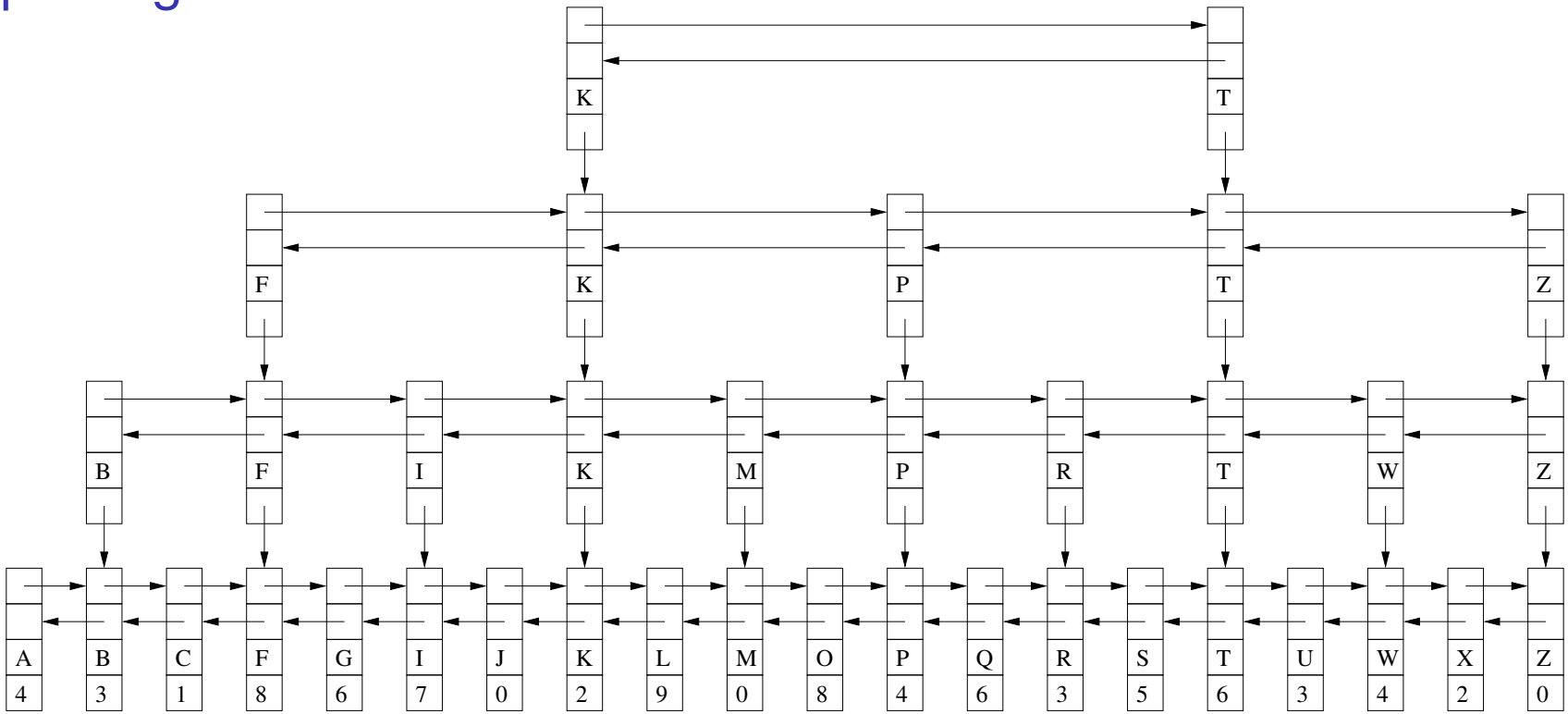


- OK, add a *third* list that skips every other key in the second list.
- So we will only have to go at most *one* step in the second list.
- And as before at most one step in the first list.
- $n/4 + 2$ is still $\mathrm{O}(n)$.

Keep doing it until...

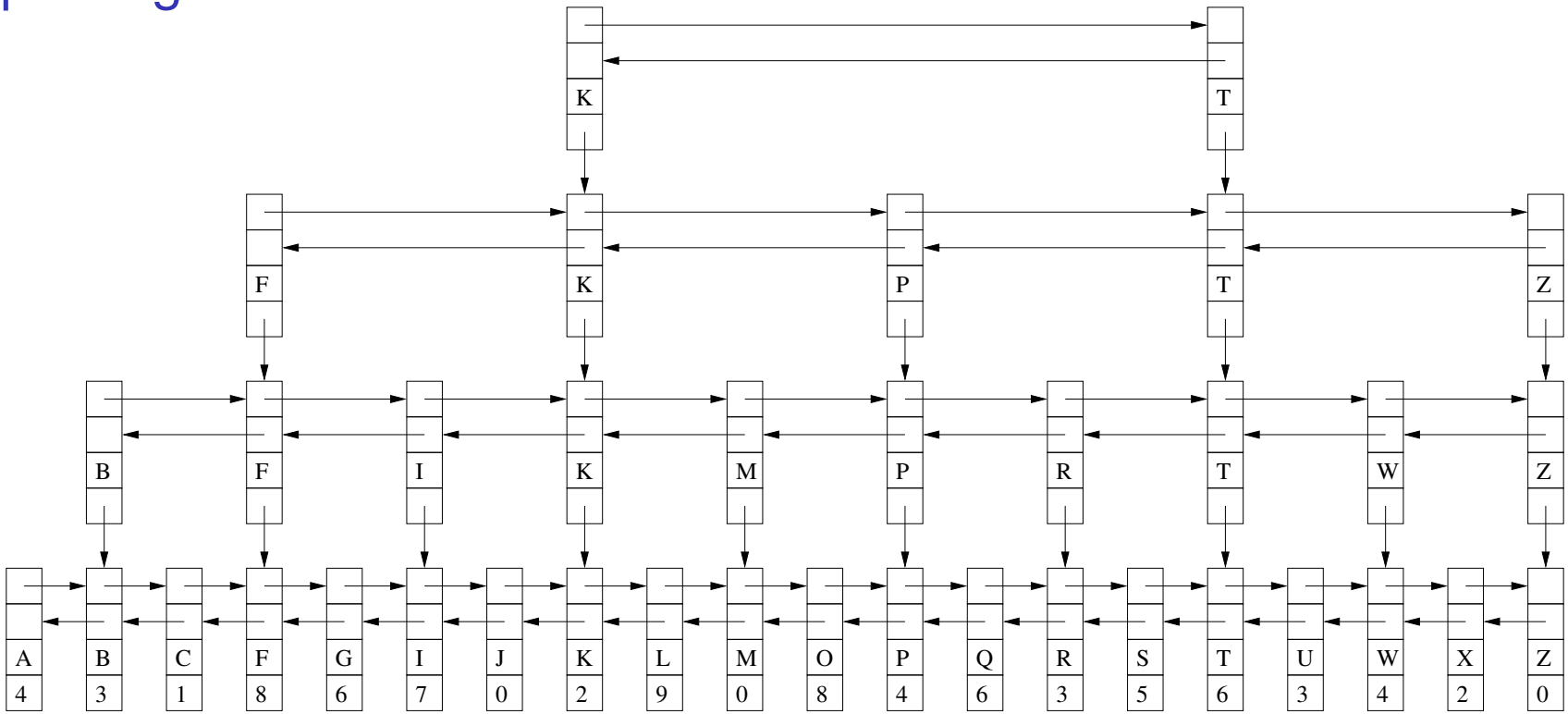


Keep doing it until...



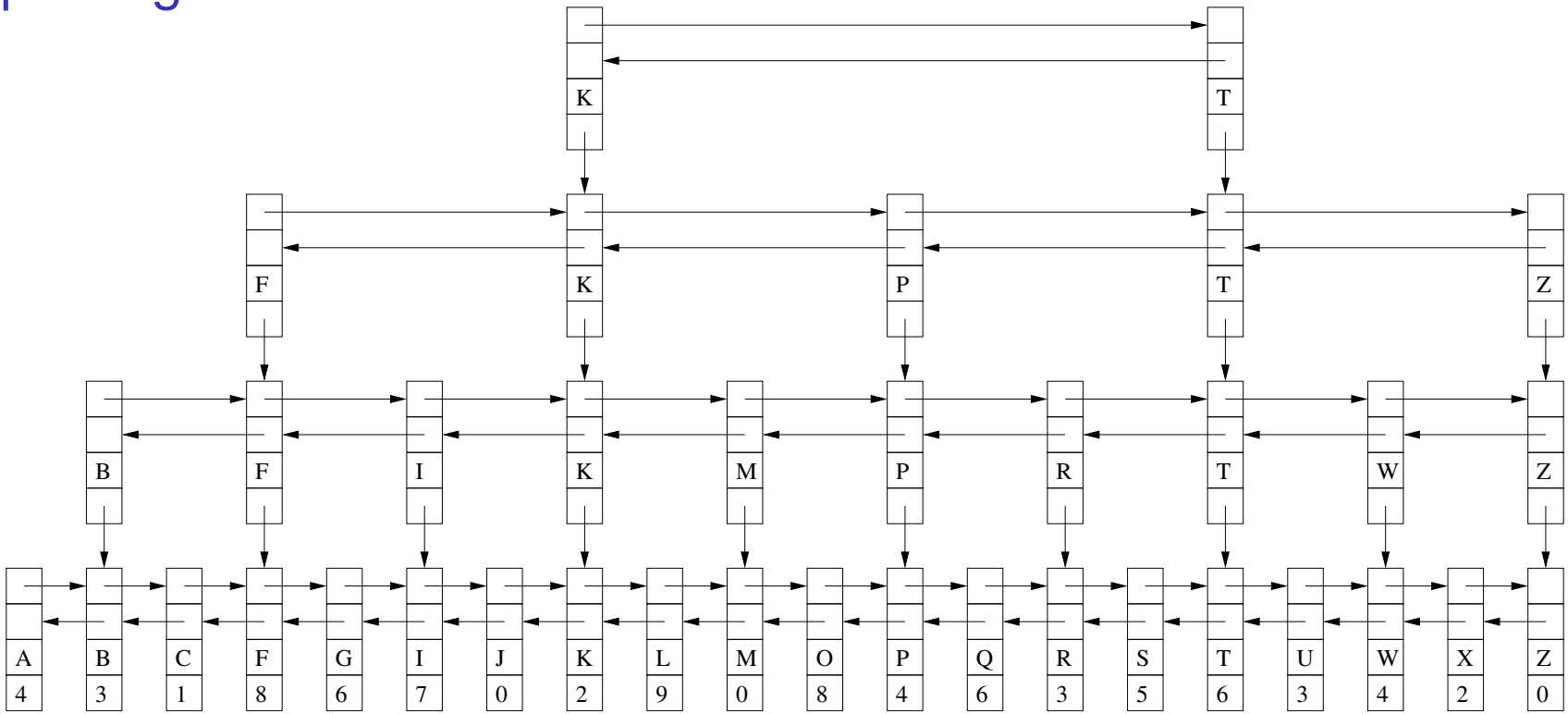
- But what if we keep doing this until there is only a *constant* number in the topmost list?

Keep doing it until...



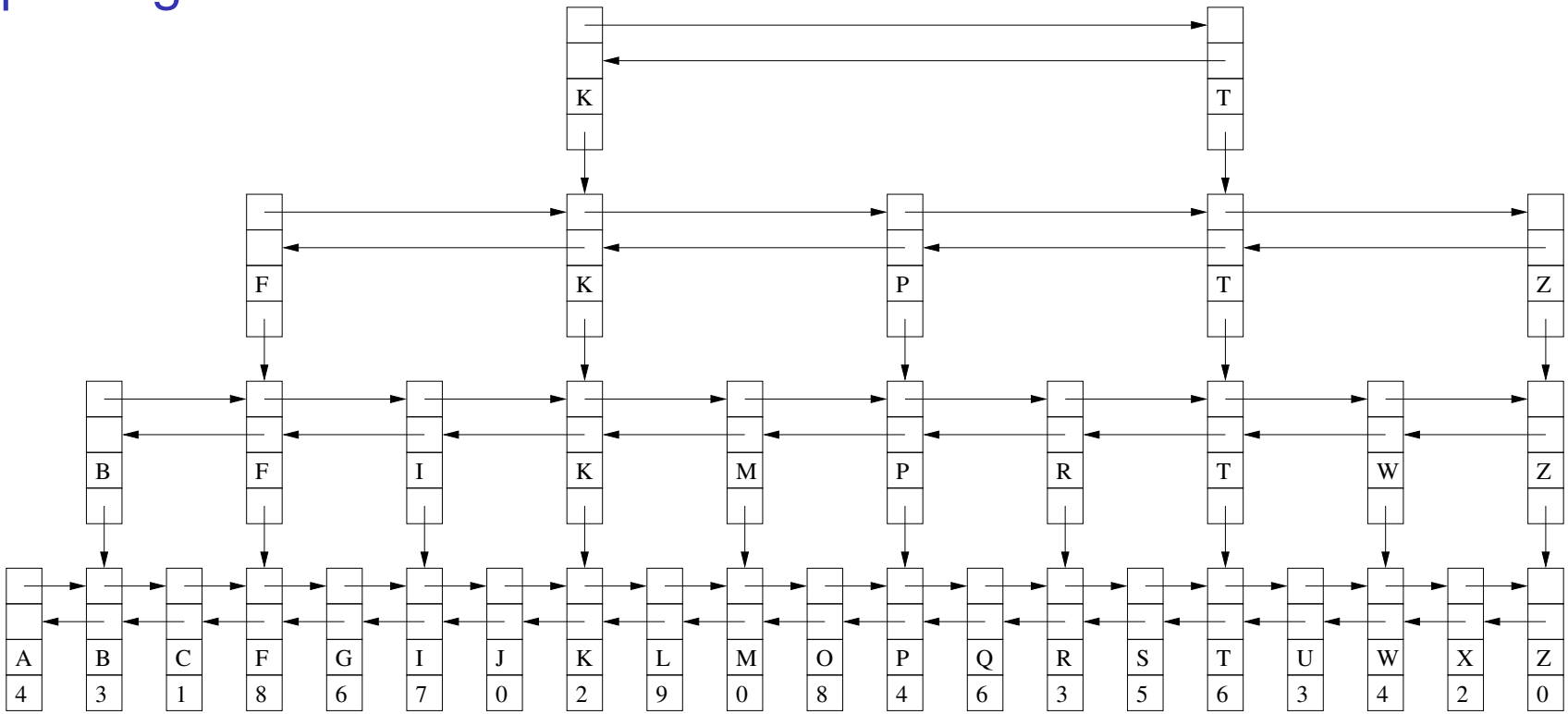
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.

Keep doing it until...



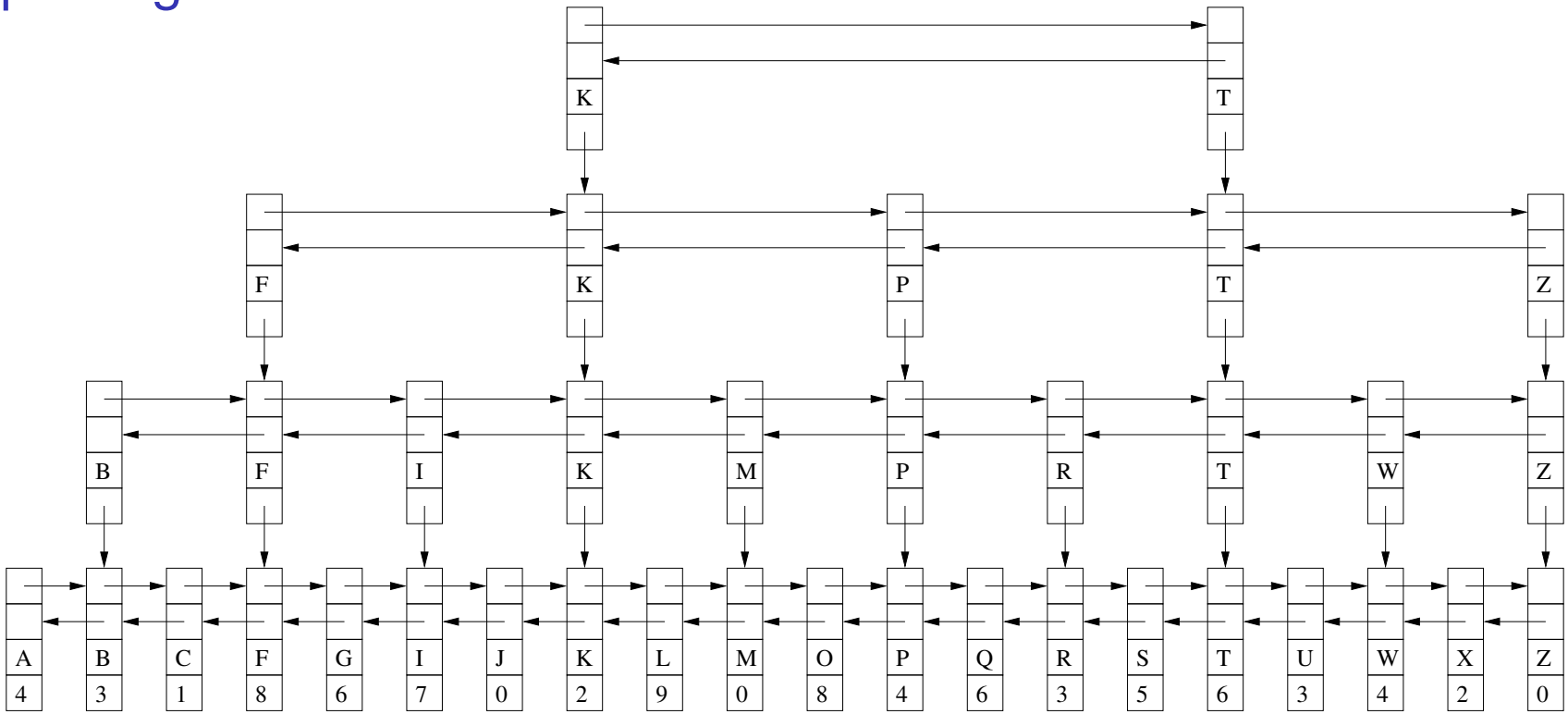
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?

Keep doing it until...



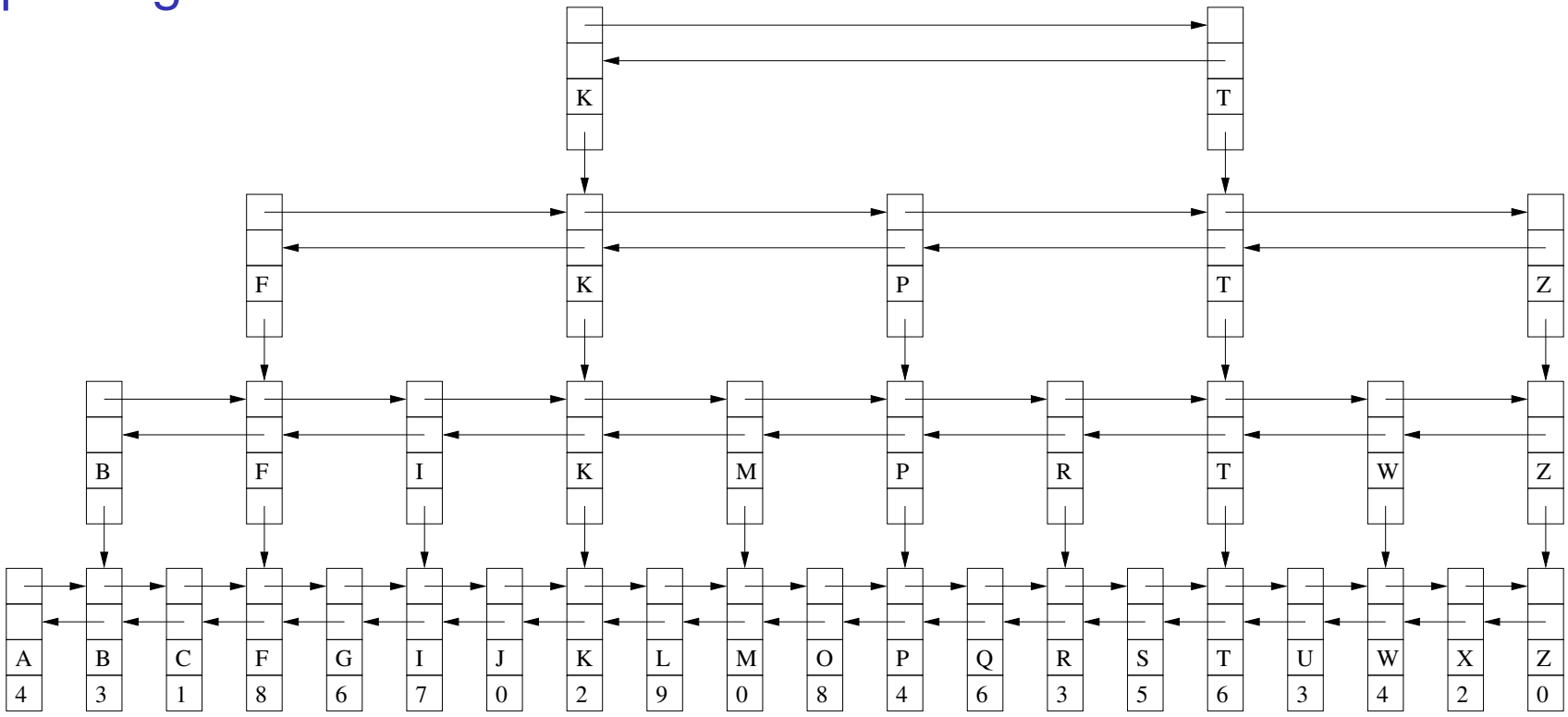
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?
- Hint: each list has half as many elements as the one below it.

Keep doing it until...



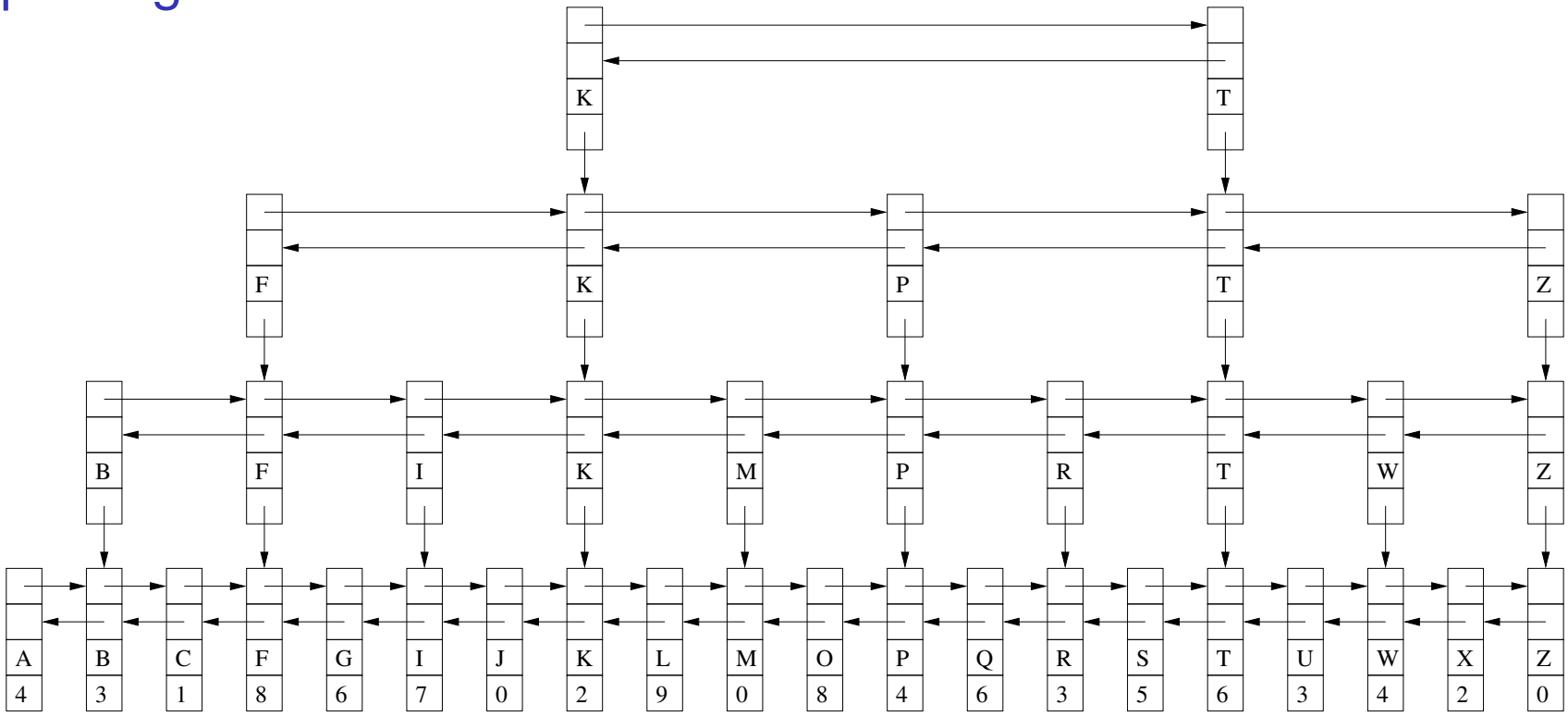
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?
- Hint: each list has half as many elements as the one below it.
- I hope you all *immediately* thought $\log_2 n$.

Keep doing it until...



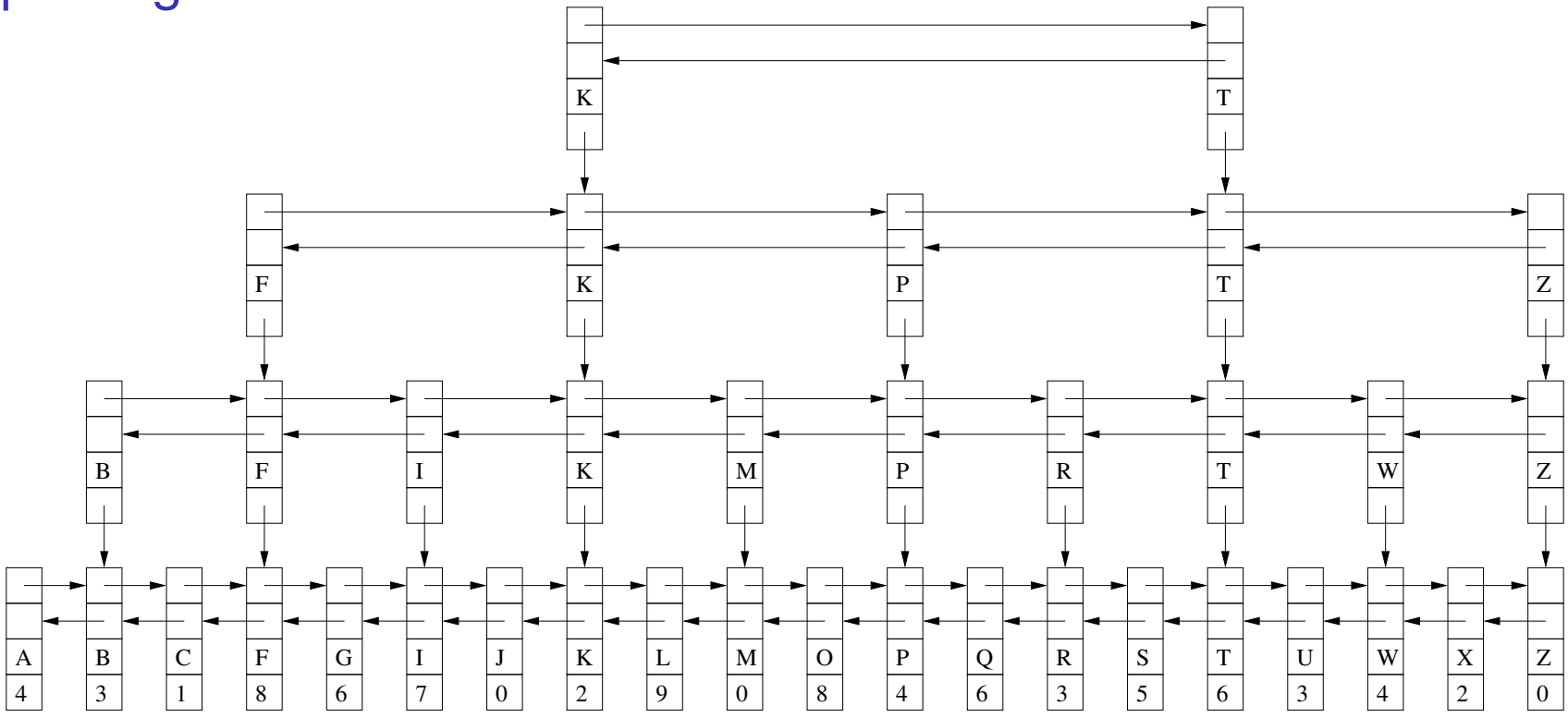
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?
- Hint: each list has half as many elements as the one below it.
- I hope you all *immediately* thought $\log_2 n$.
- So $\log_2 n$ lists and we go at most one step in each one?

Keep doing it until...



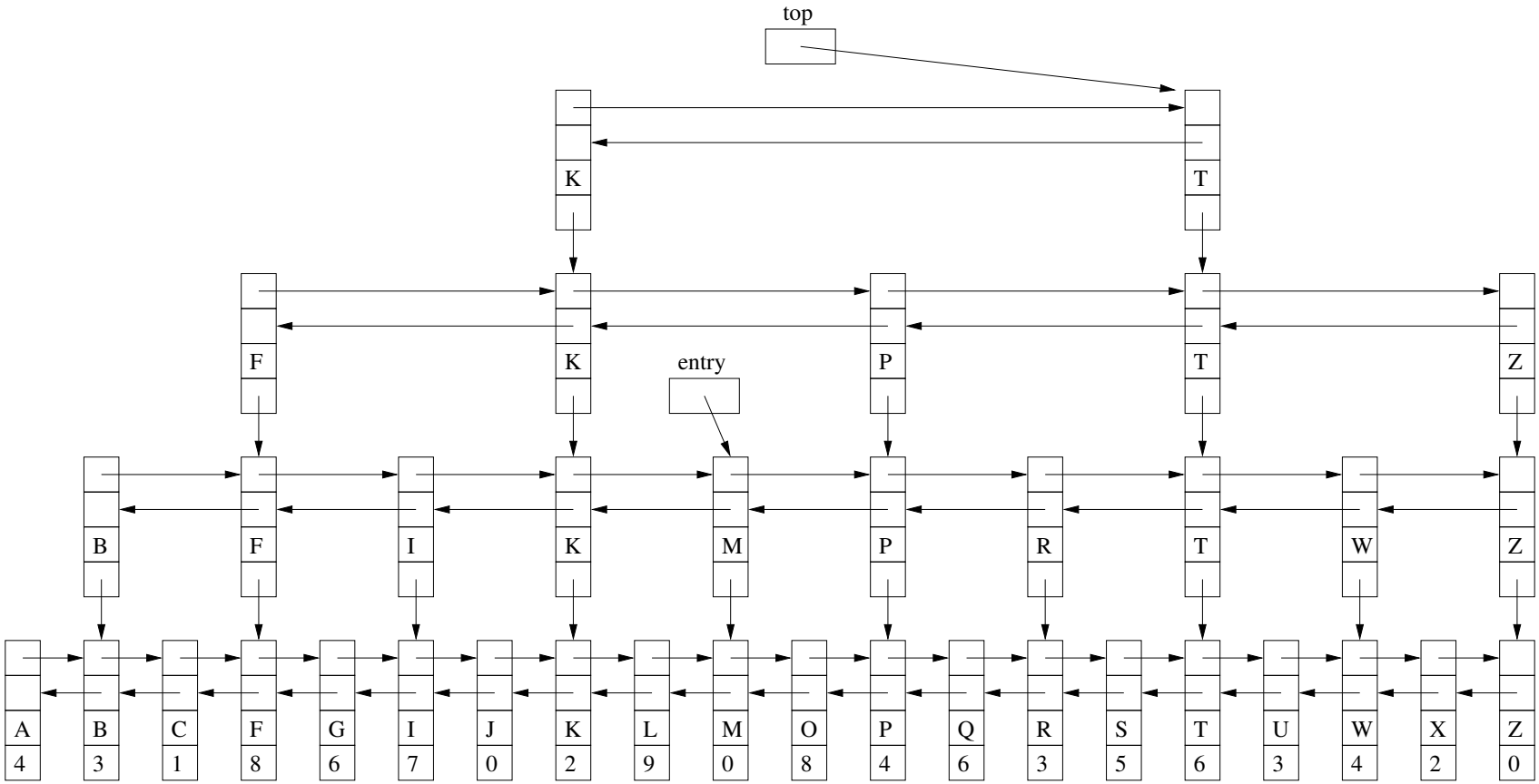
- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?
- Hint: each list has half as many elements as the one below it.
- I hope you all *immediately* thought $\log_2 n$.
- So $\log_2 n$ lists and we go at most one step in each one?
- So getting the (original) Entry for a key takes $O(\log n)$ time.

Keep doing it until...

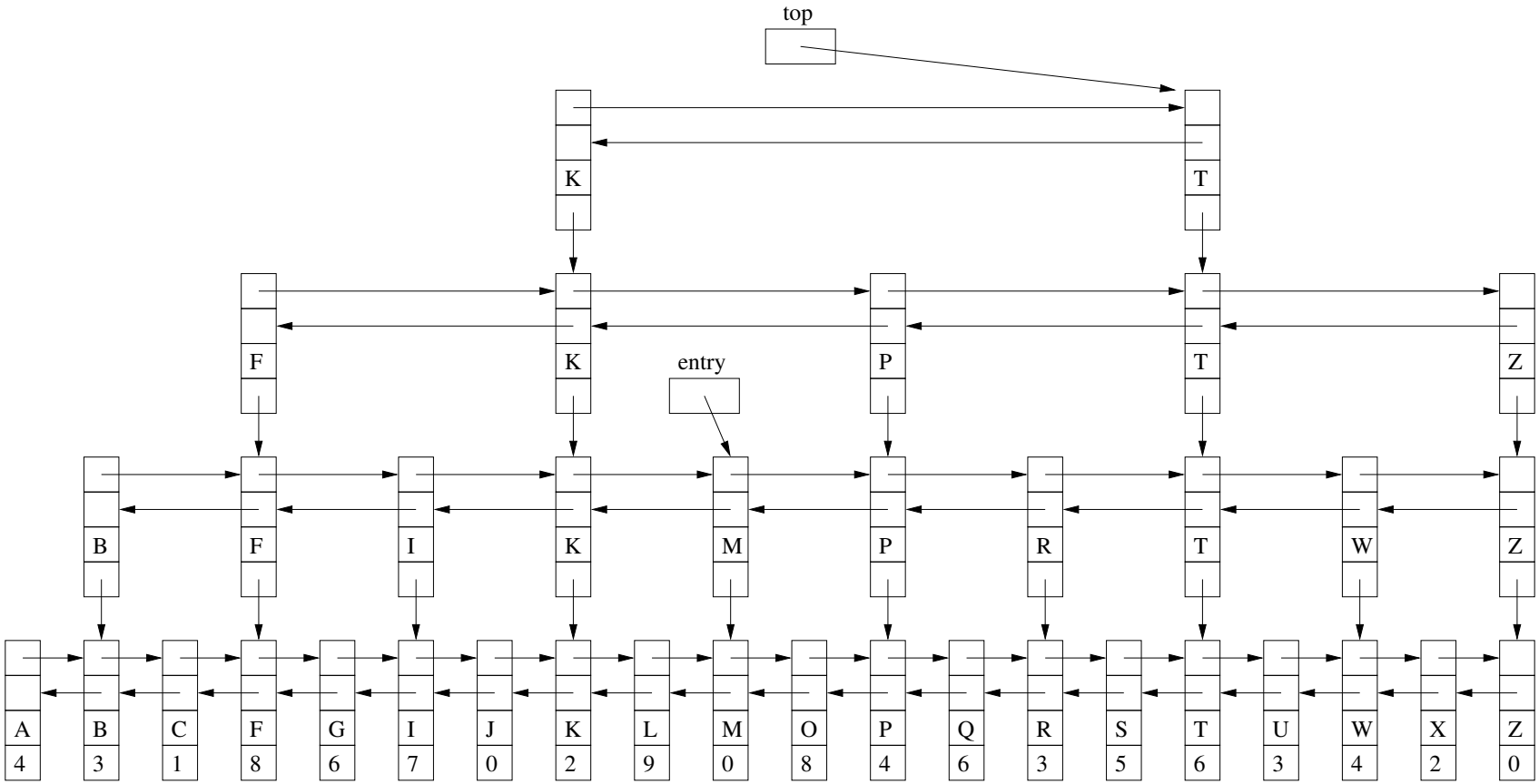


- But what if we keep doing this until there is only a *constant* number in the topmost list?
- So we will go at most one step in *every* list.
- How many lists will there be?
- Hint: each list has half as many elements as the one below it.
- I hope you all *immediately* thought $\log_2 n$.
- So $\log_2 n$ lists and we go at most one step in each one?
- So getting the (original) Entry for a key takes $O(\log n)$ time.
- This is a SKIP LIST.

Skip List

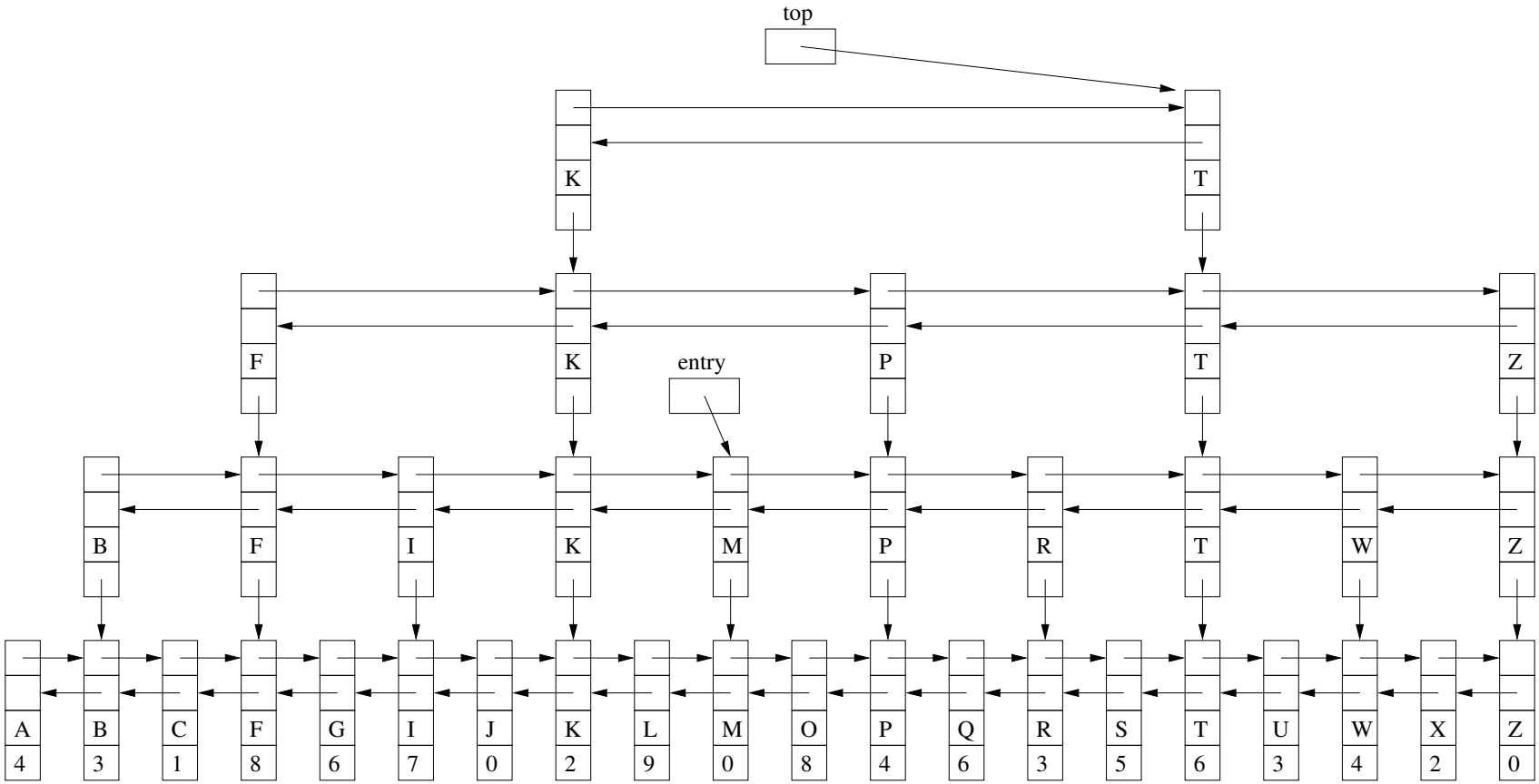

top
K
T
F
K
entry
P
T
Z
B
F
I
K
M
P
R
T
W
Z
A 4
B 3
C 1
F 8
G 6
I 7
J 0
K 2
L 9
M 0
O 8
P 4
Q 6
R 3
S 5
T 6
U 3
W 4
X 2
Z 0

Skip List



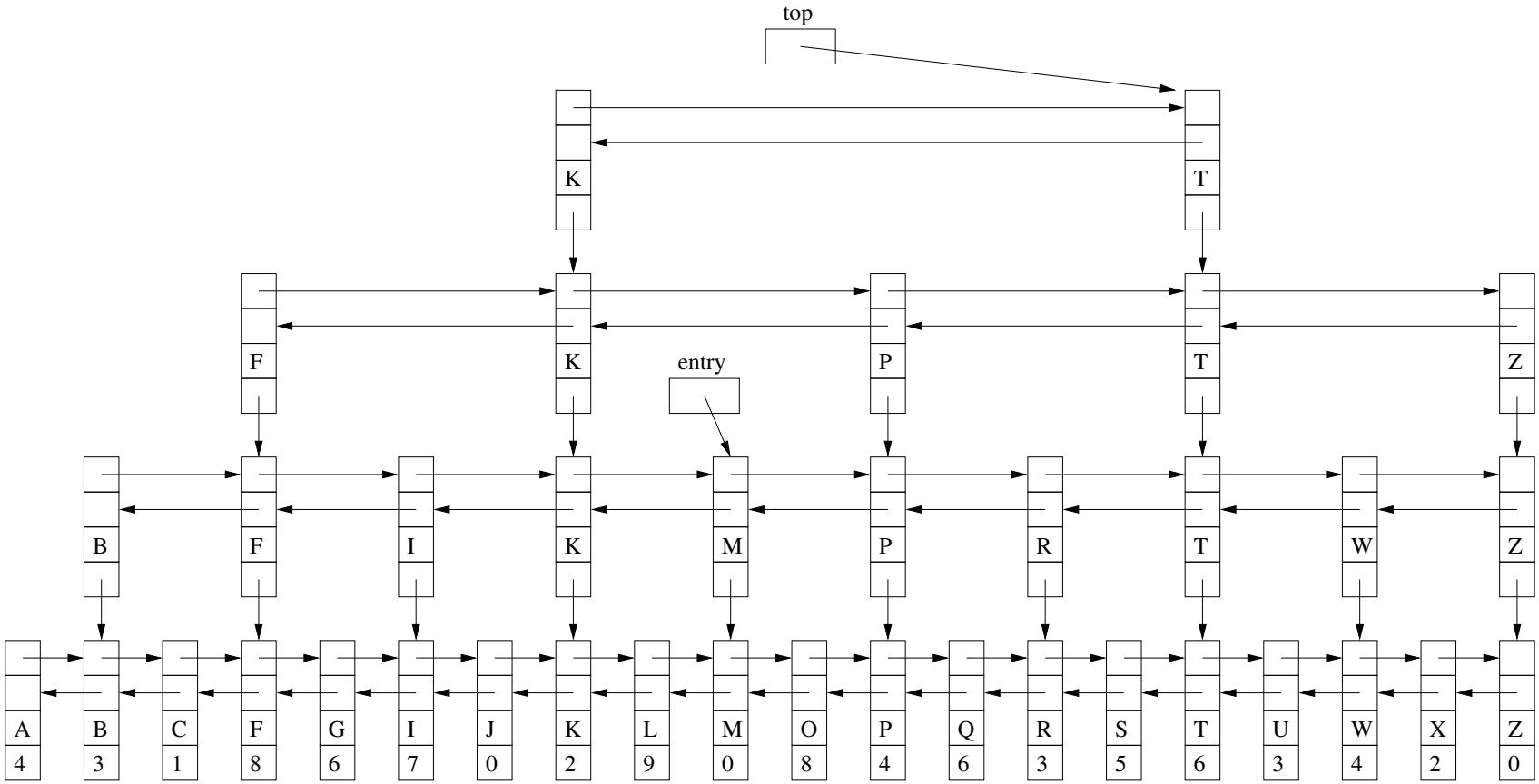
- Here is a skip list. The “top” Entry can be any entry in the top list.

Skip List



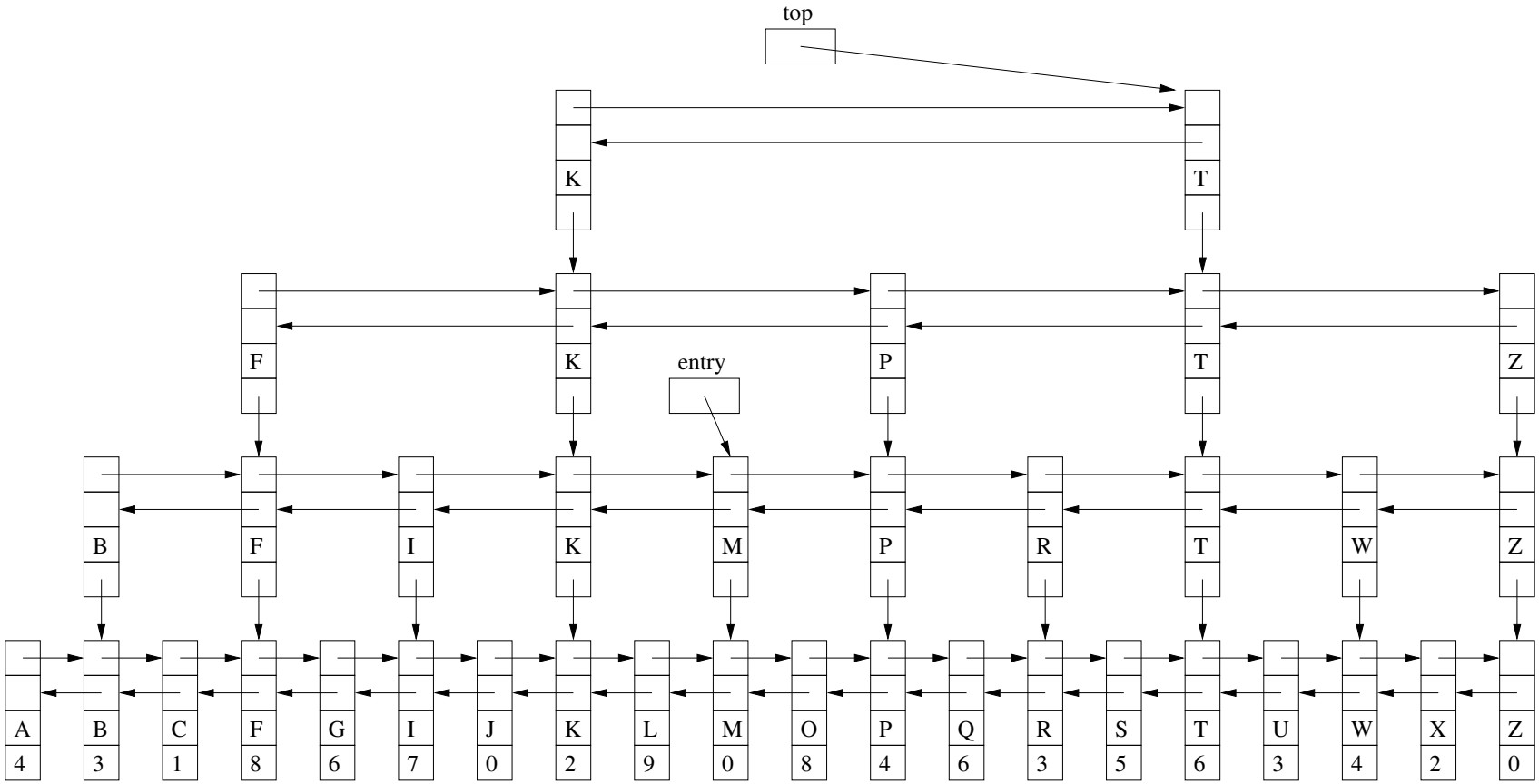
- Here is a skip list. The “top” Entry can be any entry in the top list.
- As we move around the skip list, we have a variable entry that moves left or right on a level or goes down to the next lower list.

Skip List



- Here is a skip list. The “top” Entry can be any entry in the top list.
- As we move around the skip list, we have a variable entry that moves left or right on a level or goes down to the next lower list.
- By the way, how much extra *space* does a skip list use?

Skip List



- Here is a skip list. The “top” Entry can be any entry in the top list.
- As we move around the skip list, we have a variable entry that moves left or right on a level or goes down to the next lower list.
- By the way, how much extra *space* does a skip list use?
- $n/2 + n/4 + n/8 + \cdots = ???$

Switching to ASCII

							K							@T					
			F				K				P			T				Z	
	B		F		I		K		*M		P		R		T		W		Z
A	B	C	F	G	I	J	K	L	M	O	P	Q	R	S	T	U	W	X	Z
4	3	1	8	6	7	0	2	9	0	8	4	6	3	5	6	3	4	2	0

Switching to ASCII

```
                            K                              @T

            F               K               P               T               Z

    B       F       I       K      *M       P       R       T       W       Z

A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- The figures take a lot of time to draw.

Switching to ASCII

```
                            K                              @T

            F               K               P               T               Z

    B       F       I       K      *M       P       R       T       W       Z

A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.

Switching to ASCII

```
                    K                   @T
          F         K         P         T         Z
    B     F     I   K   *M    P     R   T     W   Z
A B C F G I J K L M O P Q R S T U W X Z
4 3 1 8 6 7 0 2 9 0 8 4 6 3 5 6 3 4 2 0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.
- You have to imagine the boxes.

Switching to ASCII

```
                            K                              @T

            F               K               P               T               Z

    B       F       I       K      *M       P       R       T       W       Z

A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.
- You have to imagine the boxes.
- And the next and previous arrows.

Switching to ASCII

```
                            K                              @T

            F               K               P               T               Z

    B       F       I       K      *M       P       R       T       W       Z

A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.
- You have to imagine the boxes.
- And the next and previous arrows.
- And the downward arrow when the value is an Entry in a lower list.

Switching to ASCII

```
                            K                              @T

            F               K               P               T               Z

    B       F       I       K      *M       P       R       T       W       Z

A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.
- You have to imagine the boxes.
- And the next and previous arrows.
- And the downward arrow when the value is an Entry in a lower list.
- @ is the top.

Switching to ASCII

```
                     K                      @T

         F           K           P           T           Z

   B     F     I     K    *M     P     R     T     W     Z

A  B  C  F  G  I  J  K  L  M  O  P  Q  R  S  T  U  W  X  Z
4  3  1  8  6  7  0  2  9  0  8  4  6  3  5  6  3  4  2  0
```

- The figures take a lot of time to draw.
- Here is an ASCII version.
- You have to imagine the boxes.
- And the next and previous arrows.
- And the downward arrow when the value is an Entry in a lower list.
- @ is the top.
- * is the Entry we are currently looking at, meaning a variable points to it.

Every Other?

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.
- And get takes $\mathrm{O}(n)$ again. ($n/2$ is still $\mathrm{O}(n)$.)

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.
- And get takes $\mathrm{O}(n)$ again. ($n/2$ is still $\mathrm{O}(n)$.)
- How can we select "every other" key in each list that won't be spoiled by removals?

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.
- And get takes $\mathrm{O}(n)$ again. ($n/2$ is still $\mathrm{O}(n)$.)
- How can we select "every other" key in each list that won't be spoiled by removals?


- Answer: flip a coin!

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.
- And get takes $\mathrm{O}(n)$ again. ($n/2$ is still $\mathrm{O}(n)$.)
- How can we select "every other" key in each list that won't be spoiled by removals?


- Answer: flip a coin!
- For each key in a list, I flip a coin. If it is heads, it goes into the next higher list.

Every Other?

- What if someone *removes* every other key: B, F, I, K, M, P, R, T, W, Z?
- Then we are left with just one sorted linked list.
- And get takes $O(n)$ again. ($n/2$ is still $O(n)$.)
- How can we select "every other" key in each list that won't be spoiled by removals?


- Answer: flip a coin!
- For each key in a list, I flip a coin. If it is heads, it goes into the next higher list.
- If you can predict which of my keys will flip heads, you don't need to be studying Computer Science!

Randomized Skip List

	B								O				@T						
	B		F	G					O				T	U					
	B	C	F	G			K		M	O		Q		T	U	W		Z	
A	B	C	F	G	I	J	K	L	M	O	P	Q	R	S	T	U	W	X	Z
4	3	1	8	6	7	0	2	9	0	8	4	6	3	5	6	3	4	2	0

Randomized Skip List

```
    B                                   O                   @T
    B       F   G                       O                   T   U
    B   C   F   G           K       M   O       Q           T   U   W       Z
A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- On *average* you take one step on each level.

Randomized Skip List

```
    B                                   O                   @T
    B       F   G                       O                   T   U
    B   C   F   G           K       M   O       Q           T   U   W       Z
A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.

Randomized Skip List

```
    B                                   O                   @T
    B       F   G                       O                   T   U
    B   C   F   G           K       M   O       Q           T   U   W       Z
A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

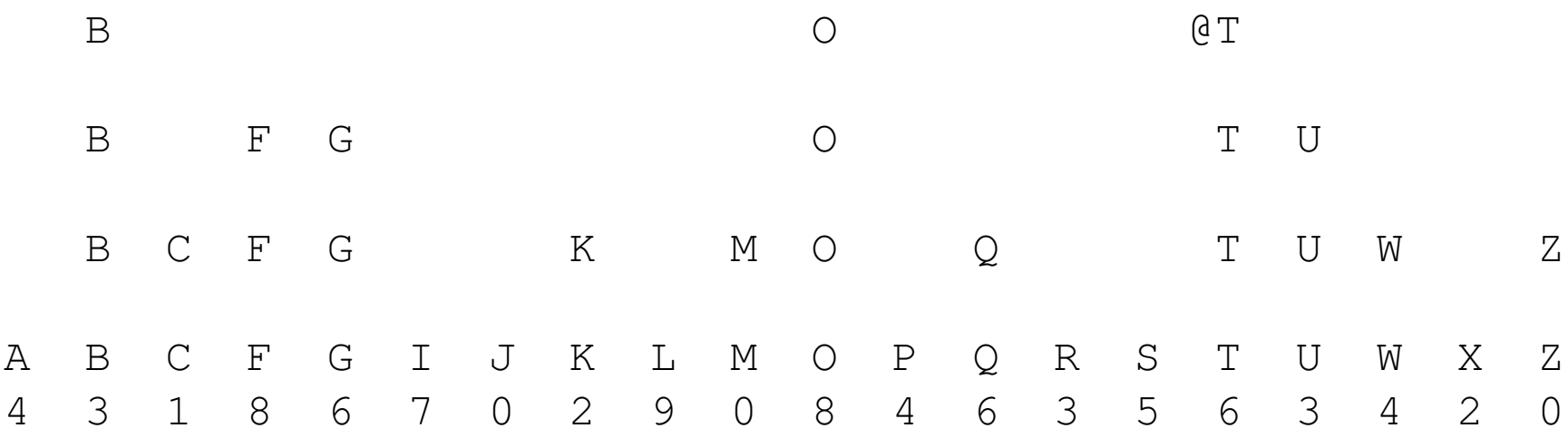
- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.

Randomized Skip List

```
    B                                   O                   @T
    B       F   G                       O                   T   U
    B   C   F   G           K       M   O       Q           T   U   W       Z
A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.

Randomized Skip List



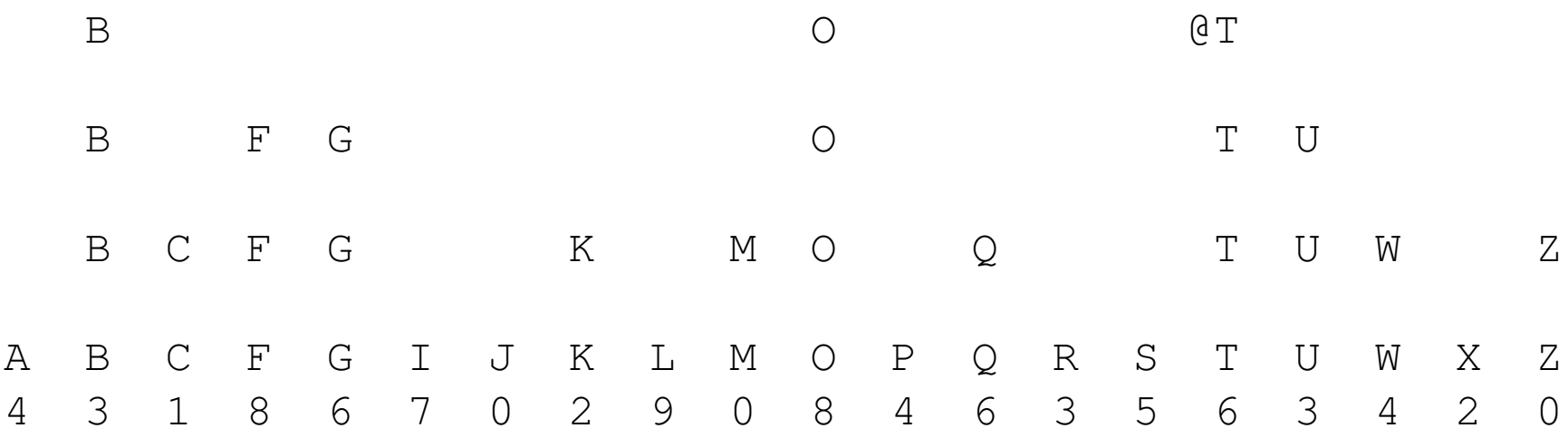
- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.
- About $n/8$ of you get tails and are allowed to flip again.

Randomized Skip List

```
    B                                   O                   @T
    B       F   G                       O                   T   U
    B   C   F   G           K       M   O       Q           T   U   W       Z
A   B   C   F   G   I   J   K   L   M   O   P   Q   R   S   T   U   W   X   Z
4   3   1   8   6   7   0   2   9   0   8   4   6   3   5   6   3   4   2   0
```

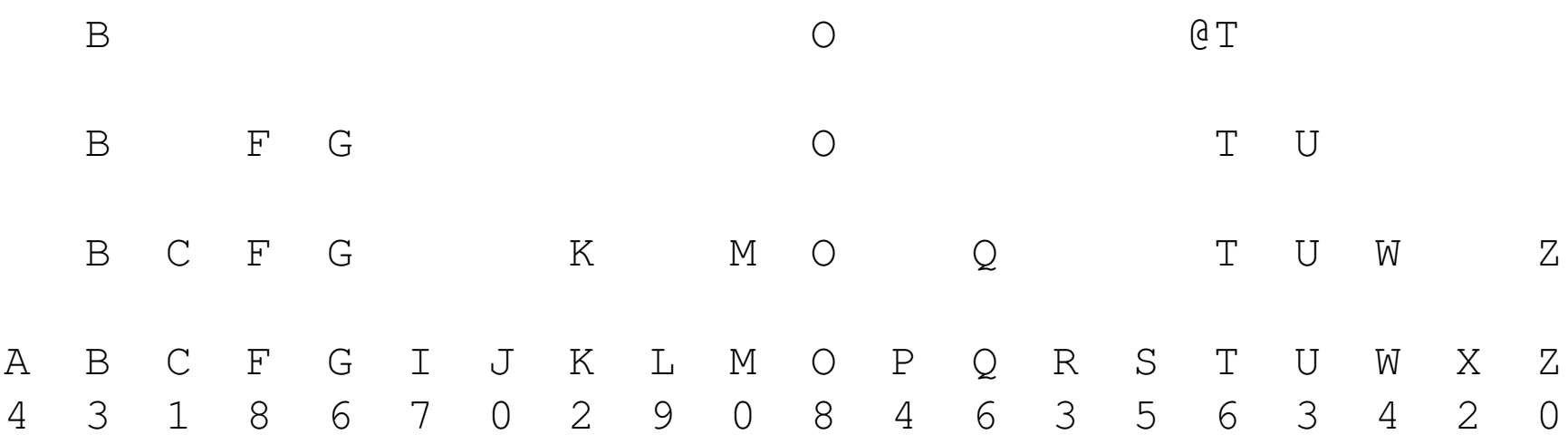
- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.
- About $n/8$ of you get tails and are allowed to flip again.
- Etc. Total number of tails flipped is $n/2 + n/4 + n/8 + \cdots = n$

Randomized Skip List



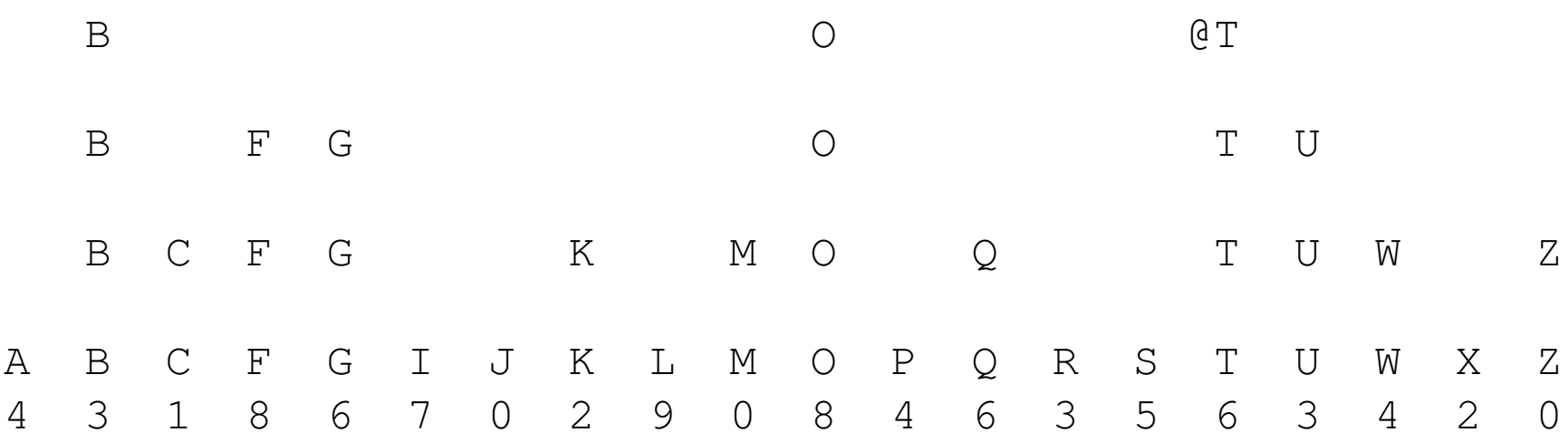
- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.
- About $n/8$ of you get tails and are allowed to flip again.
- Etc. Total number of tails flipped is $n/2 + n/4 + n/8 + \cdots = n$
- So on average everyone flips one tail before getting heads.

Randomized Skip List



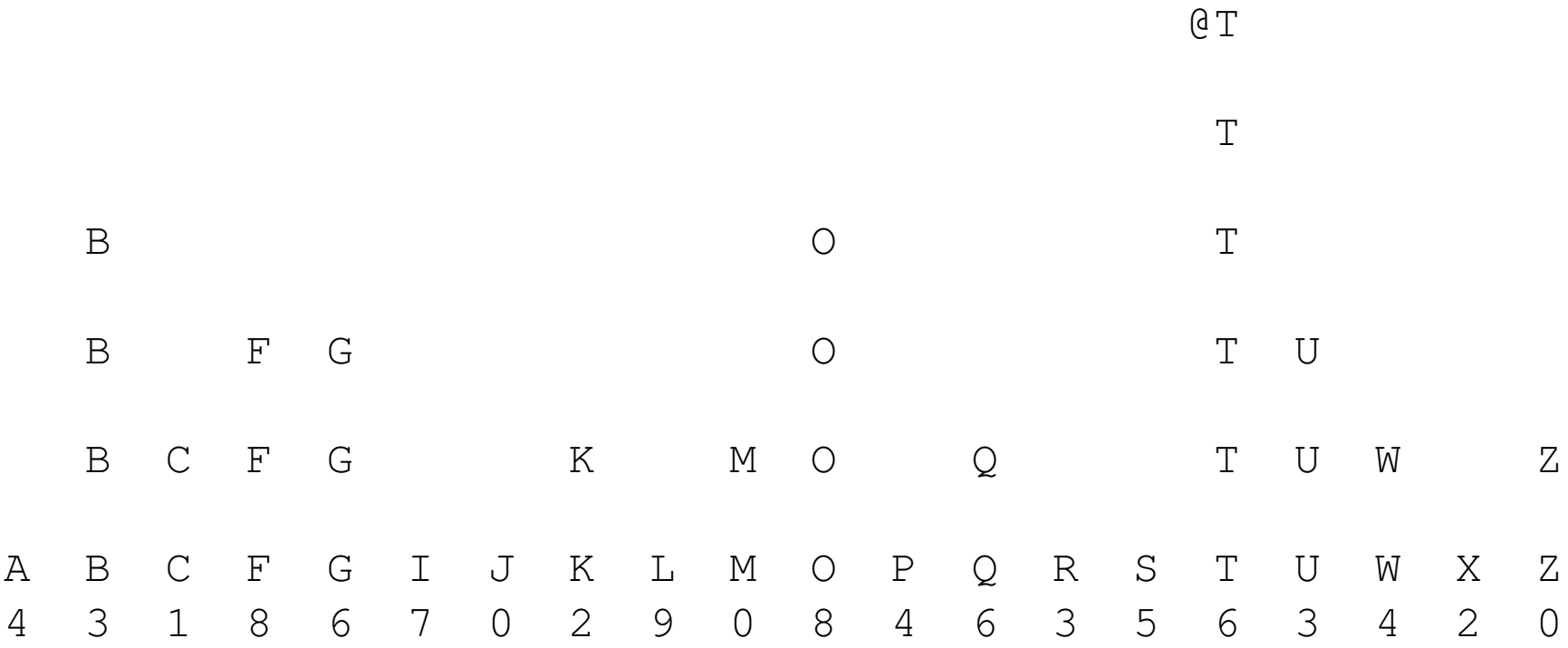
- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.
- About $n/8$ of you get tails and are allowed to flip again.
- Etc. Total number of tails flipped is $n/2 + n/4 + n/8 + \cdots = n$
- So on average everyone flips one tail before getting heads.
- So on average you skip one key before promoting the next one to a higher level.

Randomized Skip List

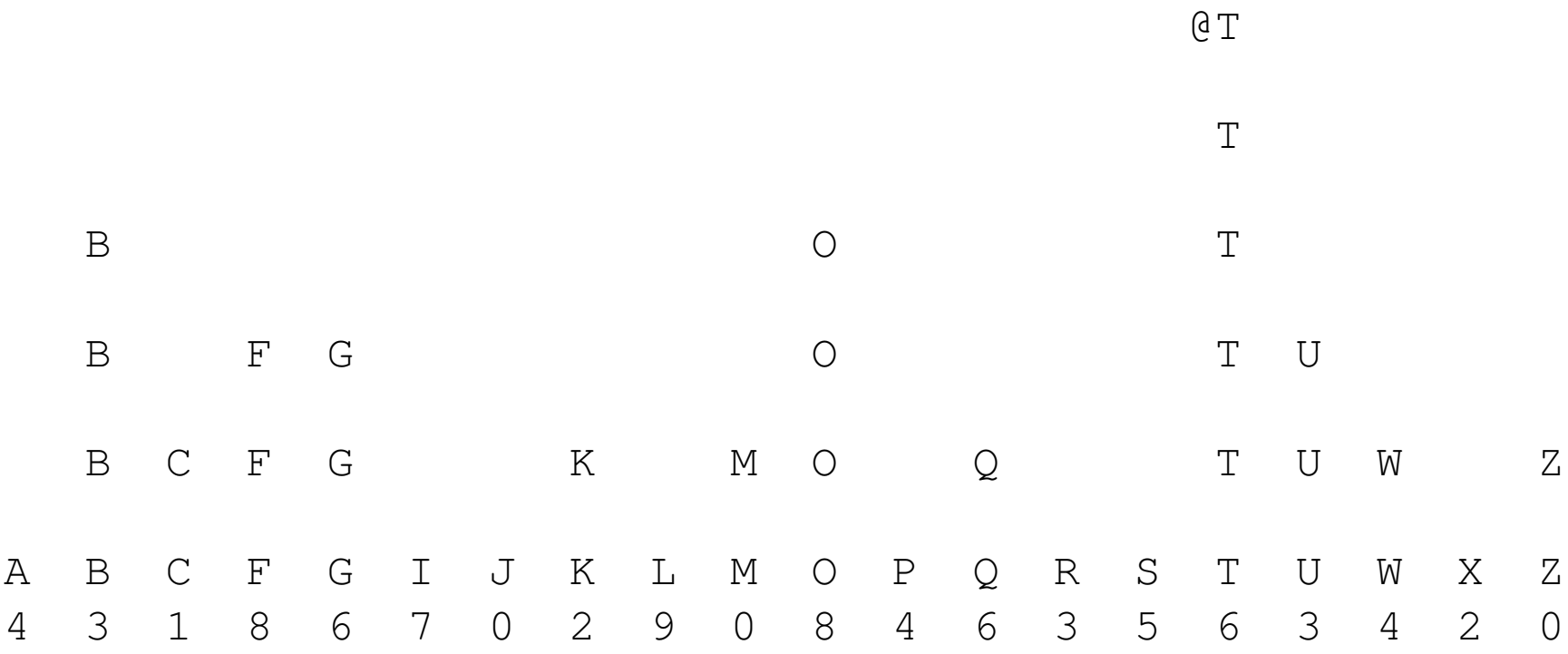


- On *average* you take one step on each level.
- Why? Suppose all n of you flipped a coin.
- About $n/2$ of you get tails and are allowed to flip again.
- About $n/4$ of you get tails and are allowed to flip again.
- About $n/8$ of you get tails and are allowed to flip again.
- Etc. Total number of tails flipped is $n/2 + n/4 + n/8 + \cdots = n$
- So on average everyone flips one tail before getting heads.
- So on average you skip one key before promoting the next one to a higher level.
- You will learn this more formally in MTH224.

Randomized Skip List

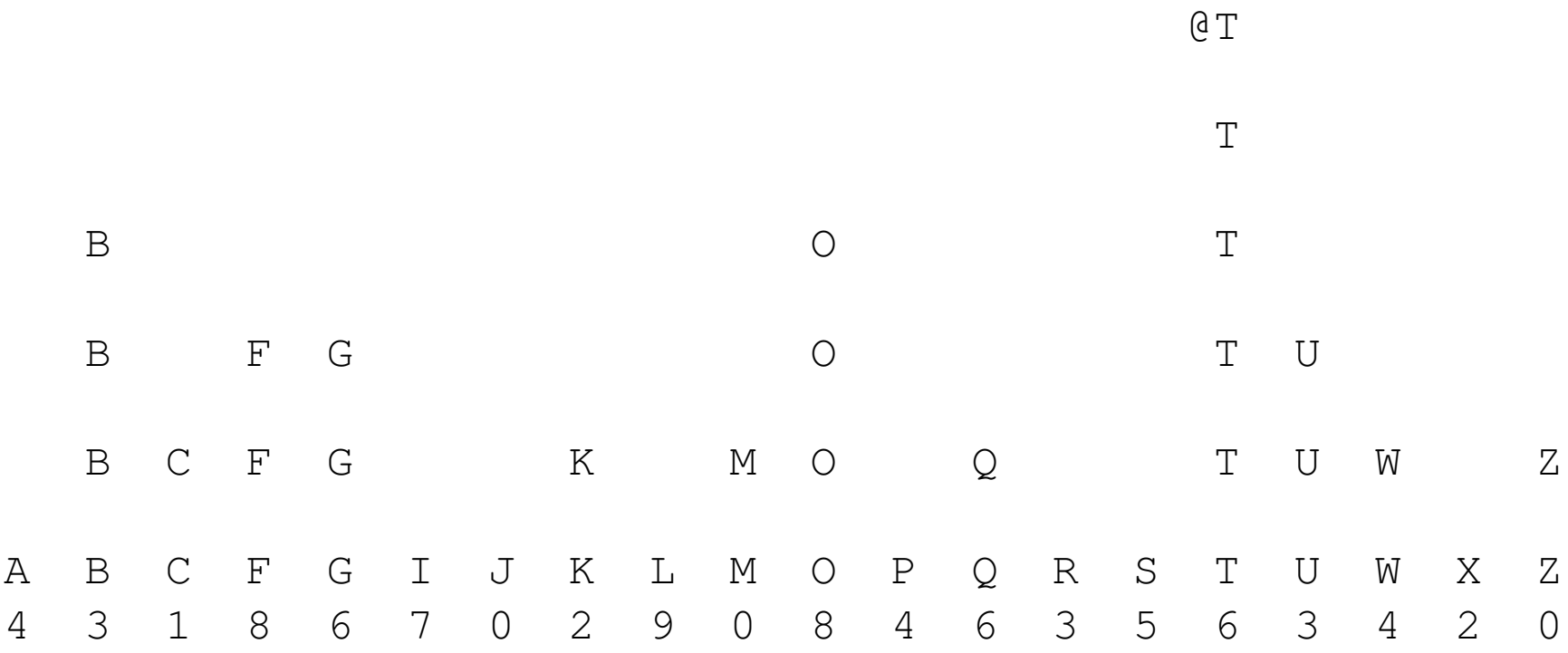


Randomized Skip List



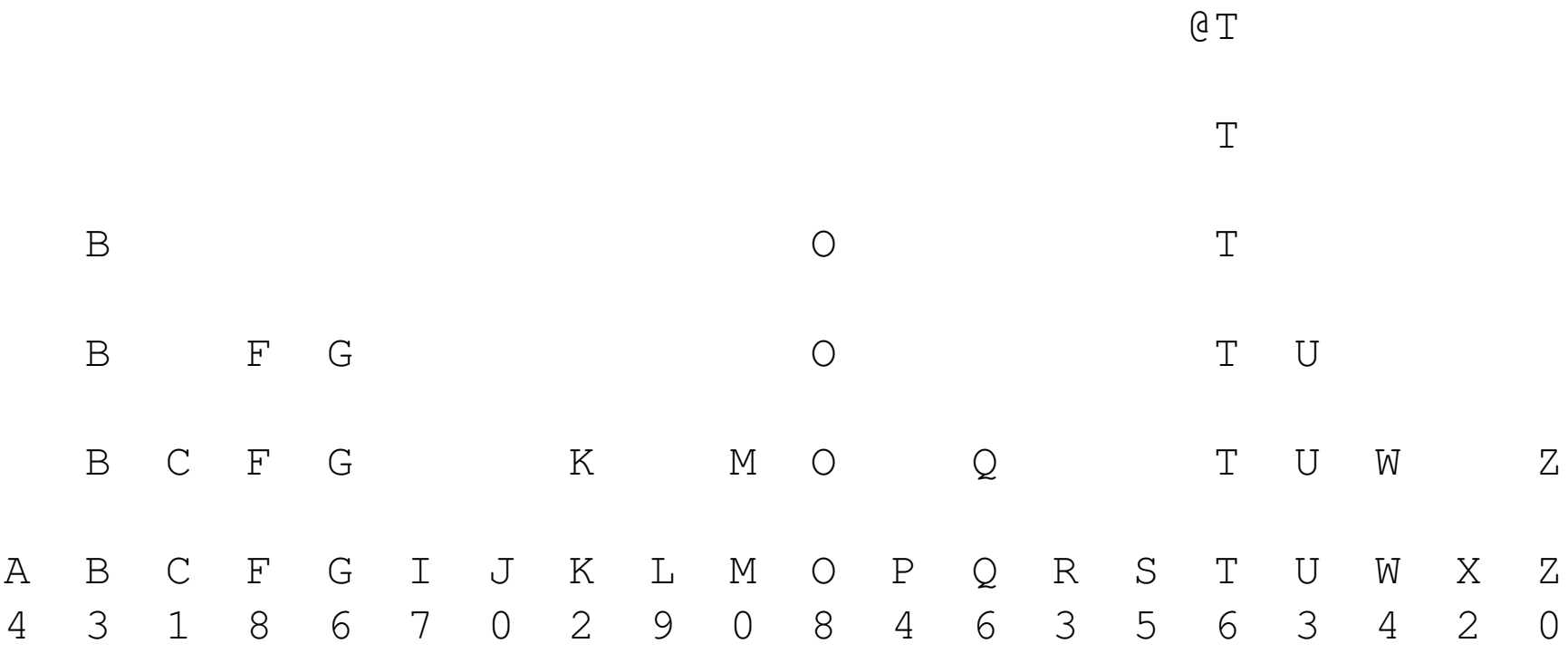
- Actually, it usually looks more like this.

Randomized Skip List



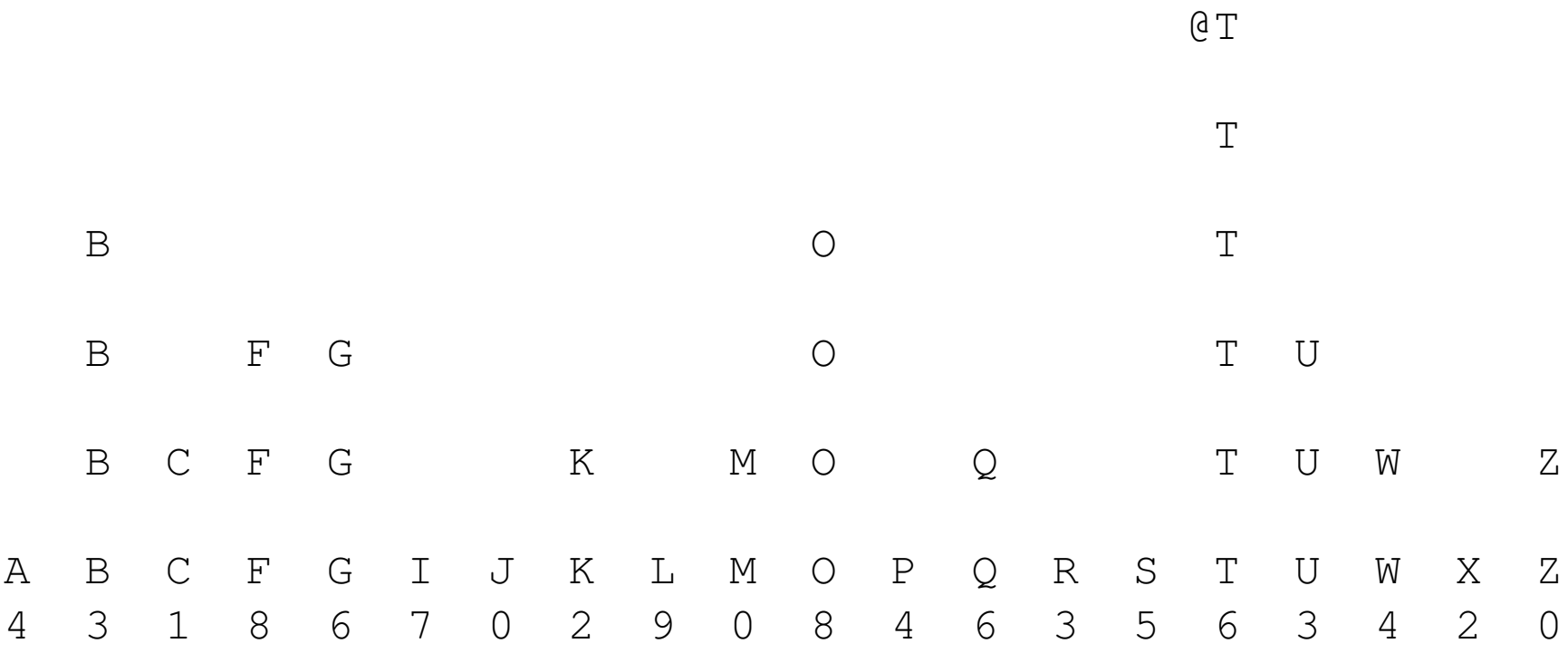
- Actually, it usually looks more like this.
- There always seems to be one key that is really lucky.

Randomized Skip List



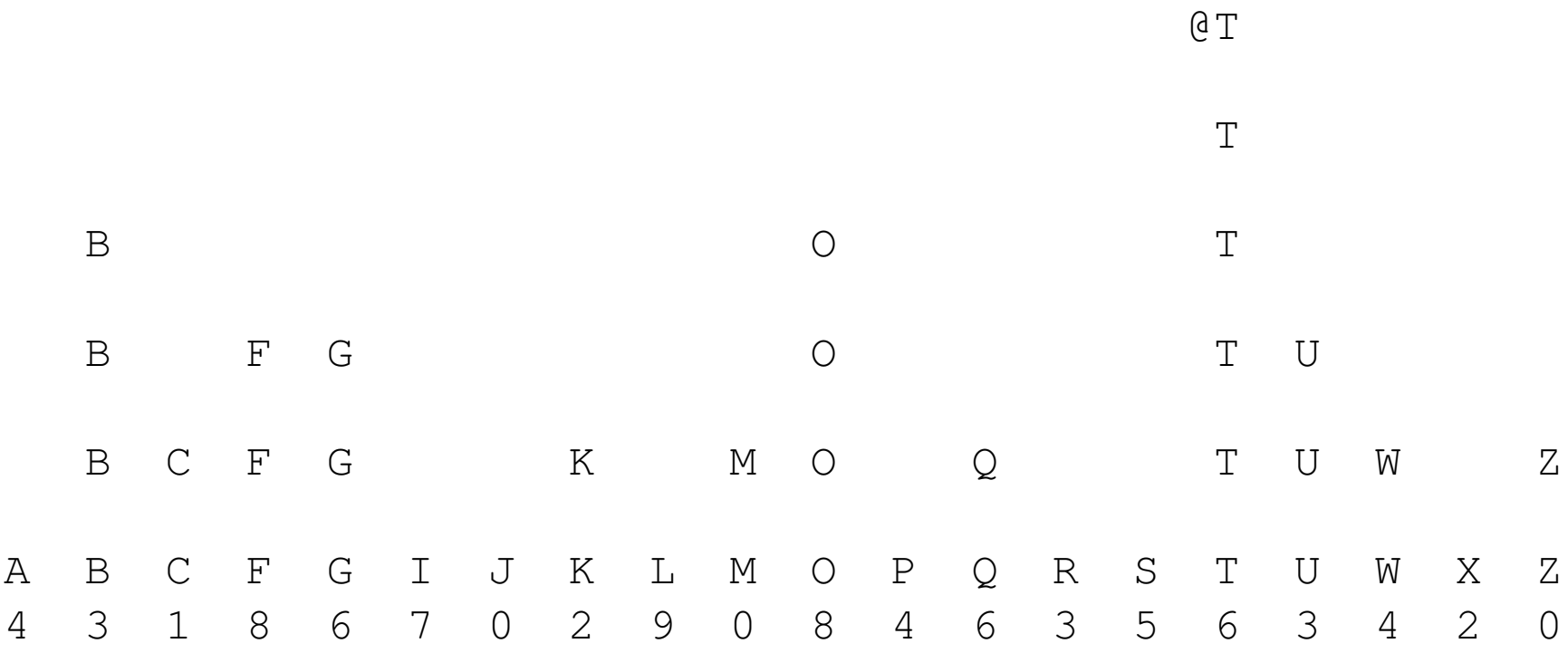
- Actually, it usually looks more like this.
- There always seems to be one key that is really lucky.
- As long as it keeps flipping heads,

Randomized Skip List



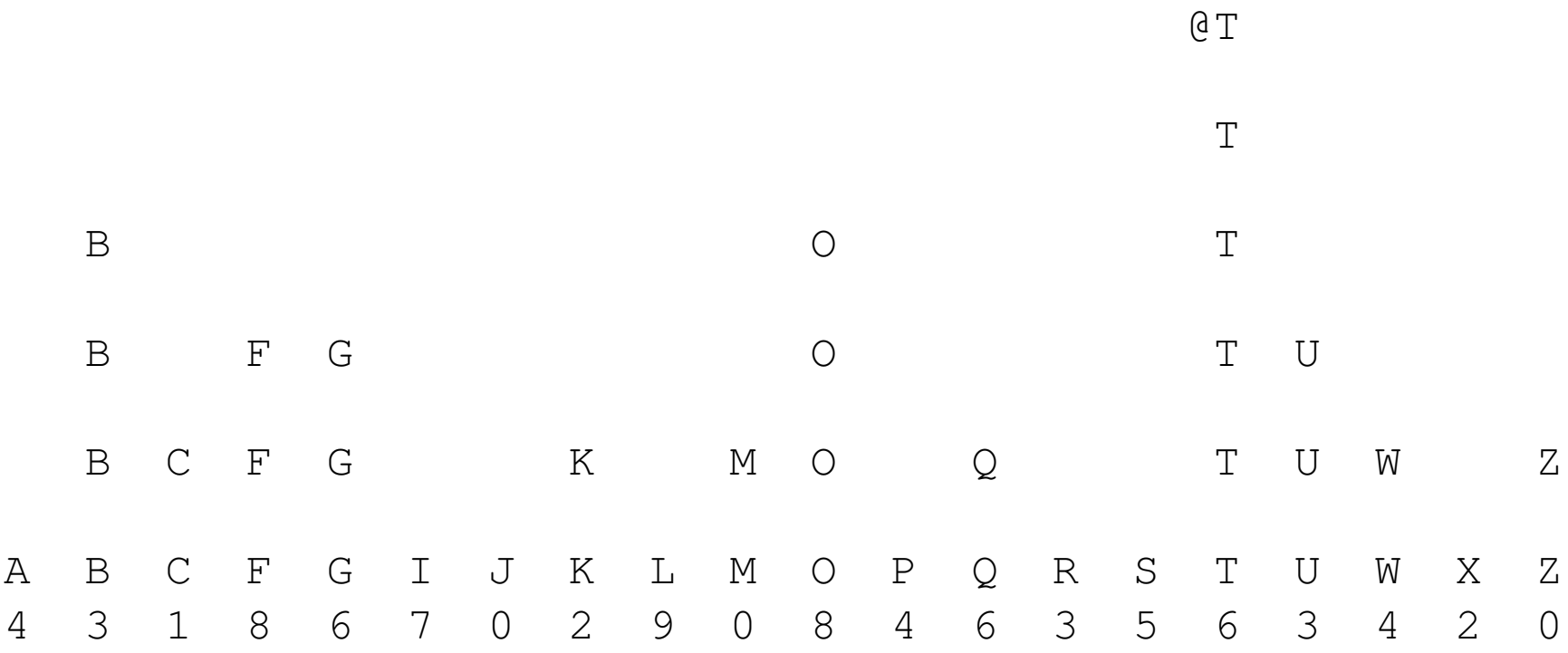
- Actually, it usually looks more like this.
- There always seems to be one key that is really lucky.
- As long as it keeps flipping heads,
- it goes into the next higher list.

Randomized Skip List



- Actually, it usually looks more like this.
- There always seems to be one key that is really lucky.
- As long as it keeps flipping heads,
- it goes into the next higher list.
- Even if that means it is by itself.

Randomized Skip List



- Actually, it usually looks more like this.
- There always seems to be one key that is really lucky.
- As long as it keeps flipping heads,
- it goes into the next higher list.
- Even if that means it is by itself.
- If the list doesn't exist already, you have to create it.

Lab and Homework

Lab and Homework

- For lab tomorrow, you will implement DLLMap

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.
  - rFind.pdf shows how its recursive find method works.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.
  - rFind.pdf shows how its recursive find method works.
  - rAdd.pdf shows how its recursive add method works.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.
  - rFind.pdf shows how its recursive find method works.
  - rAdd.pdf shows how its recursive add method works.
  - put.pdf shows how its put method works.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.
  - rFind.pdf shows how its recursive find method works.
  - rAdd.pdf shows how its recursive add method works.
  - put.pdf shows how its put method works.
  - get calls rFind and put calls rFind and rAdd.

Lab and Homework

- For lab tomorrow, you will implement DLLMap
  - which is just SortedDLLPD done formally.
  - find.pdf shows how its find method works.
- For the homework, you will implement SkipMap: a Skip List based Map.
  - rFind.pdf shows how its recursive find method works.
  - rAdd.pdf shows how its recursive add method works.
  - put.pdf shows how its put method works.
  - get calls rFind and put calls rFind and rAdd.
- Remove is extra credit: earn 50 points towards any assignment you earned less than 50 points.

Summary

Summary

- A Skip List consists of $O(\log n)$ linked lists.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.
  - Put has to keep creating new lists on top,

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.
  - Put has to keep creating new lists on top,
  - as long as the key keeps flipping heads.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.
  - Put has to keep creating new lists on top,
  - as long as the key keeps flipping heads.
- Remove just removes a key from every list it appears in.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at “top”, an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.
  - Put has to keep creating new lists on top,
  - as long as the key keeps flipping heads.
- Remove just removes a key from every list it appears in.
- Get, put, and remove are all $O(\log n)$ expected time.

Summary

- A Skip List consists of $O(\log n)$ linked lists.
  - The bottom list has all the keys and their actual values.
  - In each other list, the value is the Entry with the same key in the next lower list.
  - In each list, a key goes into the next higher list if it flips heads.
- Find starts at "top", an element of the highest list,
  - moves left or right to get as close as possible,
  - then goes down to the next lower list.
- Add adds a new key to the lowest list,
  - and a higher list for each time it flips heads.
  - Put has to keep creating new lists on top,
  - as long as the key keeps flipping heads.
- Remove just removes a key from every list it appears in.
- Get, put, and remove are all $O(\log n)$ expected time.
- The amount of space $2n$ (expected), still $O(n)$.

Summary

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.
  - You can tell find which Entry to start searching from.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is “cemoprtu”, value is “computer”.

Summary

- “Formal Phone Directory” Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn’t there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is “cemoprtu”, value is “computer”.
  - Unscramble “rtpmceuo” by sorting to “cemoprtu” and looking up its value.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".
- Analysis of SkipMap

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".
- Analysis of SkipMap
  - Only twice a much space as DLLMap.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".
- Analysis of SkipMap
  - Only twice a much space as DLLMap.
  - get, put, and remove are all $O(\log n)$ on average.

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".
- Analysis of SkipMap
  - Only twice a much space as DLLMap.
  - get, put, and remove are all $O(\log n)$ on average.
  - So total time to read in the dictionary is

Summary

- "Formal Phone Directory" Map interface has put, get, and remove.
  - Can implement as sorted doubly linked list DLLMap.
  - If key isn't there, find returns a neighbor.
  - You can tell find which Entry to start searching from.
- Map helps solve Daily Jumble.
  - Store each word under its sorted key.
  - Example: key is "cemoprtu", value is "computer".
  - Unscramble "rtpmceuo" by sorting to "cemoprtu" and looking up its value.
- DLLMap requires $O(n^2)$ to read in dictionary.
- SkipMap extends DLLMap.
  - Uses "skip lists" which skip "every other" node.
  - Uses coin flips to define "every other".
- Analysis of SkipMap
  - Only twice a much space as DLLMap.
  - get, put, and remove are all $O(\log n)$ on average.
  - So total time to read in the dictionary is
  - $O(n \log n)$.