CS2020
Data Structures and Algorithms

Welcome!

Quiz 1

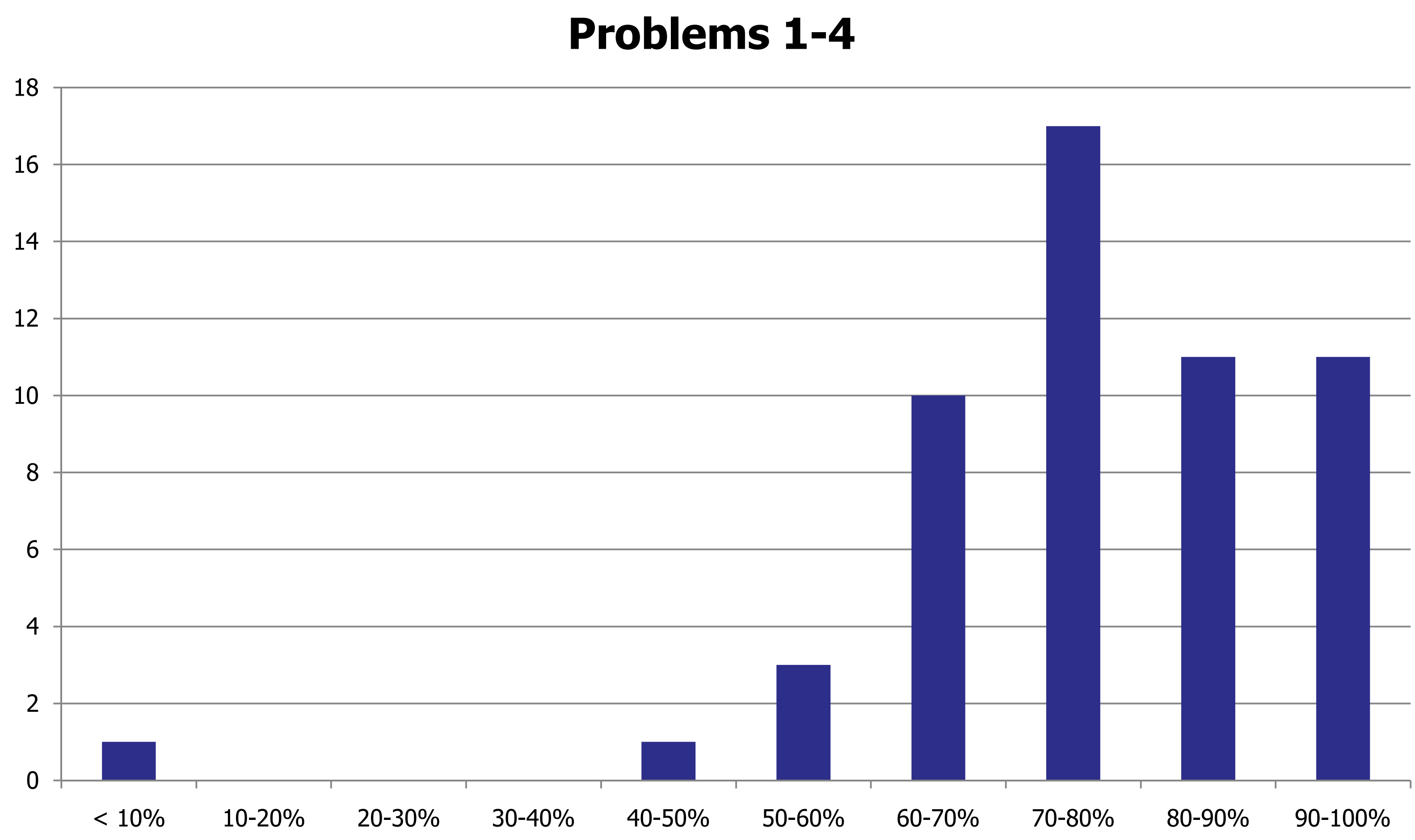
To be returned in DG this week

- Or by the end of the week…

Preliminary notes:

- Focus on basic properties.
- Always try to get partial credit.
- Java is tricky (especially without a compiler).

Quiz 1: Preliminary Results



Quiz 1: Preliminary Results

Problem	Max	Average
1. Recurrences	15	12
2. Multiple Choice	40	33
3. Java	40	27
4. Data Structure Basics	40	34

Conclusions:

- Recurrences: learned!
- Properties of basic data structures: mostly learned…
- Object-oriented programming: more needed…

Coding Quiz

Date: February 28-March 4

Time: during Discussion Group

Location: TBA

Details:

- Object-oriented basics (inheritance, interfaces).
- Implement simple data structures as classes.
- Extend implementations to add functionality.
- Use existing implementations to solve problems.
- Testing and debugging.

Questions?

Today

Two goals:

1. SkipList: an *interesting* data structure

2. SkipList: an *example* of how to implement a simple data structure in only one lecture.

Binary Search Tree

Problem Set 3

– Implement a simple binary search tree.

– Unbalanced

Problem Set 5

– Implement a simple balancing operation.

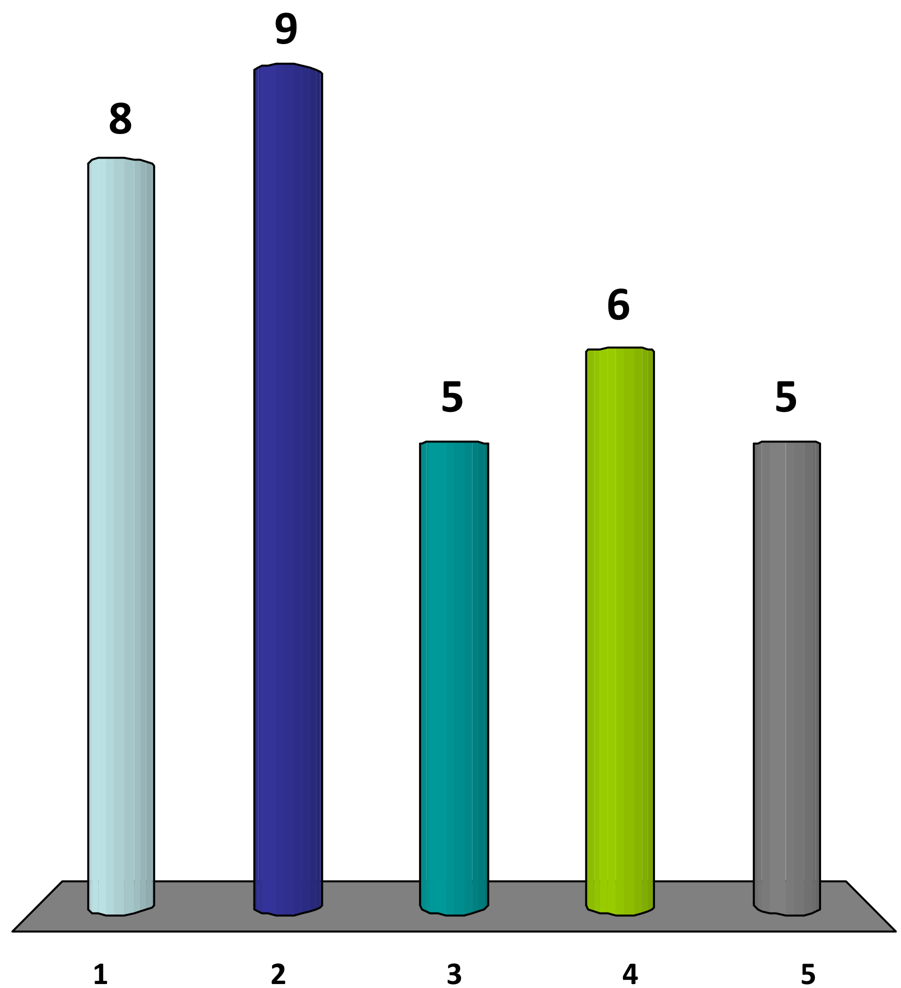
Today:

– See some common problems of what goes wrong.

– Some basic ideas on how to get it right.

Implementing a BST on PS3 was:

1. Easy.
2. Ok.
3. Pretty hard.
4. Very hard.
5. Impossible.



Problem Set 3

What went right?

- Basic java: no problem.
- Basic object-oriented design: ok.

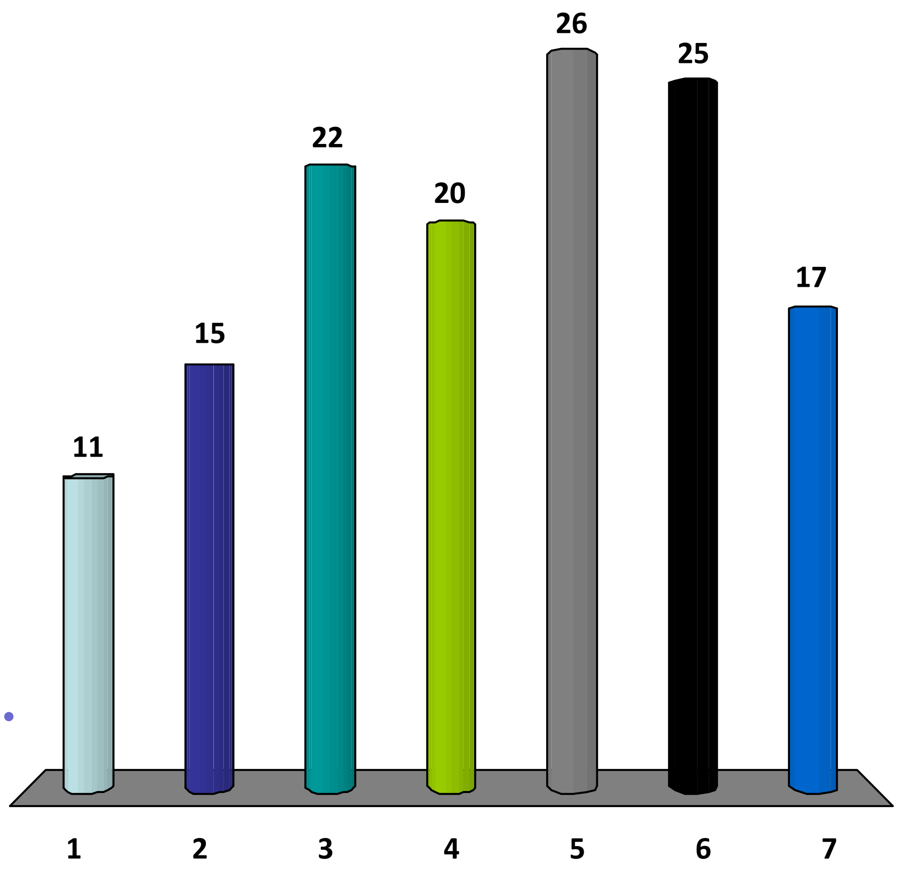
What went wrong?

- Corner-cases: checking at the edges.
- Testing: write good test cases.

- (Also: following directions: don’t change the interface!)

On my BST, I tested: (click all that apply)

1. Empty tree case.
2. One node tree.
3. In order insertions.
4. Random insertions.
5. Weight updated correctly on insert.
6. buildTree updates weight correctly.
7. buildTree is balanced.



Recipe

SkipList from scratch:

1. Linked List (easy exercise)
    - List interface
    - ListNode implementation
    - LinkedList implementation
2. SkipList
    - Search interface
    - SkipListNode interface
    - SkipList implementation
3. Analysis (randomized and slightly tricky)

SkipList Background

Simple randomized, dynamic search structure

- Invented by William Pugh in 1989
- Easy to implement

Maintains a set of n elements:

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

with high probability

SkipList Background

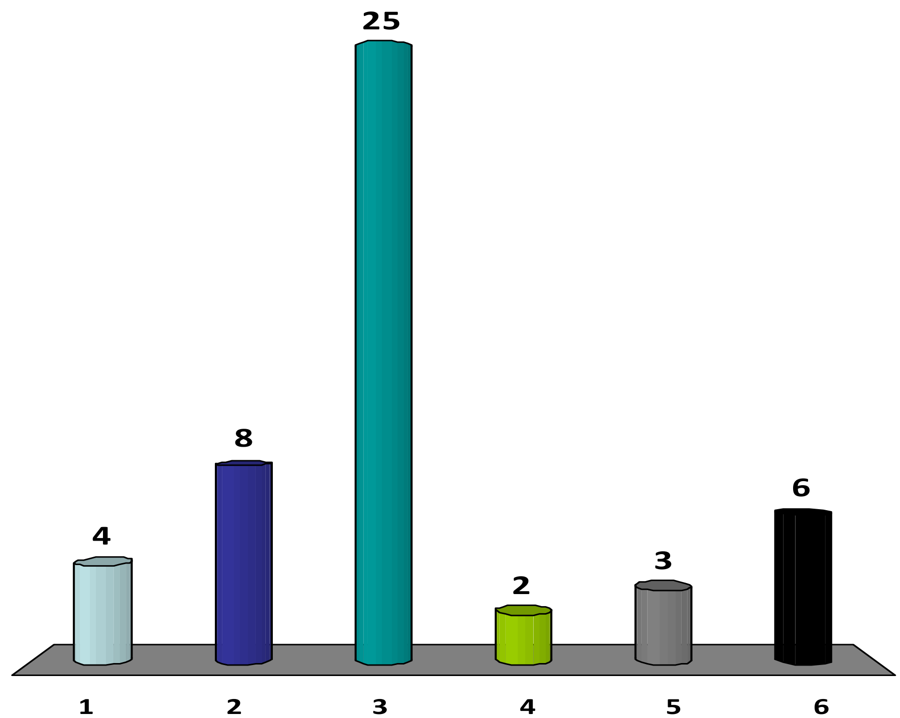
Compared:

- AVL trees
  - Deterministic.
  - More complicated to implement (many cases).
- SkipLists
  - Randomized.
  - Simple to implement.
  - More efficient “range queries” : e.g.,

    “Find all the elements with keys in the range [23, 100].”

B-trees are better than AVL trees (and SkipLists) in the following way(s):

1. Fewer steps per operation.
2. Balancing changes fewer pointers.
3. Better cache performance.
4. Uses less memory.
5. Delete has fewer cases.
6. None of the above.



SkipList Background

Compared:

- B-trees
  - Better cache-locality.
- SkipLists
  - Worse cache-locality.
  - Uses memory less efficiently.

Idea: SkipList = Randomized B-tree??

SkipList Background

Compared:

- Hash tables
  - Faster searching.
- SkipLists
  - Faster range queries.
  - Faster successor/predecessor queries.

Recipe

SkipList from scratch:

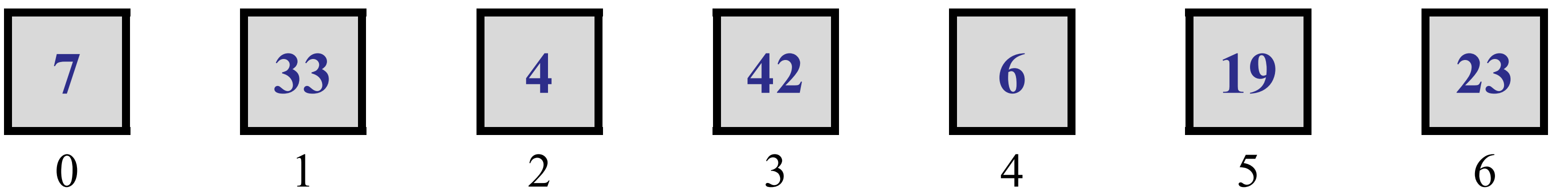
1. Linked List (easy exercise)
    - List interface
    - ListNode implementation
    - LinkedList implementation
2. SkipList
    - Search interface
    - SkipListNode interface
    - SkipList implementation
3. Analysis (randomized and slightly tricky)

Linked Lists

<u>Abstract Data Type</u>: List

– See: java.util.List

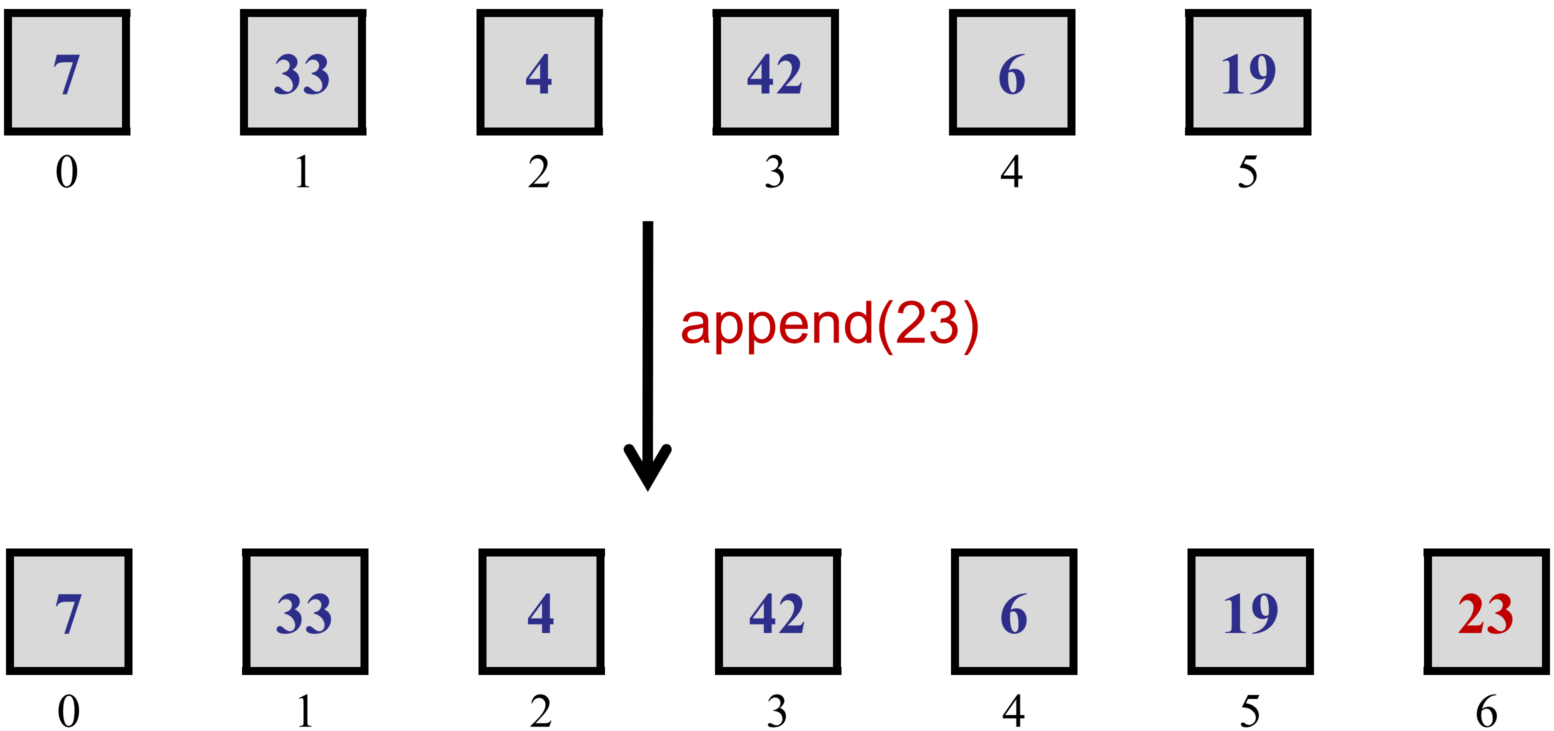
– See: Scheme



Linked Lists

Basic list functionality:

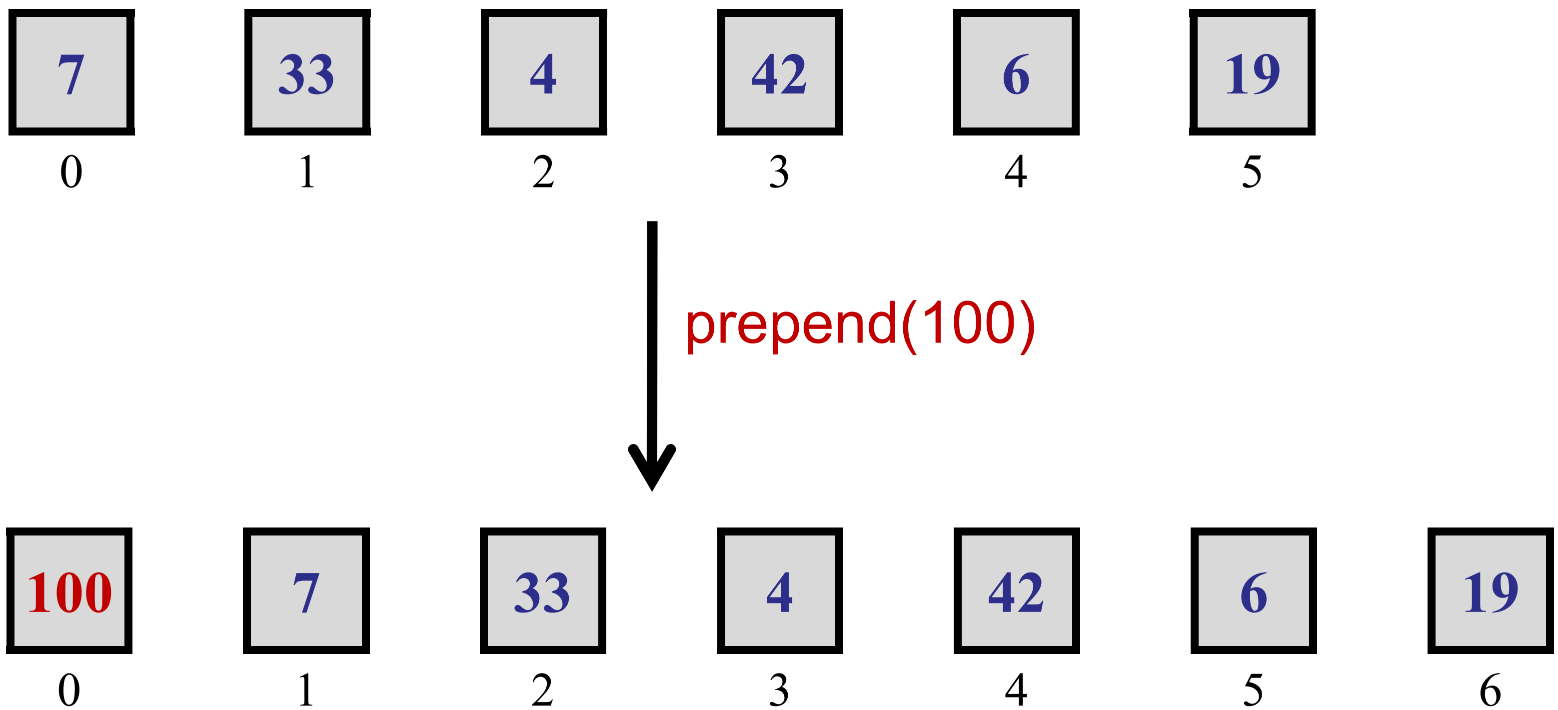
Add to end of list



Linked Lists

Basic list functionality:

Add to beginning of list



Linked Lists

Basic list functionality:

Get element from list

7	**33**	**4**	**42**	**6**	**19**
0	1	2	3	4	5

get(3) = 42

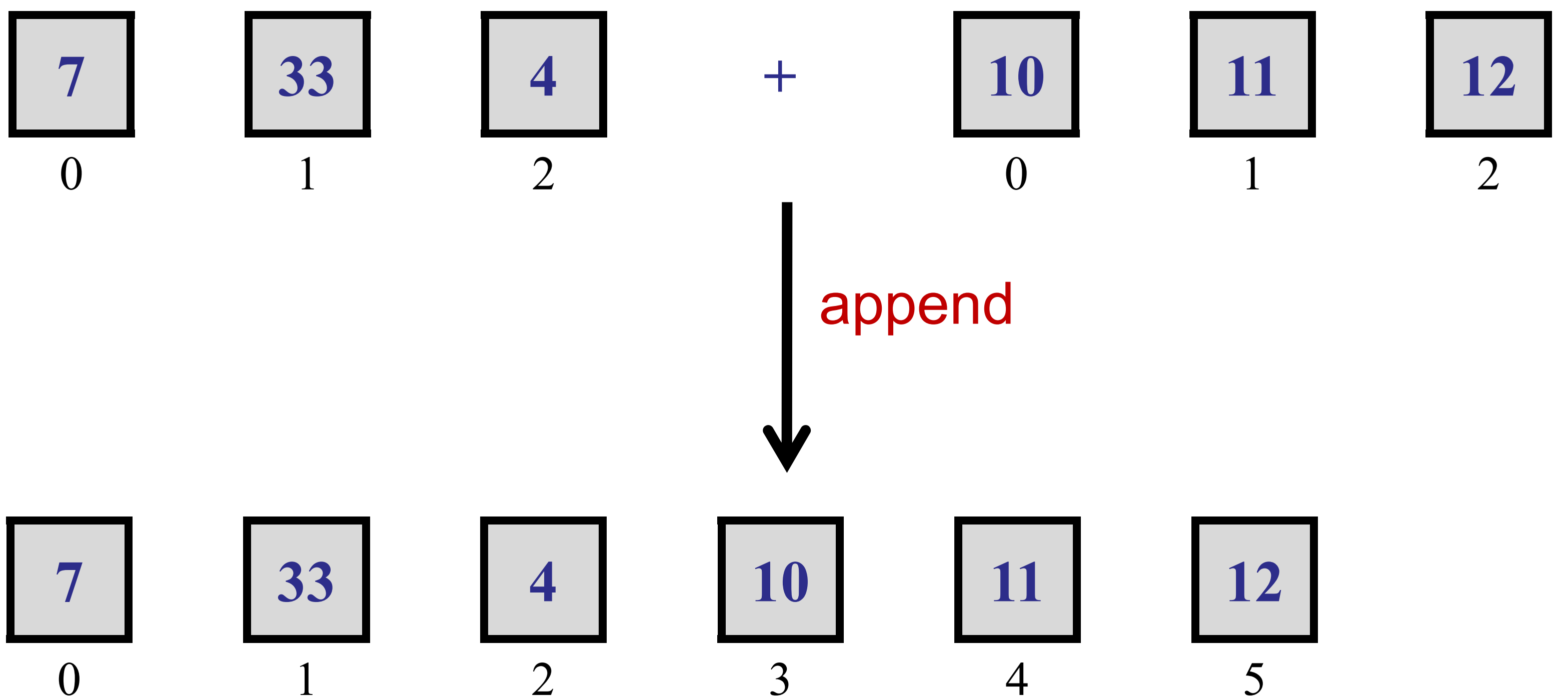
get(0) = 7

get(6) = ??

Linked Lists

Basic list functionality:

Concatenate two lists

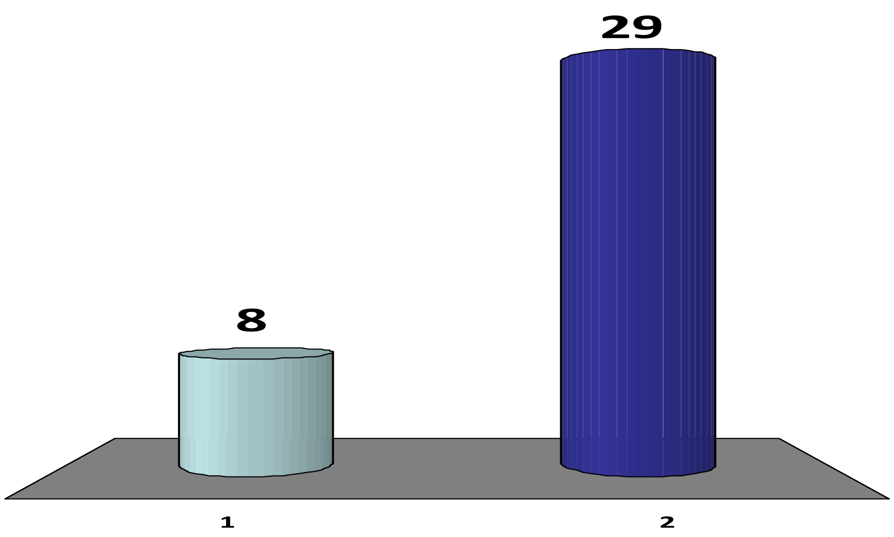


Simple List Interface

```
public interface IList<TData> {
     public int getKey(int i);
     public TData getData(int i);

     public void prepend(int key, TData data);

     public void append(int key, TData data);

     public void append(IList<TData> list);


     public boolean isEmpty();
     public int getSize();
}
```

Should we include exception-handling in the interface?

1. Yes, an interface should specify how errors are handled.
2. No, we should leave the choice of error handling to the implementer.



Simple List Interface

```
public interface IList<TData> {
     public int getKey(int i) throws LLException;
     public TData getData(int i) throws LLException;

     public void prepend(int key, TData data)
                                   throws LLException;
     public void append(int key, TData data)
                                   throws LLException;
     public void append(IList<TData> list)
                                   throws LLException;

     public boolean isEmpty() throws LLException;
     public int getSize() throws LLException;
}
```

Simple List Interface

```
public class LLException extends Exception {

}
```

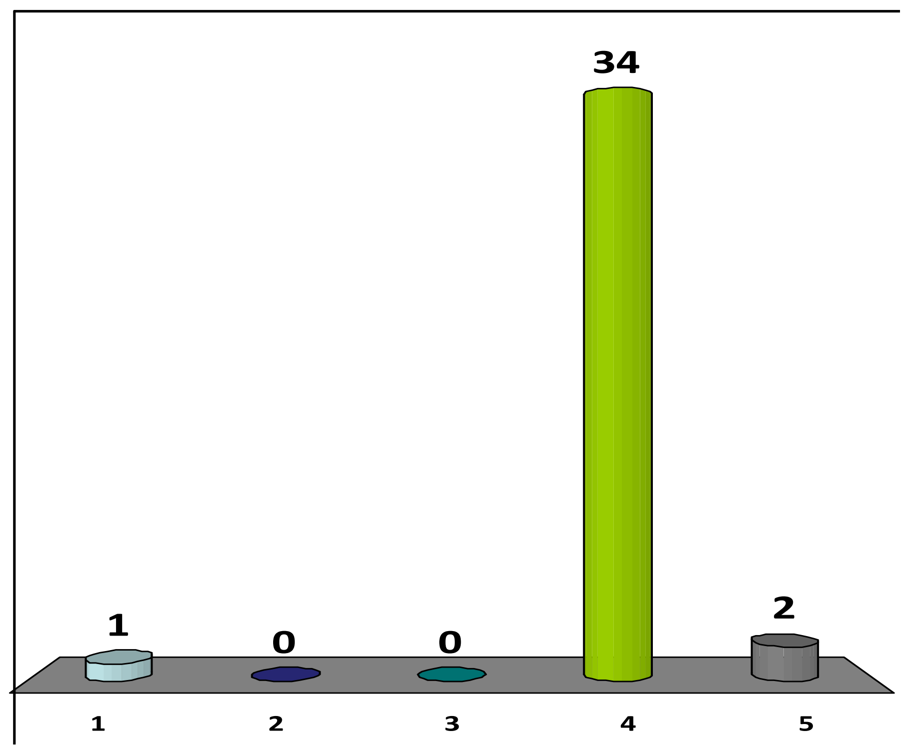
List Implementation

Possible Choices:

- Implement using an array (see: java.util.ArrayList).
- Implement using a vector (see: java.util.Vector).
- Implement using a BST.
- Implement using a queue.
- …

The challenge of implementing a List using an array is:

1. Accessing elements in an array is slow.
2. Arrays use memory inefficiently.
3. List elements are too big to store in array cells.
4. The size of the list is dynamic and the size of an array is static.
5. None of the above.



List Implementation

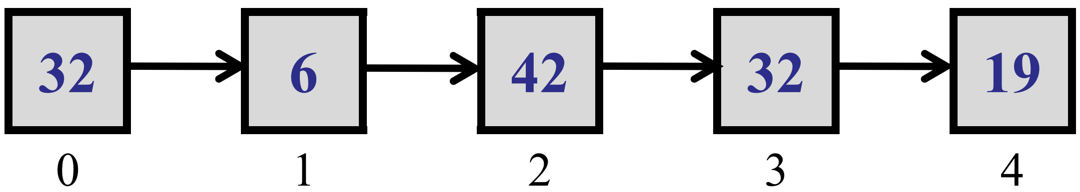
Possible Choices:

- Implement using an array (see: java.util.ArrayList).
- Implement using a vector (see: java.util.Vector).
- Implement using a BST.
- Implement using a queue.
- …
- Implement using a LinkedList

Linked List Implementation

Basic structure:

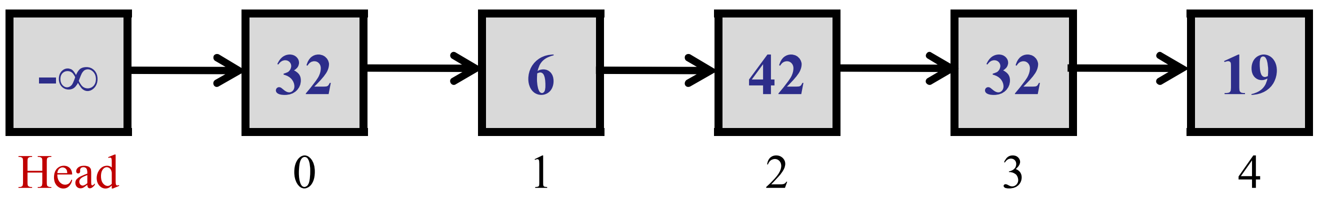
- Chained array of ListNodes.



Linked List Implementation

Basic structure:

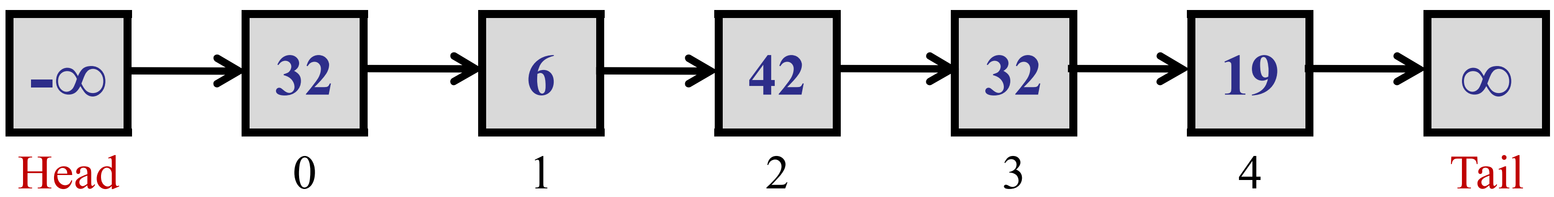
- Chained array of ListNodes.
- Special head node.



Linked List Implementation

Basic structure:

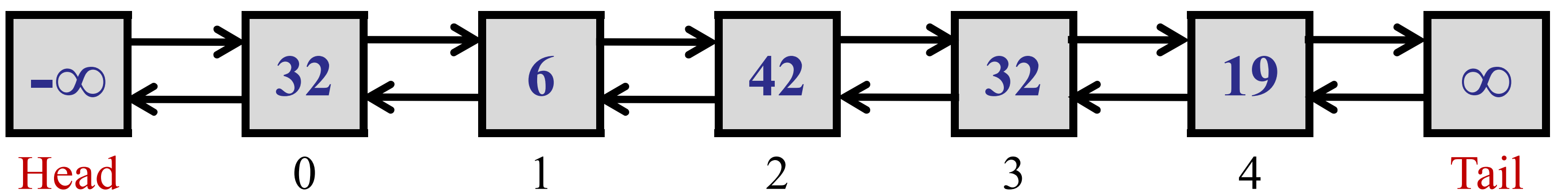
- Chained array of ListNodes.
- Special head node.
- Special tail node.



Linked List Implementation

Basic structure:

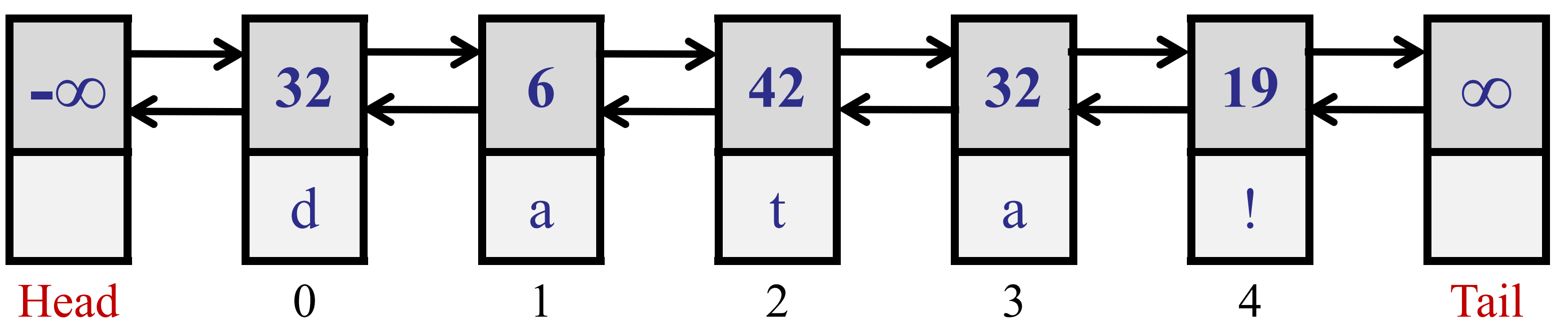
- Chained array of ListNodes.
- Special head node.
- Special tail node.
- Doubly-linked.



Linked List Implementation

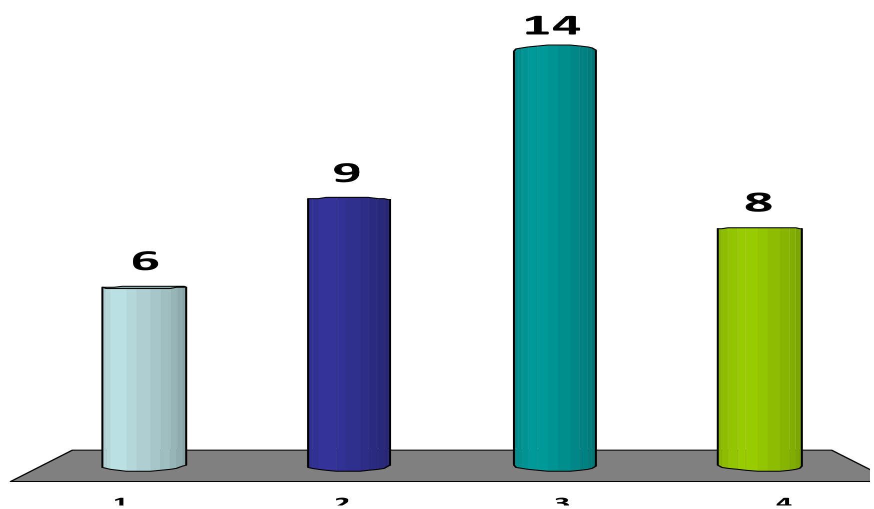
Basic structure:

- Chained array of ListNodes.
- Special head node.
- Special tail node.
- Doubly-linked.
- Add cells for data.



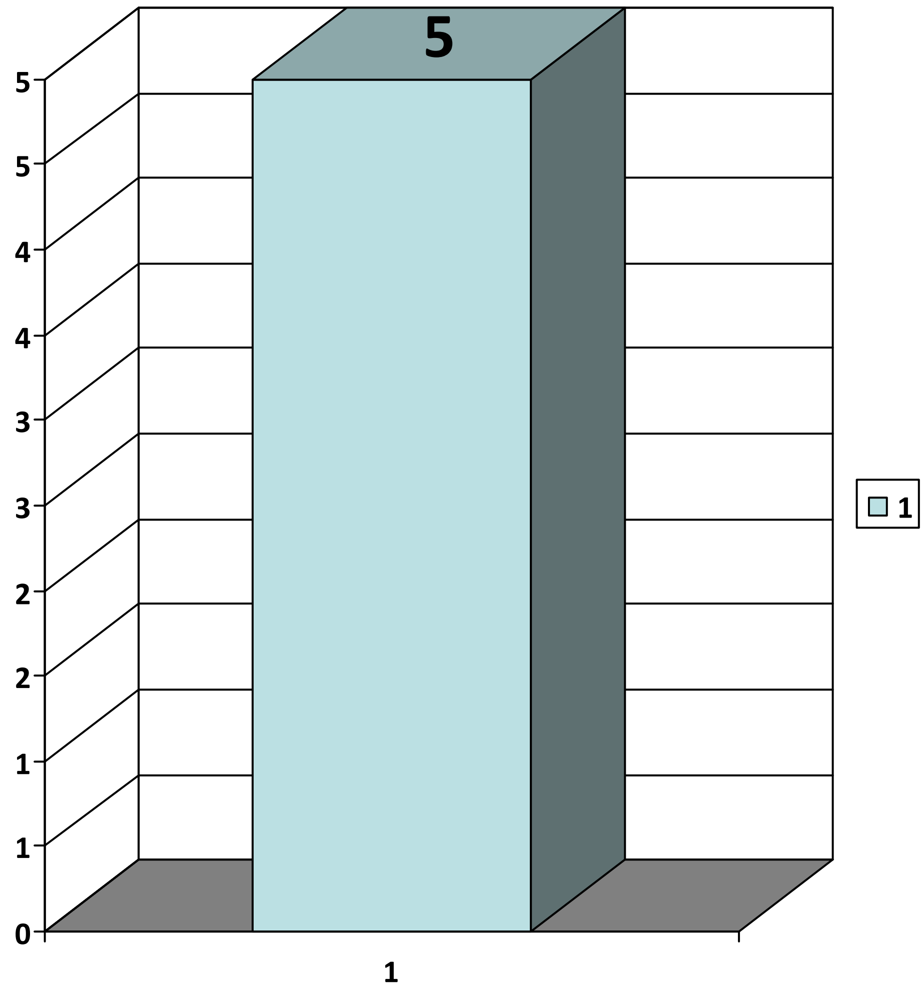
Why do we keep a separate special head node? Which of the following is WRONG?

1. To avoid special code for an empty list.
2. To store extra information (e.g., the size of the list)
3. To make the program run faster.
4. To simplify pre-pending an item to the list.



Enter question text...

1. Enter answer text...

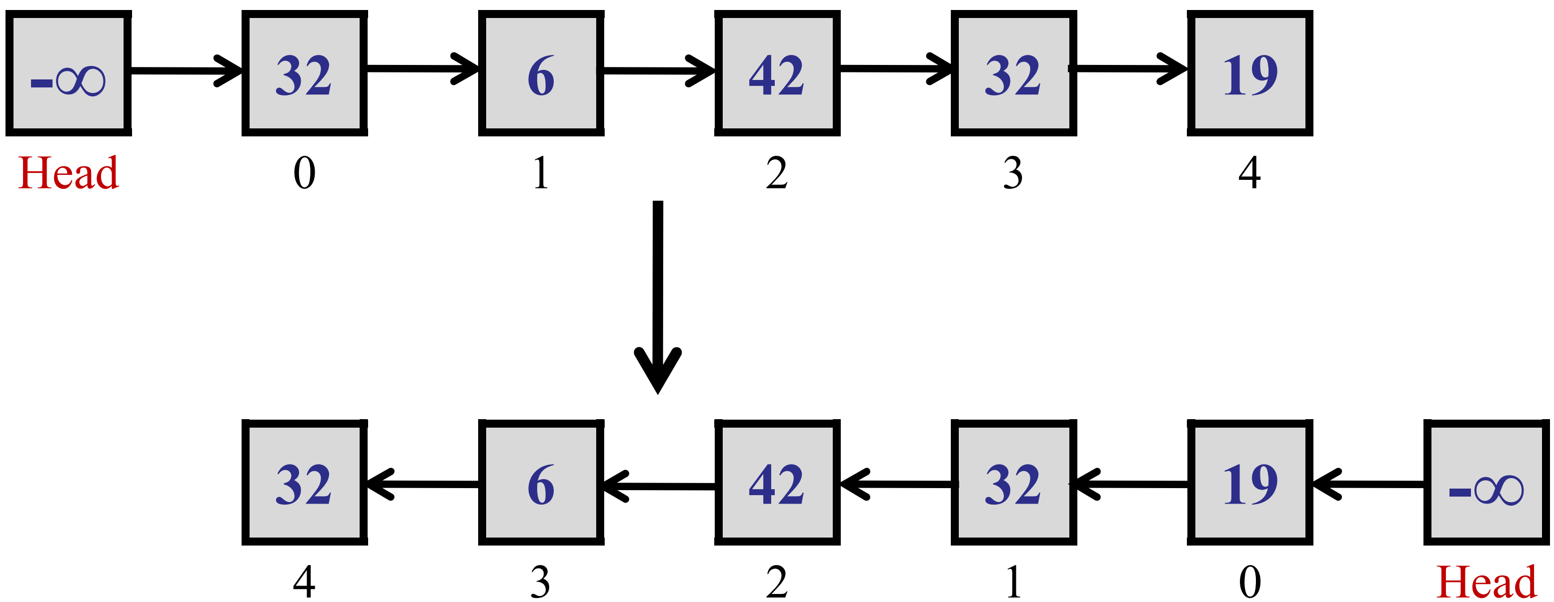


Puzzle Break

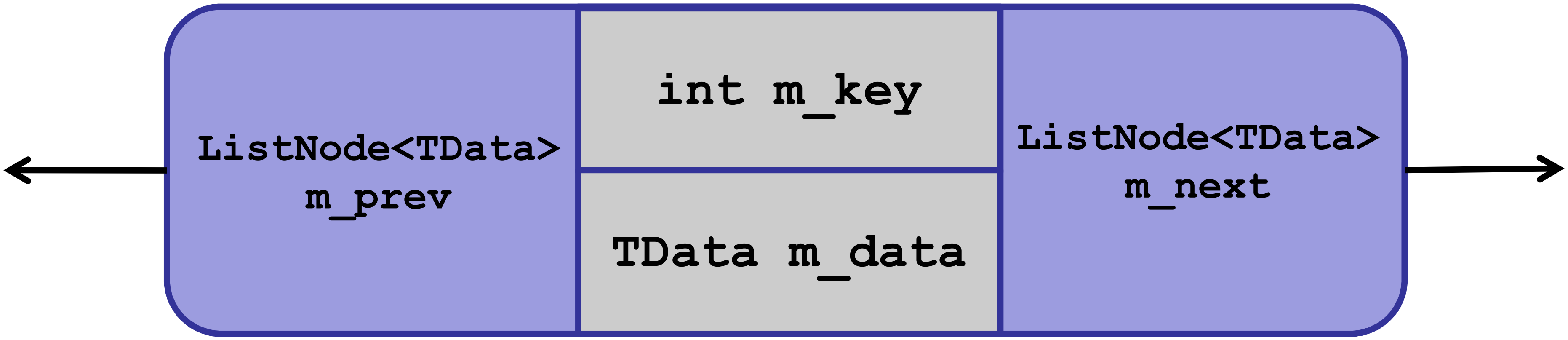
Standard Interview Question 1:

Input: Head pointer to a singly linked list.

Output: Head of reversed linked list



ListNode Implementation



Type for data

```
public class ListNode<TData> {
    int m_key;
    TData m_data;
    ListNode<TData> m_next;
    ListNode<TData> m_prev;

    ListNode(int key, TData data)
    {
        m_key = key;
        m_data = data;
        m_next = null;
        m_prev = null;
    }
```

Initialize all
in constructor

ListNode get/set methods

```
public int getKey()
{
    return m_key;
}
```

```
public TData getData()
{
    return m_data;
}
```

```
public ListNode<TData> getNext()
{
    return m_next;
}
```

```
public ListNode<TData> getPrevious()
{
    return m_prev;
}
```

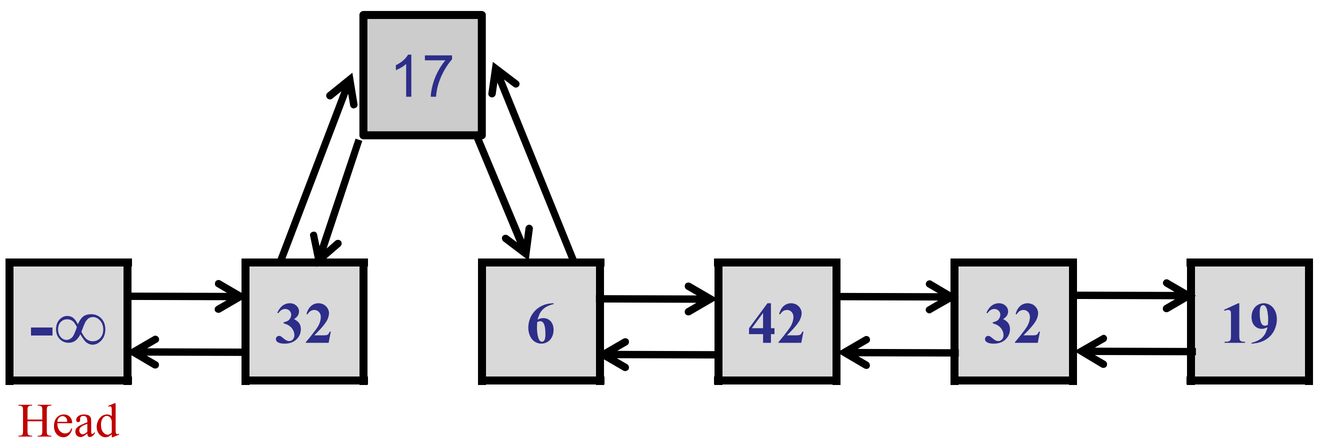
```
public void setNext(ListNode<TData> nextNode)
{
    m_next = (ListNode<TData>)nextNode;
}
```

```
public void setPrevious(ListNode<TData> prevNode)
{
    m_prev = (ListNode<TData>)prevNode;
}
```

Inserting a ListNode

```java
public void insertAfter(ListNode<TData> newNode)
{
    if (newNode == null){
        return;
    }
    newNode.setPrevious(this);
    newNode.setNext(m_next);
    if (m_next != null){
        m_next.setPrevious(newNode);
    }
    setNext(newNode);
}
```

Throw an exception?



ListNode

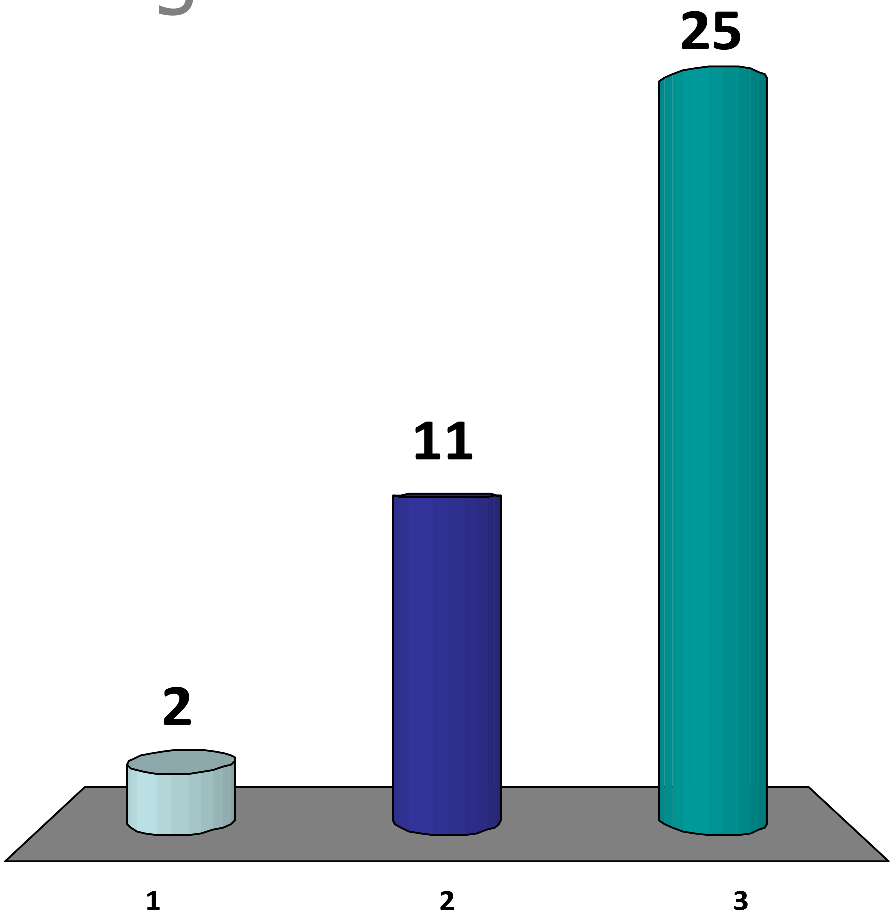
List of methods:

- `get/set …`
- `insertAfter(int key, int data)`
- `insertafter(ListNode<TData> newNode)`
- `appendList(ListNode<TData> listHead)`
- `delete()`

To use a Linked List, just use a ListNode. To search, just iterate through the ListNodes.

Are we done implementing our Linked List?

1. Yes
2. Almost: add the search methods to ListNode.
3. No, there is even more to do.



Simple List Interface

```
public interface IList<TData> {
     public int getKey(int i) throws LLException;
     public TData getData(int i) throws LLException;

     public void prepend(int key, TData data)
                                     throws LLException;
     public void append(int key, TData data)
                                     throws LLException;
     public void append(IList<TData> list)
                                     throws LLException;

     public boolean isEmpty() throws LLException;
     public int getSize() throws LLException;
}
```

IList Implementation

```
public class LinkedList<TData> implements IList<TData> {

    /* Final variable*/
    public static final int HEAD_KEY = Integer.MIN_VALUE;
    public static final int TAIL_KEY = Integer.MAX_VALUE;


    /* Class Variables */
    private ListNode<TData> m_head = null;
    private ListNode<TData> m_tail = null;
    private int m_size = 0;

    /* Constructor */
    LinkedList()
    {
        m_head = new ListNode<TData>(HEAD_KEY, null);
        m_tail = new ListNode<TData>(TAIL_KEY, null);
        m_head.insertAfter(m_tail);
        m_size = 0;
    }
```

Type for data

Sentinel / Dummy values

Head and tail of the list

Initialize all in constructor

Getting an element in the list

Throws exceptions.

```
public int getKey(int index) throws LinkedListException {
    if (index >= m_size) {
        throw new LinkedListException();
    }

    ListNode<TData> iterator = m_head.getNext();
    for (int i=0; i<index; i++)
    {
        iterator = iterator.getNext();
    }
    return iterator.getKey();
}
```

Check list size

Start at first element

Iterate to next element until (i==index)

No check for null??

Return key

Couldn’t we skip some items??

Adding an element to the list

```java
public void prepend(int key, TData data) throws LinkedListException {
    m_head.insertAfter(key,data);
    m_size++;
}
```

```java
public void append(int key, TData data) throws LinkedListException{
    m_tail.getPrevious().insertAfter(key, data);
    m_size++;
}
```

```java
public int getSize() throws LinkedListException {
    return m_size;
}
```

```java
public boolean isEmpty() throws LinkedListException{
    return (m_size == 0);
}
```

Appending a list

What type of list is it?

```
public void append(IList<TData> newList)
                              throws LinkedListException{

    // Check whether the list is a LinkedList.
    // If not, throw an exception.
    if (!(newList instanceof LinkedList)){
        throw new LinkedListException();
    }

    ListNode<TData> lastNode = m_tail.getPrevious();

    ListNode<TData> firstNewNode =
            ((LinkedList<TData>)newList).m_head.getNext();

    lastNode.appendList(firstNewNode);
    m_tail = ((LinkedList<TData>)newList).m_tail;
    m_size += ((LinkedList<TData>)newList).m_size;
}
```

Is it a linked list?

If not… exception.

We know it is a LinkedList….

Linked List

Are we done yet?? No!

Testing the Linked List

testEmptyList()

- Create an empty list.
- Check `getSize`().
- Check `isEmpty`().
- Check `getKey(0)` --- throws an exception!
- Check `getData(0)` --- throws an exception!

Testing the Linked List

testSimpleAdd()

- Create an empty list.
- Do `prepend(5, 100)`.
- Check `getKey(0), getKey(0)`.
- Do `append(20, 200)`.
- Check `getKey(0), getKey(0)`.
- Check `getKey(1), getKey(1)`.
- Check `getSize`().
- Add a few more items.
- Check getting the last element in the last.

Testing the Linked Lists

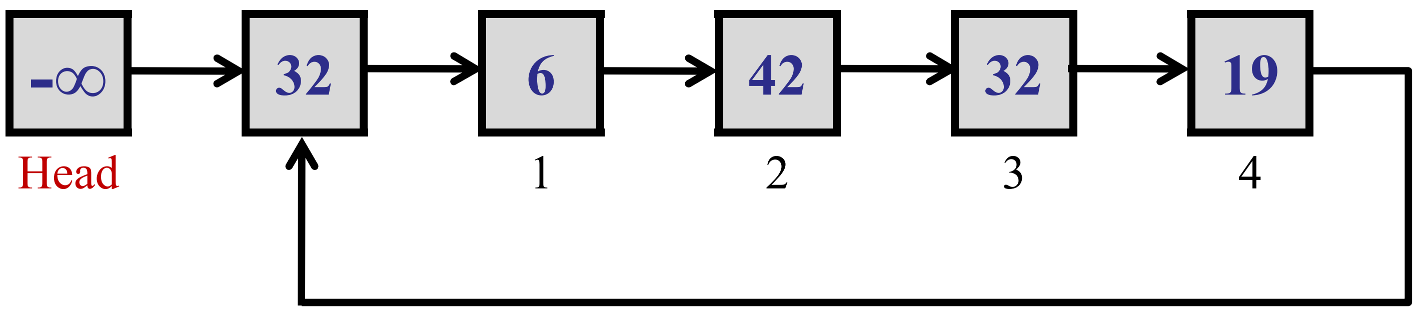
Other tests:

- Test appending a list.
- (Test deleting an element.)
- Test null/incorrect parameters.
  - What if you try to append a null list?
  - What if you try to append an empty list?
- Test with repeated keys.
- Test larger scale data.

Puzzle Break

Standard Interview Question 2:

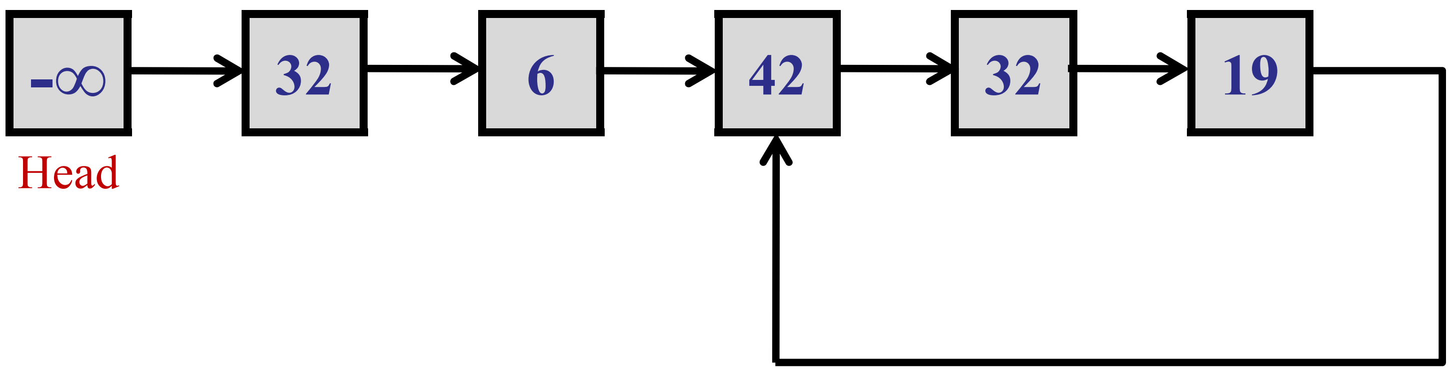
– A linked list may be circular…



Puzzle Break

Standard Interview Question 2:

– Or a linked list may contain a loop of unknown size…



Puzzle Break

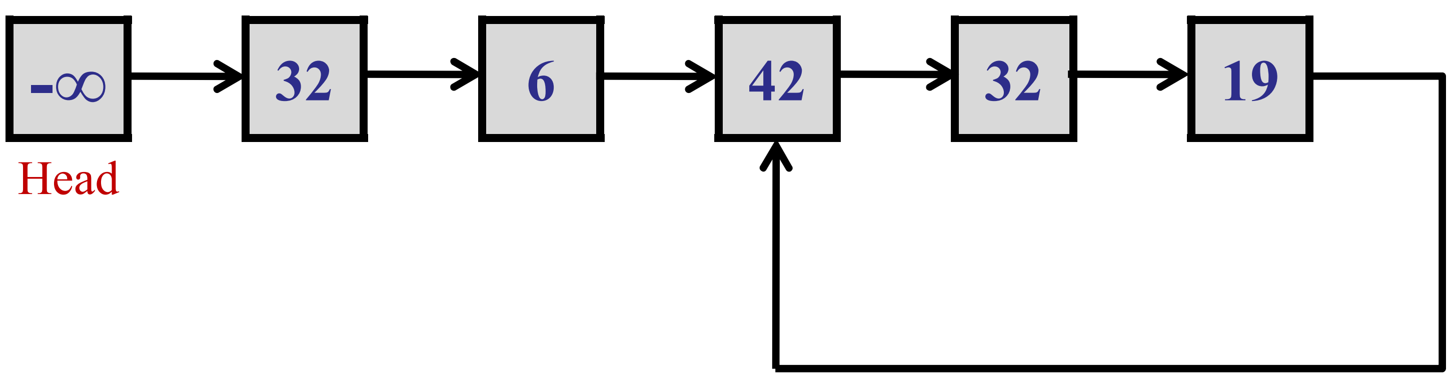
Standard Interview Question 2:

Input: Head pointer to a singly linked list of unknown size.

Output: Determine if there is a loop in the linked list.

Use only a constant amount of extra space.

What if you can't modify the original list?



SkipLists

```java
public interface ISearchTree<TKey, TData> {
    void insert(TKey key, TData data);
    boolean search(TKey key);
    TData getData(TKey key);
}
```

SkipLists

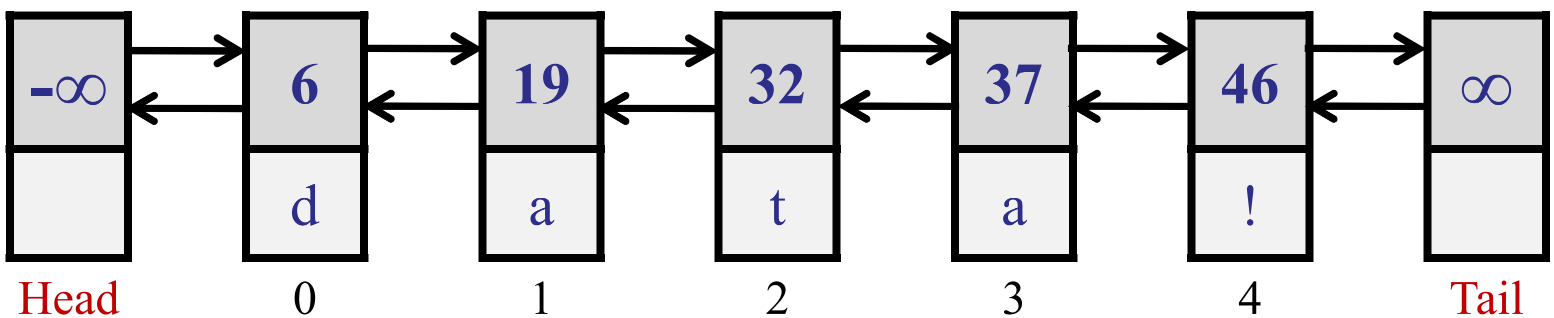
Simple search structure:

- Insert key/data pairs.
- Efficient search.
- Simple delete.

```
public interface ISearchTree<TKey, TData> {

    void insert(TKey key, TData data);

    boolean search(TKey key);

    TData getData(TKey key);

    void delete(TKey key);
}
```

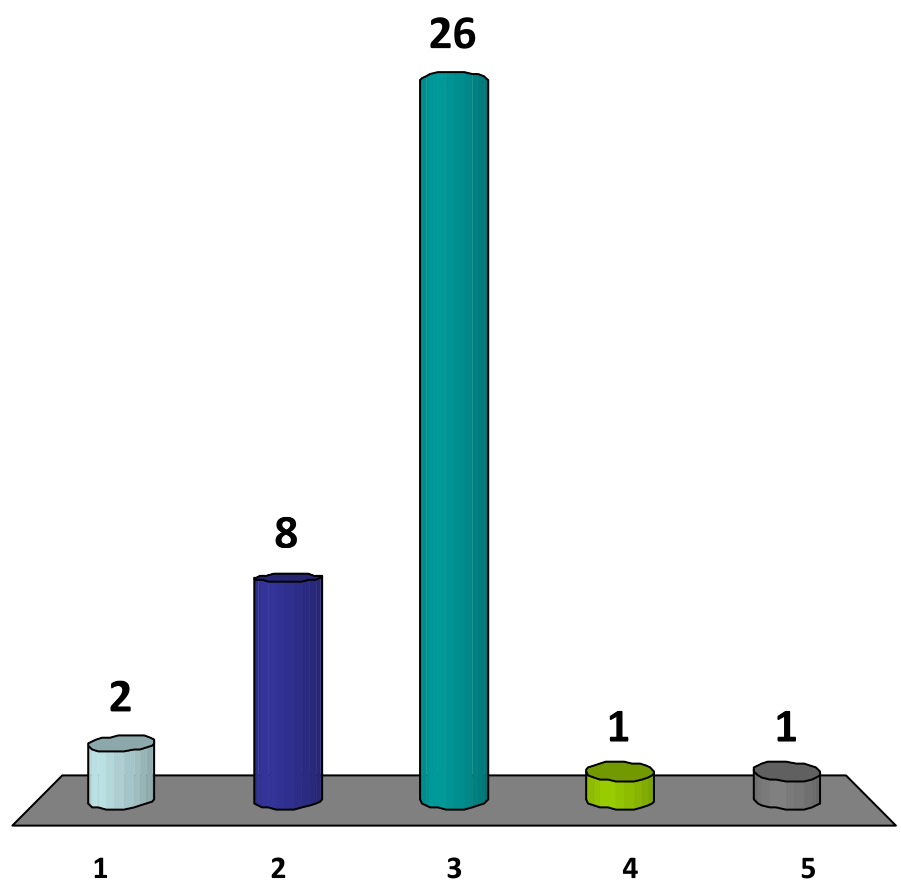
Start simple…

Store keys in a sorted linked list:



How long does it take to search?

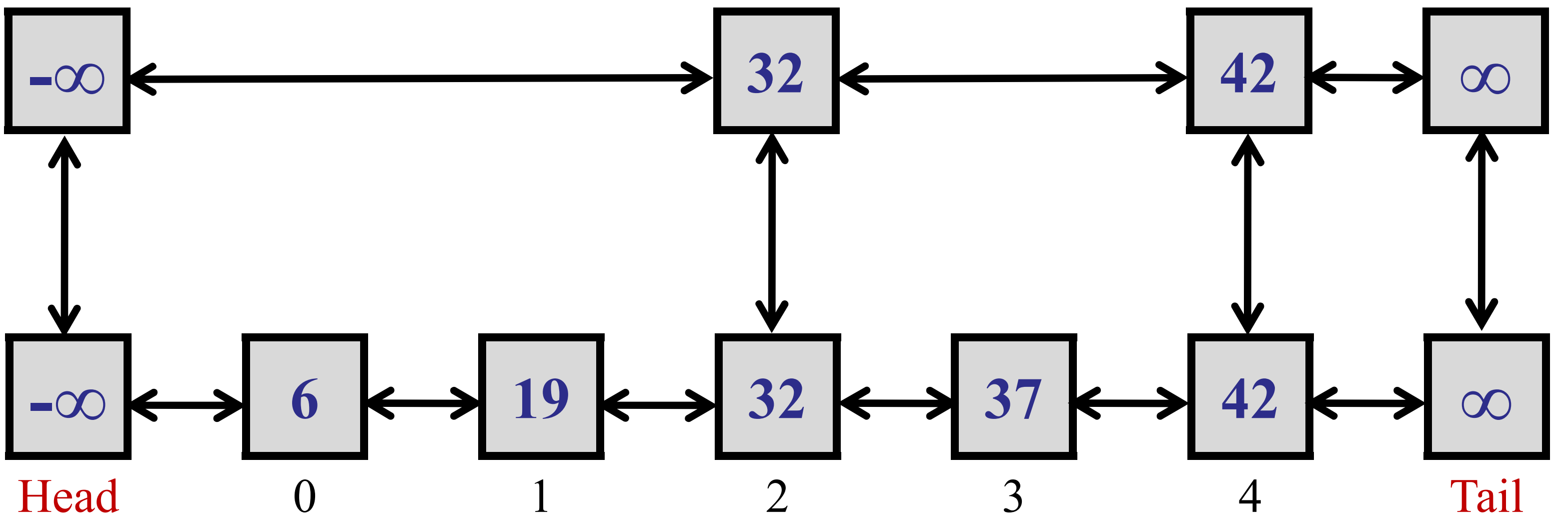
1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$



What if…

What if we use two lists?

- Express train
- Local train



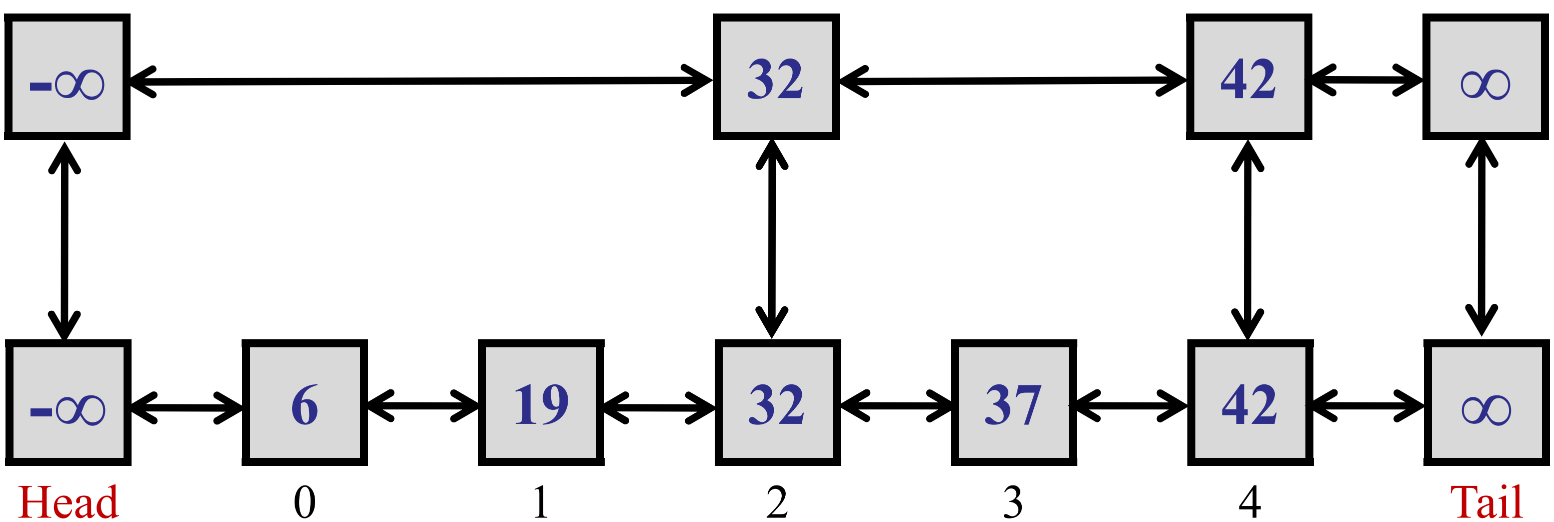
`search(37)` takes only 3 steps!

What if…

Calculation:

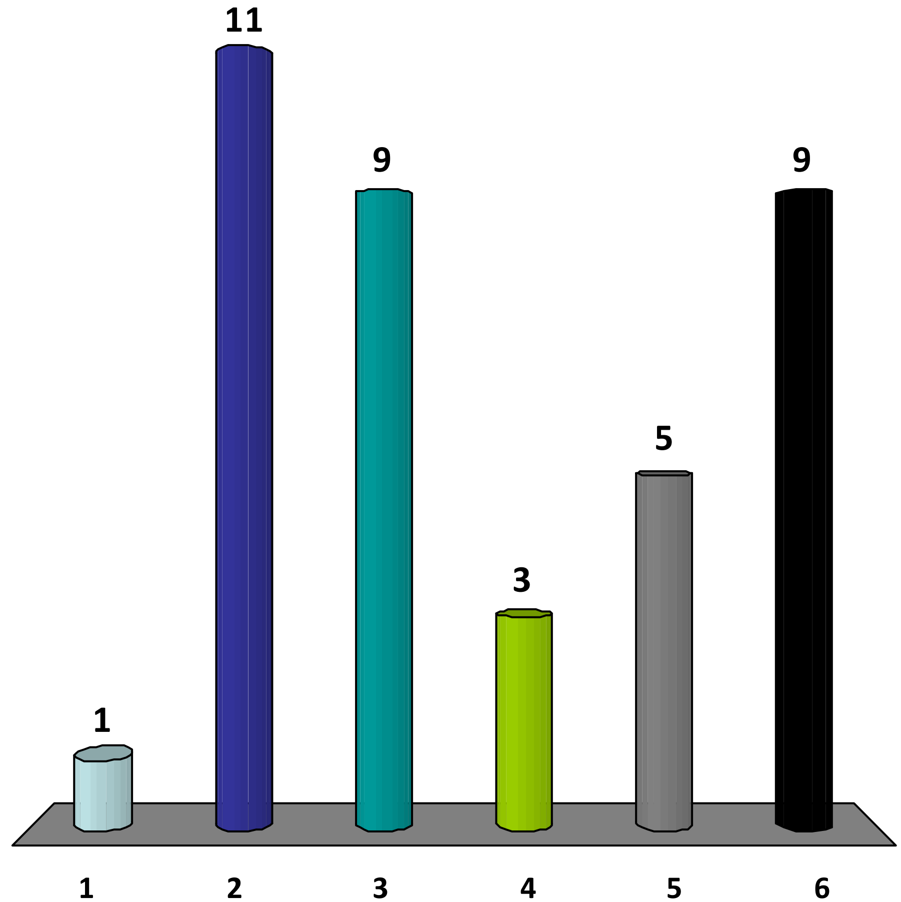
- If the “express” list skips 5 elements per “stop”, then search takes at most:

$n/5 + 5$ steps



In a two-list SkipList, how many elements should the express list skip per hop?

1. O(1)
2. log(n)
3. √n
4. n/√n
5. n/log(n)
6. Something else.

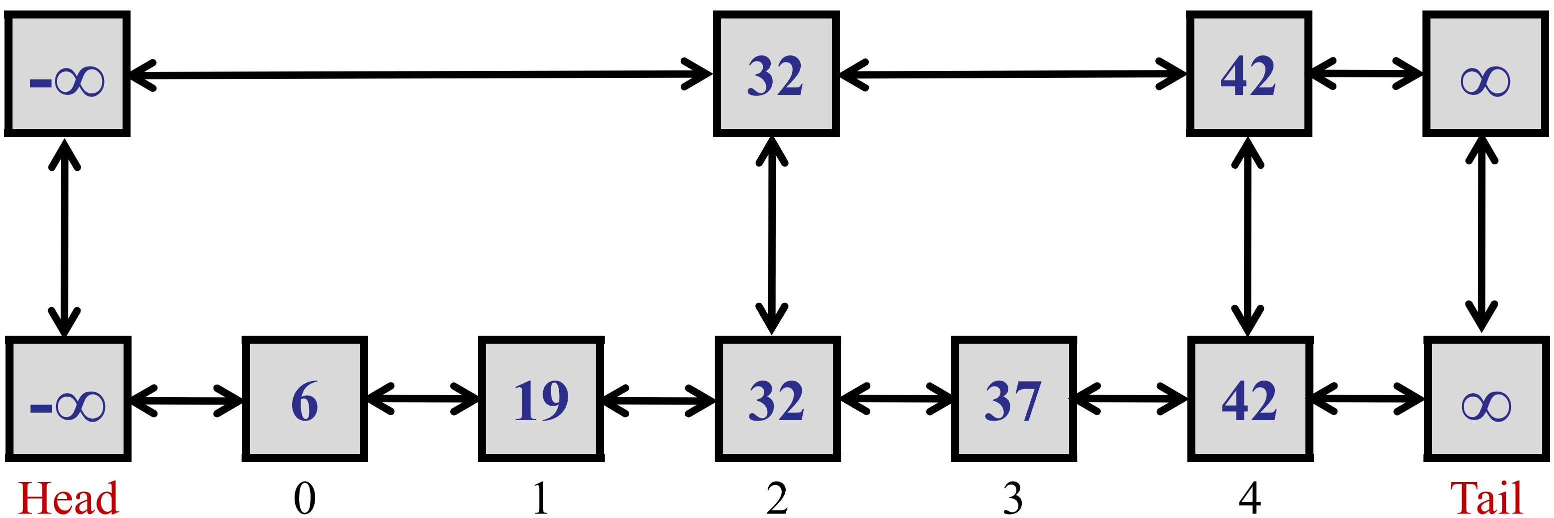


What if…

Calling:

- If the “express” list skips sqrt(n) elements per “stop”, then search takes at most:

$$\frac{n}{\sqrt{n}} + \sqrt{n} = 2\sqrt{n} = O(\sqrt{n})$$



Why stop at two?

Add more lists:

– Two lists: Cost = $2\sqrt{n}$

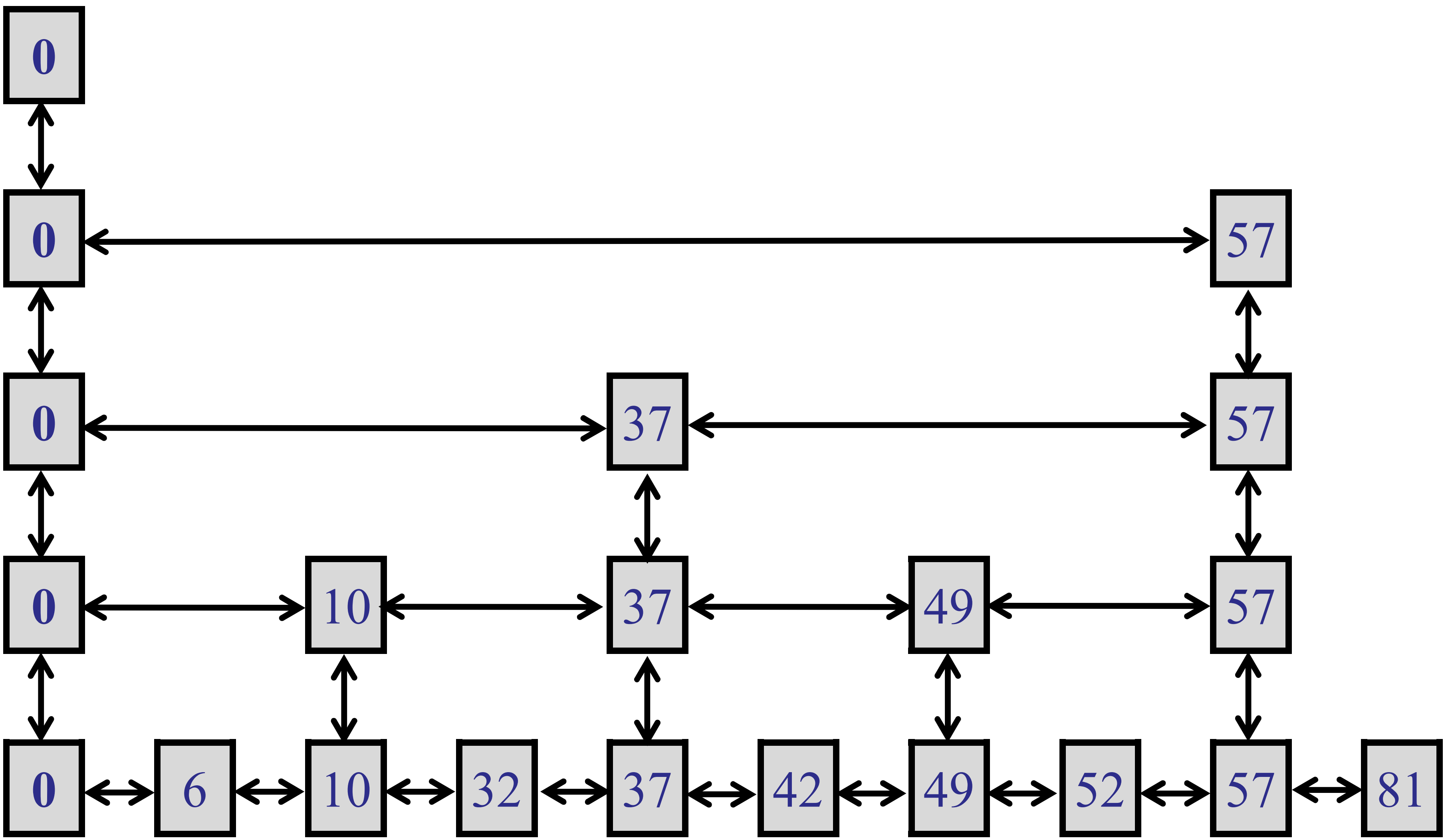
– Three lists: Cost = $3\sqrt[3]{n}$

– k lists: Cost = $k\sqrt[k]{n}$

– $\log(n)$ lists: Cost = $\log(n)\sqrt[\log(n)]{n} = \log(n)n^{1/\log(n)}$

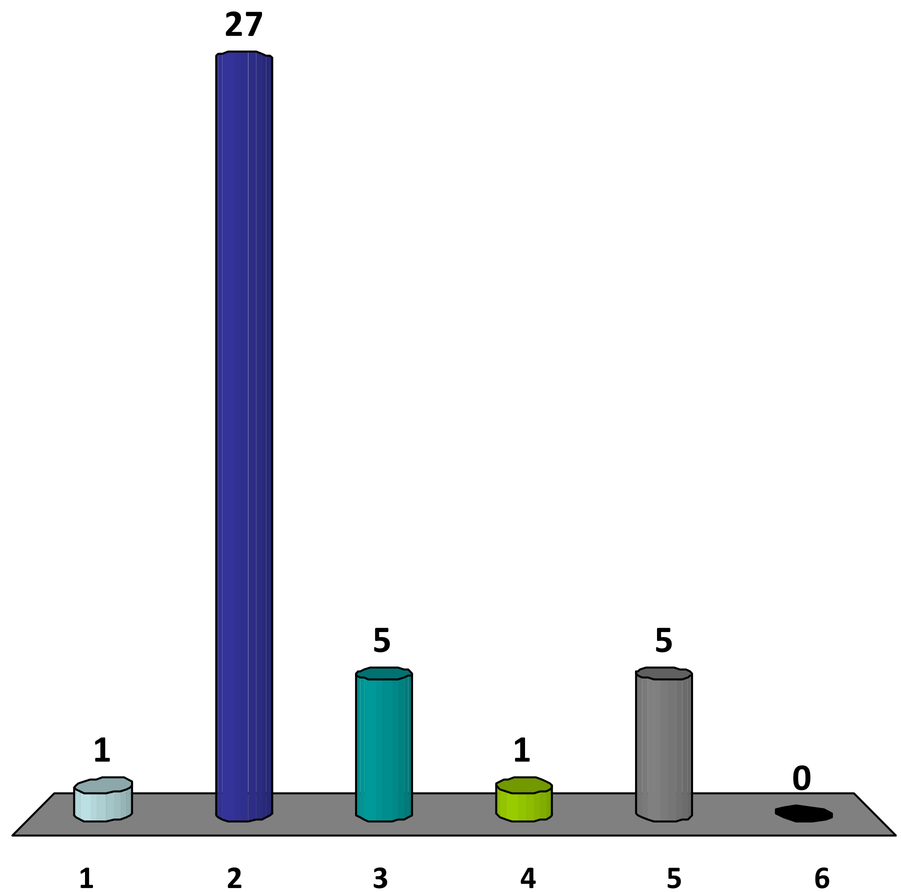
$$= 2\log(\mathrm{n})$$

Another way to think about it…

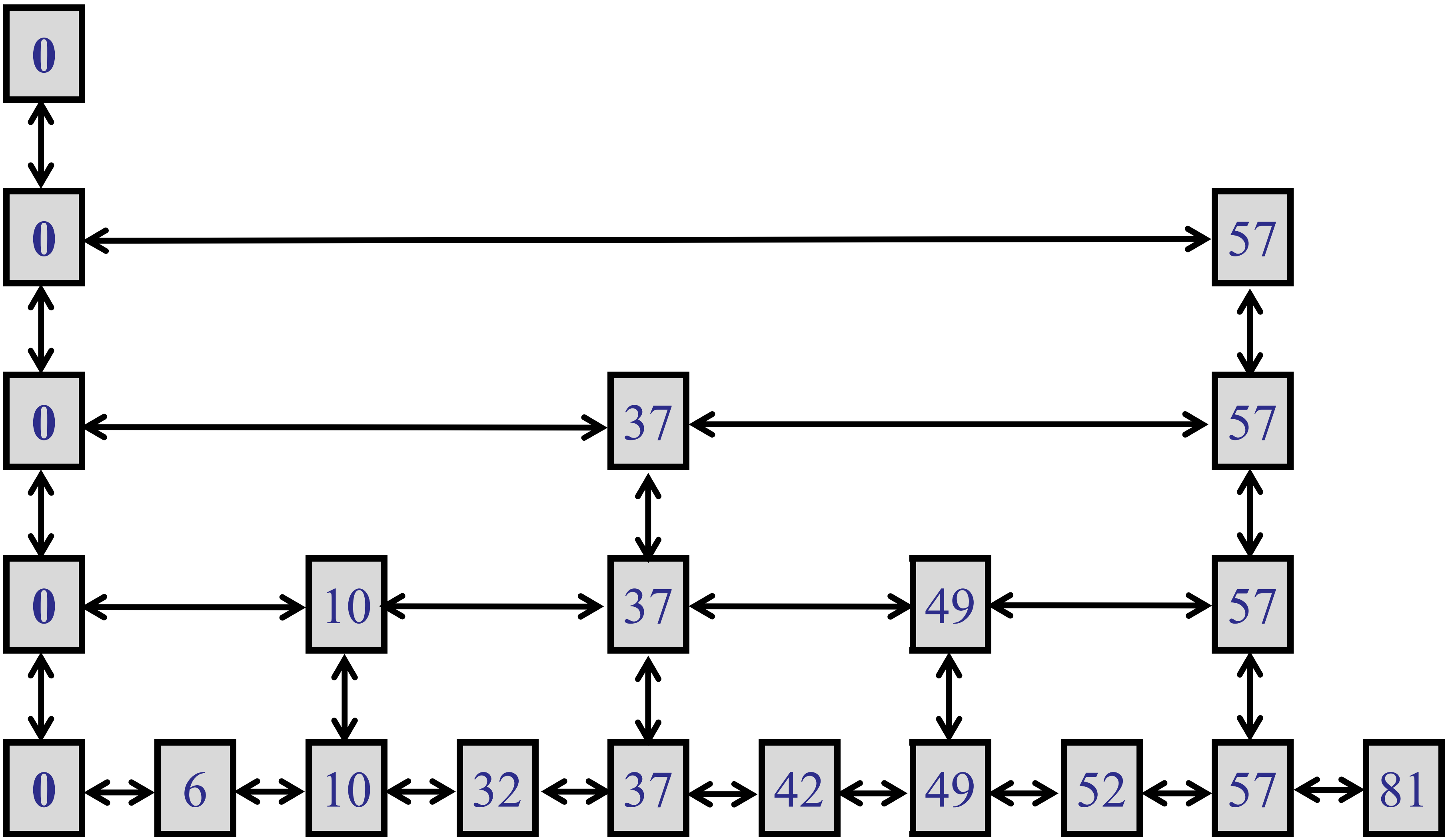


How many levels in this SkipList?

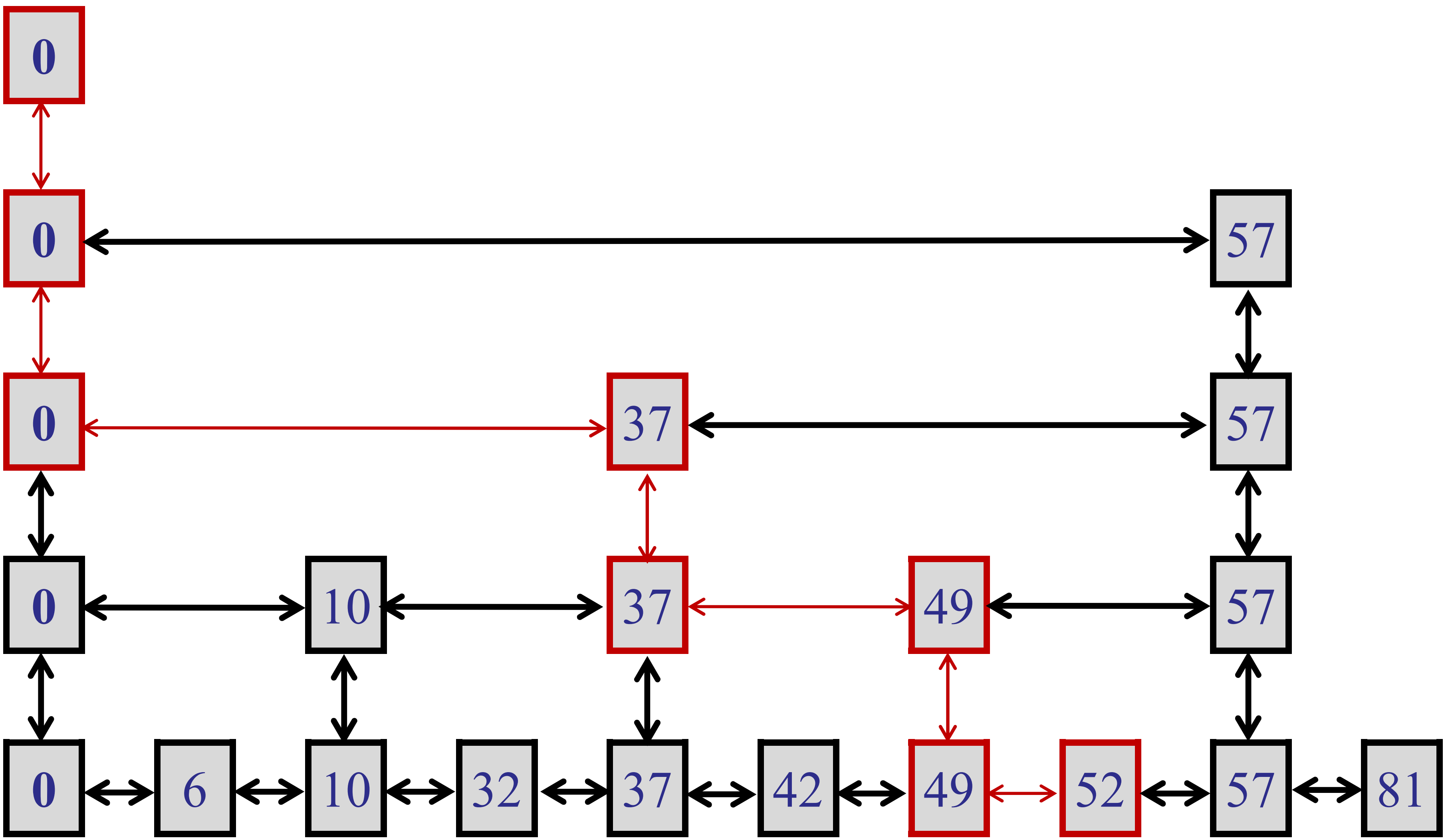
1. O(1)
2. log(n)
3. 2log(n)
4. $\log^2(n)$
5. $\sqrt{n}$
6. None of the above.



Another way to think about it…



Example: `search(52)`



Insertions

To insert a new element:

1. Add element to bottom list.

   (Invariant: bottom list contains every element.)

2. Add element to some other lists to maintain balance.

   Goal: about half of elements at level j get promoted to level $j+1$.

Insertions

Key idea: flip a coin

1. $k = 0$;
2. while (!*done*) {
3. Insert element into level k list.
4. Flip a fair coin:
5. with probability ½: *done* = true;
6. with probability ½: $k = k+1$;
7. }

Insertions

To insert a new element:

1. Add element to bottom list.

   (Invariant: bottom list contains every element.)

2. Flip coins to decide how many levels to promote.

– On average: **Level 0: *n***

Level 1: *n*/2

Level 2: *n*/4

…

Level log(*n*): O(1)

SkipList

Randomized process:

- Not as balanced as the last example.
- Good, on average.
- Really good, *almost always*.

- As usual, easy to implement, harder to analyze.

SkipListNode Implementation

Use linked list implementation.

```
public class SkipListNode<TData> extends ListNode<TData> {

    SkipListNode<TData> m_up;
    SkipListNode<TData> m_down;

    SkipListNode(int key, TData data)
    {
        super(key, data);
        m_up = null;
        m_down = null;
    }
```

Add up/down.

Call parent constructor.

Initialize new member variables.

SkipListNode Implementation

get/set methods:

- `getUp()`
- `getDown()`
- `setUp(SkipListNode<TData> newUp)`
- `setDown(SkipListNode<TData> newDown)`

SkipListNode Implementation

A few other methods:

- For example, `searchUp()` walks backward until it finds an “up” pointer.

```
public SkipListNode<TData> searchUp()
{
    SkipListNode<TData> upNode = null;
    SkipListNode<TData> iterator = this;

    while (upNode == null && iterator != null)
    {
        upNode = iterator.getUp();
        iterator = iterator.getPrevious();
    }

    return upNode;
}
```

SkipList Implementation

Implement ISearchTree where key is an Integer.

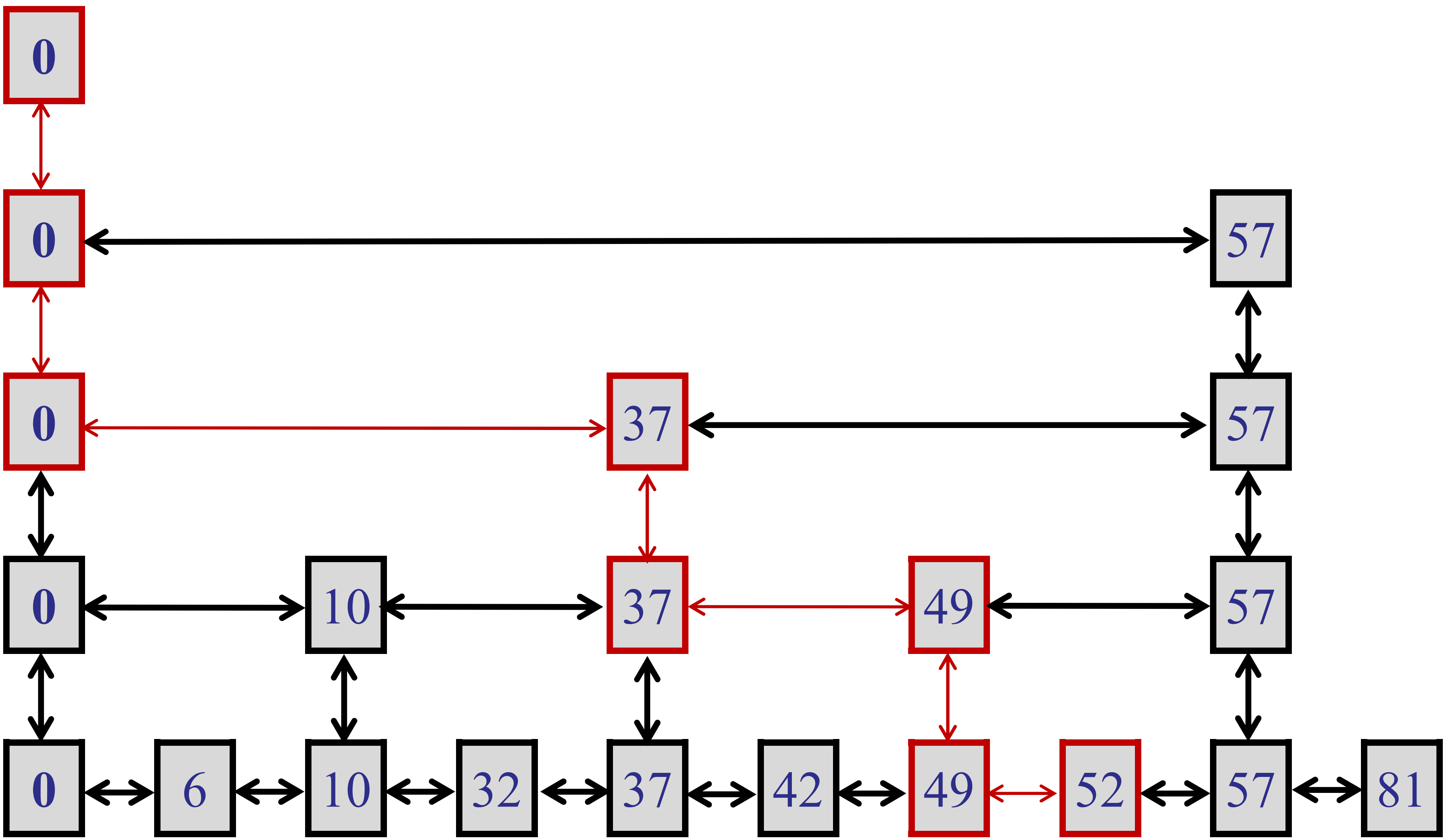
```
public class SkipList<TData> implements ISearchTree<Integer, TData>{

    /* Final variable*/
    public static final int HEAD_KEY = Integer.MIN_VALUE;

    /* Class Variables */
    SkipListNode<TData> m_ListHead;
    SkipListNode<TData> m_AllKeyLinkedList;

    SkipList()
    {
        m_ListHead = new SkipListNode<TData>(HEAD_KEY, null);
        m_AllKeyLinkedList = m_ListHead;
    }
```

Dummy value for head.

Head of top and bottom lists.

Initialize empty list.
Start with only one list.

Example: `search(52)`



SkipList Implementation

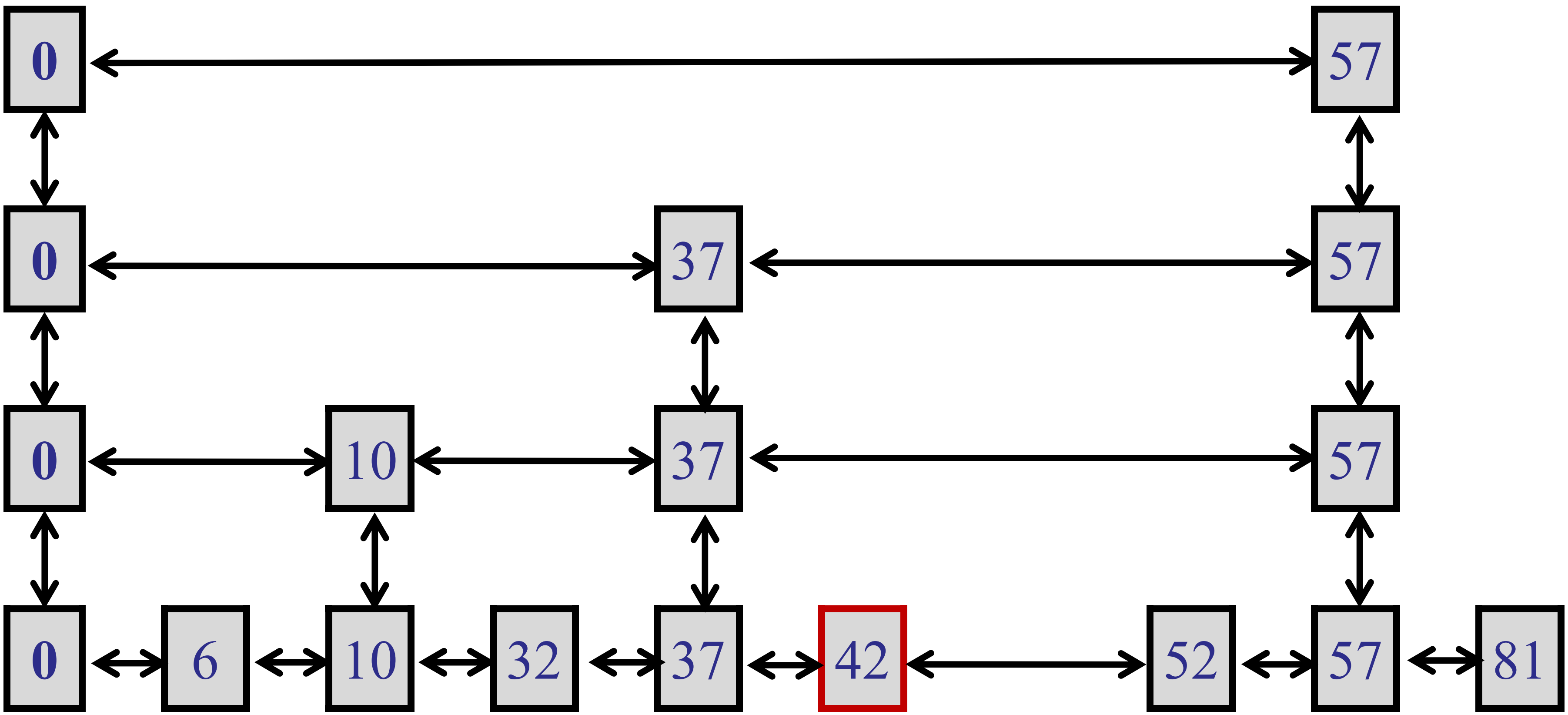
Start at the top list.

```
public SkipListNode<TData> searchSuccessorNode(Integer key)
{
    SkipListNode<TData> iterator = m_ListHead;

    while (iterator != null && (iterator.getKey() <= key)) {
        SkipListNode<TData> nextNode = iterator.getNext();
        SkipListNode<TData> downNode = iterator.getDown();

        if ((nextNode != null) && (nextNode.getKey() <= key)){
                iterator = nextNode;
        }
        else if (downNode != null){
            iterator = downNode;
        }
        else{
            return iterator;
        }
    }
    return null;
}
```

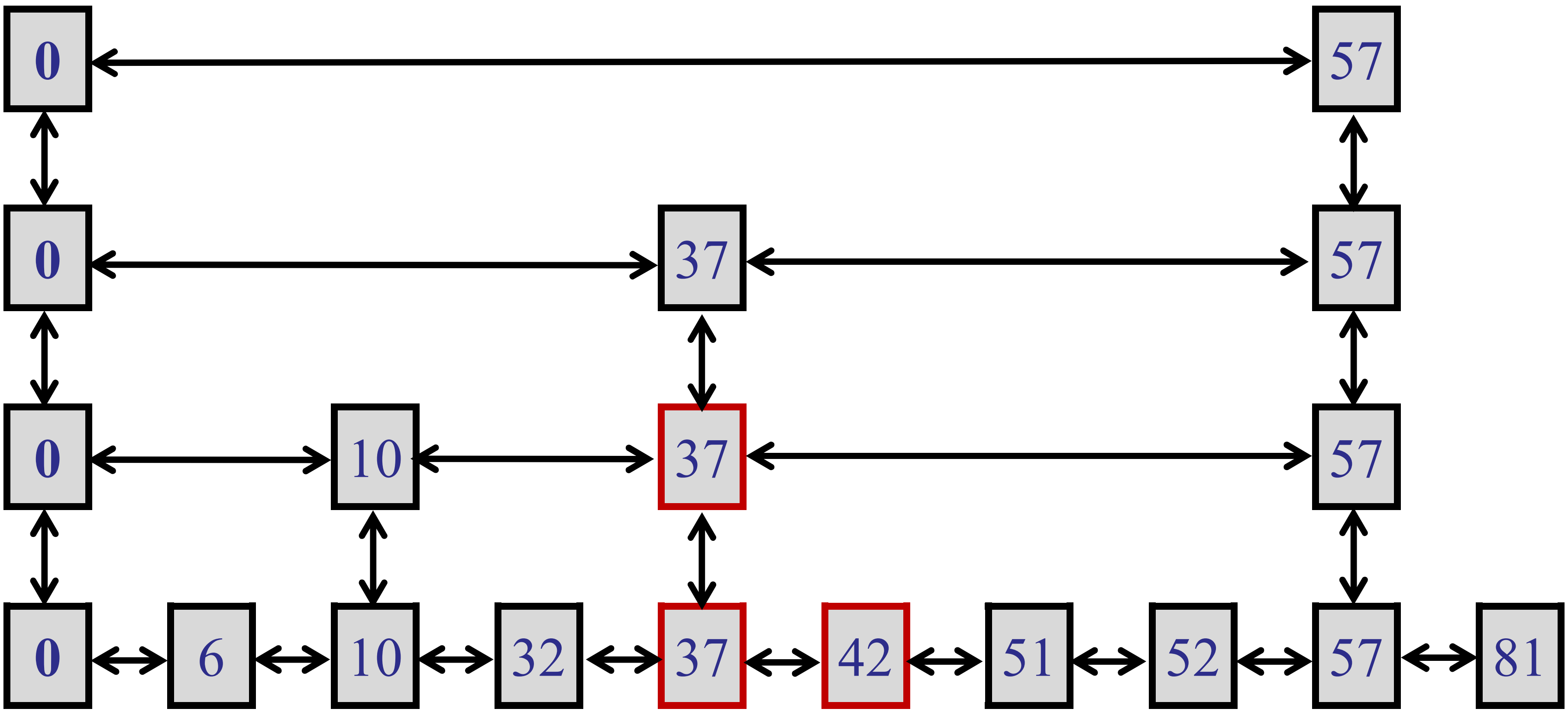
Go right if you can.
Otherwise, go down.

If you can't go any farther,
return the iterator.

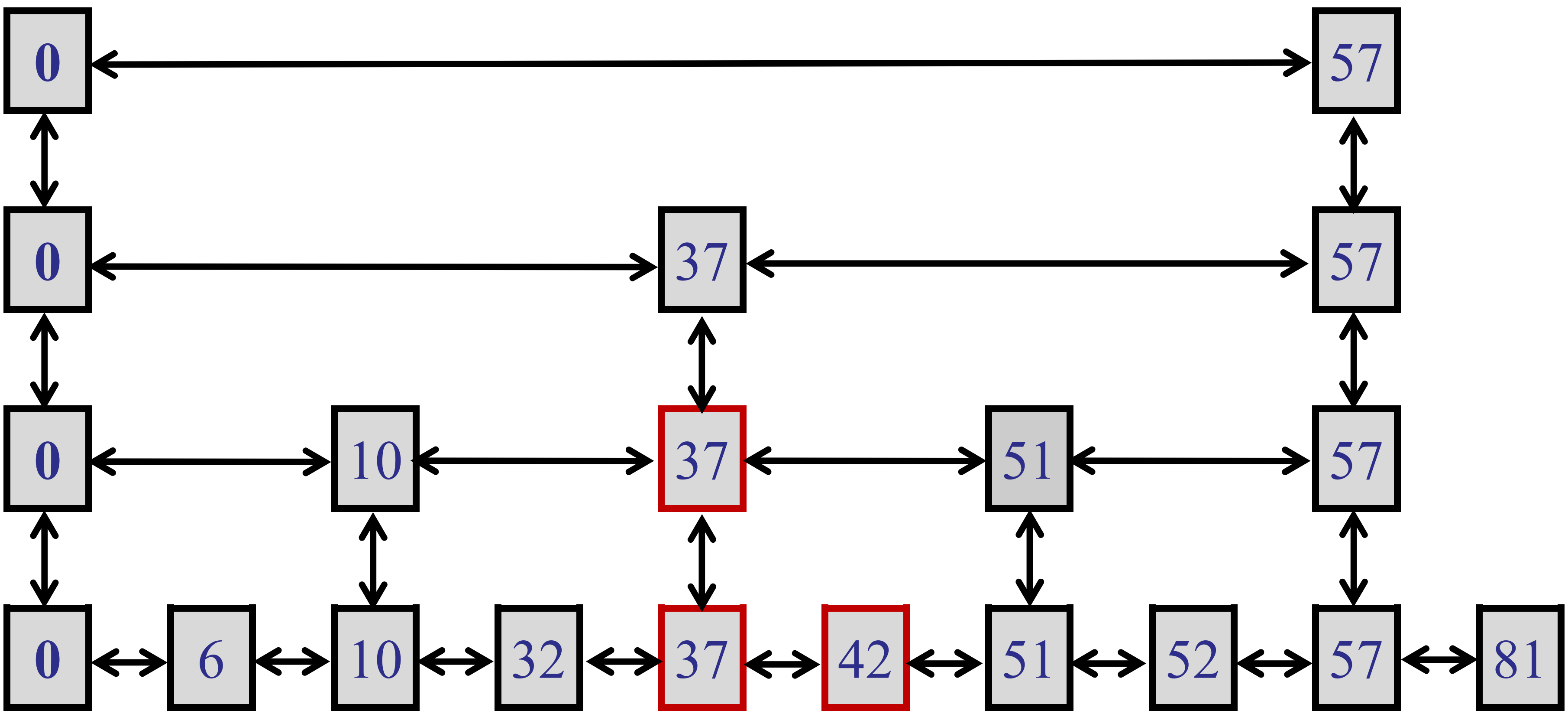
Example: `insert(51)`



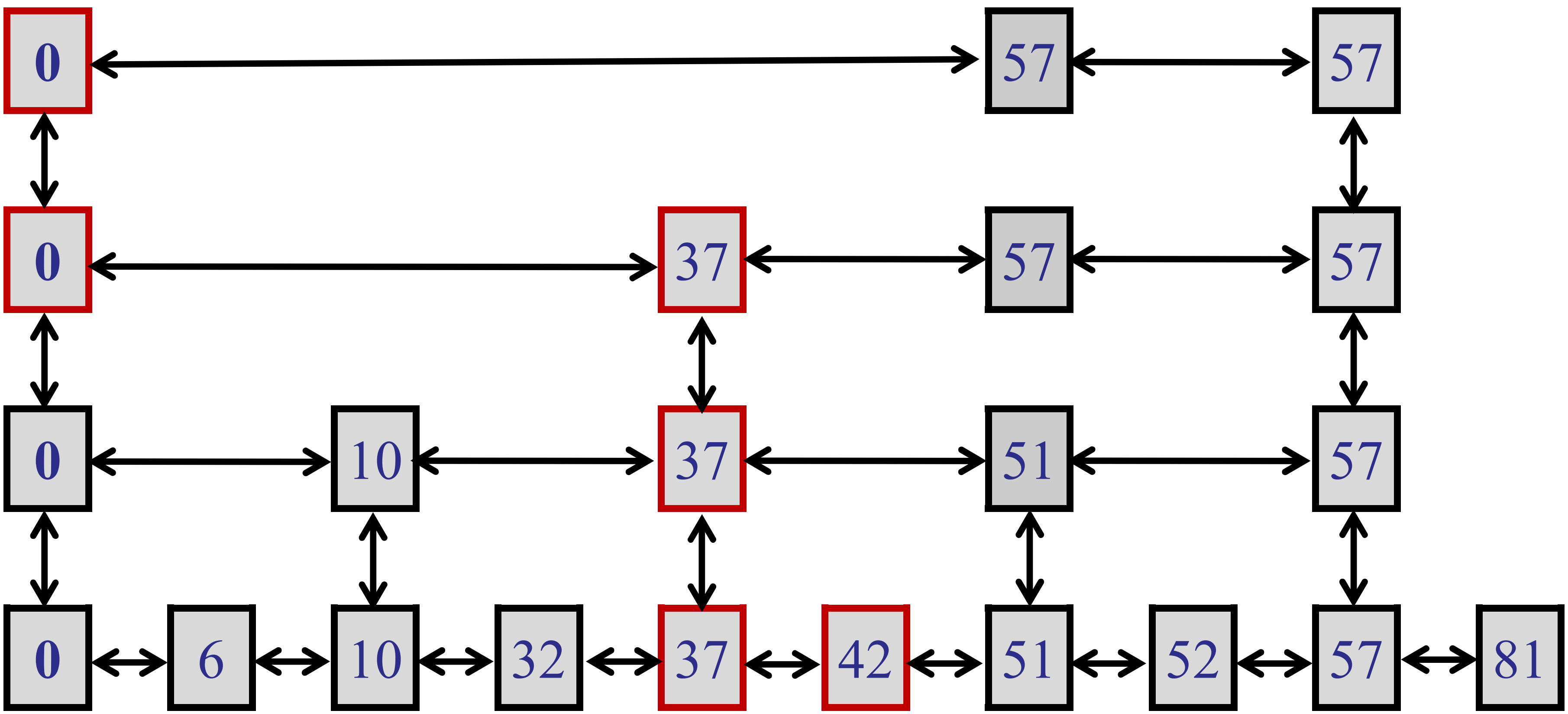
Example: `insert(51)`



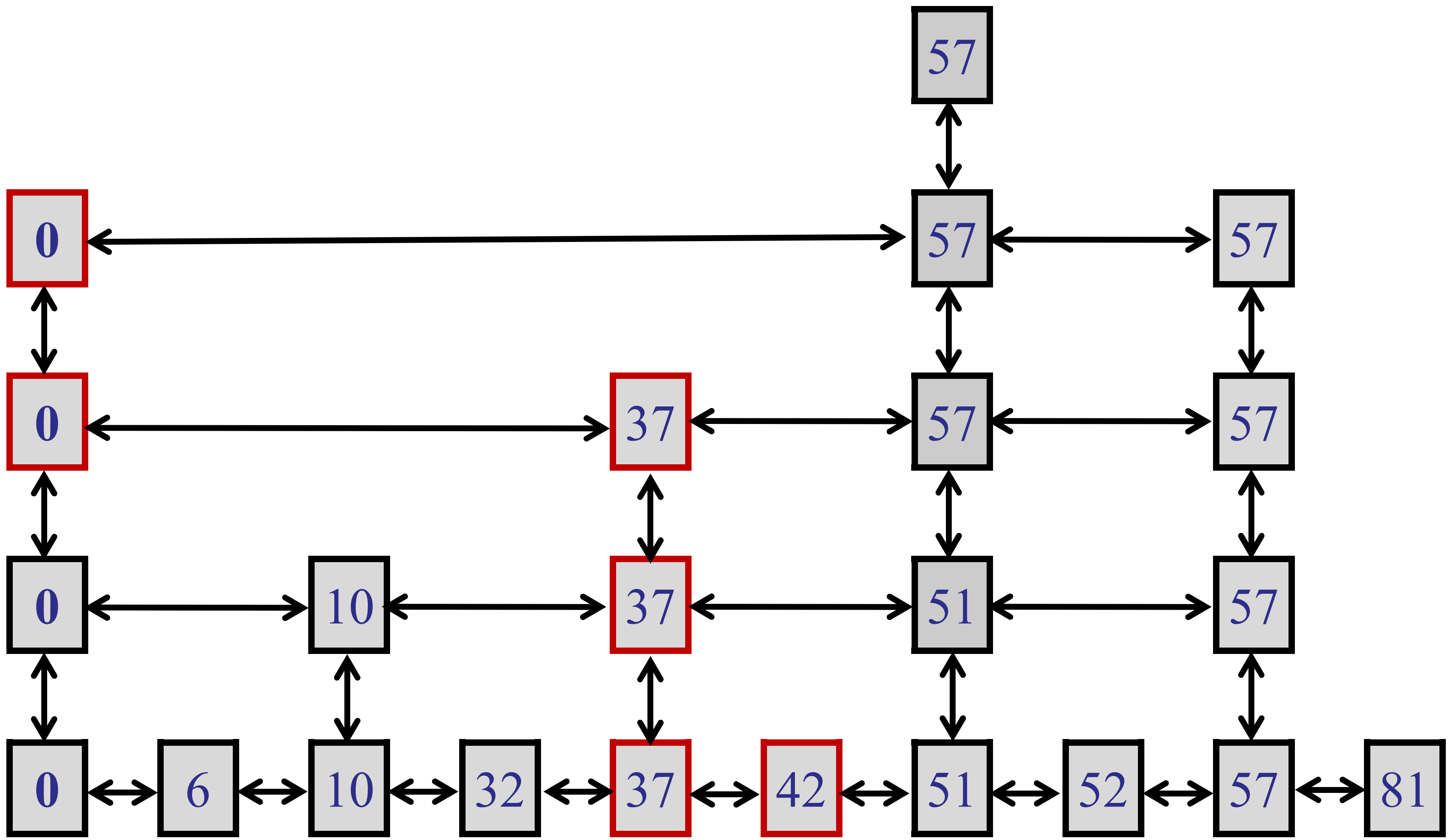
Example: `insert(51)`



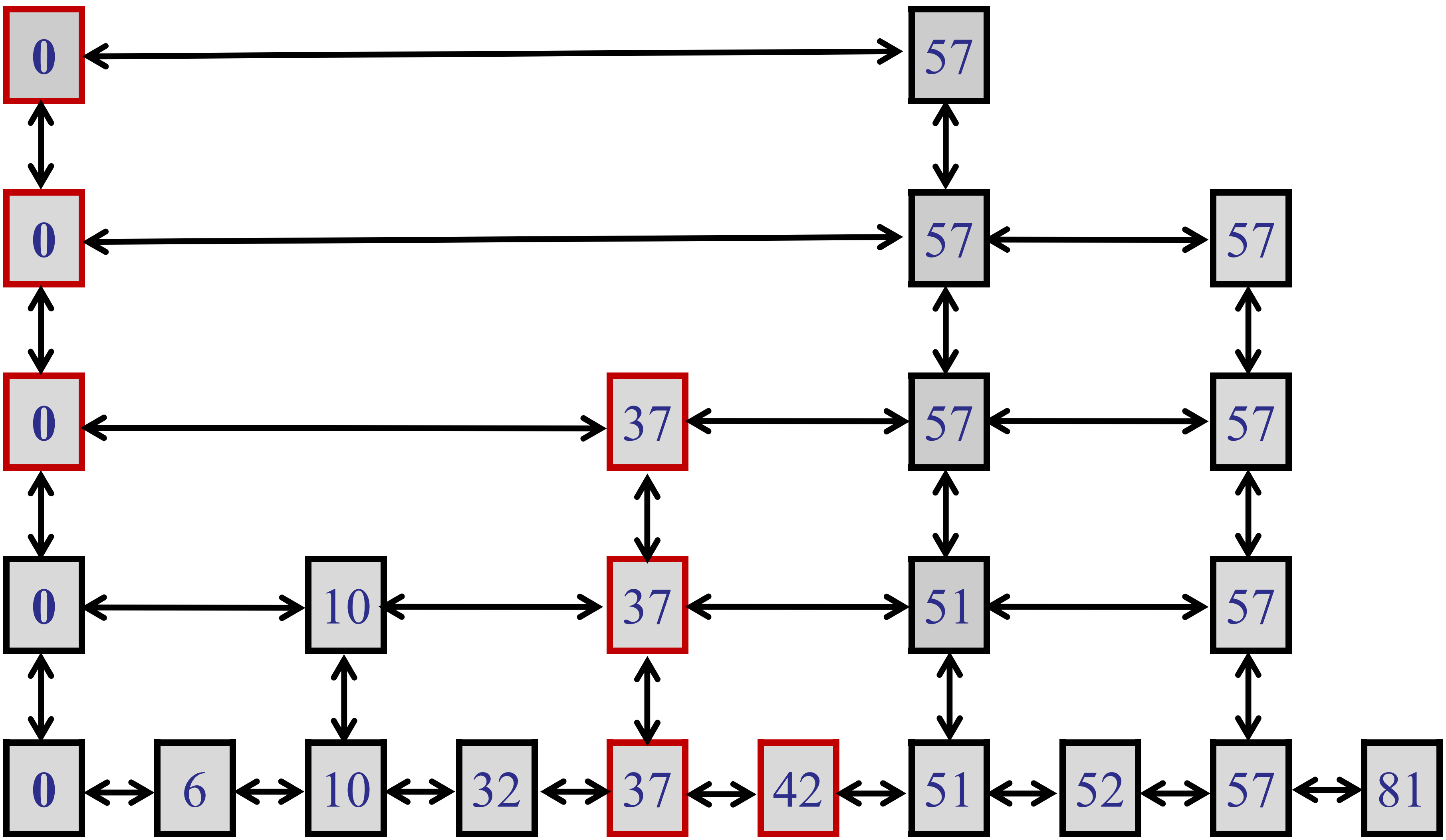
Example: `insert(51)`



Example: `insert(51)`



Example: `insert(51)`



SkipList Implementation

Create new SkipListNode to insert.

```
public void insert(Integer key, TData data) {

    SkipListNode<TData> newNode =
                new SkipListNode<TData>(key, data);

    SkipListNode<TData> insertNode = searchSuccessorNode(key);
    Random generator = new Random();
```

Search for the successor node.

Java class for generating random numbers.

SkipList Implementation

```
while (insertNode != null) {
    insertNode.insertAfter(newNode);
    boolean goUp = generator.nextBoolean();
    if (goUp) {
        insertNode = newNode.searchUp();
        if (insertNode == null) {
            insertNode = new SkipListNode<TData>(Integer.MIN_VALUE, null);
            insertNode.setDown(m_ListHead);
            m_ListHead.setUp(insertNode);
            m_ListHead = insertNode;
        }
        SkipListNode<TData> nextNewNode =
                        new SkipListNode<TData>(key, data);
        nextNewNode.setDown(newNode);
        newNode.setUp(nextNewNode);
        newNode = nextNewNode;
    }
    else insertNode = null;
}
```

Insert node in list.

Flip coin.

If heads, go up.

If no up list, add one.

Create new node for next list up.

If tails, done.

SkipList Implementation

Find the node to delete.

```
public void delete(Integer key) {
    SkipListNode<TData> node = searchSuccessorNode(key);
    if (key != node.getKey()){
        return;
    }
    SkipListNode<TData> next = node;

    while (node != null){
        next = node.getUp();
        node.delete();
        node = next;
    }
}
```

If we don't find the right key, return.

Delete the node at every level.

SkipList Implement

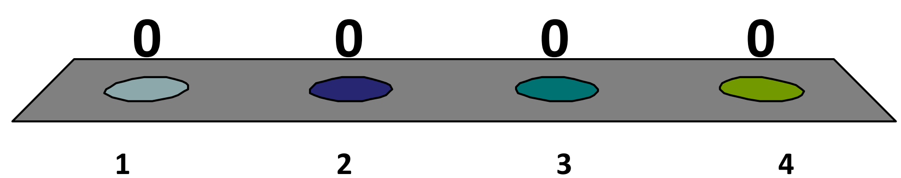
Done!

No, still need to write tests…

- Empty list
- Insert and search
- Balance (i.e., nodes per level)
- Delete

Was all the Java code in lecture useful?

1. Yes!
2. A little.
3. Not really.
4. No, it was boring!

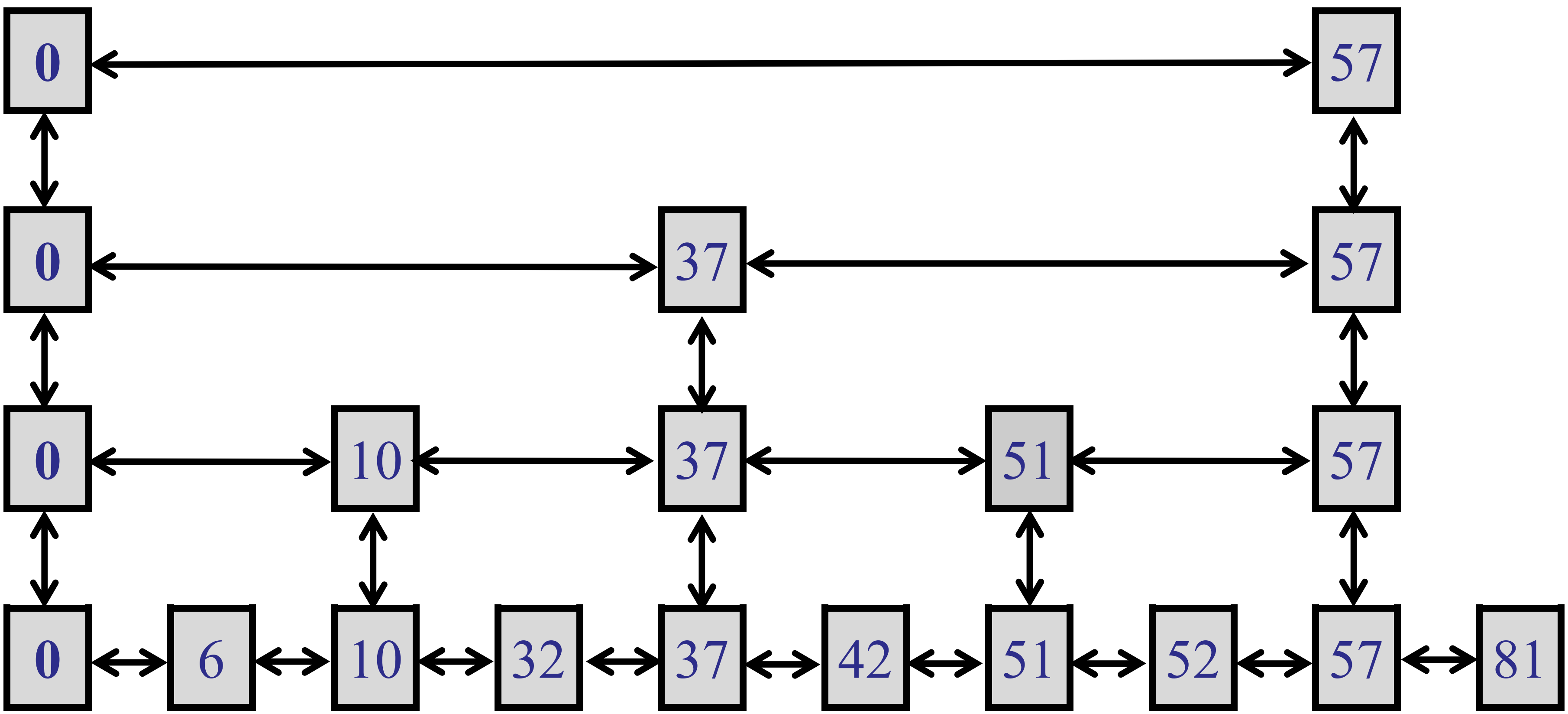


0 of 60

SkipList Extras

Very efficient for range queries.

- "Return all the elements between 30 and 60."



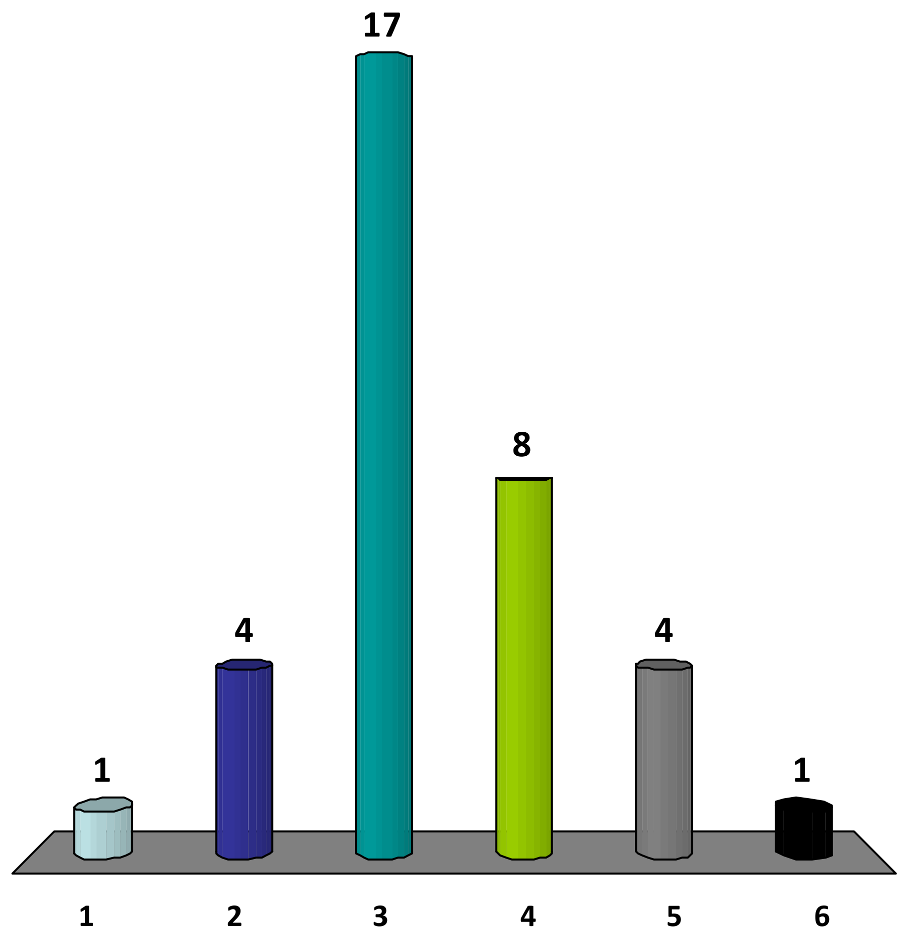
SkipList Extras

Order Statistics

- "Find me the 17th smallest element."

How fast can you find the k^{th} element in an *unsorted* list?

1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. O(n^2)
6. I don't remember.



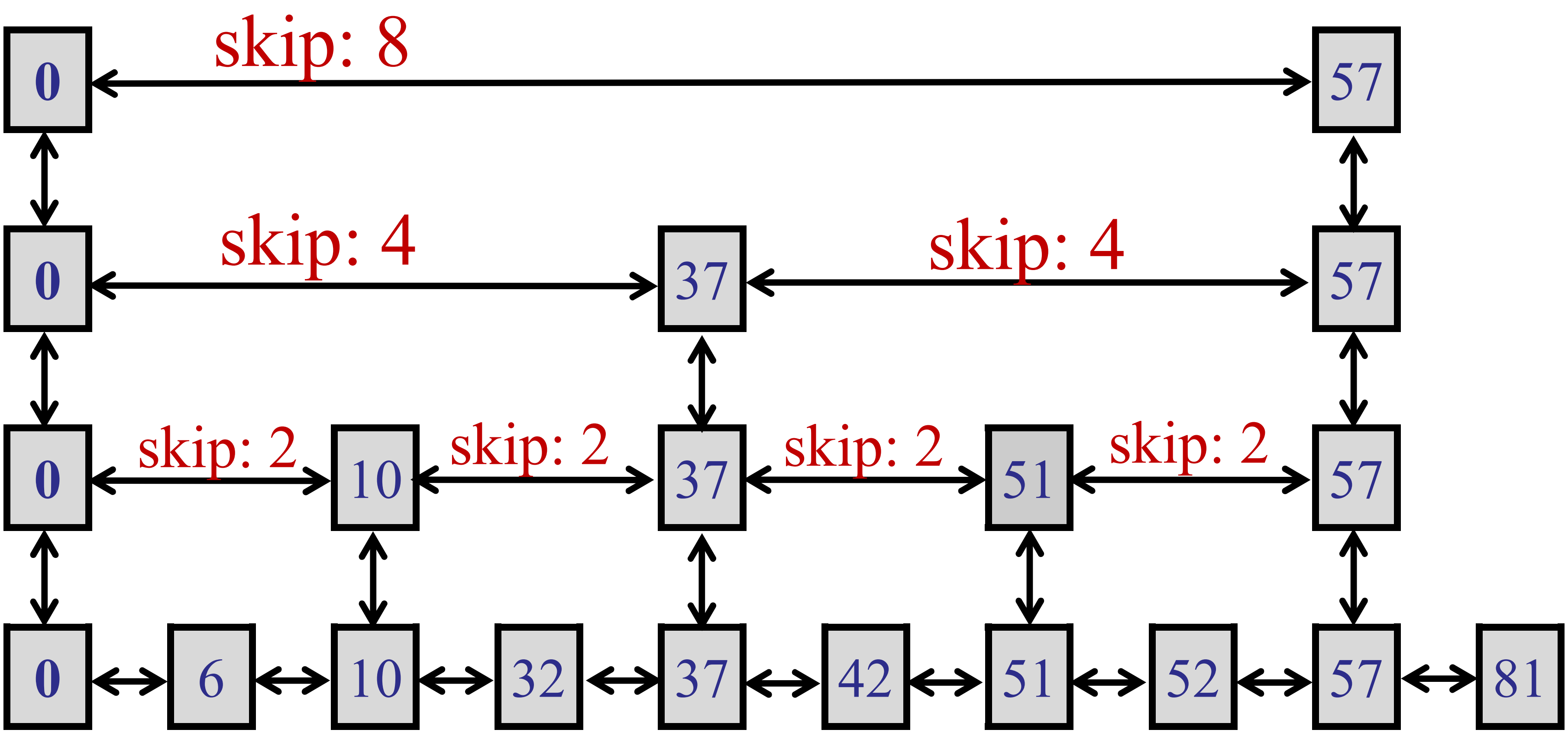
SkipList Extras

Order Statistics

- "Find me the 17th smallest element."
- Solutions:
  - Unsorted array: O(n)
  - Sorted array: O(1)
  - SkipList… (or any BST)

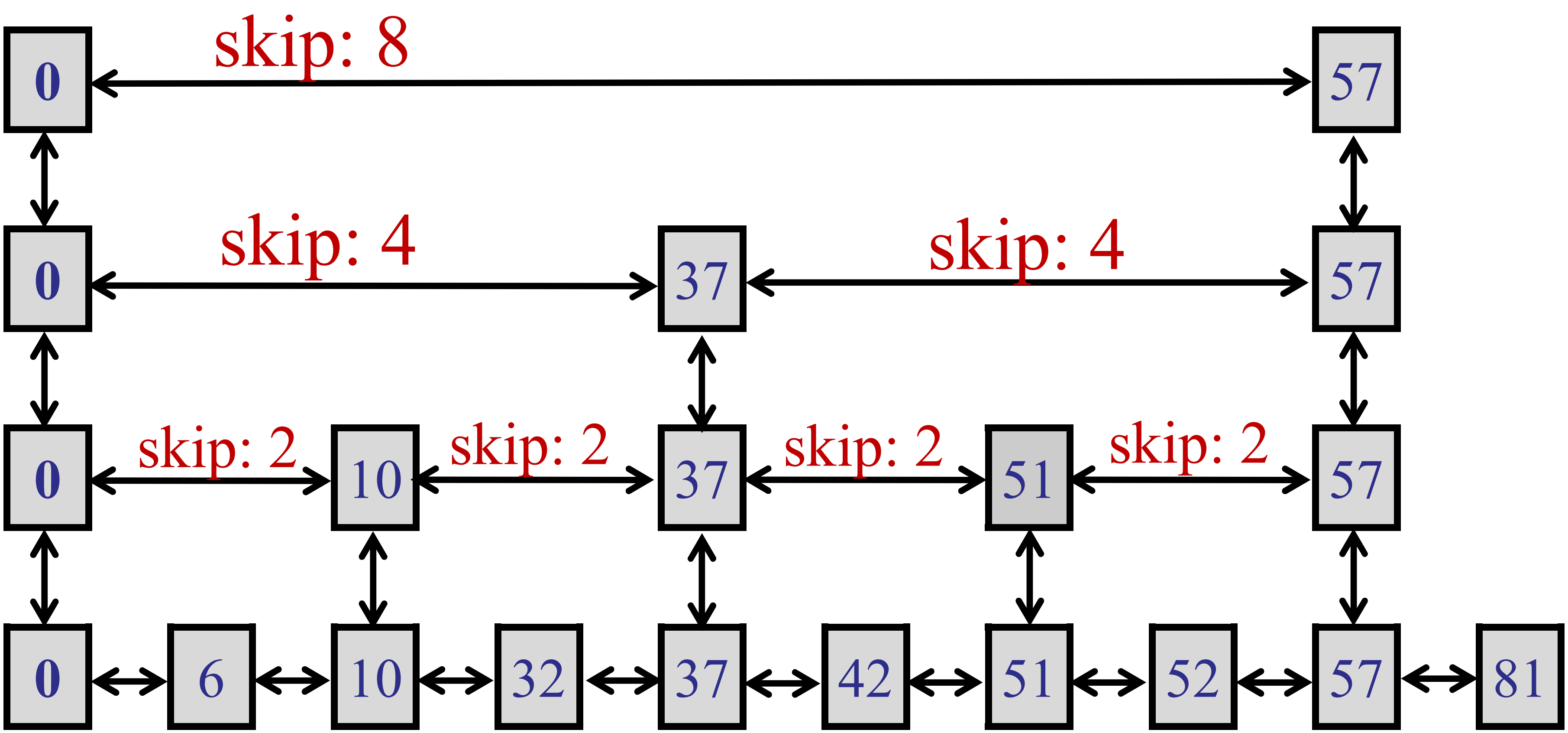
SkipList Extras

Store the "skip count" at each node.



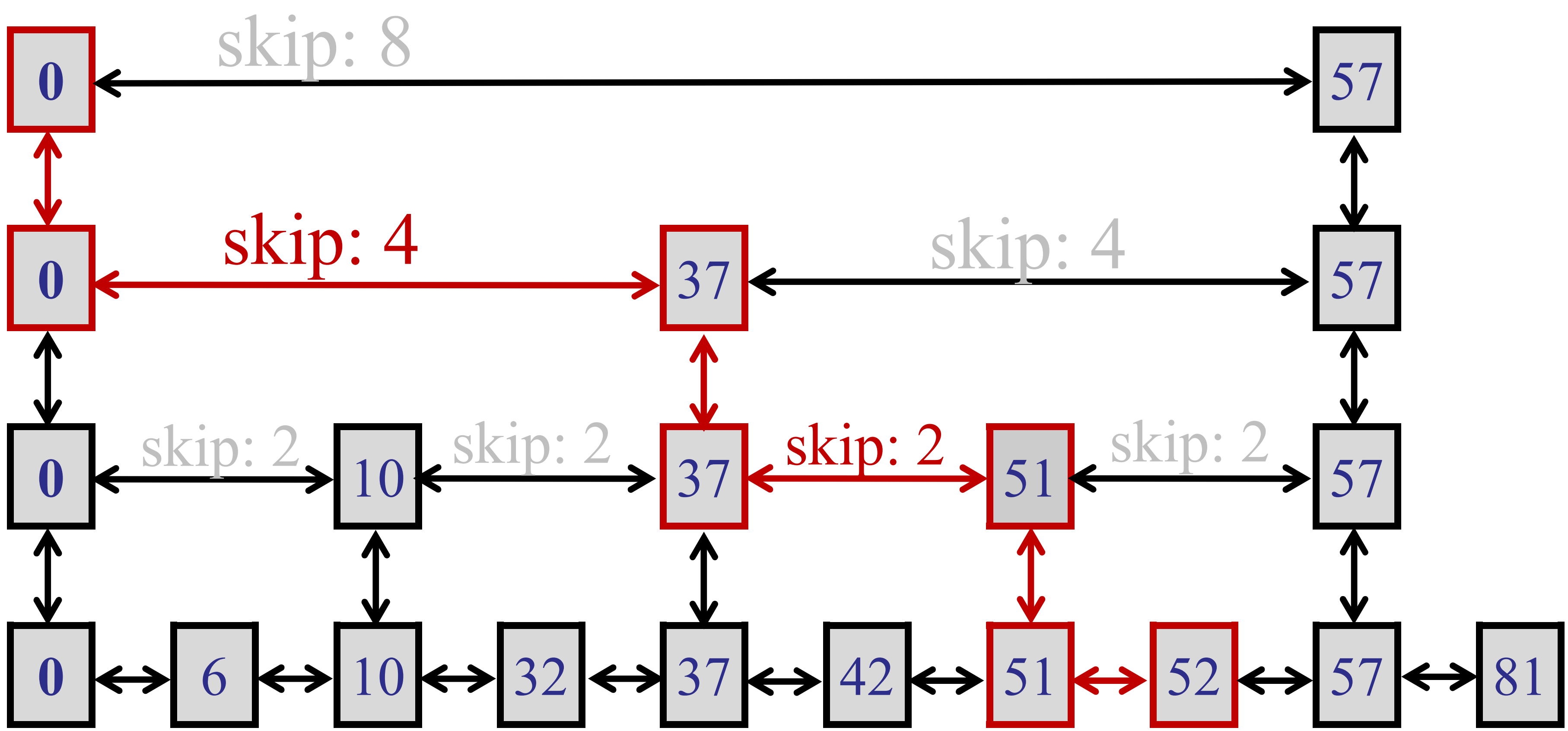
SkipList Extras

Example: find 7th element.



SkipList Extras

Example: find 7th element.



SkipList Extras

select(k) {

remainder = k;

it = head of top list.

while (*remainder* > 0){

if (*remainder* – *it.skipCount* > 0){

Go right.

else Go down.

return *it*.getData();

}

SkipList Analysis

SkipList Analysis

Claim: Every search and insert operation completes in O(log n) time *with high probability*.

Key steps:

- Analyze number of levels in a SkipList.
- Look at distribution of promotions:

  SkipList is efficient when each jump skips about the same number of elements.

SkipList Analysis

Claim: Every search and insert operation completes in O(log n) time *with high probability*.

Define *with high probability*:

An event occurs *with* ***high probability*** if, for any constant α, the event occurs with probability at least:

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

α affects hidden constants in O(.)

SkipList Analysis

Definition: with high probability

– Insert and Search terminate in O(log n) time with probability at least: $1 - \frac{1}{n^{\alpha}}$

Why?

– Set α big, error gets small (e.g., $\alpha = 100$).

– As *n* gets bigger, probability of error gets smaller.

SkipList Analysis

Warm-up:

- Assume each insert is *fast* with high probability: $1 - \frac{1}{n^{\alpha}}$
- Assume we insert n elements.
- What is the probability that ALL inserts are fast?

Boole's Inequality

Given events $e_1, e_2, \ldots, e_n$:

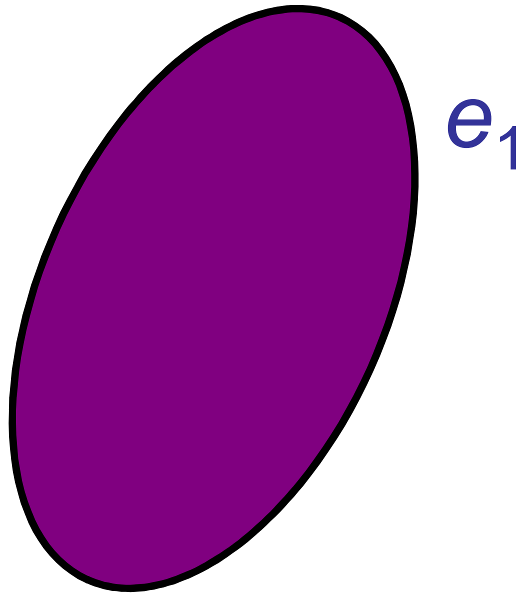
$$\Pr(e_1 \text{ or } e_2 \text{ or } e_3 \text{ or } \ldots \text{ or } e_n)$$

$$<$$

$$\Pr(e_1) + \Pr(e_2) + \Pr(e_3) + \ldots + \Pr(e_n)$$

Boole's Inequality

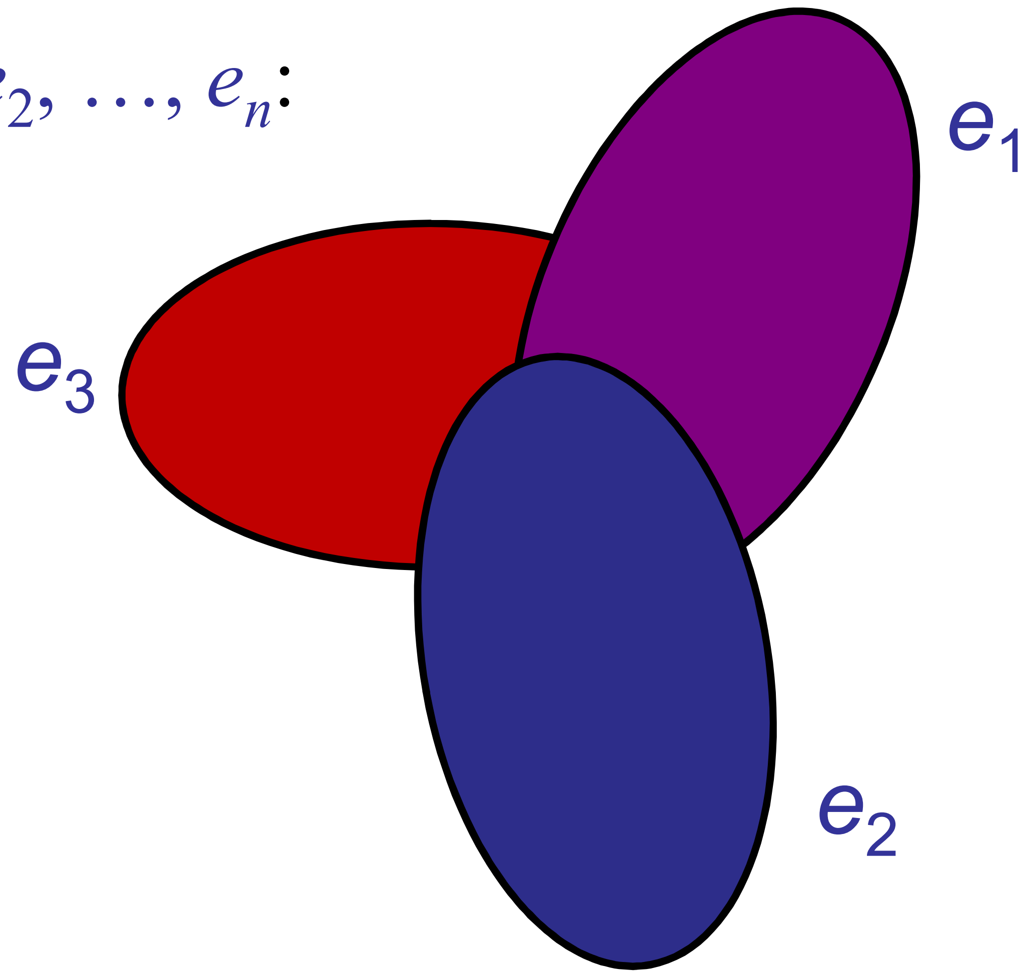
Given events $e_1, e_2, \ldots, e_n$:



$\Pr(e_1)$ = area of purple object.

Boole's Inequality

Given events $e_1, e_2, \ldots, e_n$:



$\Pr(e_1 \text{ or } e_2 \text{ or } e_3)$ = area three overlapping objects.

SkipList Analysis

Warmup:

- Assume each insert is *fast* with high probability: $1 - \frac{1}{n^{\alpha}}$
- Assume we insert *n* elements.
- What is the probability that ALL inserts are fast?

SkipList Analysis

Define:

- e_1 = probability first insert is slow $< 1/n^{\alpha}$
- e_2 = probability second insert is slow $< 1/n^{\alpha}$

...

- e_n = probability n^{th} insert is slow $< 1/n^{\alpha}$

SkipList Analysis

Define:

- e_1 = probability first insert is <u>slow</u> $< 1/n^{\alpha}$
- e_2 = probability second insert is <u>slow</u> $< 1/n^{\alpha}$

…

- e_n = probability nth insert is <u>slow</u> $< 1/n^{\alpha}$

$$\Pr(e_1 \text{ or } e_2 \text{ or } e_3 \text{ or } \ldots \text{ or } e_n) < \frac{n}{n^{\alpha}} < \frac{1}{n^{\alpha-1}}$$

$$\Pr(\textit{\textbf{all}} \text{ inserts are fast}) \geq 1 - \frac{1}{n^{\alpha-1}}$$

SkipList Analysis

Claim: *With high probability*, a SkipList with n elements has $O(\log n)$ levels.

SkipList Analysis

Claim: *With high probability*, a SkipList with n elements has $O(\log n)$ levels.

Proof:

Fix an element x.

$\Pr[x \text{ is higher than } c\log(n)] \leq \dfrac{1}{2^{c \log n}} \leq \dfrac{1}{n^c}$

Probability of flipping more than $c\log(n)$ heads in a row!

SkipList Analysis

Proof:

Fix an element x.

$\Pr[x \text{ is higher than } c\log(n)] \leq \dfrac{1}{2^{c\log n}} \leq \dfrac{1}{n^c}$

Define:

– e_1 = probability first element is <u>too high</u> $< 1/n^c$

– e_2 = probability second element is <u>too high</u> $< 1/n^c$

…

– e_n = probability n^{th} element is <u>too high</u> $< 1/n^c$

$\Pr(\textbf{\textit{any}} \text{ element is too high}) \leq \dfrac{n}{n^c} \leq \dfrac{1}{n^{c-1}}$

SkipList Analysis

Claim: *With high probability*, a SkipList with n elements has $O(\log n)$ levels (for sufficiently large c).

SkipList Analysis

Done!

Claim: Every search and (insert) operation completes in O(log n) time *with high probability*.

Done!

Key steps:

– Analyze number of levels in a SkipList.

– Look at distribution of promotions:

SkipList is efficient when each jump skips about the same number of elements.

SkipList Analysis

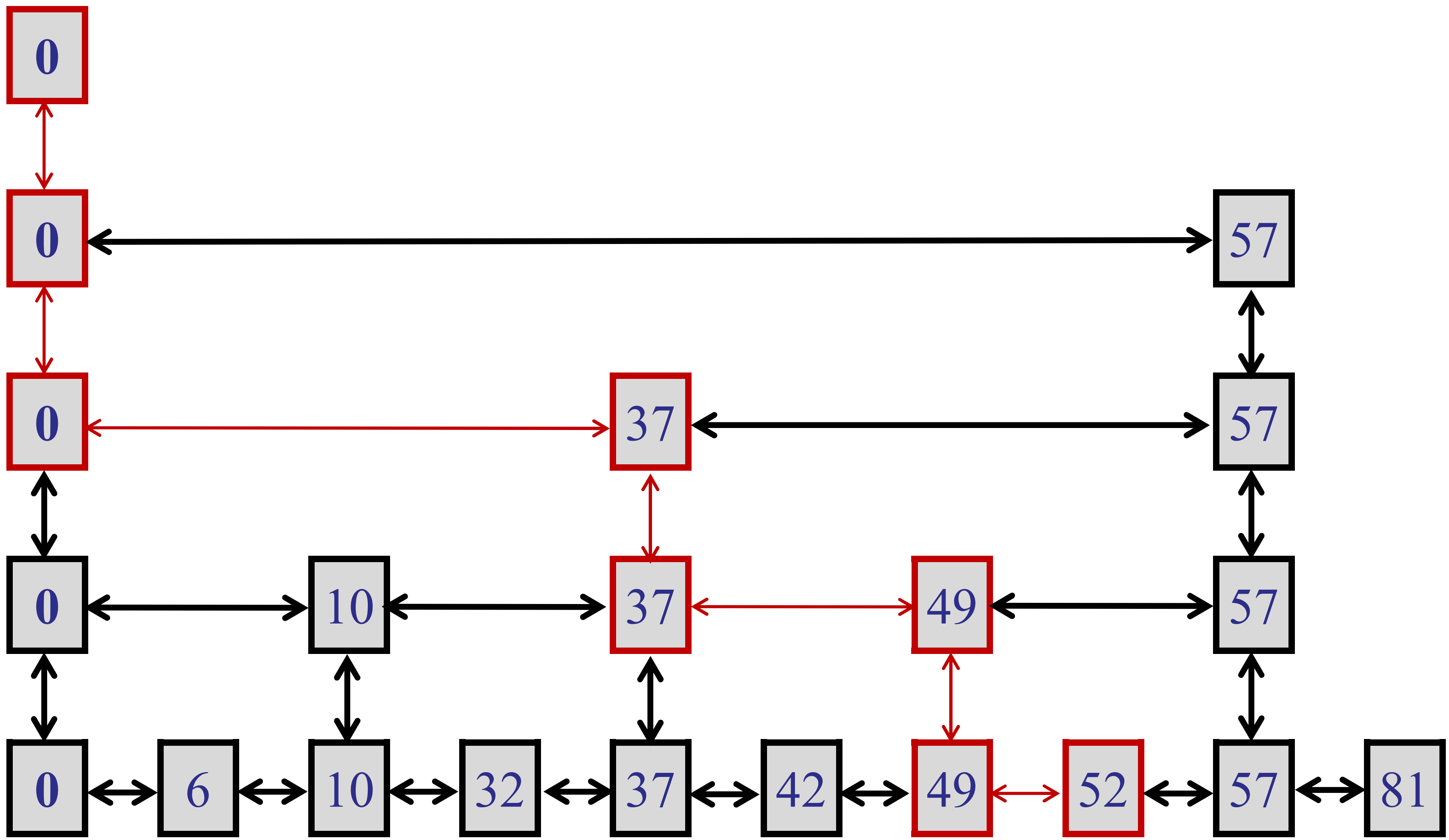
Analyzing a search:

Neat idea: analyze the search backwards.

- Start at leaf.
- For each node visited:
  - If node was not promoted (TAILS), go left.
  - If node was promoted (HEADS), go up.
- Stop at root of tree.

Occurs at most O(log n) times!

Example: `search(52)`



SkipList Analysis

Analyzing a search:

Neat idea: analyze the search backwards.

- Start at leaf.
- For each node visited:
  - If node was not promoted (TAILS), go left.
  - If node was promoted (HEADS), go up.
- Stop at root of tree.

At most O(log n)!

New question: How many times to flip a coin until we get $c\log(n)$ heads?

SkipList Analysis

Claim: *With high probability*, after $O(\log n)$ coin flips, you get $c \log n$ heads.

SkipList Analysis

Proof:

- Say we flip $10c\log(n)$ coins.
- Pr[exactly $c\log(n)$ heads] =

$$\binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{c\log n}\left(\frac{1}{2}\right)^{9c\log n}$$

Number of ways to choose
clog(n) heads out of all the flips:
TTTTHH
TTTHTH
…

SkipList Analysis

Proof:

- Say we flip $10c\log(n)$ coins.
- Pr[exactly $c\log(n)$ heads] =

$$\binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{c\log n}\left(\frac{1}{2}\right)^{9c\log n}$$

Probability each of the H comes up heads.

Probability each of the T comes up tails.

SkipList Analysis

Proof:

- Say we flip $10c\log(n)$ coins.
- Pr[exactly $c\log(n)$ heads] =

bad case!

$$\binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{c\log n}\left(\frac{1}{2}\right)^{9c\log n}$$

- Pr[at most $c\log(n)$ heads] ≤

If all 9clog(n) are tails, then not enough heads!

$$\binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{9c\log n}$$

SkipList Analysis

Bounding binomials:

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(\frac{ey}{x}\right)^x$$

SkipList Analysis

Bounding binomials:

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(\frac{\mathrm{e}y}{x}\right)^x$$

$$\binom{10c\log n}{c\log n} \le \left(\frac{\mathrm{e}10c\log n}{c\log n}\right)^{c\log n} \le (10\mathrm{e})^{c\log n}$$

$$\le n^{c\log(10\mathrm{e})}$$

SkipList Analysis

Proof:

- Say we flip $10c\log(n)$ coins.
- Pr[at most $c\log(n)$ heads] $\leq \binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{9c\log n}$

$$\leq n^{c\log(10e)}\frac{1}{n^{9c}}$$

$$\leq \frac{1}{n^{\alpha}}$$

$\alpha = c(9 - \log(10) - \log(e))$ Generalize for other values of 10…

SkipList Analysis

Claim: *With high probability*, after O(log n) coin flips, you get $c$log n heads.

Conclusion: Each search takes O(log n) steps with high probability.

Conclusions

SkipLists

- Simple, efficient, randomized search structure.
- Easy to implement.
- Reasonably good performance in practice.

Analysis:

- Tricky randomized calculations.
- Key idea: analyze backwards!
- Reduce to the problem of flipping coins.