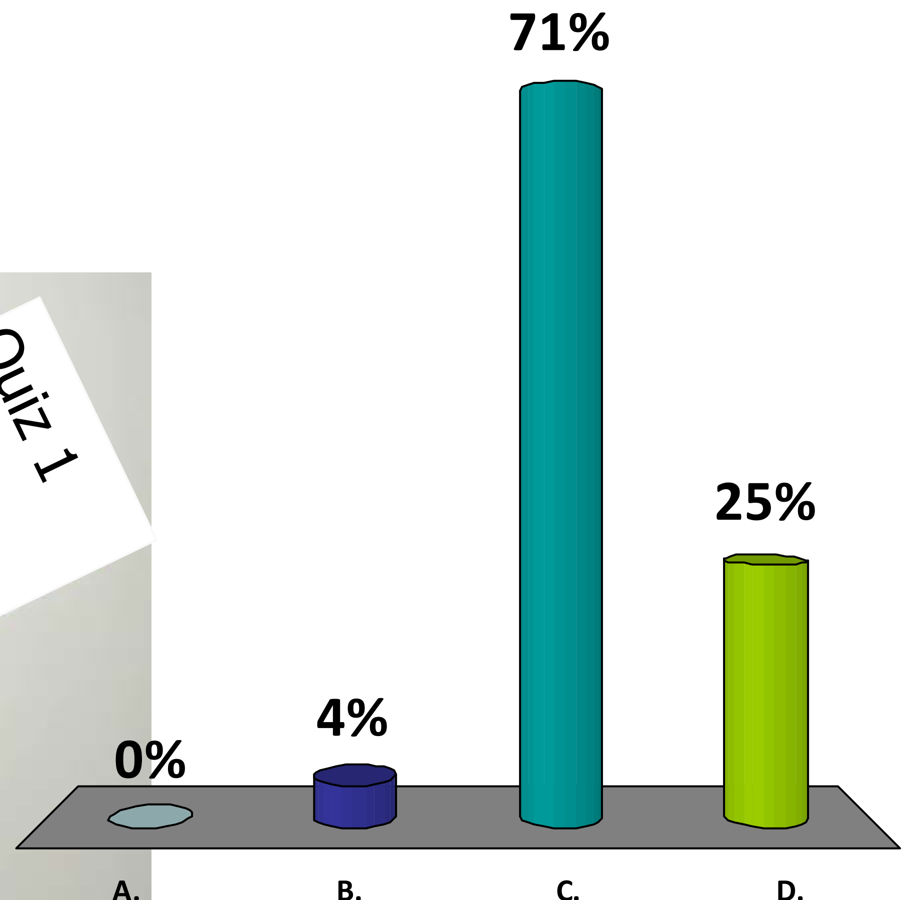
CS2020
Data Structures and Algorithms

Welcome!

When is the date for the Quiz?

A. On the Eve of CNY
B. On the Day of CNY
C. 12 Feb 2039
D. 12 Feb 2016



Announcements

Quiz 1 : February 12

- In class: be there!
- Be on time.
- Covers material through today's lecture

Bring to quiz:

- One sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else.

Announcements

Quiz 1 : February 12

- In class: be there!
- Be on time.
- Covers material through today's lecture

Practice Quizzes (from 2013 and 2015):

- Posted today
- Covered in Discussion Group
- Quiz will be very similar in structure

Quiz Topics

Theory:

- Asymptotic analysis
- Simple recurrences
- Simple probability

Quiz Topics

Algorithms and data structures:

- Abstract Data Types
  - Bags, Lists, Stacks, Queues
- Divide-and-conquer
  - Binary search
  - Peak finding
- Sorting
  - BubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort
  - Reversal Sorting, The Birthday Party, etc.

Quiz Topics

Java:

- Object-oriented programming
  - Basic principles and implementation in Java
- Basic Java
  - classes and inheritance: class, interface, implements, extends
  - access control: public, private, protected, static
  - Generics
  - Comparable
  - Iterators
  - etc…..

Announcements

Quiz Advice:

Announcements

Quiz Advice:

Get the maximum number of points you can.

Announcements

Quiz Advice:

Get the maximum number of points you can.

- Do not leave easy questions blank.
- Give a simple solution for partial credit.
- Bypass questions instead of getting stuck.
- Show me what you know.

Announcements

Quiz Advice:

Be as clear as possible.

- Do not be ambiguous.
- Circle your final answer (if it is unclear).
- Cross out incorrect answers.
- Write neatly.

Announcements

Quiz Advice:

State your assumptions.

- If the question is ambiguous, state precisely what you are assuming.
- If your assumptions are reasonable, and your answer is correct subject to those assumptions, you will (most likely) get full credit!

Announcements

Quiz Advice:

Review the basics:

- Know the basic recurrences.
- Review how the algorithms we have studied work.
- Know the running time of the algorithms we have studied.

Announcements

Quiz Advice:

Review Java:

- Make sure you know what the different keywords mean.
- Try to generate examples that are correct **and** incorrect (e.g., DG 3).

Announcements

Quiz Advice:

Review problem solving strategies:

- Review problems we have solved on problem sets, in DG, in recitation, in class.
- What is the basic strategy used in the question?
  - Binary search? Divide-and-conquer? Sorting?
- What strategy is good for which types of problems?

Today's Plan

1. ~~Exceptions and error handling~~

2. Problem: Scheduling Airplanes

3. Abstract Data Types:
   - Symbol Tables
   - Dictionaries

4. Linked Data Structures

Today's Plan

1. ~~Exceptions and error handling~~

2. Problem: Scheduling Airplanes

3. Abstract Data Types:
   - Symbol Tables
   - Dictionaries

4. Linked Data Structures

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi!) has 1 runway.
- Airplanes want to land:
  - Input: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

Harder Airport Scheduling

Multiple Runway Problem:

- Changi airport has k runways.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

Not today…

- Think of scheduling computing jobs on a network.

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi) has 1 runway.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no

```
interface IRunway{
   // return true if scheduled for time t
   // return false if scheduling fails
   boolean requestLanding(Time t, Plane p);
}
```

Airport Scheduling

Simple Runway Problem:

- Small airport (not Changi) has 1 runway.
- Airplanes want to land:
  - Given: requested landing time
  - Requirement: 3 minutes between planes
  - Output: yes/no
- Additional requirements:
  - How many planes scheduled between: 9:00-11:00am?
  - Cancel landing reservation.

Airport Scheduling

Implementing ideas?

Airport Scheduling

Algorithm 1:

- Maintain an **unsorted** list of landing times.

7:00	6:35	14:23	12:21	7:19	8:21	14:42			

- On a request for time t, scan the list.
- If time t is safe, then add t to the end of the list.

7:00	6:35	14:23	12:21	7:19	8:21	14:42	t		

Airport Scheduling

Better ideas?

Airport Scheduling

Algorithm 2:

- Maintain a **sorted** list of landing times.

6:35	7:00	7:19	8:21	12:21	14:23	14:42			

- On a request for time t, binary search the list.
- If time t is safe, then add t to the end of the list.

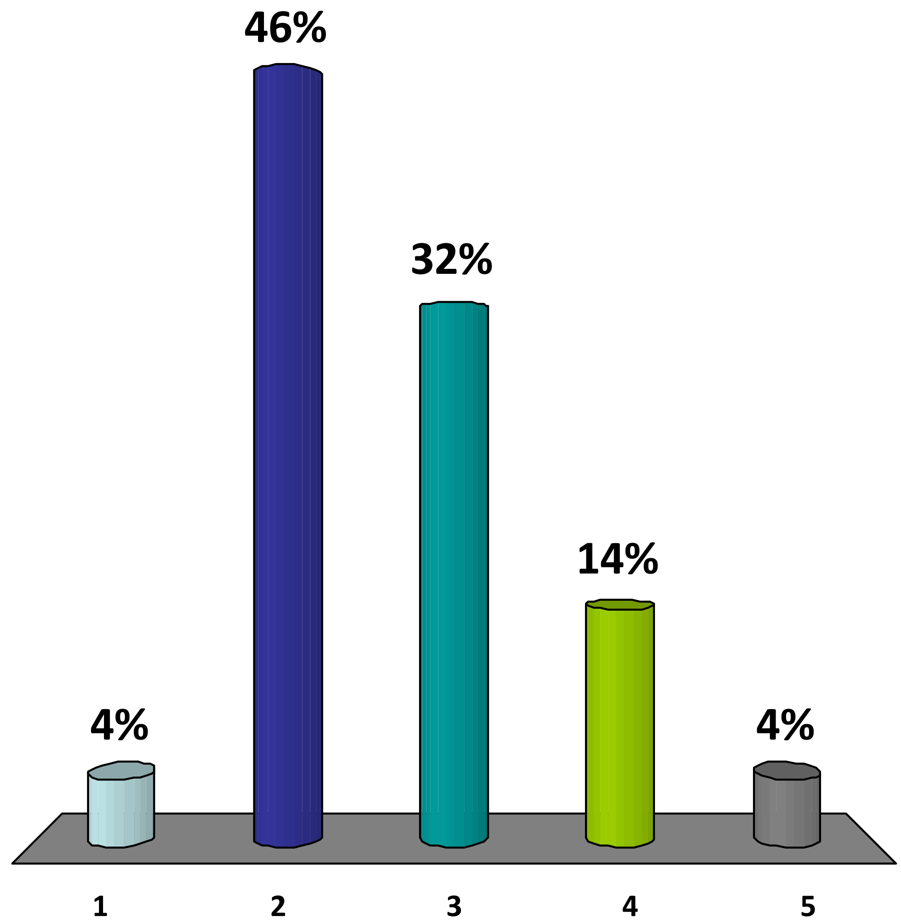
6:35	7:00	7:19	8:21	12:21	14:23	14:42	t		

- **Re-sort**:

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		

Running time for insertion Algorithm 2:

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



Airport Scheduling

Algorithm 2:

- Maintain a **sorted** list of landing times.

6:35	7:00	7:19	8:21	12:21	14:23	14:42			

- On a request for time t, binary search the list.
- If time t is safe, then add t to the end of the list.

6:35	7:00	7:19	8:21	12:21	14:23	14:42	t		

- Re-sort:

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		

Airport Scheduling

Algorithm 2b:

- Maintain a **sorted** list of landing times.

6:35	7:00	7:19	8:21	12:21	14:23	14:42			

- On a request for time t, binary search the list.
- If time t is safe, then make space for t by moving other times over.

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		

Running time: O(n)

Airport Scheduling

Algorithm 3:

– Maintain a list of all times.

...	7:00	7:01	7:02	7:03	7:04	7:05	7:06	7:07	...
		X			?			X	

– If times: [t-2, t-1, t, t+1, t+2] are free, schedule plane.

Running time: O(1)

Space: (24*60)

What if arrival times are not on-the-minute?

Airport Scheduling

Algorithm 3:

- Maintain a list of all times.

...	7:00	7:01	7:02	7:03	7:04	7:05	7:06	7:07	...
		X			?			X	

Problems:

What if arrival times are not on-the-minute?

Total number of scheduled planes?

Next scheduled plane after 7:00pm?

Scheduling

Simple Runway Problem:

- 1 runway.
- Airplanes want to land

Complicated Runway Problem

- Multiple runways
- Airplanes want to land

Discrete Event Simulation

- System with ongoing events
- Subject to constraints

Abstraction

Key advantages

- Separate interface and implementation
- Hide implementation details
- Modularity: implement/analyze components separately

Abstract Data Types

Specification:

- Interface
- Behavior

Implementation:

- Application dependent
- Array or List or Stack or Queue or …

Abstract Data Types

What type of data?

Key-Value pairs

- key : search index for the pair
- value : main object being stored

Examples:

- (time, airplane)
- (pilot, airplane)
- (pilot, identification-number)

Abstract Data Types

Example (key, value) pairs

Key	1	2	5	3	4	5	6	7	8	9
Data	a	b	C	g	h	D	j	k	l	m

Abstract Data Types

Symbol Table

Interface:

- void insert(KType key, VType value)
- VType search(KType key)

Behavior:

- insert : adds (key, value) to table
- search : returns value associated with specified key (or null, if not available)

Symbol Table

Examples:

Dictionary:	key = word
	value = definition
Phone Book	key = name
	value = phone number
Internet DNS	key = website URL
	value = IP address
Java compiler	key = variable name
	value = type and value

Symbol Table

Example:

Insert data:

insert(9:30, planeAAA)

insert(10:00, plane BBB)

Query data:

search(9:30) => **planeAAA**

search(10:00) => **planeBBB**

Abstract Data Types

Symbol Table

Interface:

- void insert(KType key, VType value)
- VType search(KType key)

Behavior:

- insert : adds (key, value) to table
- search : returns value associated with specified key (or null, if not available)

Abstract Data Types

Symbol Table

Class:

```
public class  SymbolTable<Key extends Comparable<Key>, Value>
```

	`SymbolTable()`	*constructor*
`void`	`insert(Key k, Value v)`	*insert (k,v) into table*
`Value`	`search(Key k)`	*get value paired with k*
`void`	`delete(Key k)`	*remove key k (and value)*
`boolean`	`contains(Key k)`	*is there a value for k?*
`int`	`size()`	*number of (k,v) pairs*

Symbol Table

Interface Ambiguities

Can you insert a null value? **NO**

insert(10, null)

If you search for a non-existent key?

search(10) = **null**

If you re-insert a key with a new value?

insert(10, AAA)

insert(10, BBB)

BBB overwrites AAA

Symbol Table

Why?

Easy to implement `contains()`:

```
public boolean contains(Key k)}
      return (search(k) != null);
}
```

Easy to implement (lazy) `delete()`:

```
public void delete(Key k)}
      insert(k, null);
}
```

Symbol Table

Key Mutability

```
SymbolTable<Time, Plane> t =
           new SymbolTable<Time, Plane>();

Time  t1 = new Time(9:00);
Time  t2 = new Time(9:15);

t.insert(t1, “SQ0001”);
t.insert(t2, “SQ0002”);

t1.setTime(10:00);

x = new Time(9:00);
t.search(x);
```

Symbol Table

Key Mutability

```
SymbolTable<Time, Plane> t =
           new SymbolTable<Time, Plane>();

Time  t1 = new T
Time  t2 = new T

t.insert(t1, “SC
t.insert(t2, “SC

t1.setTime(10:00);

x = new Time(9:00);
t.search(x);
```

Moral: Keys should be immutable.

Examples: Integer, String

Abstract Data Types

Symbol Table

Class:

```
public class  SymbolTable<Key extends Comparable<Key>, Value>
-------------------------------------------------------------------
                   SymbolTable()              constructor
            void   insert(Key k, Value v)     insert (k,v) into table
           Value   search(Key k)              get value paired with k
            void   delete(Key k)              remove key k (and value)
         boolean   contains(Key k)            is there a value for k?
             int   size()                     number of (k,v) pairs
-------------------------------------------------------------------
```

Abstract Data Types

Dictionary

Interface:

- void insert(KType key, VType value)
- VType search(KType key)
- KType successor(KType key)
- KType predecessor(KType key)

Abstract Data Types

Dictionary

Interface:

- void insert(KType key, VType value)
- VType search(KType key)
- KType successor(KType key)
- KType predecessor(KType key)

Adds new item to dictionary.

Abstract Data Types

Dictionary

Interface:

- void insert(KType key, VType value)
- VType search(KType key)
- KType successor(KType key)
- KType predecessor(KType key)

Searches for item in dictionary.

Abstract Data Types

Dictionary

Interface:

- void insert(KType key, VType value)
- VType search(KType key)
- KType successor(KType key)
- KType predecessor(KType key)

Find first item in dictionary that is bigger than **key**.

Find biggest item in dictionary that is smaller than **key**.

KType must implement Comparable<KType>

Airport Scheduling

Dictionary

6:35	7:00	7:19	8:21	12:21	14:23	14:42			
SIA-07	SIA-12	SIA-01	UA-10	DAL-32	JAL-42	SIA-09			

insert(t, A)

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		
SIA-07	SIA-12	SIA-01	A	UA-10	DAL-32	JAL-42	SIA-09		

Airport Scheduling

Dictionary

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		
SIA-07	SIA-12	SIA-01	A	UA-10	DAL-32	JAL-42	SIA-09		

- search(8:24) $\rightarrow$ **false**
- search(8:21) $\rightarrow$ **true**

Airport Scheduling

Dictionary

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		
SIA-07	SIA-12	SIA-01	A	UA-10	DAL-32	JAL-42	SIA-09		

- successor(8:24) = 12:21

Airport Scheduling

Dictionary

6:35	7:00	7:19	t	8:21	12:21	14:23	14:42		
SIA-07	SIA-12	SIA-01	A	UA-10	DAL-32	JAL-42	SIA-09		

– predecessor(8:24) = 8:21

Question: should successor/predecessor return a key or a value?

Airport Scheduling

```
class SimpleRunway implements IRunway{
    Dictionary dict<Time, Plane>;

    boolean requestLanding(Time t, Plane p){
        if (dict.search(t) != null) return false; // Check time t
        Integer pred = dict.predecessor(t); // Check predecessor
        if ((t – pred) <= 3) return false;
        Integer succ = dict.successor(t);  // Check successor
        if ((succ – t) <= 3) return false;
        dict.insert(t, p);  // Insert new item
        return true;
    }
}
```

Airport Scheduling

```
class SimpleRunway implements IRunway{
    Dictionary dict<Time, Plane>;

    boolean requestLanding(Time t, Plane p){
        if (dict.search(t) != null) return false; // Check time t
        Time pred = dict.predecessor(t); // Check predecessor
        if ((t – pred) <= 3) return false;
        Time succ = dict.successor(t);  // Check successor
        if ((succ – t) <= 3) return false;
        dict.insert(t, p);  // Insert new item
        return true;
    }
}
```

Bug: can’t subtract Time class

Bug: What if pred/succ is null?

Summary

1. Problem: Scheduling Airplanes
   - Implement via Dictionary

2. Abstract Data Types:
   - Symbol Tables
   - Dictionaries

Dictionary

Implementation

Option 1: Unsorted array

- insert: add to end of array --- O(1)
- search : linear search through array --- O(n)

Dictionary

Implementation

Option 1: Unsorted array

- insert: add to end of array --- O(1)
- search : linear search through array --- O(n)

Option 2: Sorted array

- insert: add to middle of array --- O(n)
- search : binary search in array --- O(log n)

Dictionary

Array Implementation Problems

Problem 1: Too slow

- Operations have O(n) cost.
- Not today…

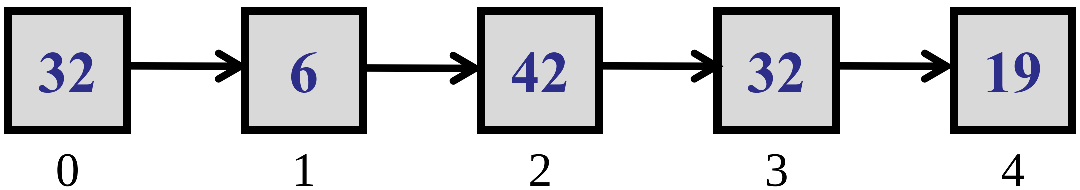
Problem 2: How big an array?

- Solution 1: today
- Solution 2: in a few weeks

Linked List Implementation

Basic structure:

– Chained array of ListNodes.

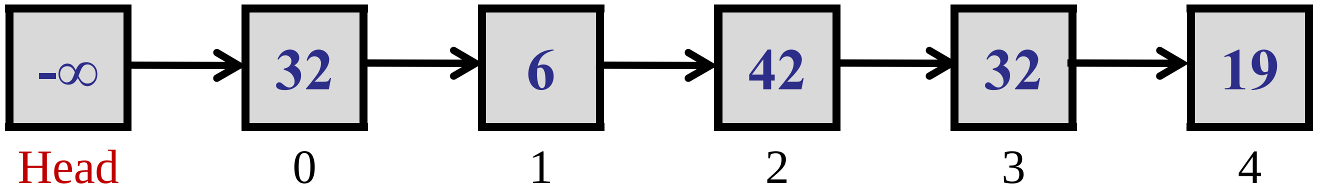



```
class ListNode<Key extends Comparable<Key>>{
      private ListNode<Key> m_next = null;
      private Key m_key = null;
      …
}
```

Linked List Implementation

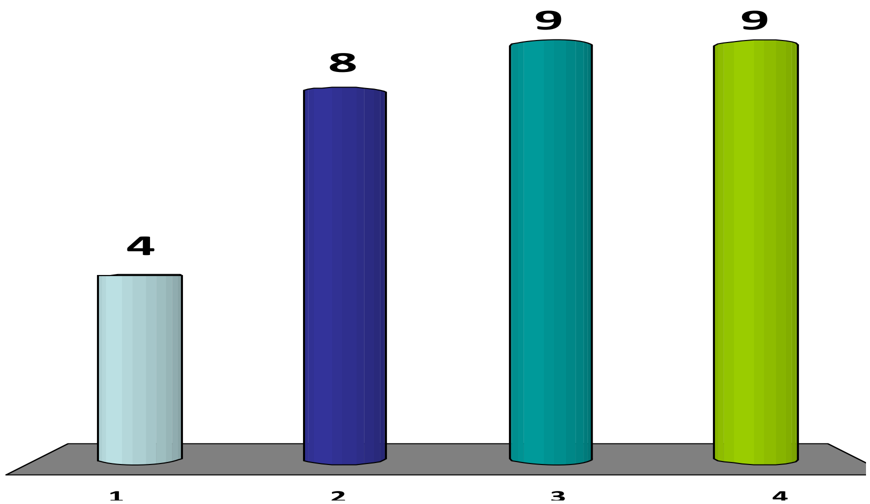
Basic structure:

- Chained array of ListNodes.
- Special head node.



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

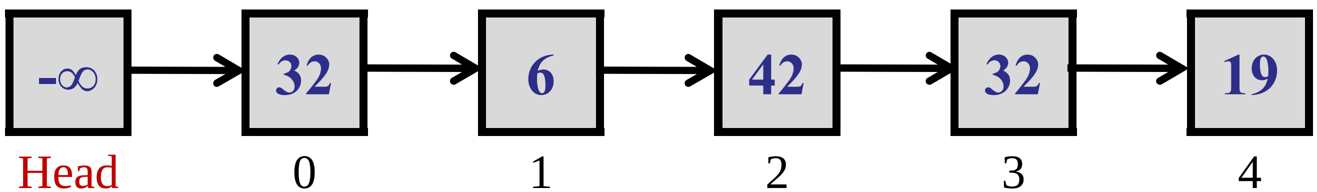
1. To avoid special code for an empty list.
2. To store extra information (e.g., the size of the list)
3. To make the program run faster.
4. To simplify pre-pending an item to the list.



Linked List Implementation

Basic structure:

- Chained array of ListNodes.
- Special head node.

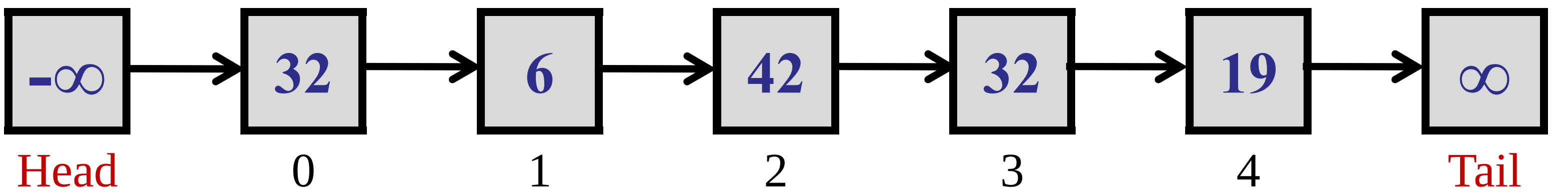



```
private ListNode<Key> m_list = null;

void doSomething(Key k){
      m_list.delete(k);
}
```

Linked List Implementation

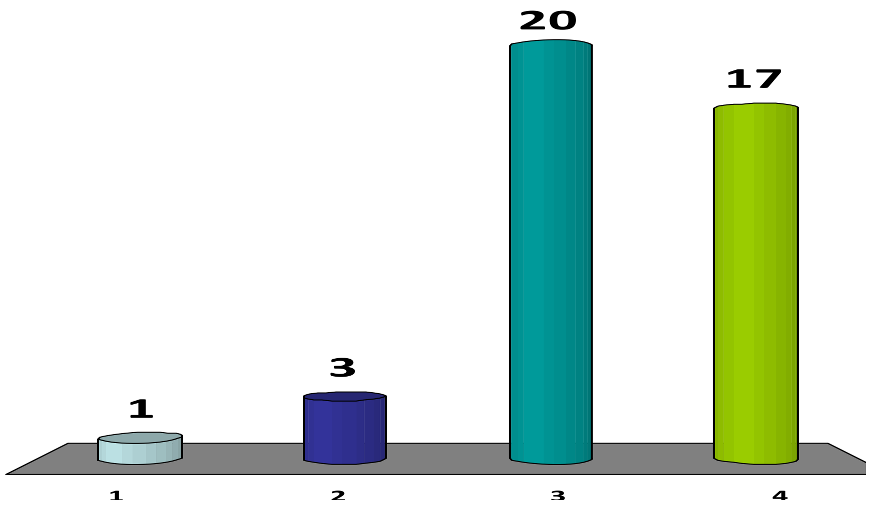
Basic structure:

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



Why do we add special tail node?

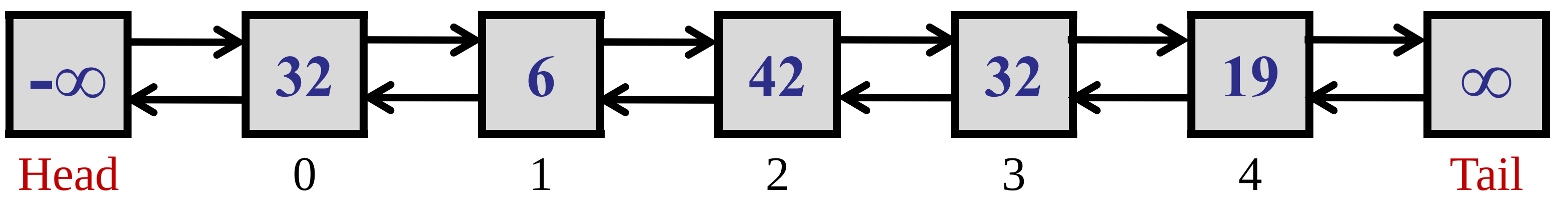
1. To avoid special code for an empty list.
2. To store extra information.
3. To avoid accidently traversing off the end of the list.
4. To allow for fast appending at the end.



Linked List Implementation

Basic structure:

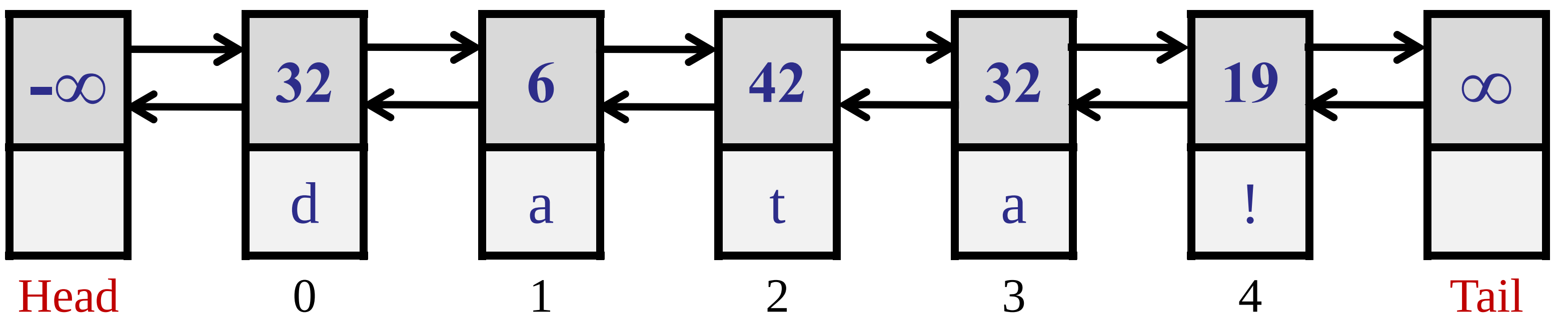
- Chained array of ListNodes.
- Special head node.
- Special tail node.
- Doubly-linked.



Linked List Implementation

Basic structure:

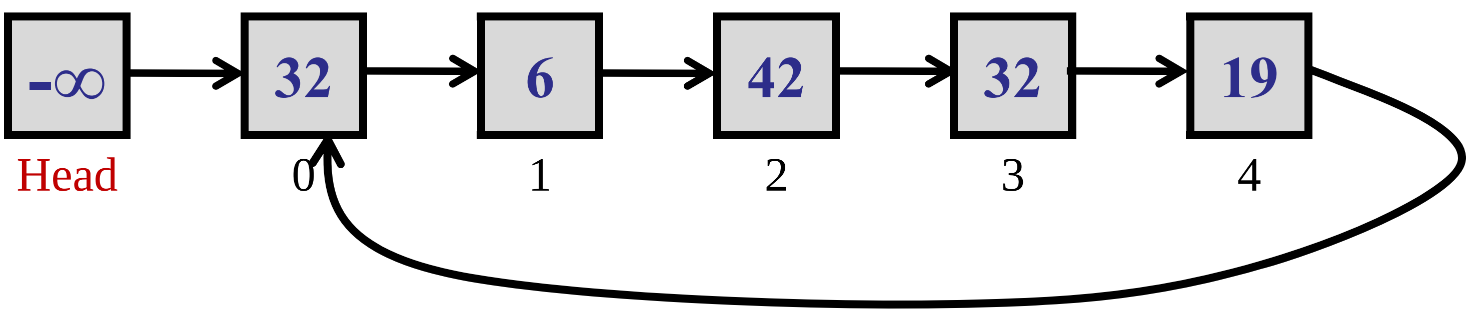
- Chained array of ListNodes.
- Special head node.
- Special tail node.
- Doubly-linked.
- Add cells for data.



Linked List Implementation

Basic structure:

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

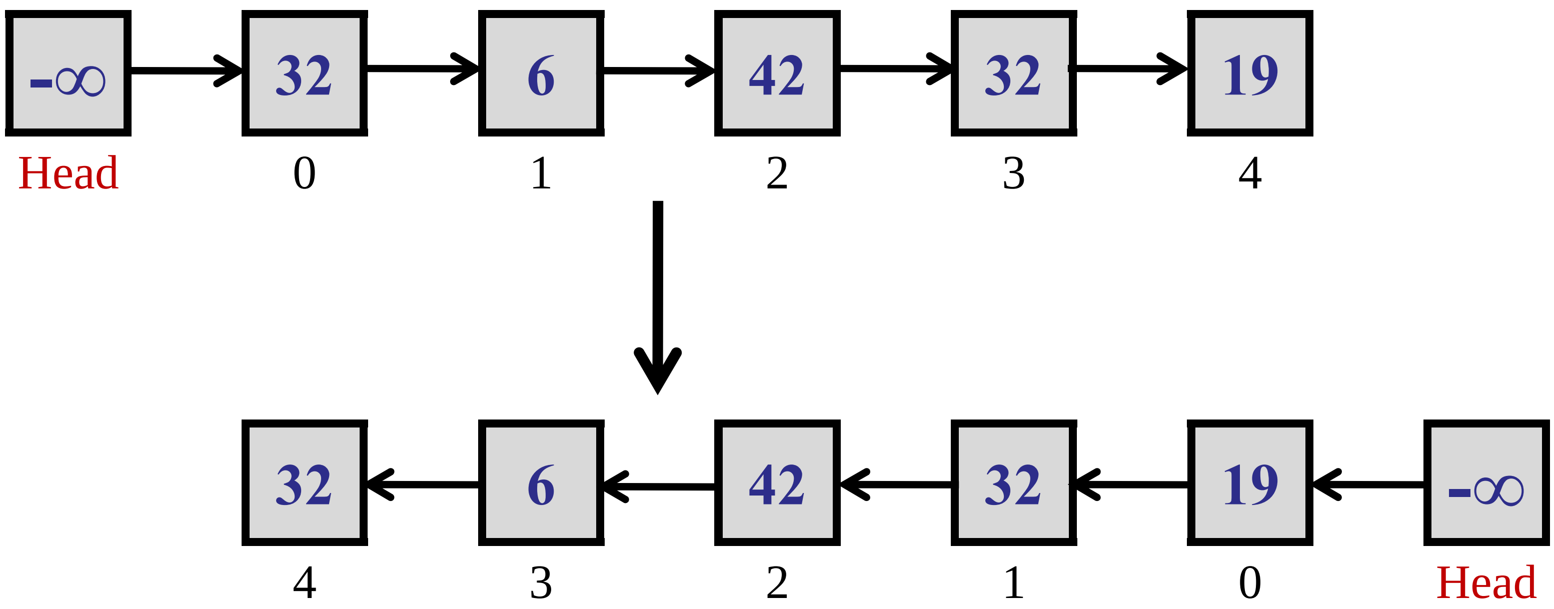


Puzzle Break

Standard Interview Question 1:

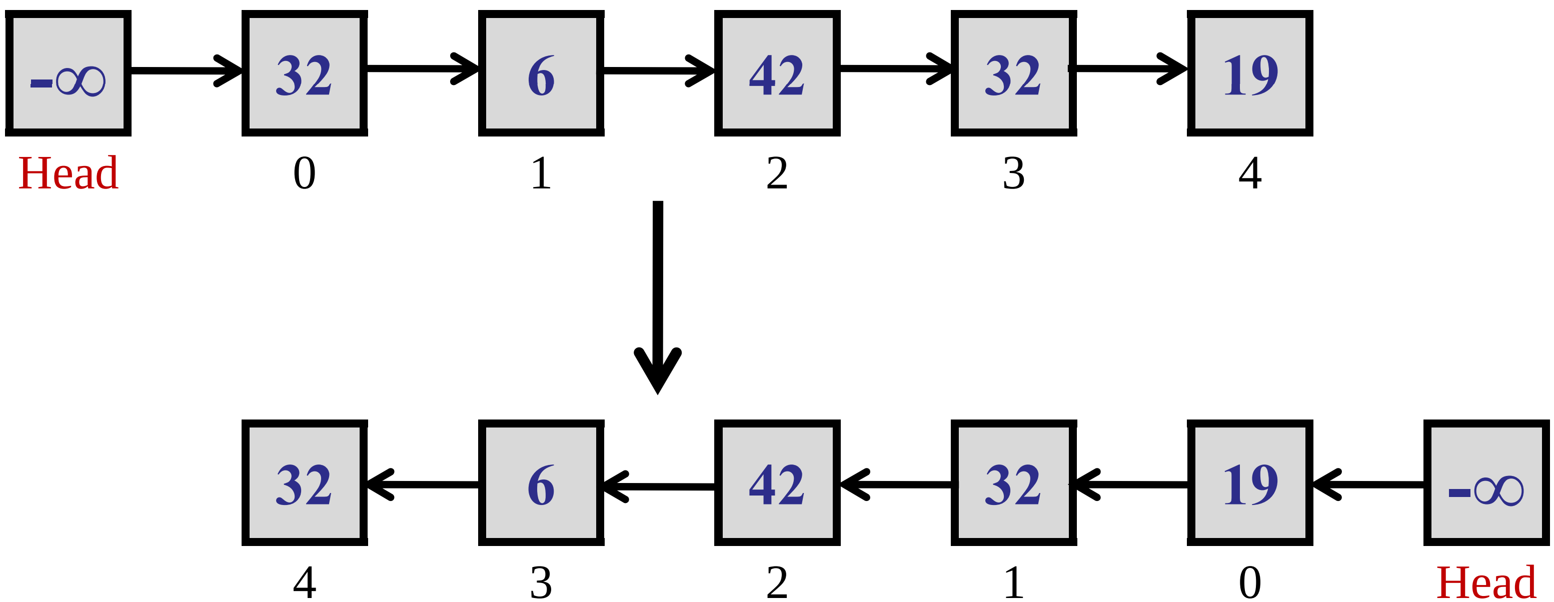
Input: Head pointer to a singly linked list.

Output: Head of reversed linked list



Puzzle Break

```
void reverseList(ListNode<Key> list){
      // Your code here (3-4 lines?)
}
```



Linked Data Structures

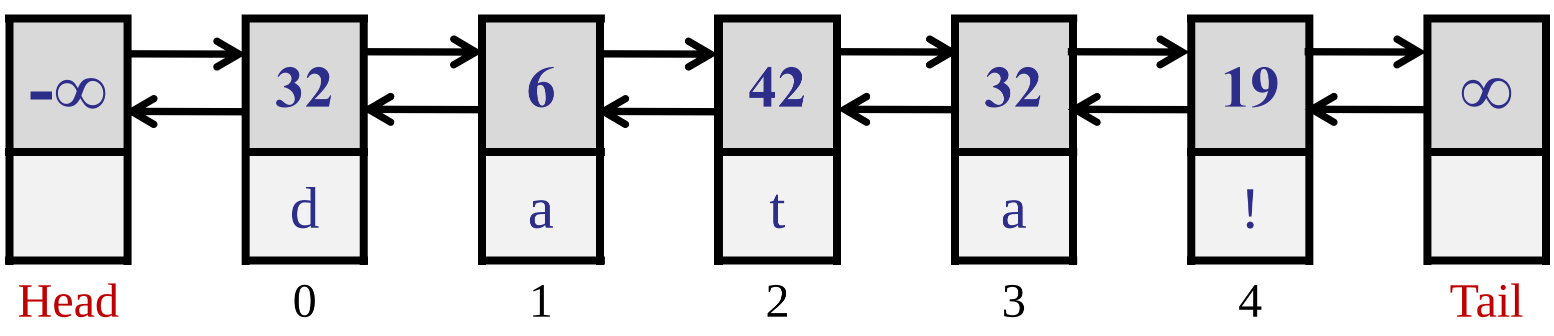
Challenge:

- You only have access to one node in the structure at a time.
- You cannot access arbitrary parts of the data structure.
- If you “lose” your handle, the whole structure is lost.

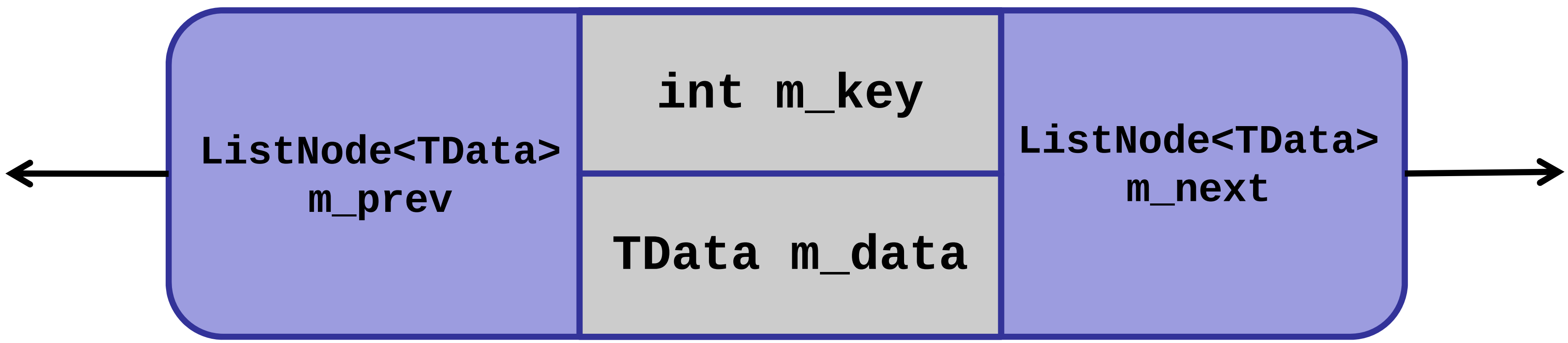
Linked Lists and Dictionaries

Plan:

- Implementing a ListNode.
- Adding List functionality.
- Implement a Dictionary.



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;
}
```

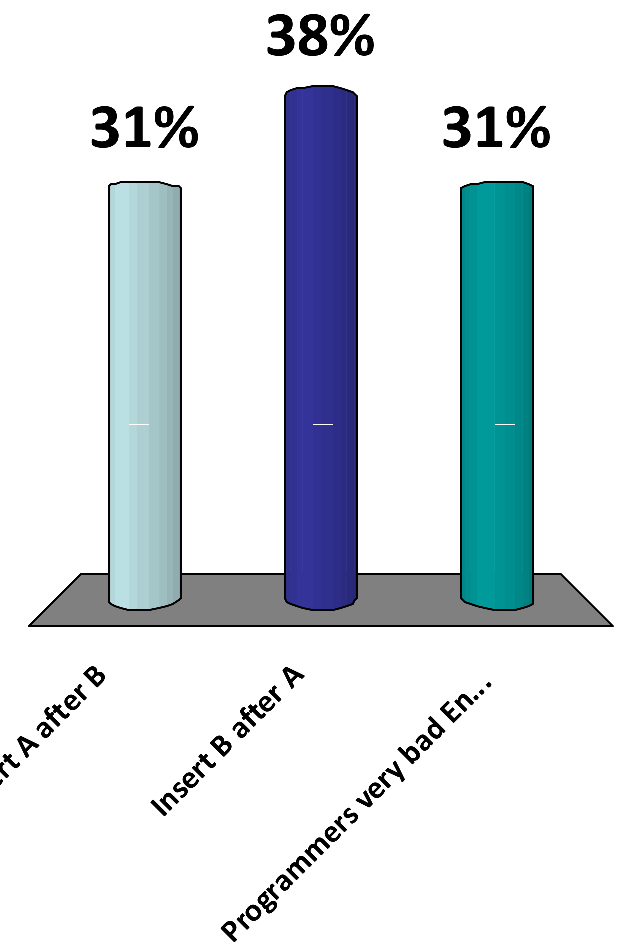
```
public ListNode<TData> getPrevious()
{
    return m_prev;
}
```

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

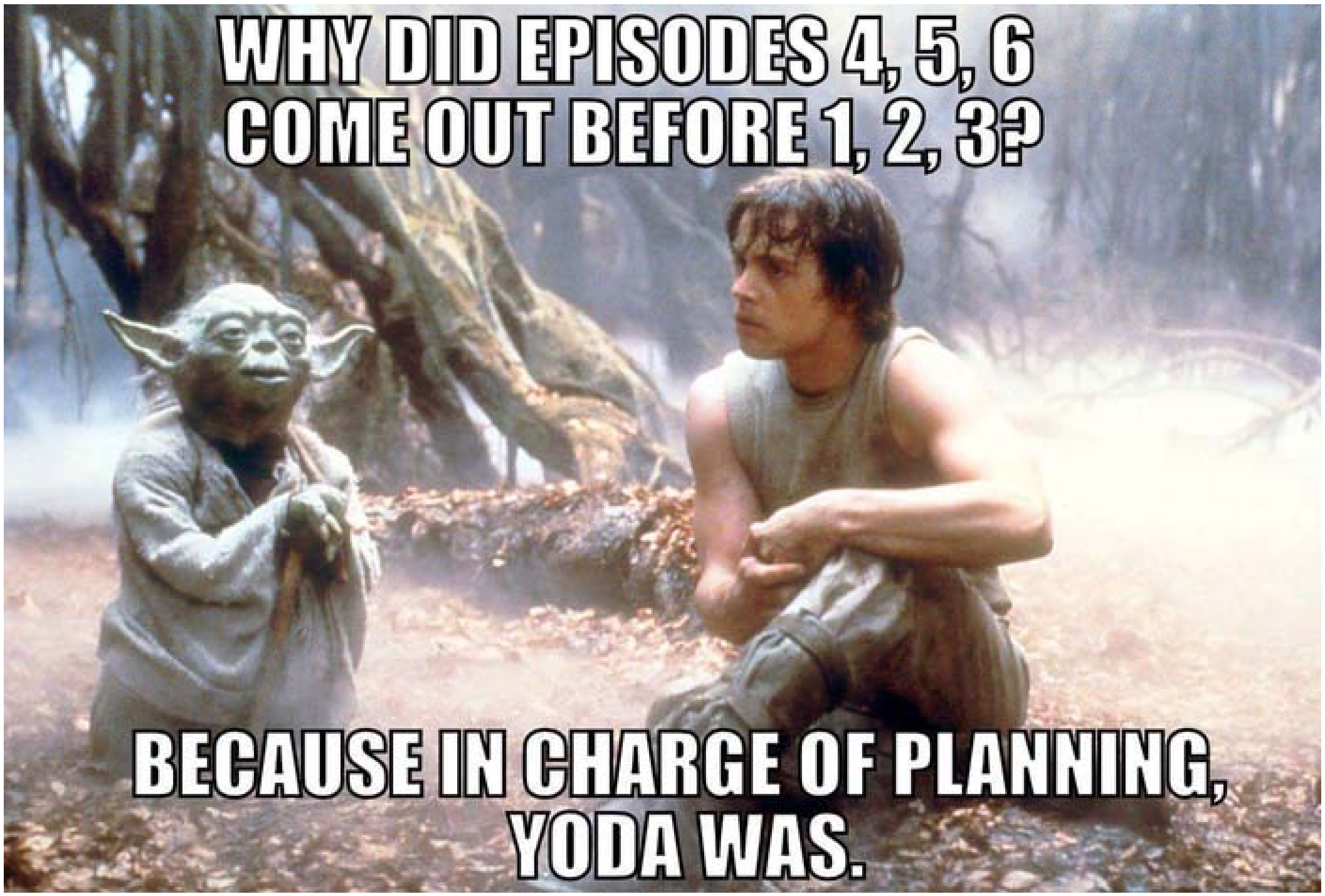
```
public void setPrevious(ListNode<TData> prevNode)
{
    m_prev = (ListNode<TData>)prevNode;
}
```

If A and B are two nodes, callin "`A.insertAfter(B)`" means

A. Insert A after B
B. Insert B after A
C. Programmers very bad English One!



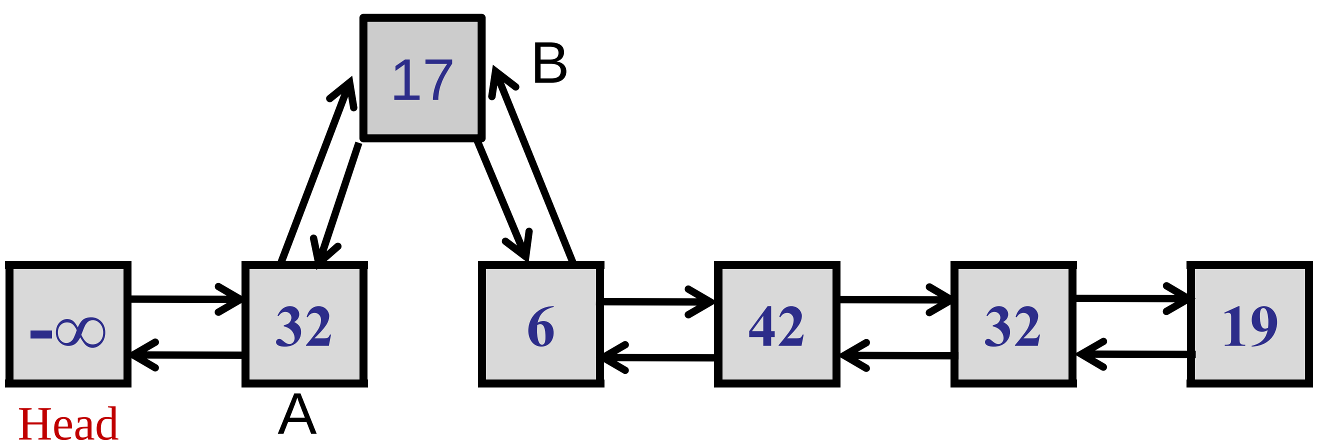
Starwars…


WHY DID EPISODES 4, 5, 6
COME OUT BEFORE 1, 2, 3?
BECAUSE IN CHARGE OF PLANNING,
YODA WAS.

Inserting a ListNode

Plan:

– When we call A.insertAfter(B) it means insert the node "B" after "A"

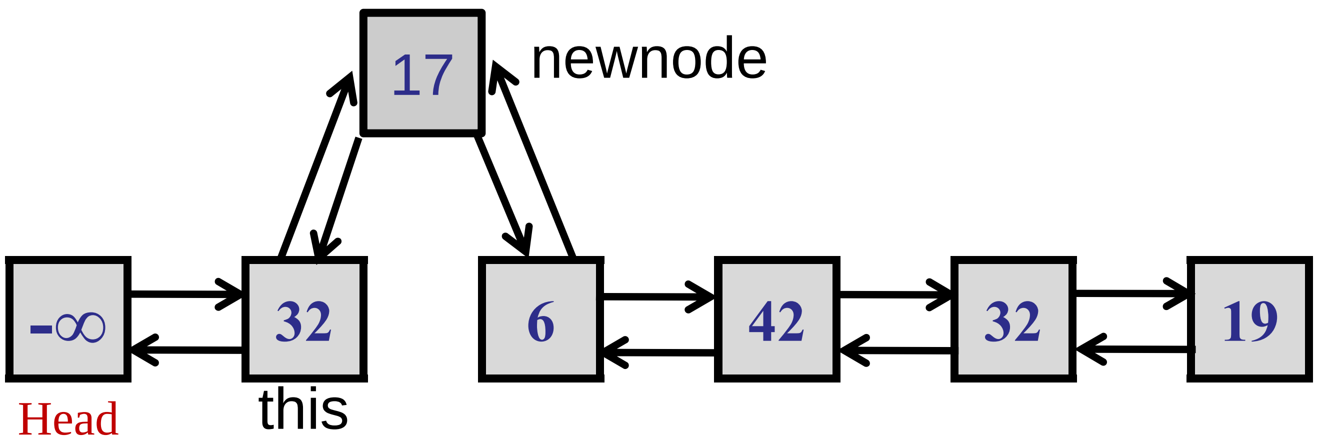


Inserting a ListNode

```
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);
}
```

Error?

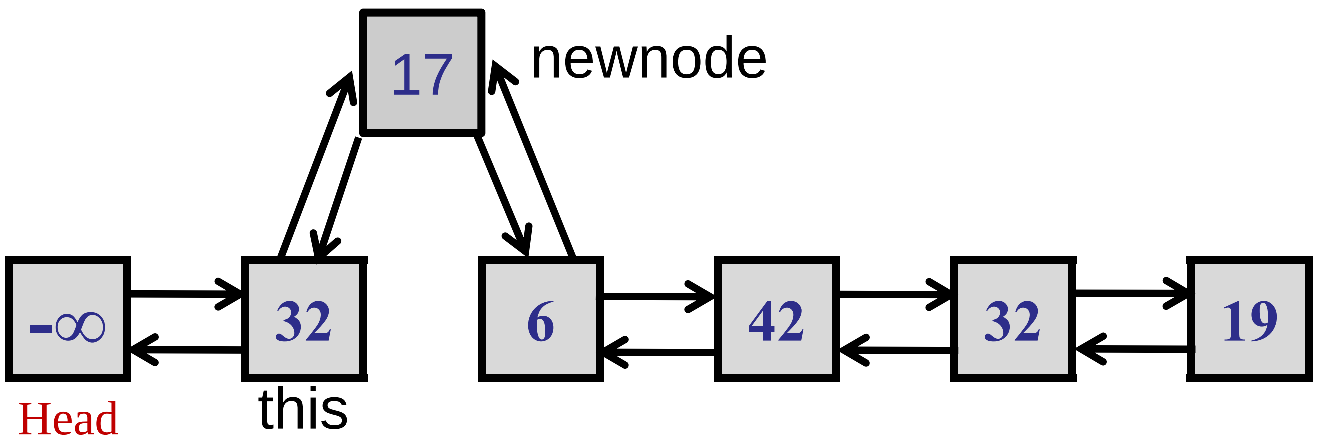
Pass reference to this object.



Inserting a ListNode

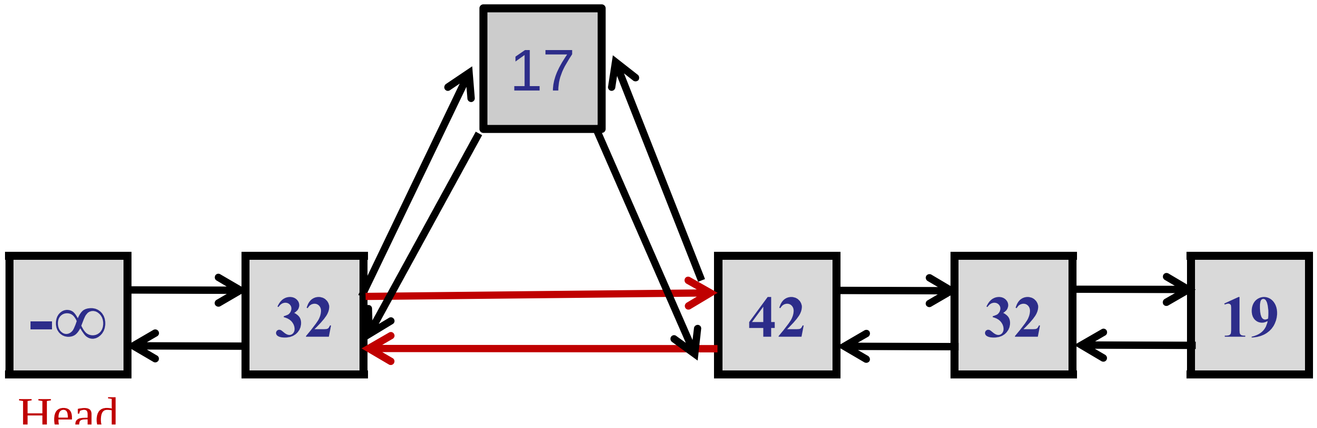
```
public void insertAfter(ListNode<TData> newNode)
{
    if (newNode == null){
        return;
    }
    newNode.setPrevious(this);
    newNode.setNext(m_next);
    if (m_next != null){
        m_next.setPrevious(newNode);
    }
    setNext(newNode);
}
```

Can we do this first in this section?



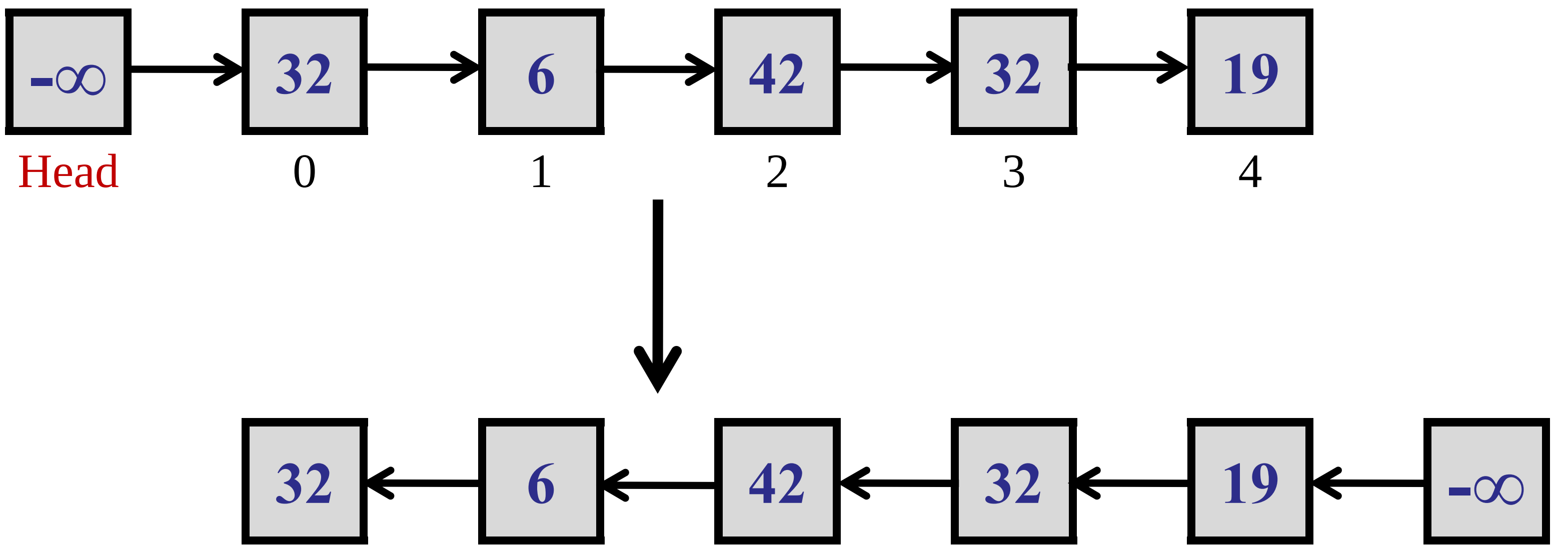
Deleting a ListNode

```
void delete(){
      if (m_prev != null){
            m_prev.setNext(m_next);
      }
      if (m_next != null){
            m_next.setPrev(m_prev);
      }
      m_next = null;
      m_prev = null;
}
```



Puzzle Break

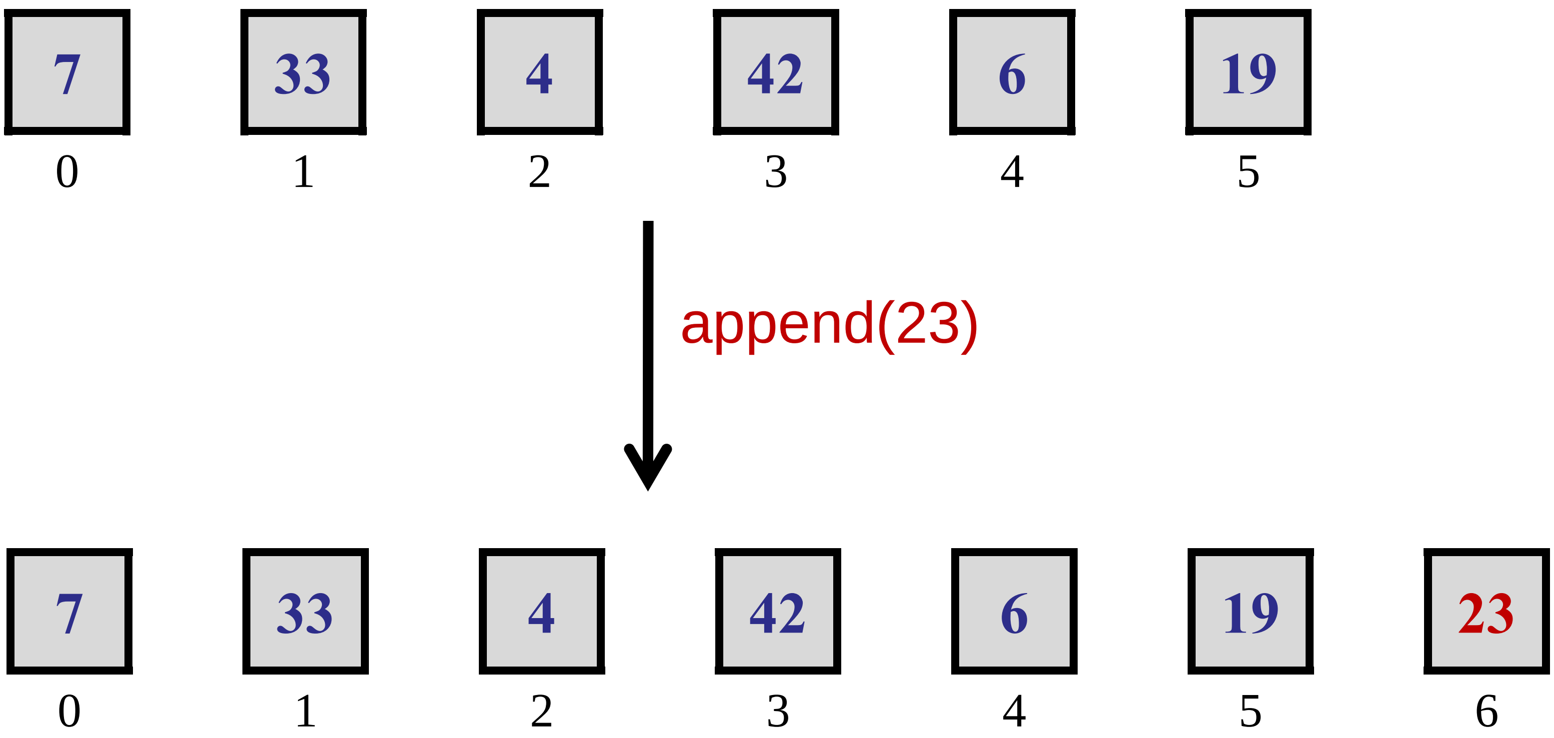
```
void reverseList(ListNode<Key> list){
     next = list.getNext();
     reverseList(next);
     next.setNext(list);
     list.setNext(null);
}
```



Linked Lists

Basic list functionality:

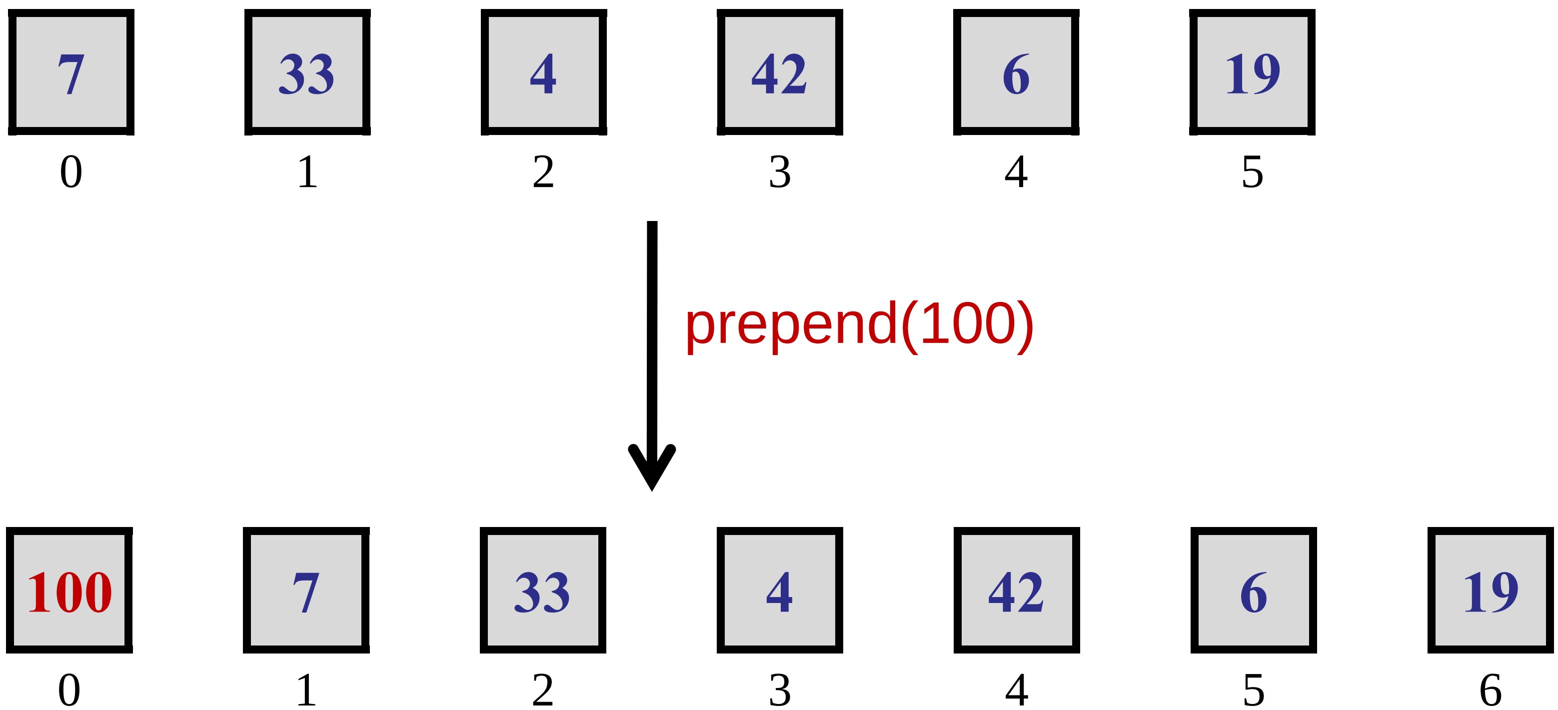
Add to end of list



Linked Lists

Basic list functionality:

Add to beginning of list



Linked Lists

Basic list functionality:

Get the k^{th} element from list

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

get(3) = 42

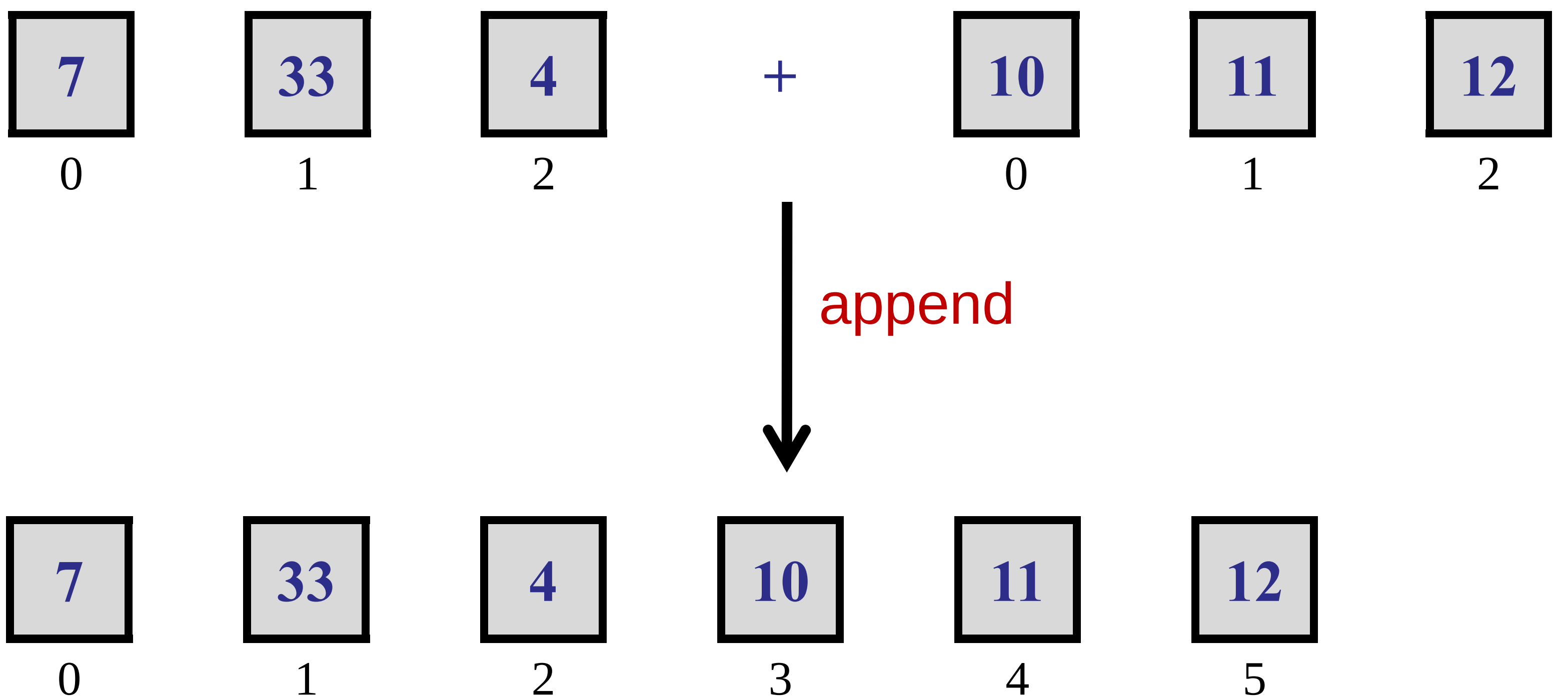
get(0) = 7

get(6) = ??

Linked Lists

Basic list functionality:

Concatenate two lists



Linked List

- Where to put the list functionality?
  - In the ListNode?

- How do we hide the list implementation?
  - Users don't care if it's a linked list or some other type of list.
  - The list should just work…

Linked List Implementation

Type for data

```
public class LinkedList<TData> implements IDictionary<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;
    }
```

Sentinel / Dummy values

Head and tail of the list

Initialize all in constructor

Adding an element to the linked list

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

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

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

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

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
      ...
}
```

```
interface IDictionary<TData> {
      . . .
      void append(IDictionary<Tdata> addList);
}
```

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
    ...
}
```

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
      ...
}
```

Problem:

- What if `addList` is implemented in an array?
- Option 1: Copy the entire list. [O(n)]
- Option 2: Only append linked lists. [O(1)]

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
      if (!addList instanceof LinkedList){
            // Error!  Not a linked list.
      }
      LinkedList<TData> addLink = addList;
      ...
}
```

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
      if (!addList instanceof LinkedList){
            // Error!  Not a linked list.
      }
      LinkedList<TData> addLink = addList;
      ...
}
```

Oops! Can't assign IDictionary to LinkedList

Linked List Implementation

Append: adds one list to another

```
void append(IDictionary<TData> addList){
     if (!addList instanceof LinkedList){
          // Error!  Not a linked list.
     }
     LinkedList<TData> addLink;
     addLink = (LinkedList<TData>)(addList);
     ...
}
```

Cast: unsafe operation?

Appending a linked list

```
public void append(IDictionary<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… error.

We know it is
a LinkedList….

Beware instanceof

In general, this is a bad design!

Beware instanceof

In general, this is a bad design!

- Fragile: breaks in unexpected ways.
- Violates the contract: it promises to append lists, but doesn't.

Beware instanceof

What is wrong here?

```
void exercise(Animal myPetRoofus){
    if (myPetRoofus instanceof Dog){
        Dog d = (Dog)myPetRoofus;
        d.run();
    }
    else if (myPetRoofus instanceof Fish){
        Fish f = (Fish)myPetRoofus;
        f.swim();
    }
```

What if you implement new classes of GoldFish and GoldenRetriever?

Beware instanceof

Polymorphism!

```
void exercise(Animal myPetRoofus){
     myPetRoofus.move();
}
```

Override `move` in Dog/Fish class to run/swim.

Linked List Implementation

Append: adds one linked list to another

```
void append(IDictionary<TData> addList){
      ...
}
```

Problem:

- What if `addList` is implemented in an array?
- Option 1: Copy the entire list. [O(n)]
- Option 2: Only append linked lists. [O(1)]

Linked List Implementation

Append: adds one linked list to another

```
void append(IDictionary<TData> addList){
      ...
}
```

One solution:

- Add to the interface:
  - toLinkedList()
  - toArray ()
- Append requests the Dictionary be translated…

Linked List Implementation

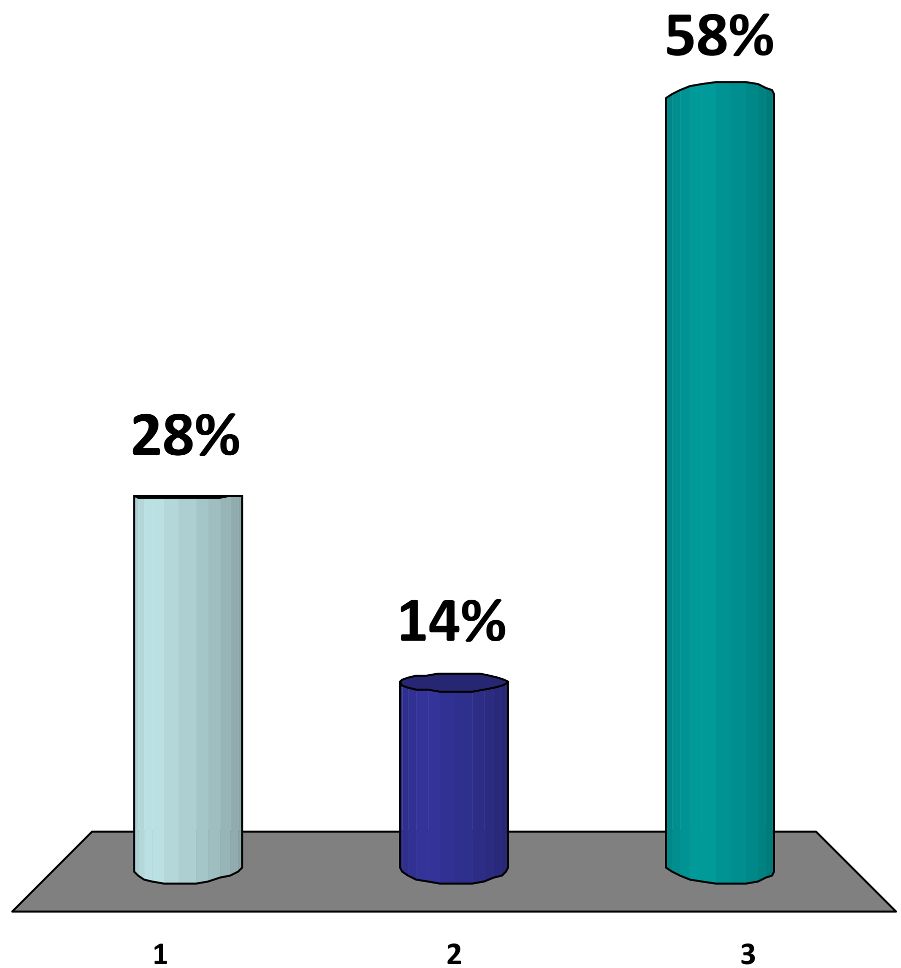
Dictionary Interface

interface IDictionary<Key extends Comparable<Key>, Value>

`void`	`insert(Key k, Value v)`	*insert (k,v) into table*
`Value`	`search(Key k)`	*get value paired with k*
`Key`	`successor(Key k)`	*find next key > k*
`Key`	`predecessor(Key k)`	*find next key < k*
`void`	`delete(Key k)`	*remove key k (and value)*
`boolean`	`contains(Key k)`	*is there a value for k?*
`int`	`size()`	*number of (k,v) pairs*

How to implement dictionary?

1. Sorted linked list
2. Unsorted linked list
3. Does not matter



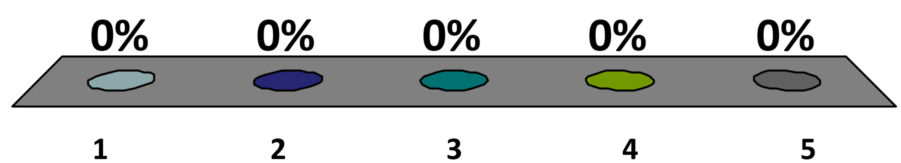
Dictionary Implementation

- Unsorted list:
  - Search entire list: O(n)
  - Insert at head (or tail): O(1)

- Sorted list:
  - Search (almost) entire list: O(n)
  - Insert in middle: O(n)

Can we use binary search?

1. Yes: as usual.
2. No: must start at head or tail.
3. No: keys may not be Comparable
4. No: requires too much space
5. None of the above.



Dictionary Implementation

- Unsorted list:
  - Search entire list: O(n)
  - Insert at head (or tail): O(1)

- Sorted list:
  - Search (almost) entire list: O(n)
  - Insert in middle: O(n)

Dictionary Implementation

- Unsorted list:
  - Search entire list: O(n)
  - Insert at head (or tail): O(1)
  - Successor: O(n)

- Sorted list:
  - Search (almost) entire list: O(n)
  - Insert in middle: O(n)
  - Successor: ??

Dictionary Implementation

Sorted list:

- Search (almost) entire list: O(n)
- Insert in middle: O(n)
- Successor: O(n)

```
Key successor(Key k){
     ListNode keyNode = list.search(k);
     return keyNode.getNext().getKey();
}
```

Dictionary Implementation

Sorted list:

- Search (almost) entire list: O(n)
- Insert in middle: O(n)
- Successor: O(1)

```
Key successor(ListNode keyNode){
    // Directly return successor
    return keyNode.getNext().getKey();
}
```

Searching the Dictionary

```
Value search(Key key){
     ListNode<Key, Value> current = m_head;
     while (current != null) && (current != m_tail) {
          if (current.m_key.compareTo(key)==0)
               return current.m_value;
          else if (current.m_key.compareTo(key)>0)
               return null;
          current = current.m_next;
     }
     return null;
}
```

Dictionary Implementation

Sorted list:

- Search (almost) entire list: O(n)
- Insert in middle: O(n)
  - Search for location to insert
  - Append new ListNode
- Successor: O(n)
  - Search for key in list
  - Return next ListNode

Linked List Implementation

Dictionary Interface

interface IDictionary<Key extends Comparable<Key>, Value>

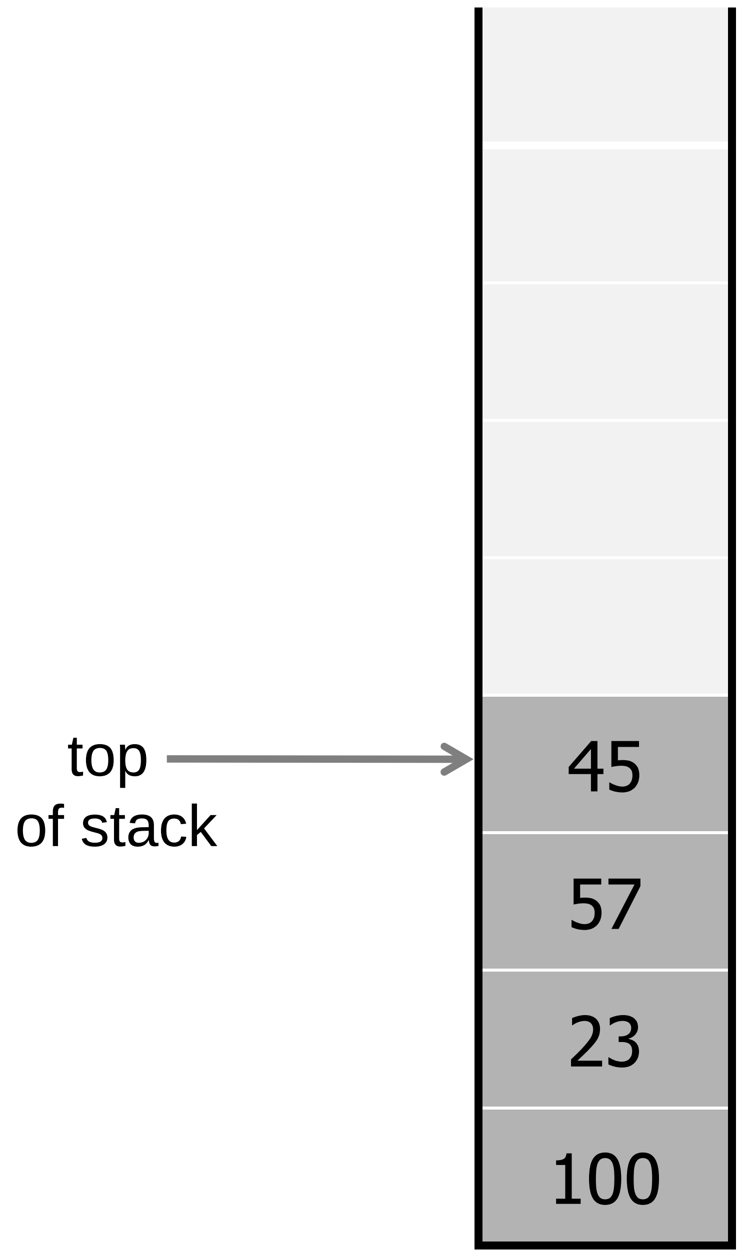
void	insert(Key k, Value v)	*insert (k,v) into table*
Value	search(Key k)	*get value paired with k*
Key	successor(Key k)	*find next key > k*
Key	predecessor(Key k)	*find next key < k*
void	delete(Key k)	*remove key k (and value)*
boolean	contains(Key k)	*is there a value for k?*
int	size()	*number of (k,v) pairs*

Abstract Data Types

Stack

Interface:

- void push(element x)
- element pop()
- empty()



Exercise:

Implement Stack using a LinkedList.

Abstract Data Types

Queue

Interface:

- void enqueue(element x)
- element dequeue()

Exercise:

Implement Queue using a LinkedList.

Iterators

Dictionary

What if I want cheaper successor queries?

Iterators

Dictionary

What if I want to list everything in the dictionary in sorted order?

```
for (word in dictionary) {
      // Print word
      System.out.println(word);
}
```

Iterators

List or Bag or Stack or Queue

- Implemented as Linked List or in array

```
for (item in list) {
      // operate on item
      process(item);
}
```

Iterators

Iterator interface

interface	**Iterator<Type>**	
boolean	hasNext()	*true: if there are more elements* *false: if no more elements in list*
Type	next()	*returns next element*
void	remove()	*Optional*

Iterators

Iterator interface

```
interface  Iterator<Type>
```

`boolean`	`hasNext()`	*true: if there are more elements* *false: if no more elements in list*
`Type`	`next()`	*returns next element*
`void`	`remove()`	*Optional*

```
interface  Iterable<Type>
```

`Iterator<Type>`	`iterator()`	*returns an iterator for the collection*

Iterator Example

class ListOfStrings can be iterated:

```
class ListOfStrings implements Iterable<String> {

	Iterator<String> iterator(){
		Iterator<String> iter = new SListIterator(this);
		return iter;
	}
}
```

Iterator Example

Using an iterator:

```
Iterator<String> stackIterator = myStack.iterator();

while (stackIterator.hasNext()){
      String nextString = stackIterator.next();
      System.out.println(nextString);
}
```

Iterator Example

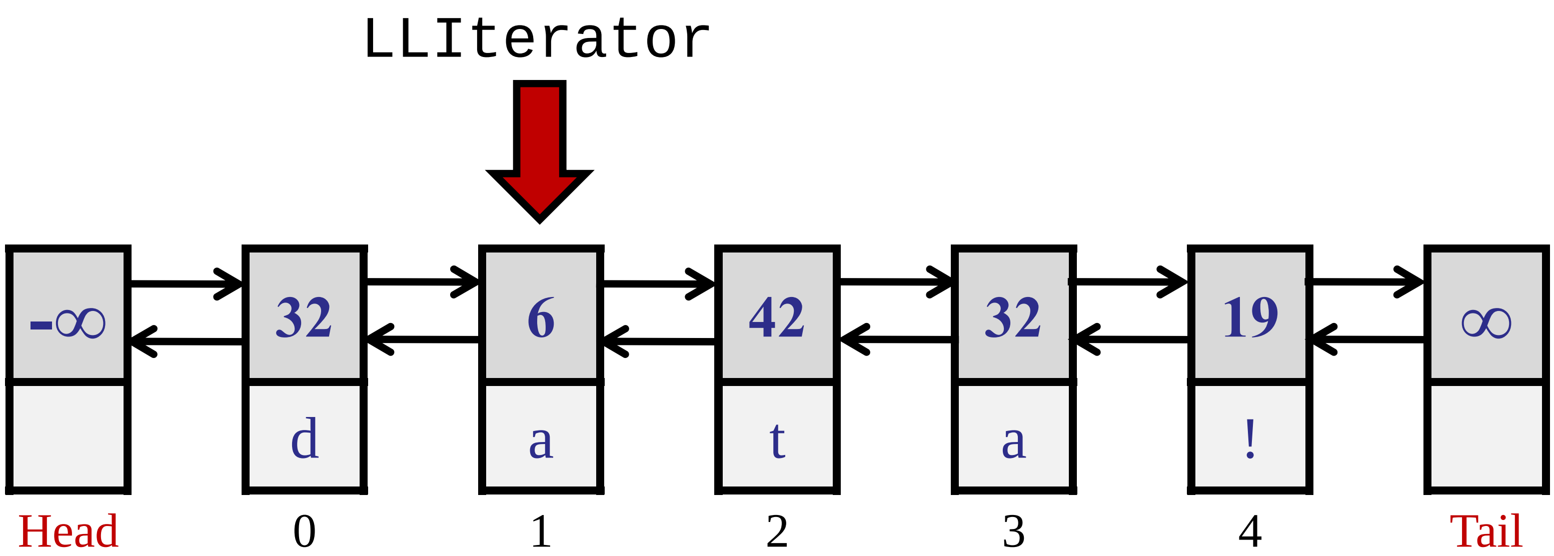
Using an iterator: enhanced for loops

```
Iterator<String> stackIterator = myList.iterator();


for (String s : stackIterator){
      System.out.println(s);
}
```

Linked List Iterator

Implement a LinkedListIterator:

- Reference (pointer) to some node in the list.
- Goal: hide implementation
- Problem: cannot simple use ListNode reference.



Iterator Implementation

Inner (nested) class

```
public class LinkedList<T> {
      ListNode m_head;
      ... // Code for List goes here
      private class LLIterator implements Iterator<T> {
            private ListNode current = m_head.getNext();
            public boolean hasNext() {return current != null;}
            public T next(){
                  if (!hasNext())
                        throw new NoSuchElementException();
                  T item = current;
                  current = current.getNext();
                  return item;
            }
      }
}
```

Nested Classes

- Inner Class
  - Defined within another class
  - Associated with an instance.
  - Has access to private variables within class.

- Static Nested Class
  - Defined with another class
  - Associated with a class.
  - Otherwise, unrelated.
  - Mainly for organization.

Iterator Implementation

Inner (nested) class

```
public class Outer {
        int m_number;

        public class Inner {
                public int getNumber() {return m_number;}
        }
}

Outer.Inner foo = new Outer.Inner();
```

No!! Which instance is this for?

Iterator Implementation

Inner (nested) class

```
public class Outer {
        int m_number;

        public static class Inner {
                public int getNumber() {return m_number;}
        }
}

Outer.Inner foo = new Outer.Inner();
```

No!! Which instance is this for?

Yes!! Outer.Inner is static.

Iterator Implementation

Inner (nested) class

```
public class Outer {
        int m_number;

        public static class Inner {
                public int getNumber() {return 17;}
        }
}

Outer.Inner foo = new Outer.Inner();
```

Yes!! Outer.Inner is static.

Iterator Implementation

Inner (nested) class

```
public class Outer {
        int m_number = 17;

        public class Inner {
                public int getNumber() {return m_number;}
        }

        public Inner getInner() {return new Inner();}
}

Outer foo = new Outer();
Outer.Inner bar = foo.getInner();
System.out.println(bar.getNumber());
```

Iterator Implementation

Inner (nested) class

```
public class LinkedList<T> {
      ListNode m_head;
      ... // Code for List goes here
      private class LLIterator implements Iterator<T> {
            private ListNode current = m_head.getNext();
            public boolean hasNext() {return m_head != null;)
            public T next(){
                  if (!hasNext())
                        throw new NoSuchElementException();
                  T item = current;
                  current = current.getNext();
                  return item;
            }
      }
}
```

List Iterator Implementations

Exercise:

- Implement a FixedLengthList that is Iterable.
- Implement an Iterator for your list.

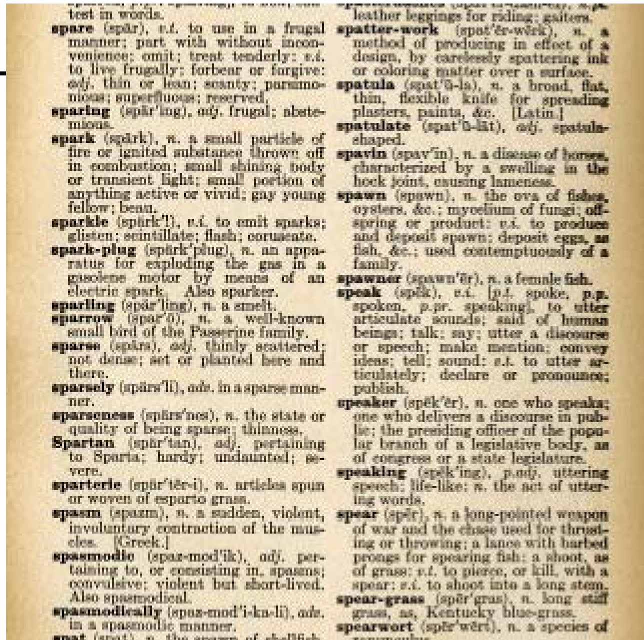
Today's Plan

1. Exceptions and error handling

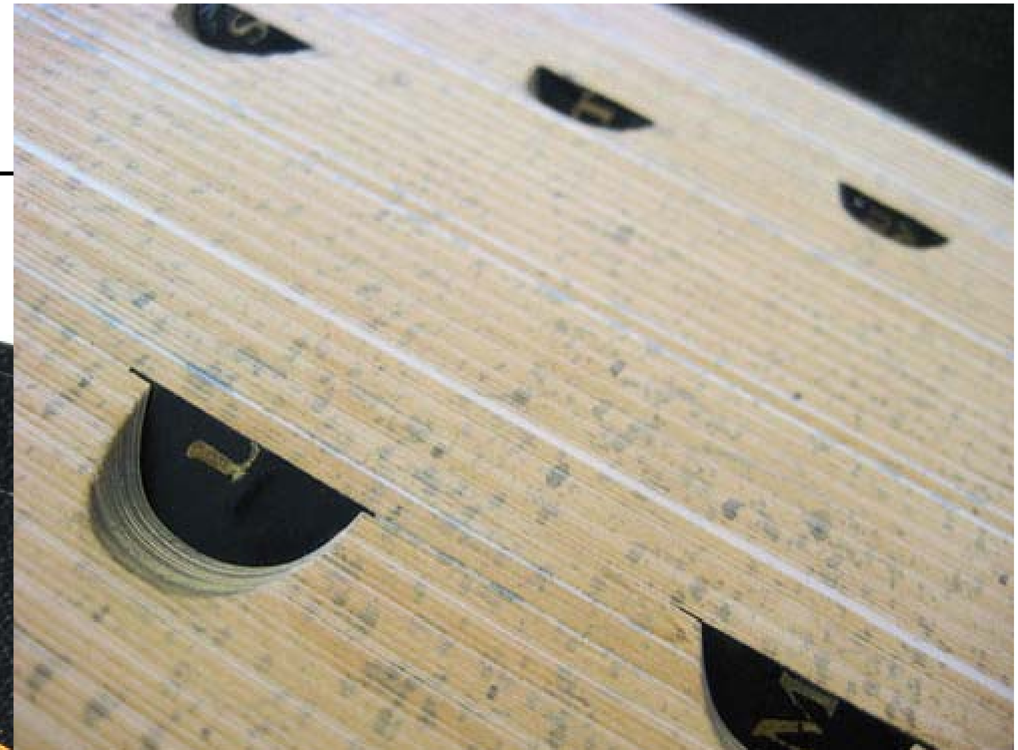
2. Problem: Scheduling Airplanes

3. Abstract Data Types:
   - Symbol Tables
   - Dictionaries

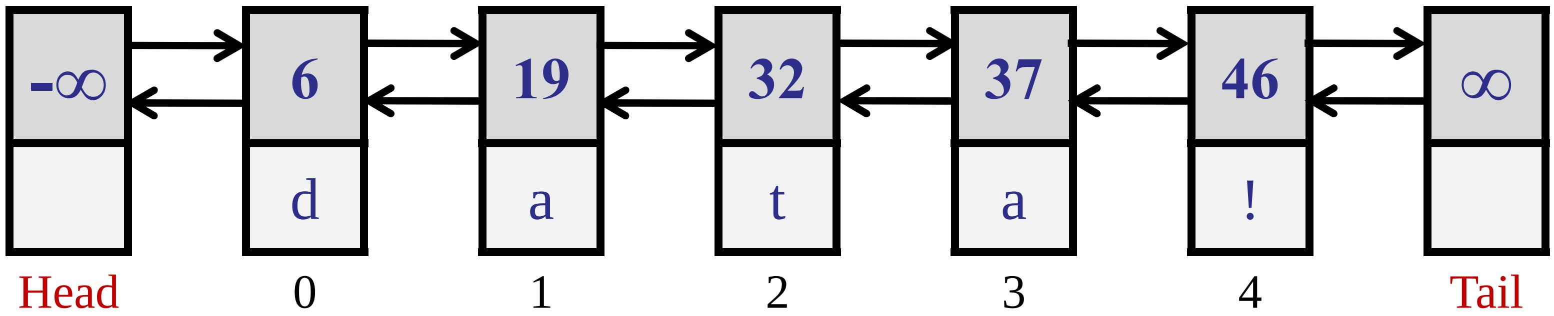
4. Linked Data Structures



test in words.
spare (spâr), *v.t.* to use in a frugal manner; part with without inconvenience; omit; treat tenderly: *v.i.* to live frugally; forbear or forgive: *adj.* thin or lean; scanty; parsimonious; superfluous; reserved.
sparing (spâr'ing), *adj.* frugal; abstemious.
spark (spärk), *n.* a small particle of fire or ignited substance thrown off in combustion; small shining body or transient light; small portion of anything active or vivid; gay young fellow; beau.
sparkle (spärk'l), *v.i.* to emit sparks; glisten; scintillate; flash; coruscate.
spark-plug (spärk'plug), *n.* an apparatus for exploding the gas in a gasolene motor by means of an electric spark. Also sparker.
sparling (spär'ling), *n.* a smelt.
sparrow (spar'ō), *n.* a well-known small bird of the Passerine family.
sparse (spärs), *adj.* thinly scattered; not dense; set or planted here and there.
sparsely (spärs'li), *adv.* in a sparse manner.
sparseness (spärs'nes), *n.* the state or quality of being sparse; thinness.
Spartan (spär'tan), *adj.* pertaining to Sparta; hardy; undaunted; severe.
sparterie (spär'tēr-i), *n.* articles spun or woven of esparto grass.
spasm (spazm), *n.* a sudden, violent, involuntary contraction of the muscles. [Greek.]
spasmodic (spaz-mod'ik), *adj.* pertaining to, or consisting in, spasms; convulsive; violent but short-lived. Also spasmodical.
spasmodically (spaz-mod'i-ka-li), *adv.* in a spasmodic manner.

leather leggings for riding; gaiters.
spatter-work (spat'ēr-wûrk), *n.* a method of producing in effect of a design, by carelessly spattering ink or coloring matter over a surface.
spatula (spat'ū-la), *n.* a broad, flat, thin, flexible knife for spreading plasters, paints, &c. [Latin.]
spatulate (spat'ū-lāt), *adj.* spatula-shaped.
spavin (spav'in), *n.* a disease of horses, characterized by a swelling in the hock joint, causing lameness.
spawn (spawn), *n.* the ova of fishes, oysters, &c.; mycelium of fungi; offspring or product: *v.t.* to produce and deposit spawn; deposit eggs, as fish, &c.; used contemptuously of a family.
spawner (spawn'ēr), *n.* a female fish.
speak (spēk), *v.i.* [*p.t.* spoke, *p.p.* spoken, *p.pr.* speaking], to utter articulate sounds; said of human beings; talk; say; utter a discourse or speech; make mention; convey ideas; tell; sound: *v.t.* to utter articulately; declare or pronounce; publish.
speaker (spēk'ēr), *n.* one who speaks; one who delivers a discourse in public; the presiding officer of the popular branch of a legislative body, as of congress or a state legislature.
speaking (spēk'ing), *p.adj.* uttering speech; life-like: *n.* the act of uttering words.
spear (spēr), *n.* a long-pointed weapon of war and the chase used for thrusting or throwing; a lance with barbed prongs for spearing fish; a shoot, as of grass: *v.t.* to pierce, or kill, with a spear: *v.i.* to shoot into a long stem.
spear-grass (spēr'gras), *n.* long stiff grass, as, Kentucky blue-grass.
spearwort (spēr'wērt), *n.* a species of

Implementing a dictionary, again…

Store keys in a sorted linked list:



Time:

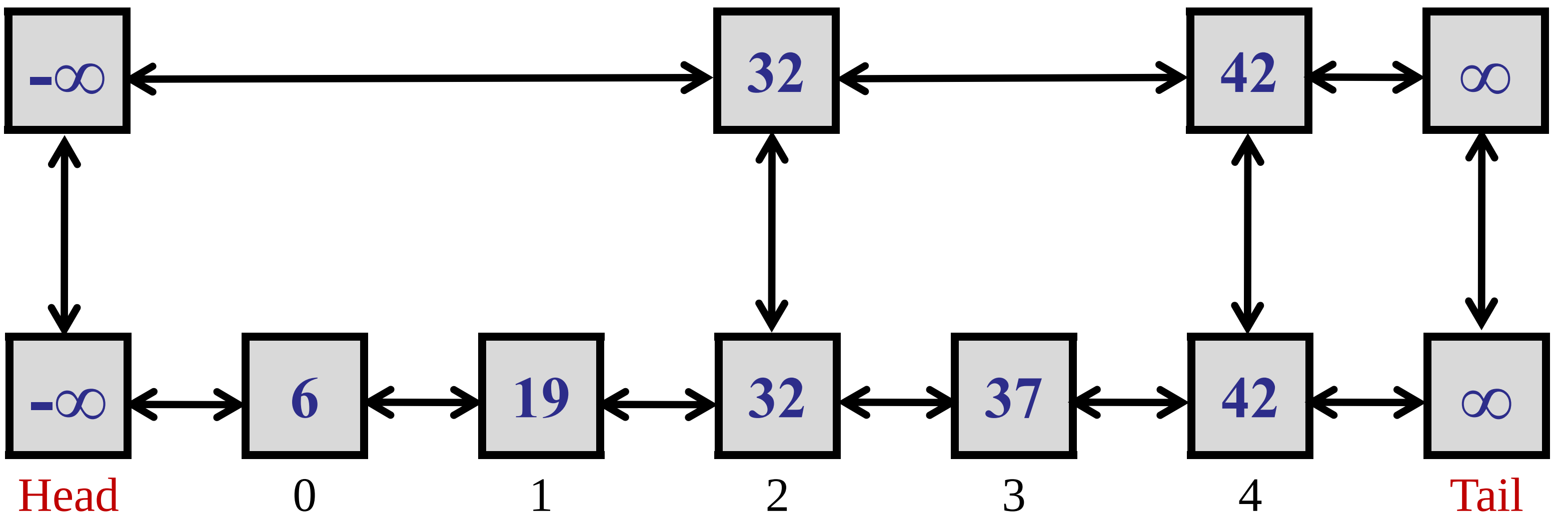
- Search: O(n)
- Insert: O(n)

What if…

What if we use two lists?

– Express train

– Local train



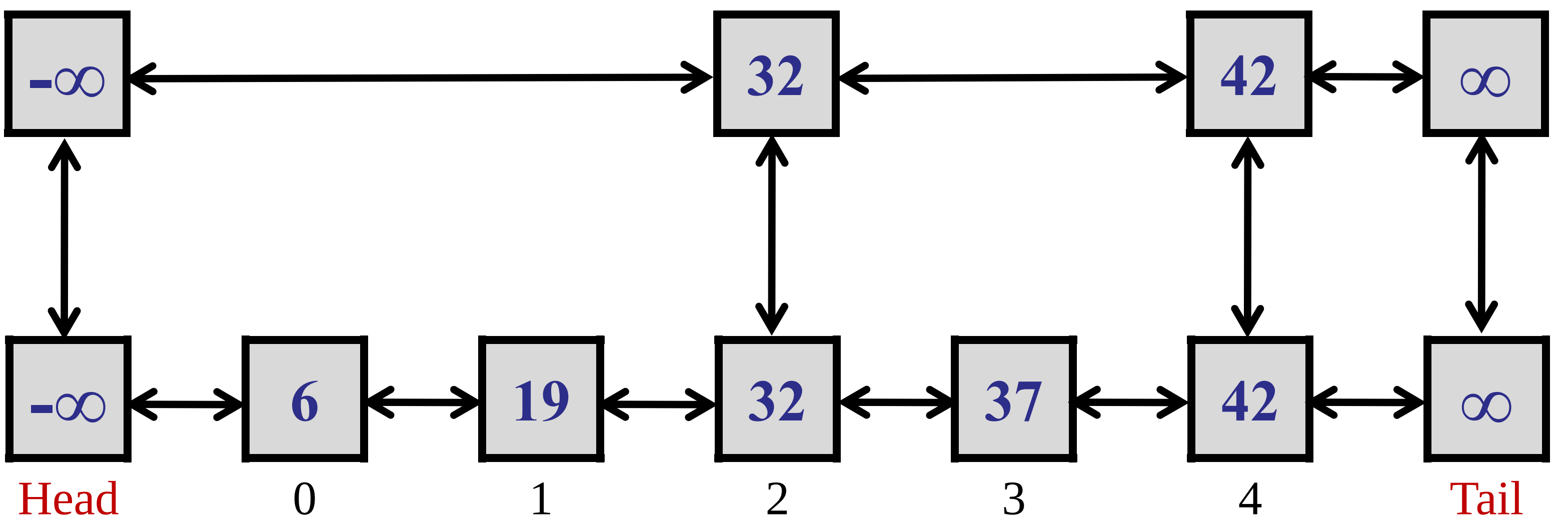
`search(37)` takes only 3 steps!

What if…

Calculation:

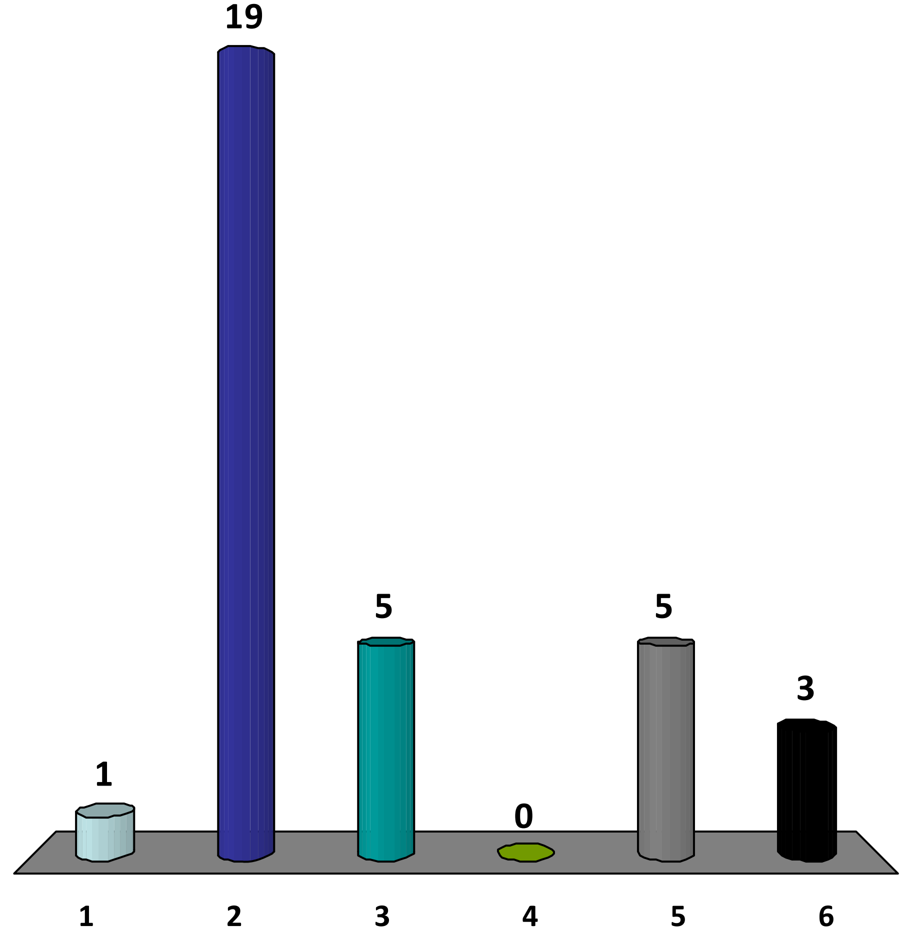
– If the “express” list skips 5 elements per “stop”, then search takes at most:

$$n/5 + 5 \text{ steps}$$



With two lists, 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.

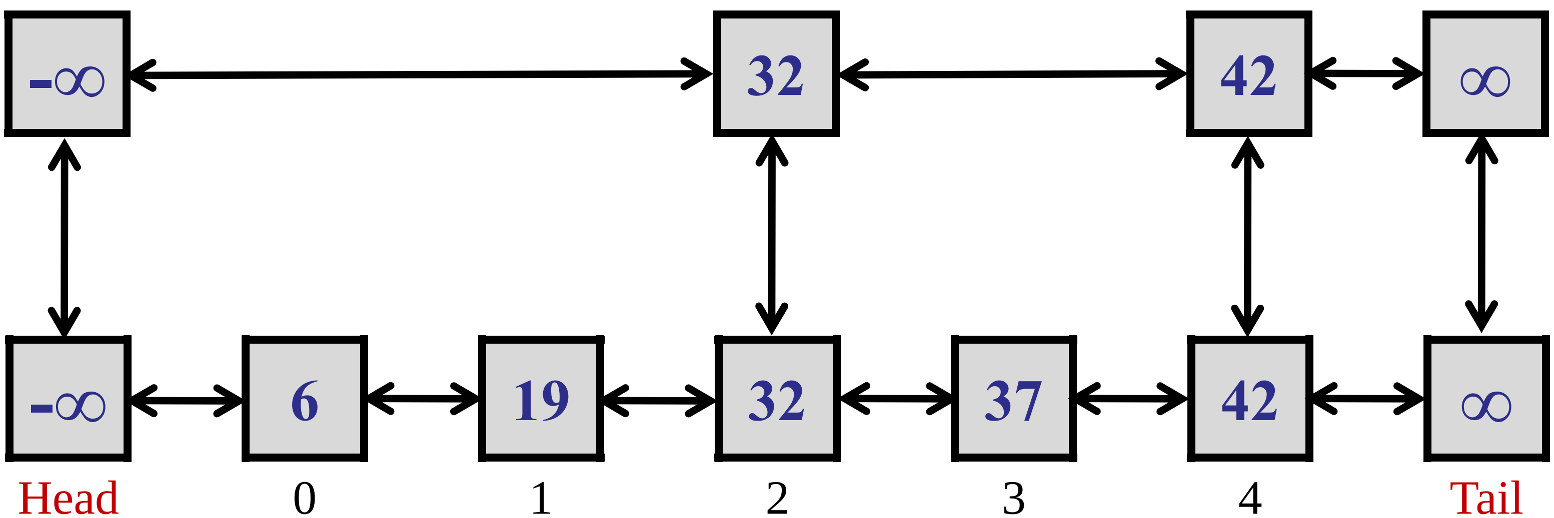


What if…

Calculation:

– 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: $2\sqrt{n}$

– Three lists: $3\sqrt[3]{n}$

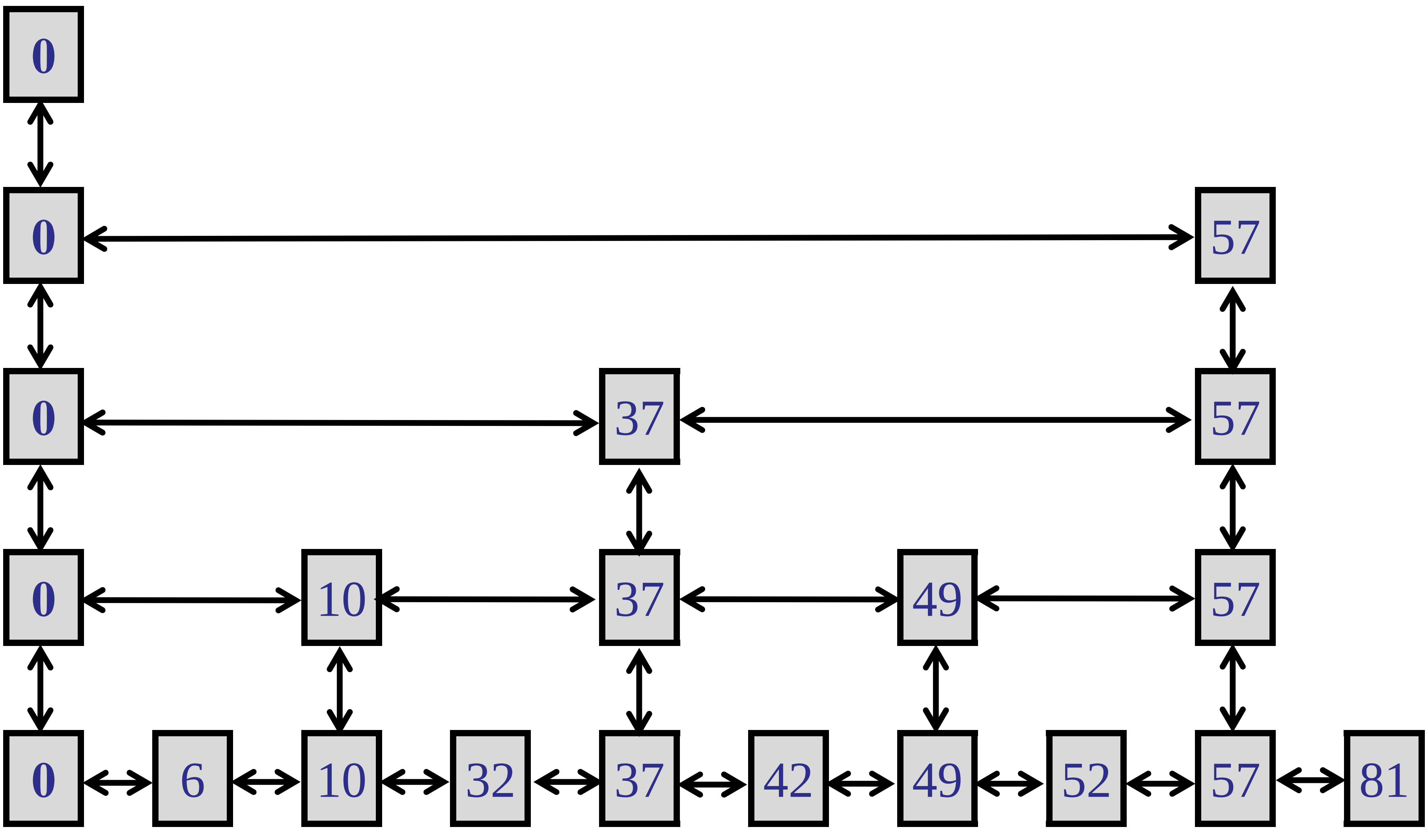
…

– *k* lists: $k\sqrt[k]{n}$

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

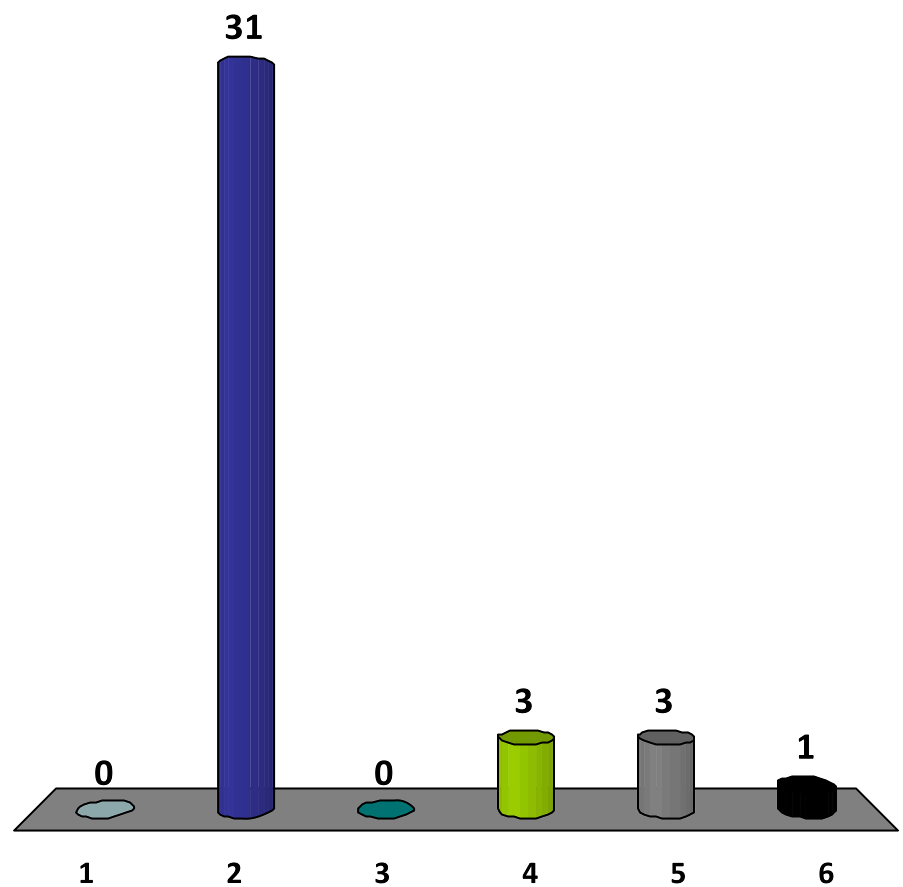
$= 2\log(\mathrm{n})$

Another way to think about it...



How many levels?

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



SkipList Background

Simple randomized, dynamic search structure

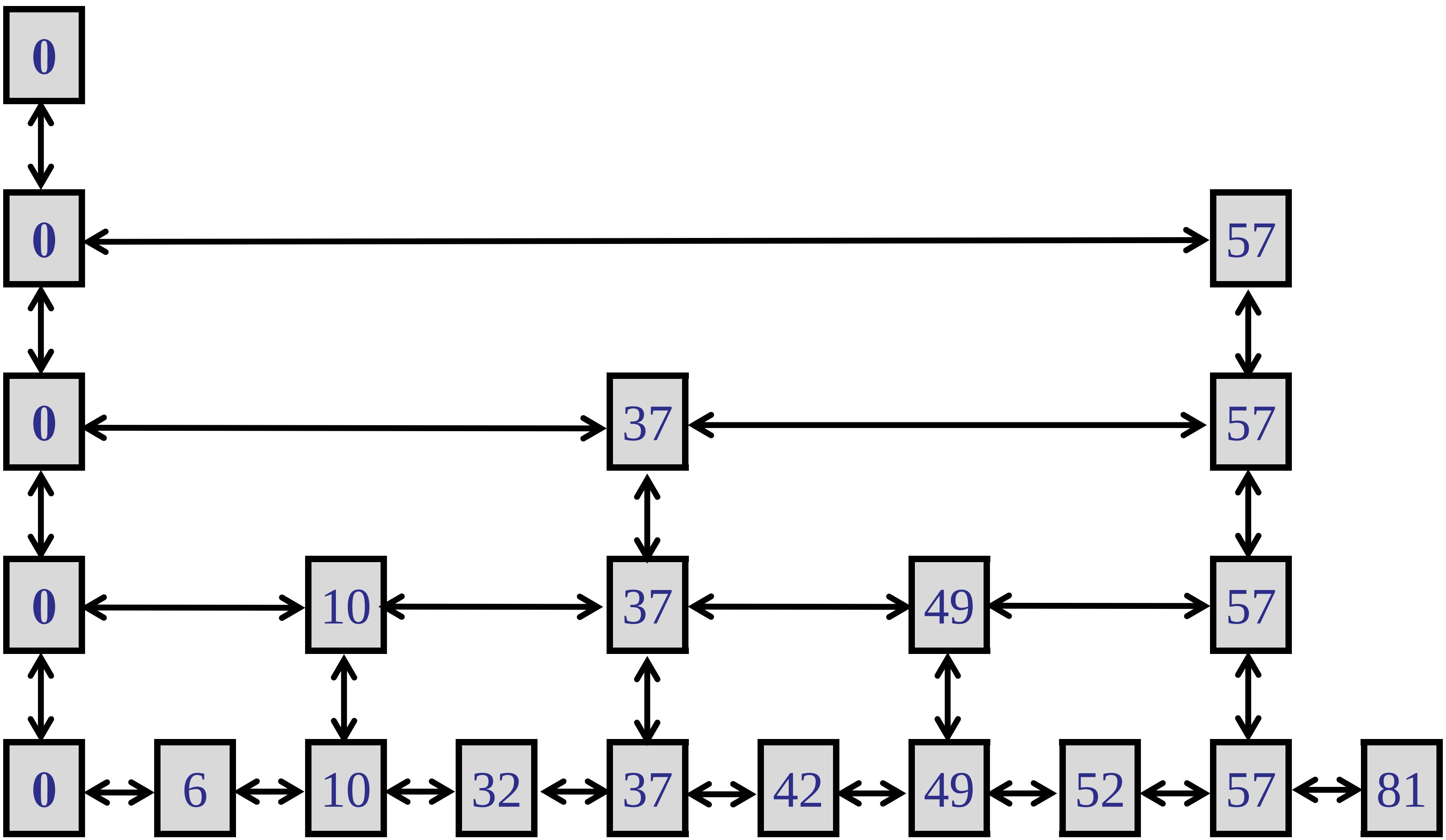
- Invented by William Pugh in 1989
- Easy to implement

Maintains a set of n elements:

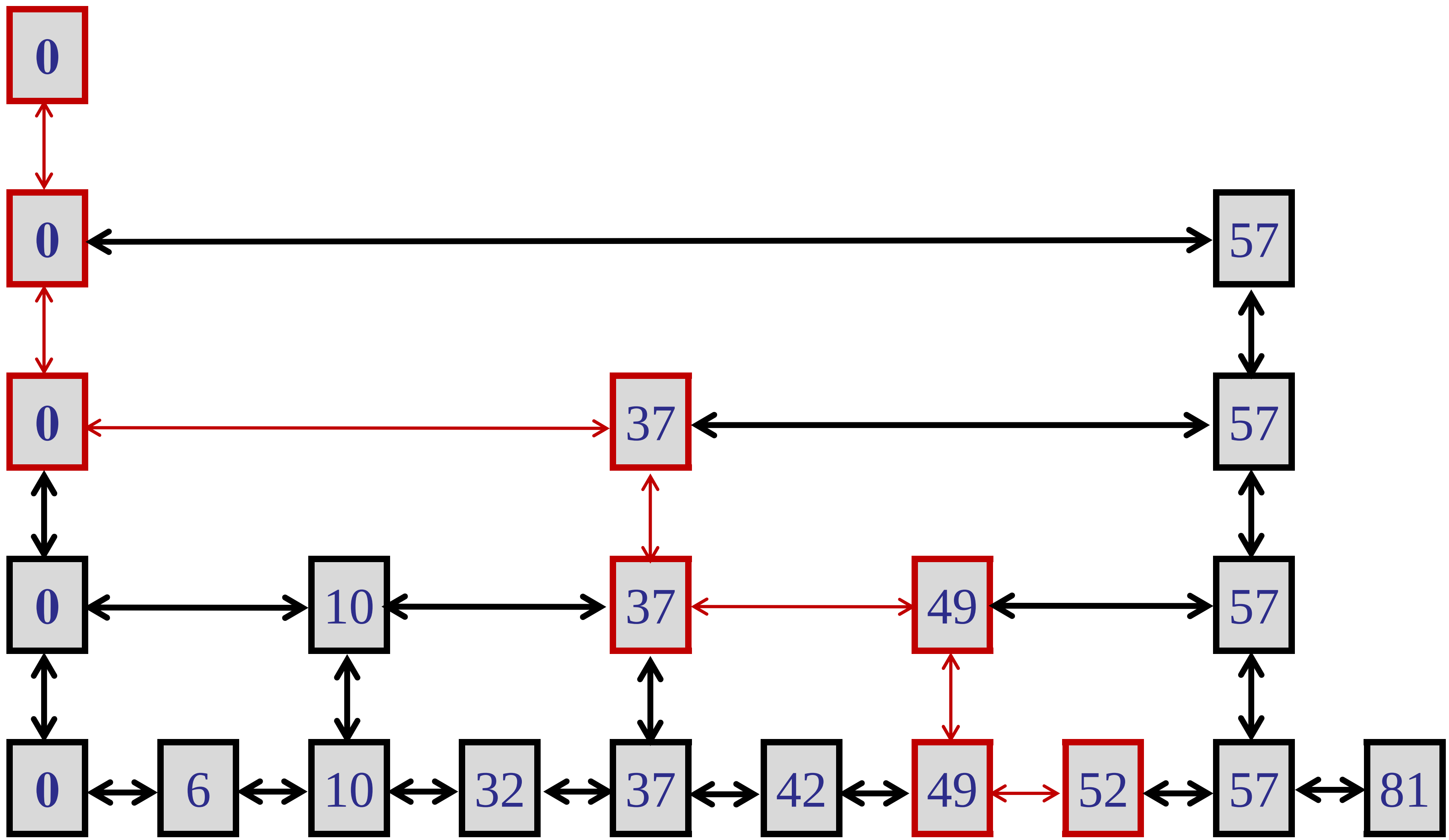
- search: O(log n) time
- insert/delete: O(log n) time

with high probability

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

```
k = 0;
while (!done) {
        Insert element into level k list.
        Flip a fair coin:
                with probability ½: done = true;
                with probability ½: k = k+1;
}
```

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 a perfect distribution.
- Good, on average.
- Really good, *almost always*.
- As usual with randomized algorithms, easy to implement, harder to analyze.

SkipListNode 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;
    }
```

Use linked list implementation.

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 IDictionary with parameterized type.

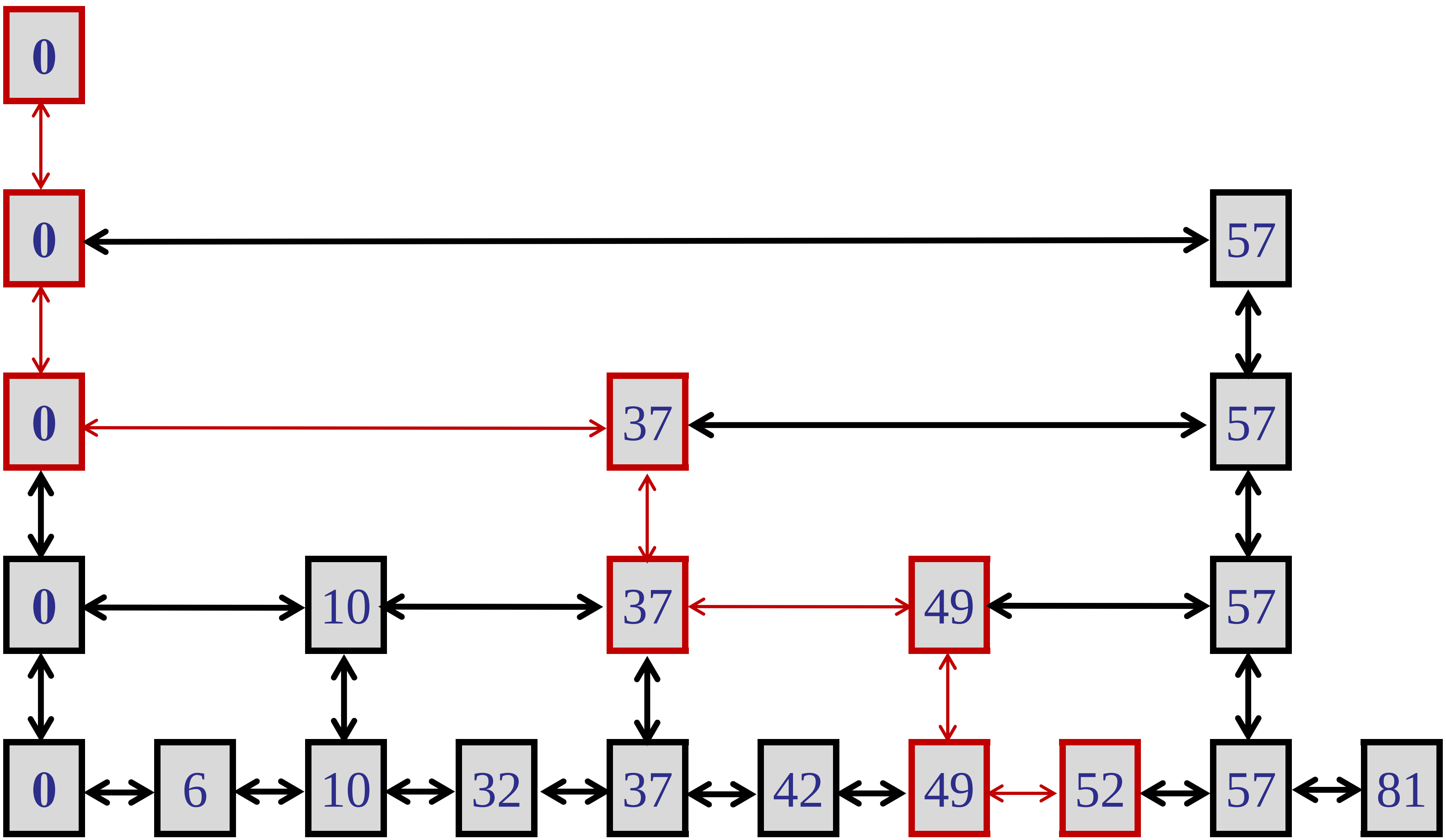
```
public class SkipList<TData> implements IDictionary<TData> {

    /* 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

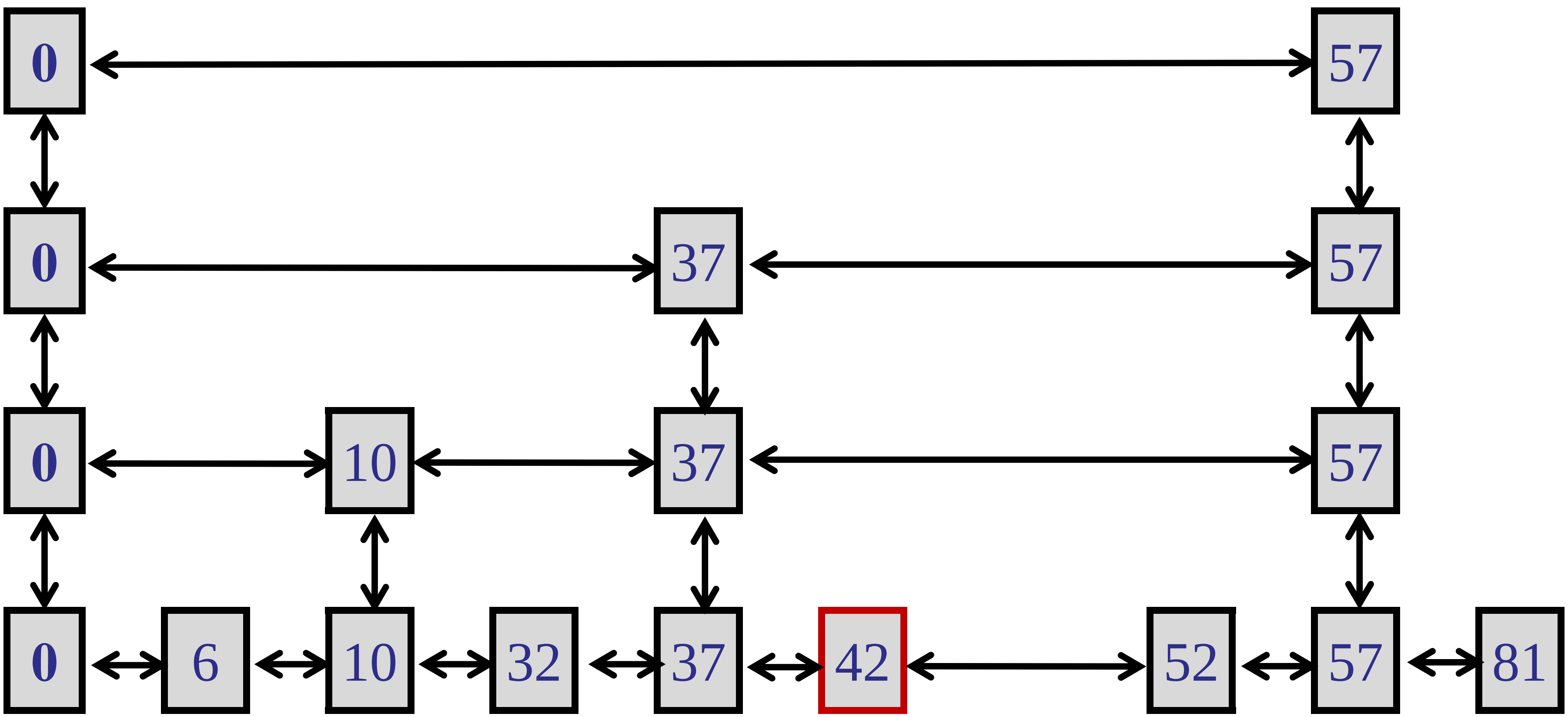
Start at the top list.

```
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;
}
```

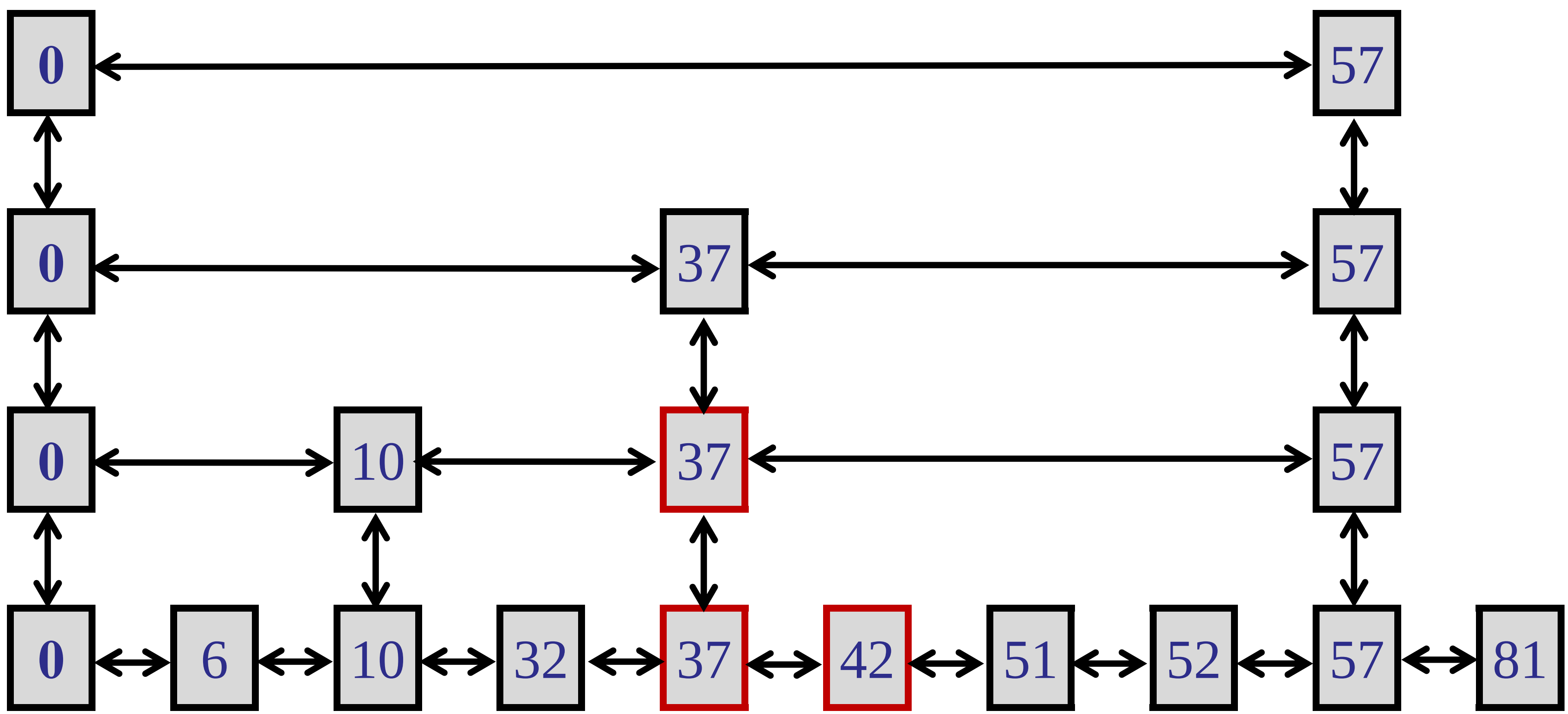
Go right if you can.
Otherwise, go down.

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

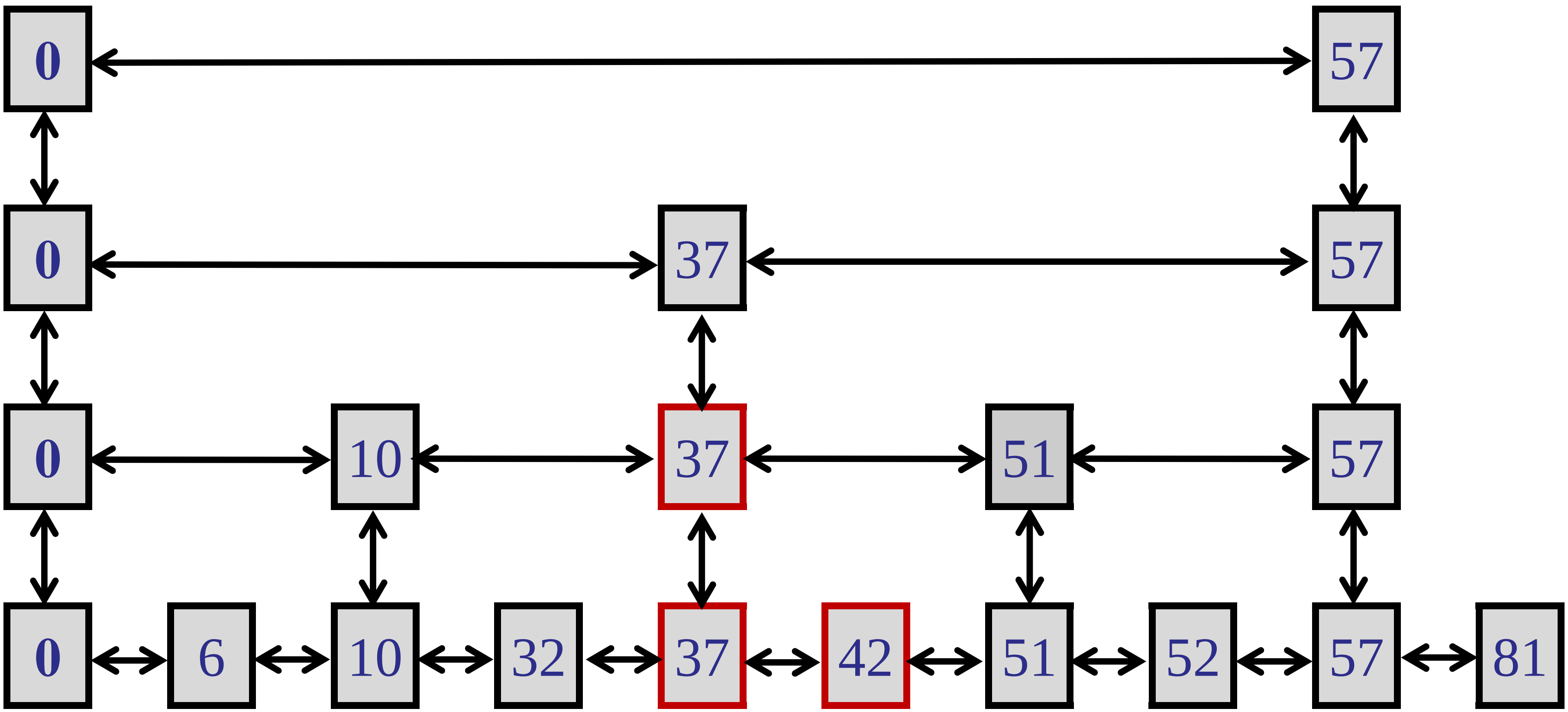
Example: `insert(51)`



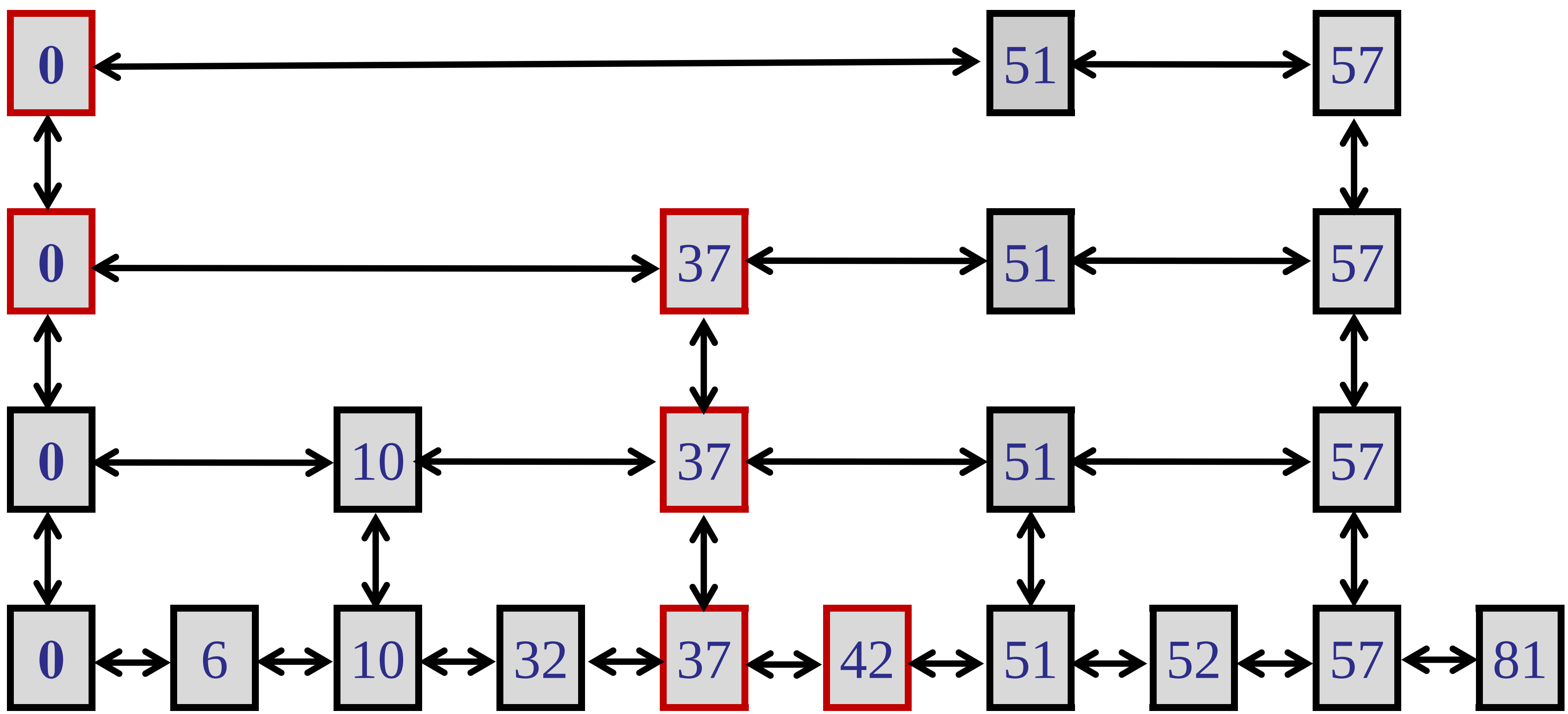
Example: `insert(51)`



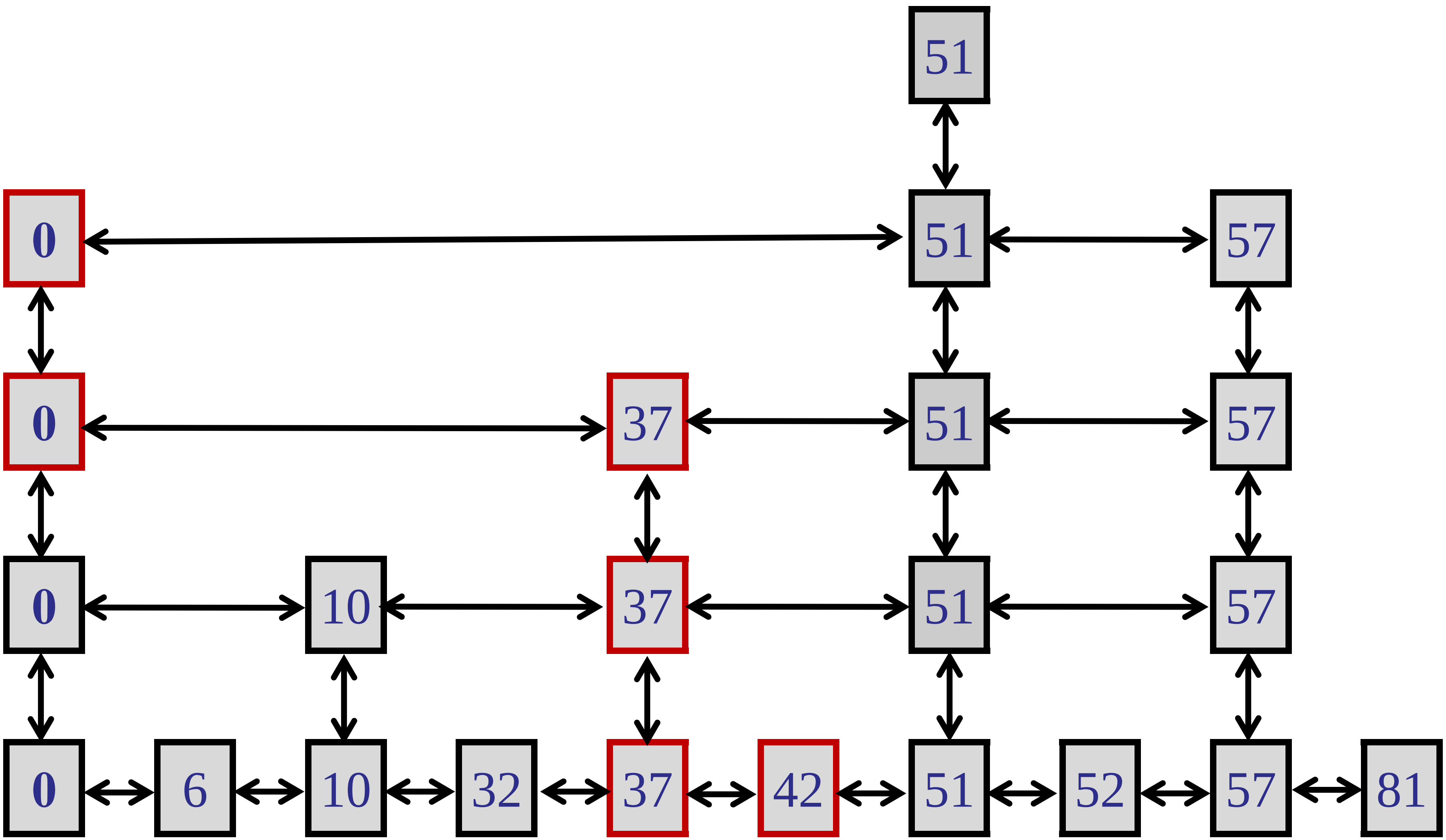
Example: `insert(51)`



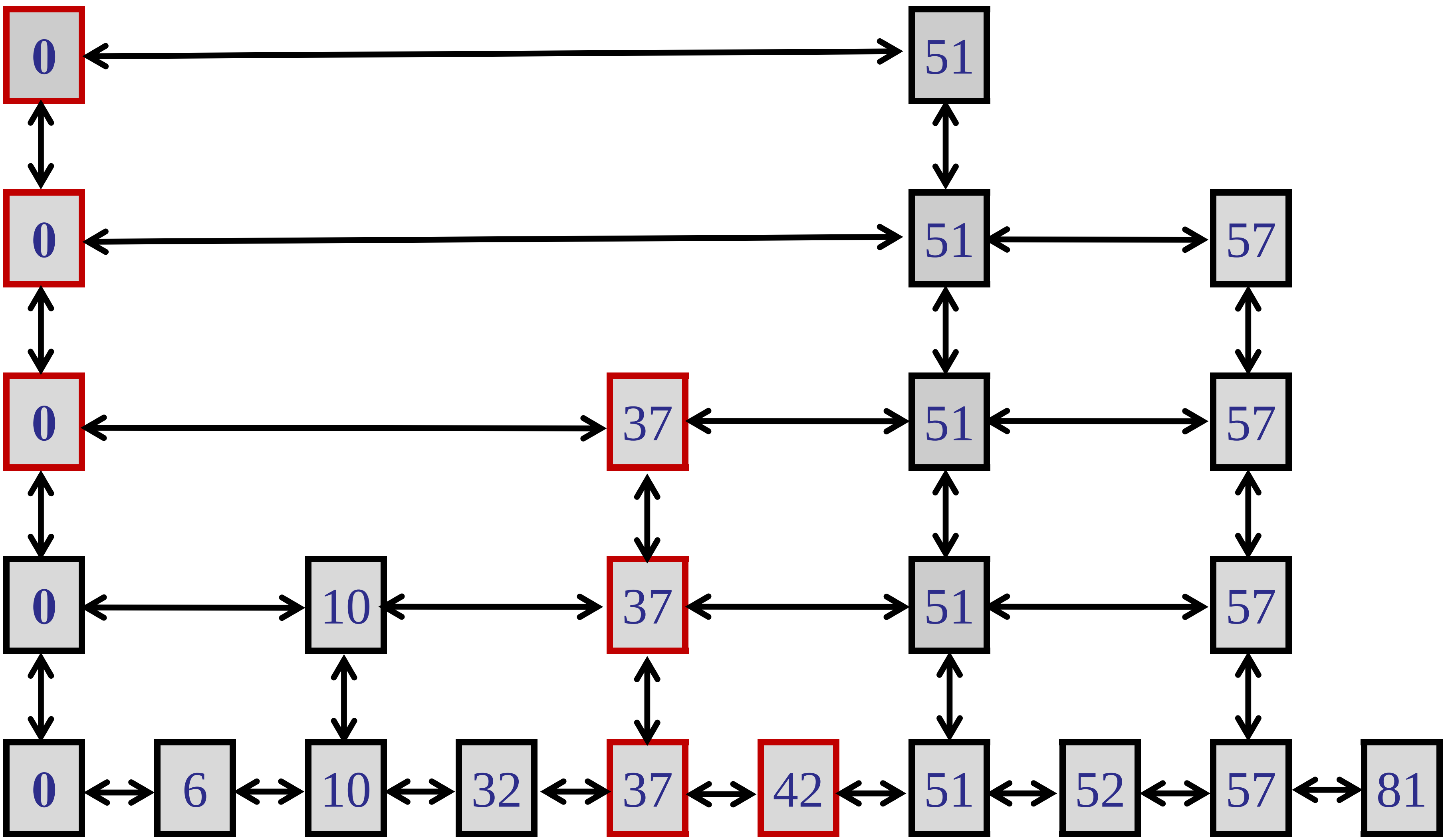
Example: `insert(51)`



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

Insert node in list.

Flip coin.

If heads, go up.

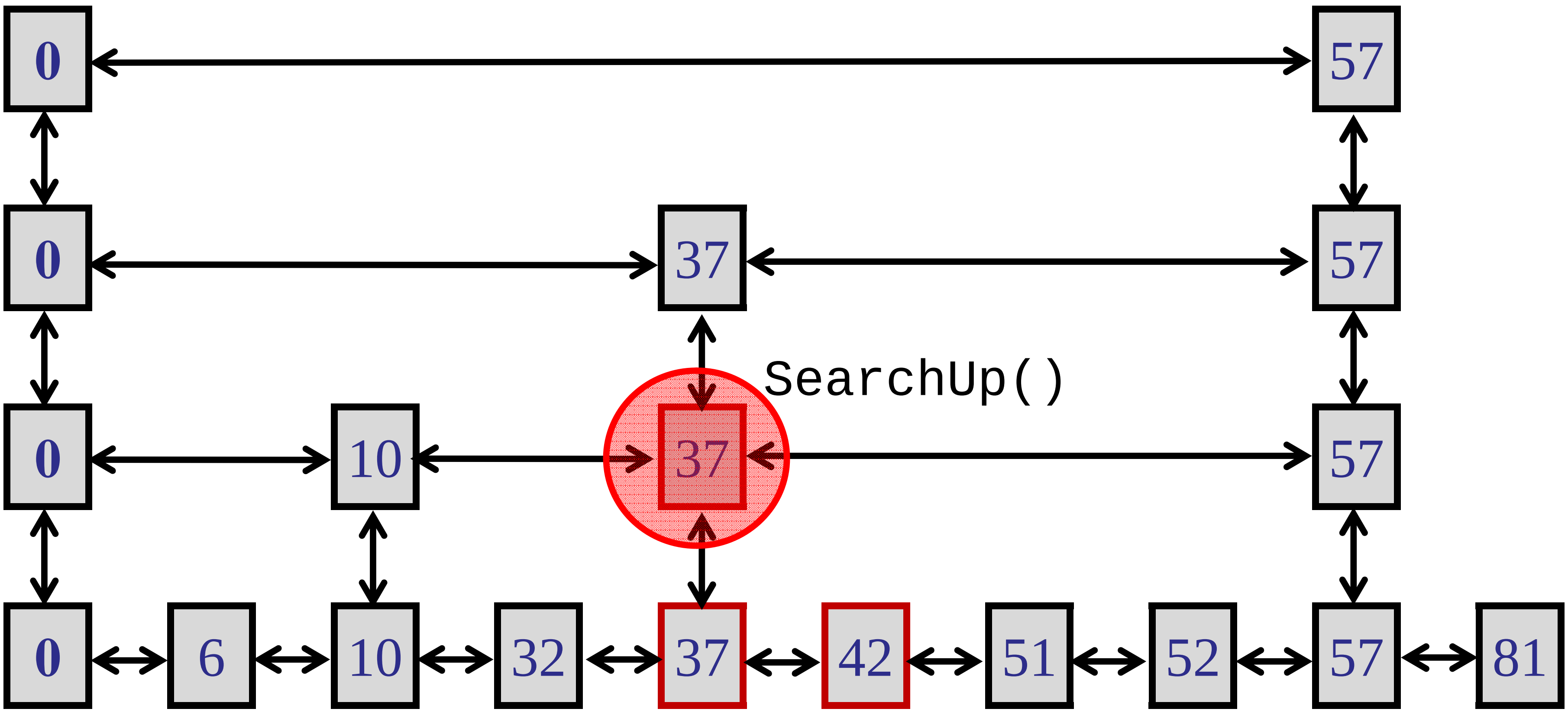
If no up list, add one.

Create new node for next list up.

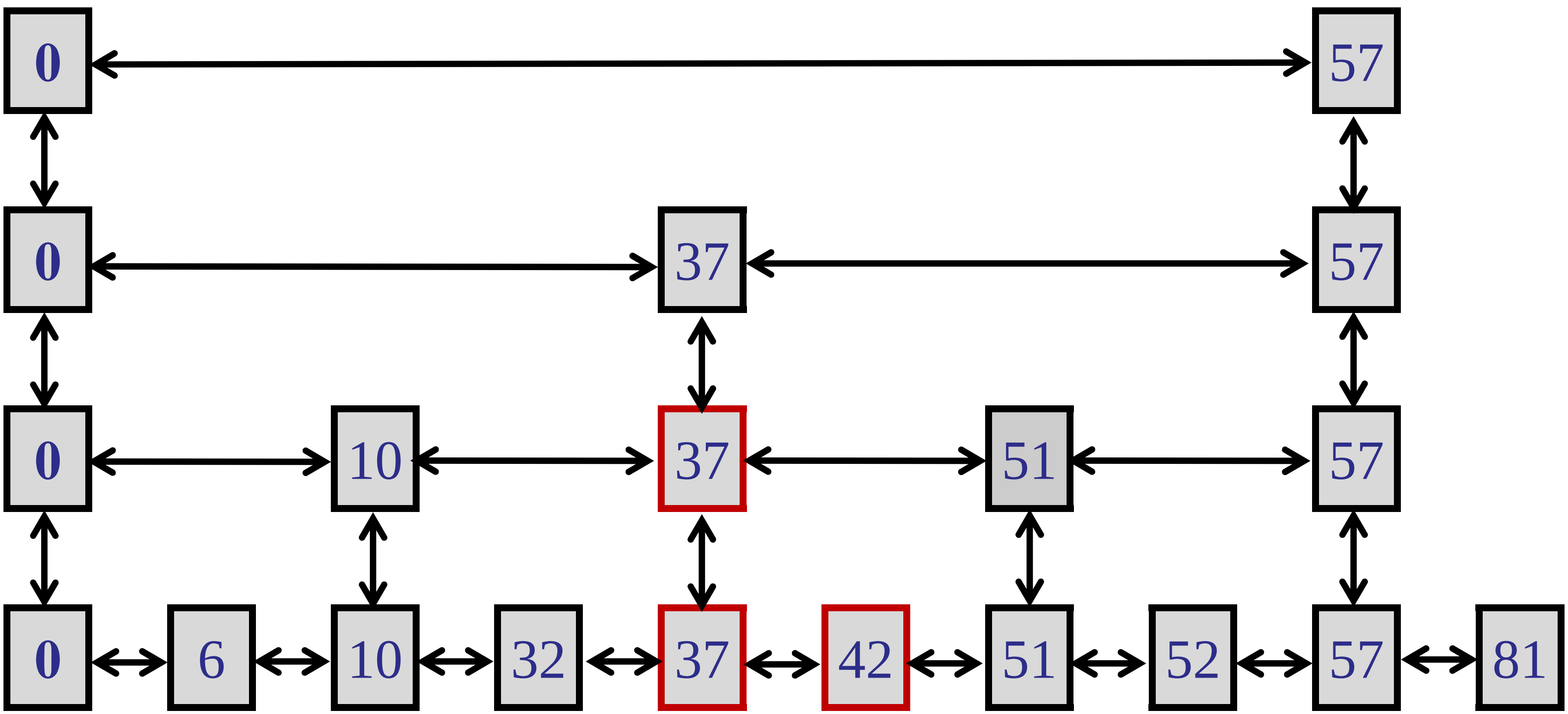
If tails, done.

```
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;
}
```

Example: `insert(51)`

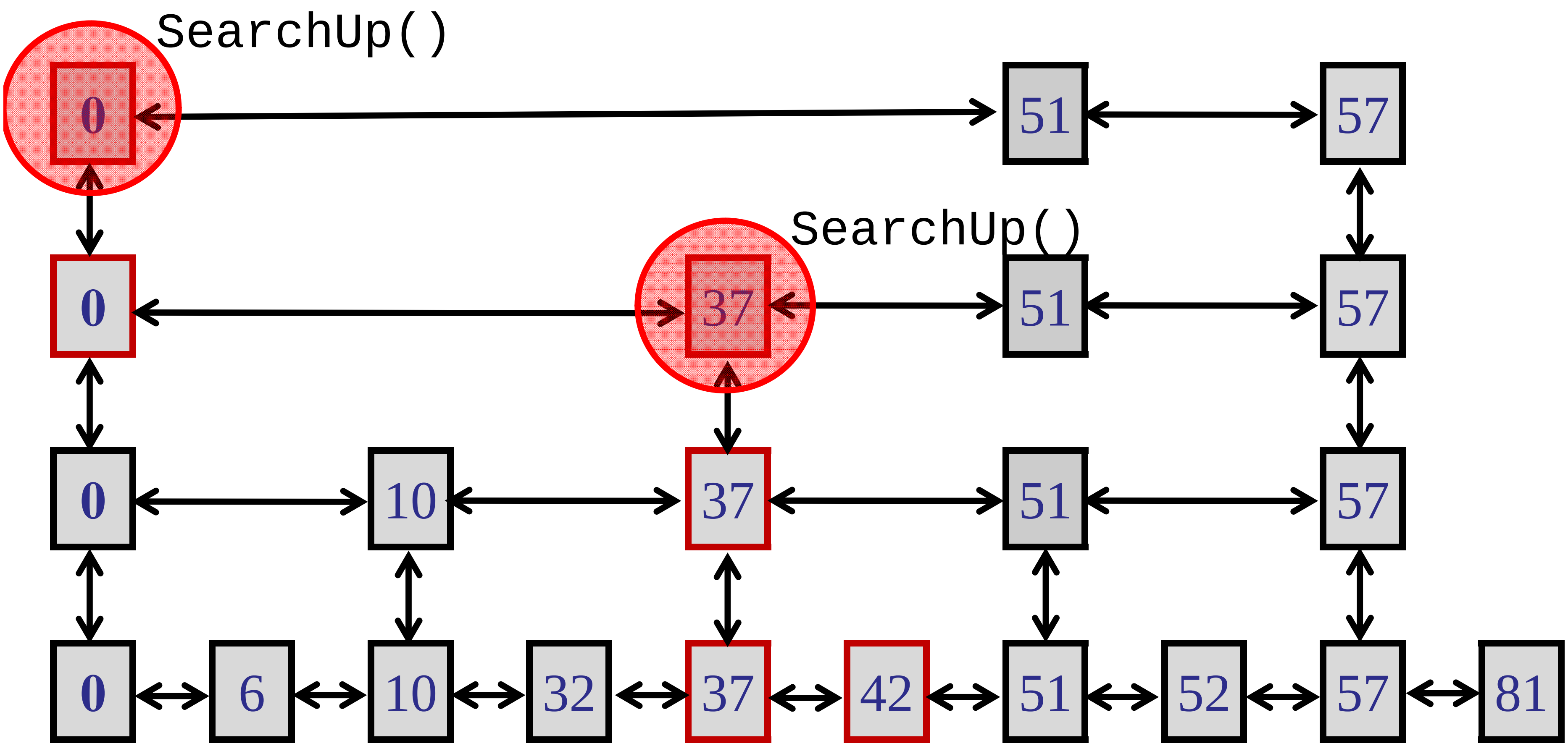


Example: `insert(51)`

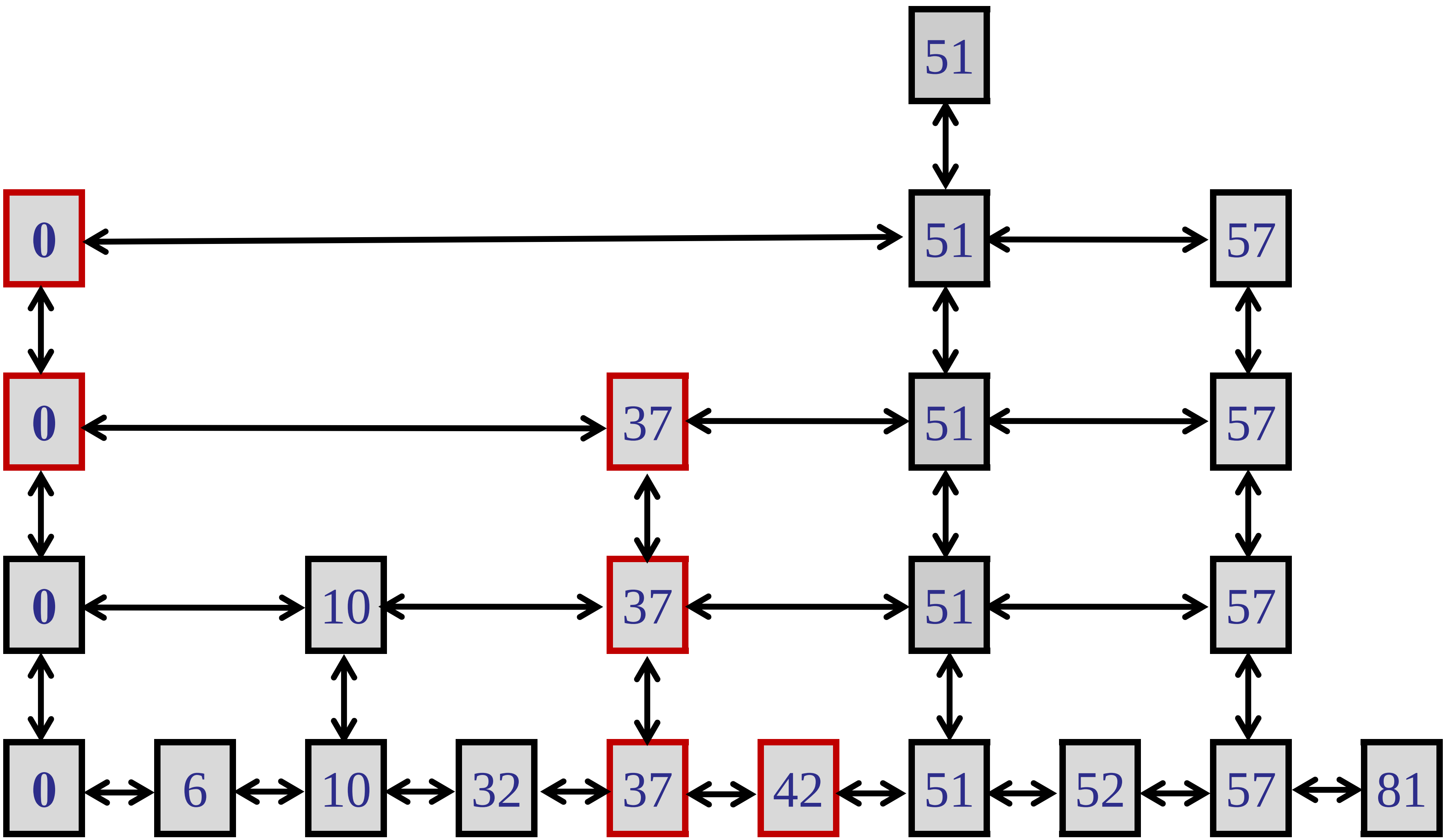


Example: `insert(51)`

Oop! No
SearchUp()

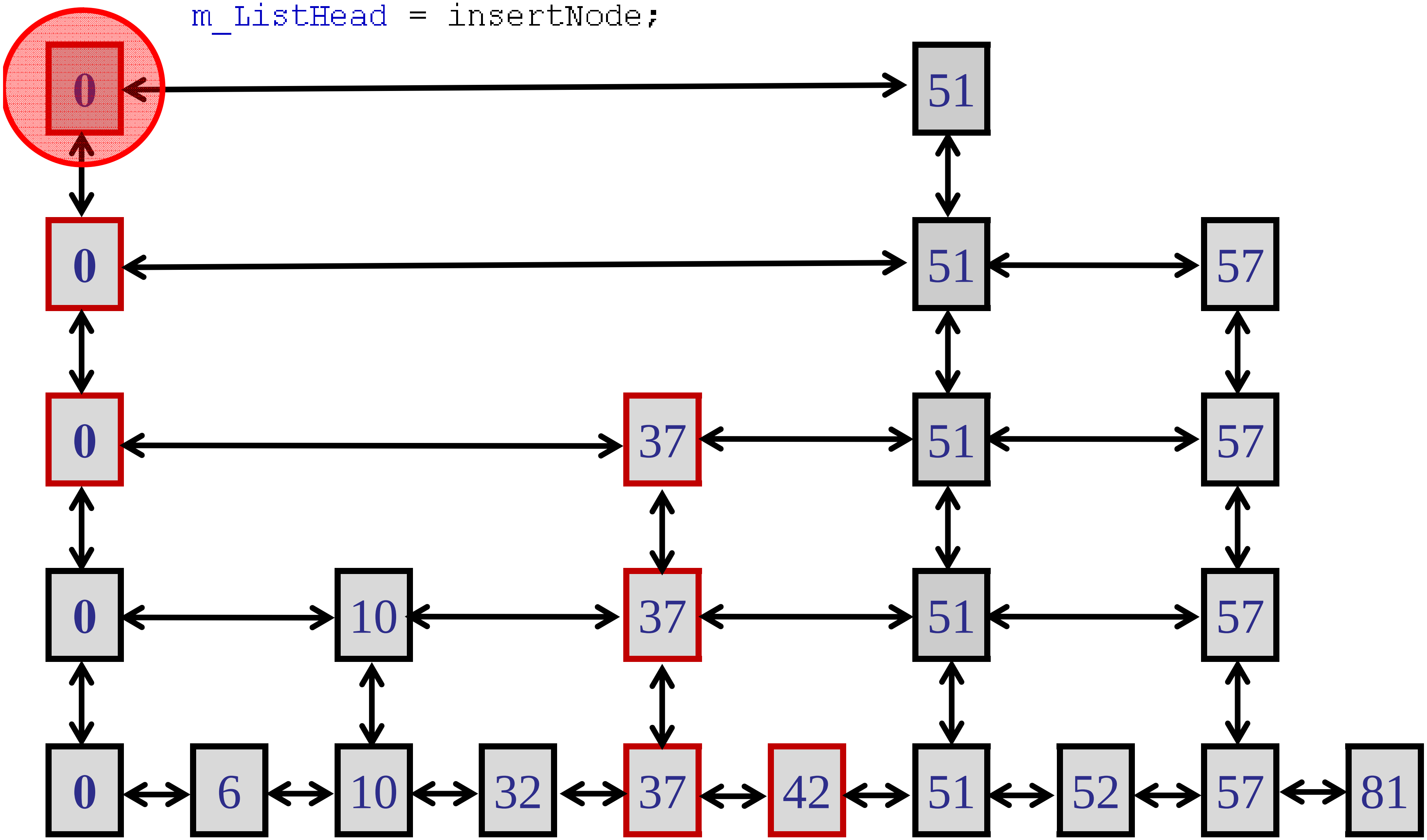


Example: `insert(51)`



Example: insert(51)

```
insertNode = new SkipListNode<TData>(Integer.MIN_VALUE, null);
insertNode.setDown(m_ListHead);
m_ListHead.setUp(insertNode);
m_ListHead = insertNode;
```



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 Implementation

Done!

SkipList Analysis

SkipList Analysis

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

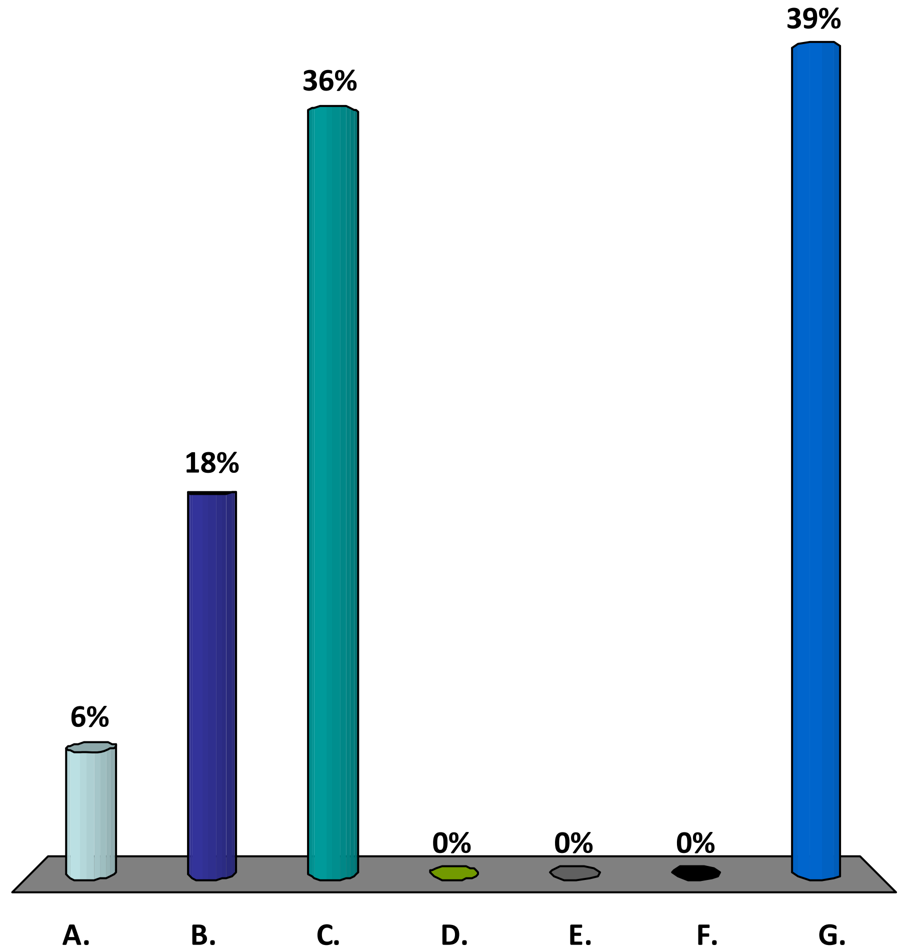
Key steps:

- 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.

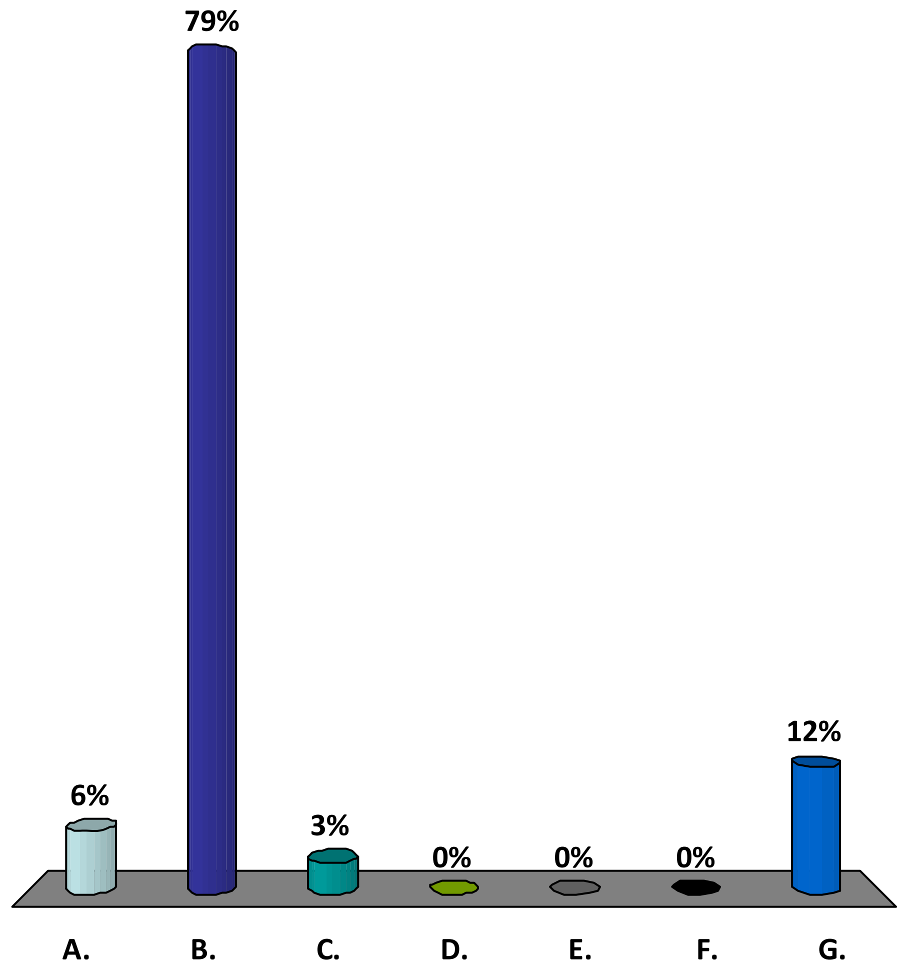
What is the largest number of level you can build in SkipList?

A. Sqrt(n)
B. Log n
C. n
D. 2n
E. n^2
F. 2^n
G. Angpao give me!



What is the expected number of level you can build in SkipList?

A. Sqrt(n)
B. Log n
C. n
D. 2n
E. n^2
F. 2^n
G. Angpao give me!



SkipList Analysis

Claim: *In expectation*, a SkipList with n elements has $O(\log n)$ levels.

SkipList Analysis

Claim: *In expectation*, a SkipList with n elements has $O(\log n)$ levels.

Proof:

E[#node at Level 1] = n

E[#node at Level 2] = n/2

E[#node at Level 3] = n/4

.

.

E[#node at Level log n] = 1

SkipList Analysis

Claim: *In expectation*, 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 is higher than $c\log(n)$] $\leq \frac{1}{2^{c\log n}} \leq \frac{1}{n^c}$

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(***any*** element is too high) $\leq \frac{n}{n^c} \leq \frac{1}{n^{c-1}}$

SkipList Analysis

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

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

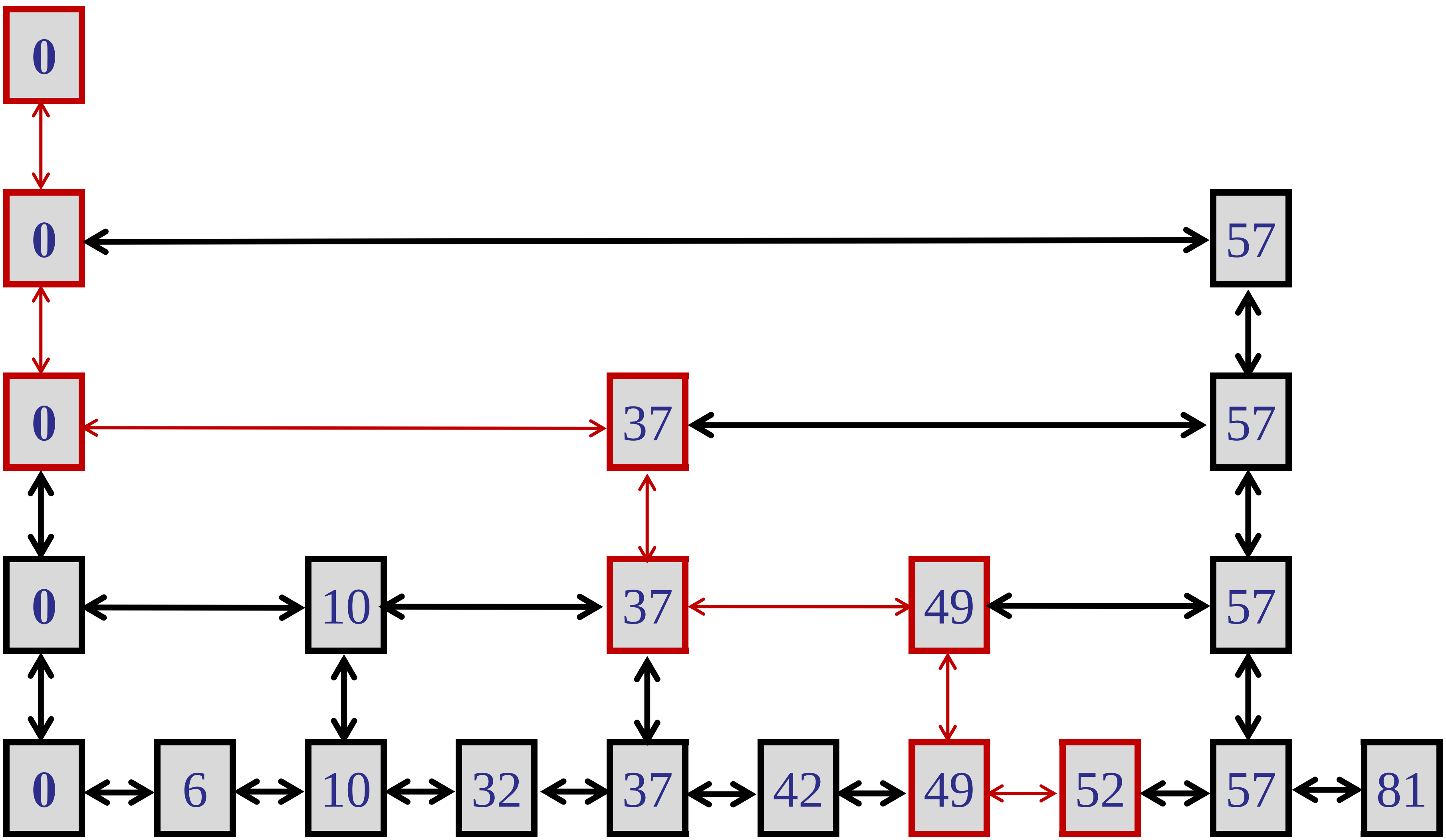
Analyzing a search:

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: *In expectation*, after O(log n) coin flips, you get c log n heads.

SkipList Analysis

Proof:

- Say we flip 10*c*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 10*c*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:

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

SkipList Analysis

Bounding binomials:

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

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

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

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: *In expectation*, after O(log n) coin flips, you get c log n heads.

Conclusion: Each search takes O(log n) steps in expectation.

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.