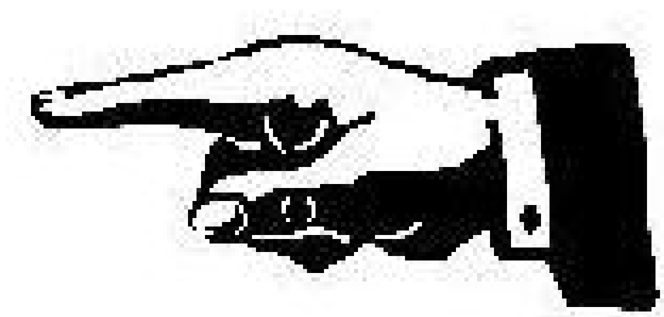
Data Structures and Algorithms

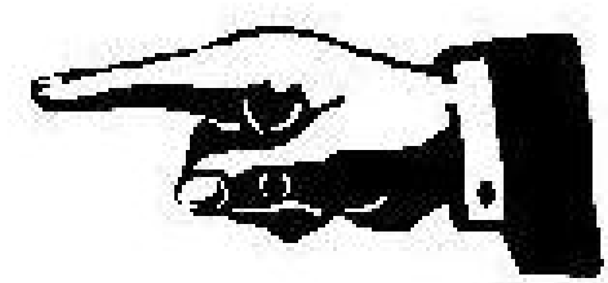
Lesson 3: *Point to where you are going: links*



Linked lists

Outline

1. **Arrays**
2. Non-contiguous Data Structures
3. Singly Linked List
4. Implementing Stacks and Queues
5. Java Linked List Class
6. Using Linked Lists
7. Skip Lists



Arrays

- An array is a contiguous chunk of memory
- It has an access **time** of $\Theta(1)$
  - ⋆ The constant factor is small
- Arrays provide a very efficient use of **memory**
- 95% of the time using arrays is going to give the best performance
- Disadvantages:
  - ⋆ fixed length (but can use variable-length arrays, at extra cost)
  - ⋆ insertion/deletion to/from the middle have $\Theta(n)$ time complexity

Variable-Length Arrays: Time Analysis

- Most `add(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- How efficient is resizing?
- Adding to a full array is slow but this is **amortised** by other quick adds – see next two slides!

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10+20

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10+20+40

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10+20+40+80

Example

- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10+20+40+80
  - ⋆ `new int[]`: 4

Example

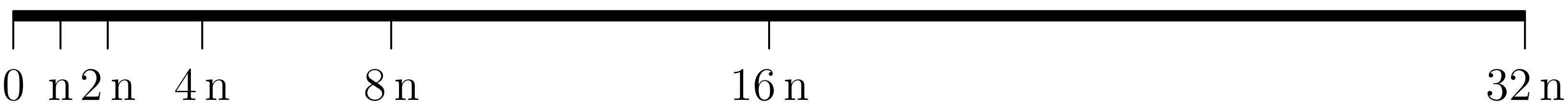
- If we have an initial capacity of 10 and add 100 elements, doubling the array size whenever required, then the number of operations needed is
  - ⋆ adds: 100
  - ⋆ copies: 10+20+40+80
  - ⋆ **`new int`**`[]`: 4
- 250 `add` and `copy` operations + 4 **`new`** operations

General Time Analysis

- If we perform N adds with an initial capacity of n,

General Time Analysis

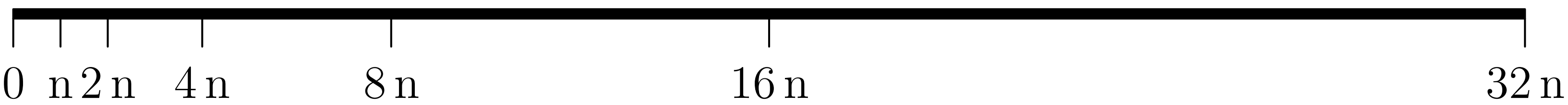
- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m$$

General Time Analysis

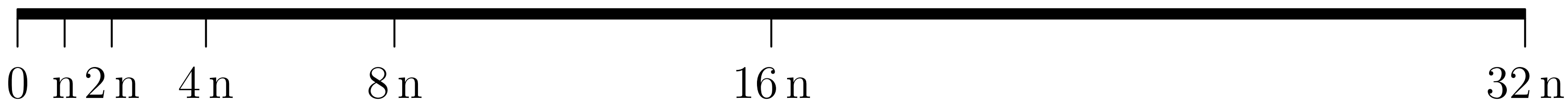
- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \le n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



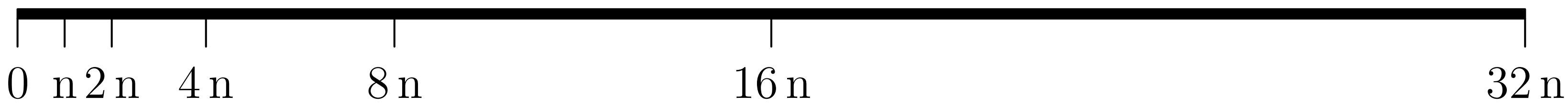
$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}\, n$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



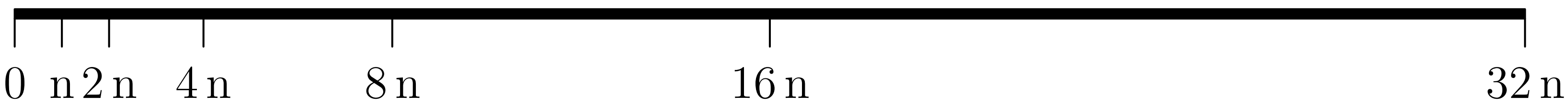
$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1} n = n(1 + 2 + \cdots + 2^{m-1})$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



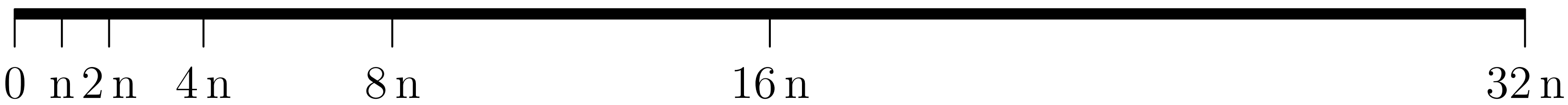
$$n \times 2^{m-1} < N \leq n \times 2^{m} \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}\, n = n(1 + 2 + \cdots + 2^{m-1}) = n\,(2^{m} - 1)$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

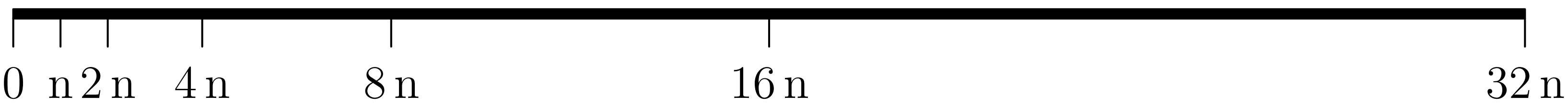
$$n + 2n + 4n + \cdots + 2^{m-1}\, n = n(1 + 2 + \cdots + 2^{m-1}) = n\left(2^m - 1\right)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\left(2^m - 1\right) + m$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

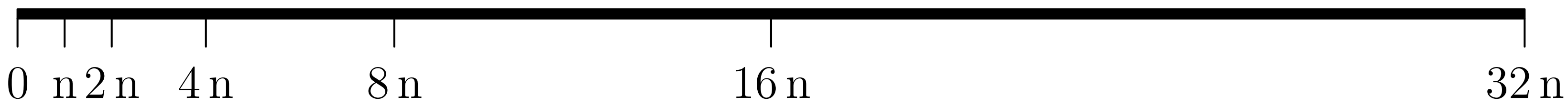
$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1}) = n\left(2^m - 1\right)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\left(2^m - 1\right) + m = N + n2^{\left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil} - n + \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2\left(\frac{N}{n}\right)\right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}\, n = n(1 + 2 + \cdots + 2^{m-1}) = n\,(2^m - 1)$$

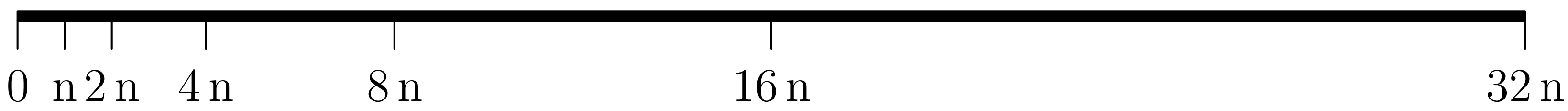
- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\,(2^m - 1) + m = N + n2^{\left\lceil \log_2\left(\frac{N}{n}\right)\right\rceil} - n + \left\lceil \log_2\left(\frac{N}{n}\right)\right\rceil$$

$$< N + 2N - n + \log_2\left(\frac{N}{n}\right) + 1$$

General Time Analysis

- If we perform N adds with an initial capacity of n,
- we must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \qquad \text{i.e. } m = \left\lceil \log_2\left(\frac{N}{n}\right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}\,n = n(1 + 2 + \cdots + 2^{m-1}) = n\left(2^m - 1\right)$$

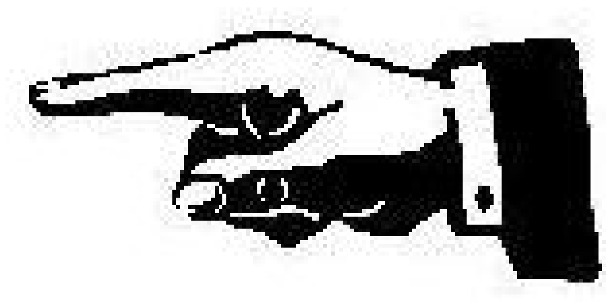
- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\left(2^m - 1\right) + m = N + n2^{\left\lceil \log_2\left(\frac{N}{n}\right)\right\rceil} - n + \left\lceil \log_2\left(\frac{N}{n}\right) \right\rceil$$

$$< N + 2N - n + \log_2\left(\frac{N}{n}\right) + 1 < 4N$$

Outline

1. Arrays
2. **Non-contiguous Data Structures**
3. Singly Linked List
4. Implementing Stacks and Queues
5. Java Linked List Class
6. Using Linked Lists
7. Skip Lists



Non-Contiguous Data

- Storing data in a contiguous chunk of memory has the great advantage of allowing random access
- It has the disadvantage that it is expensive to add or remove data from the middle of the list or to rearrange the data
- A different approach is to use units of data that point to other units

Non-Contiguous Data Structures

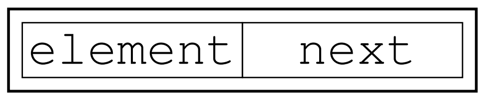
- There are a lot of important data structures that use non-contiguous memory
  - ⋆ Binary trees
  - ⋆ Graphs
- In this lecture we consider **linked lists**
- This is a classic data structure, which is almost entirely useless...
- However, it serves as a good introduction to much more useful data structures!

Self-Referential Classes

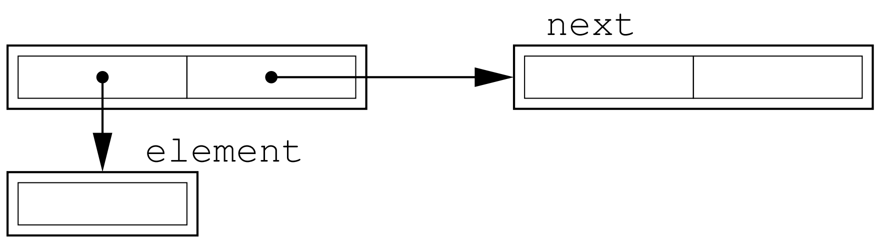
- The building block for a linked list is a node

```
class Node<T>
{
  private T element;
  private Node<T> next;
}
```

- This contains a reference to another node object



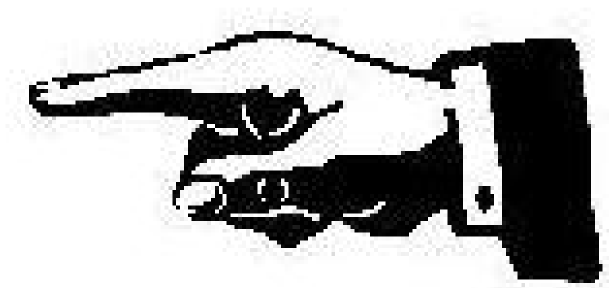
- Both element and node can point to an object



  ⋆ we represent the address by the outer box

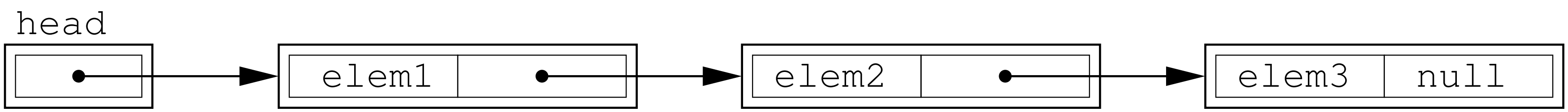
Outline

1. Arrays
2. Non-contiguous Data Structures
3. **Singly Linked List**
4. Implementing Stacks and Queues
5. Java Linked List Class
6. Using Linked Lists
7. Skip Lists



Singly Linked List

- We can build a linked list by stringing nodes together



  - ⋆ We don't show the "pointer" to `element`
- A singly linked list has a single "pointer" to the next element
- A doubly linked list has "pointers" to the next and previous element – we will see this later
- We should be able to create a linked list, add elements, remove elements, see if an element exists, etc.

Java Implementation

- We consider a lightweight implementation
- The class will have a head, a size counter and have `Node` as a nested class

```
public class MyLinkedList<E>
{
    private Node<E> head;
    private int no_elements;

    private static class Node<T>
    {
        private T element;
        private Node<T> next;
    }
```

Simple Methods

- The constructor is simple (and not strictly necessary)

```
public MyLinkedList()
{
    head = null;
    noElements = 0;
}
```

- Other simple methods are

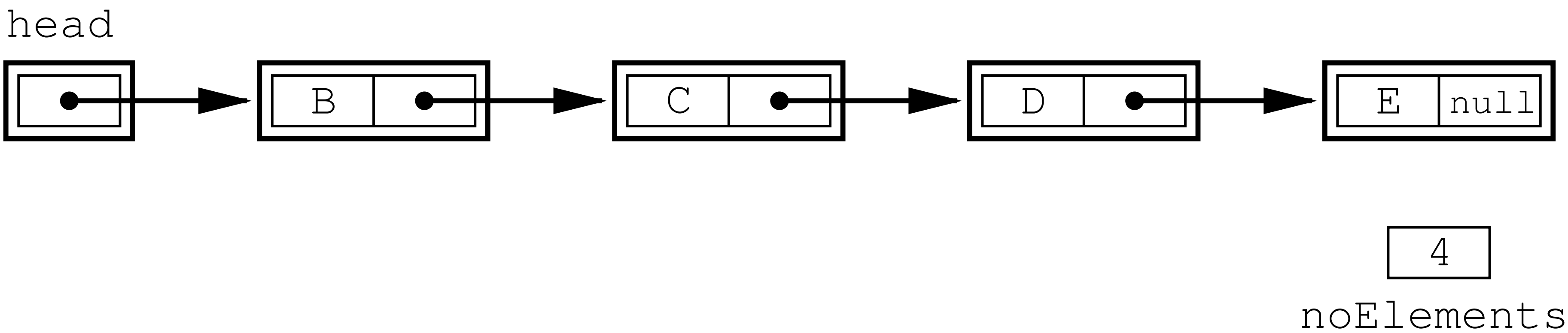
```
public int size()
{
    return noElements;
}

public boolean isEmpty()
{
    return head == null;
}
```

Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```

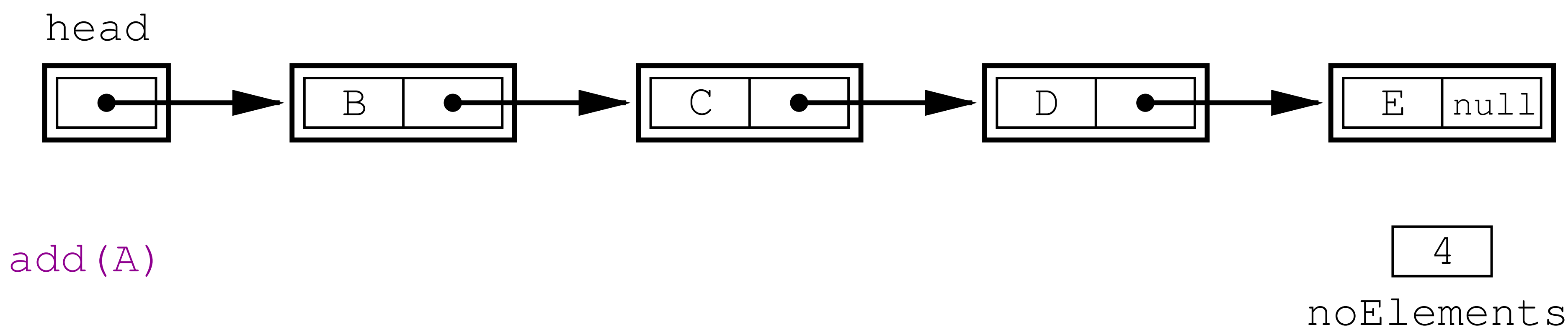
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



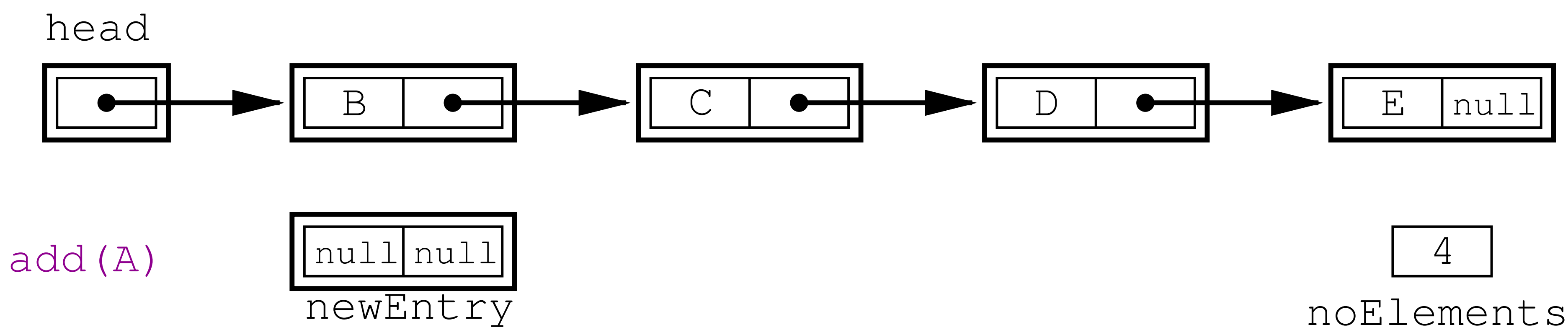
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



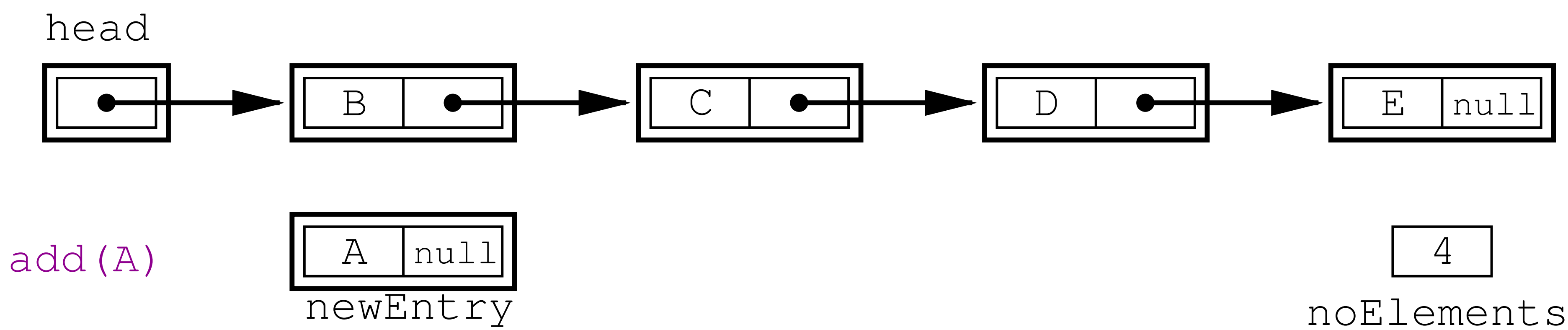
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



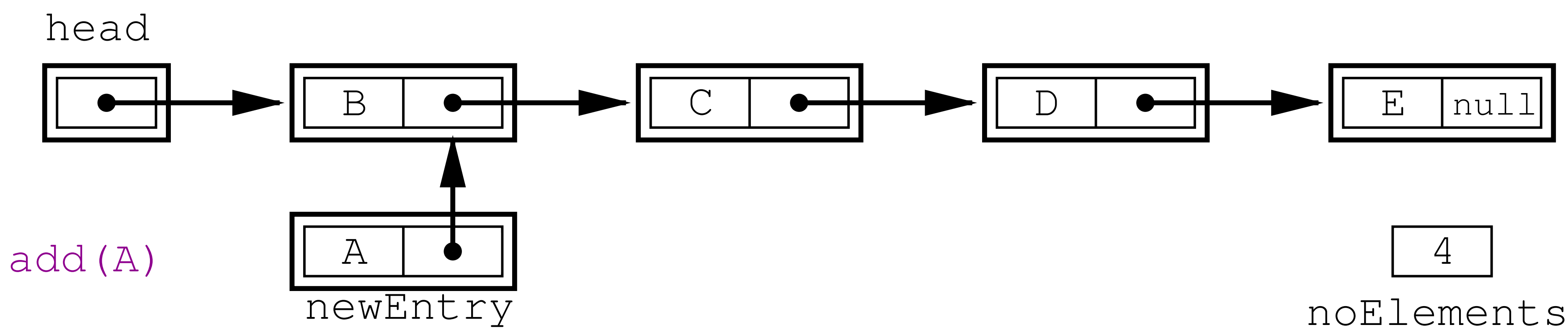
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



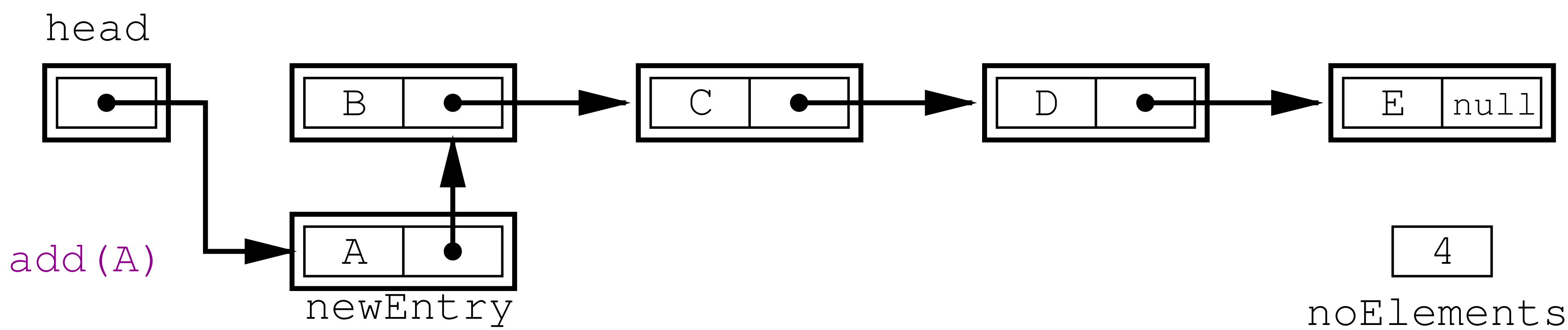
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



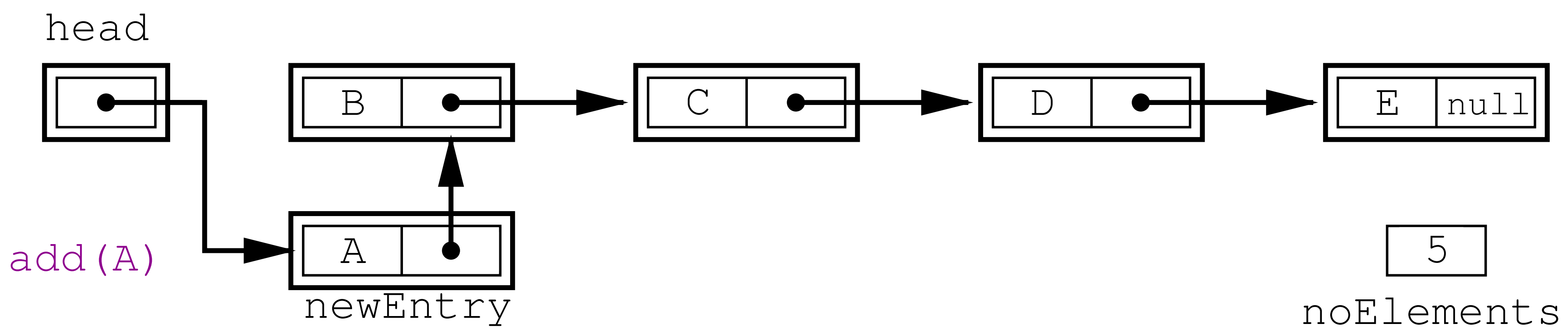
Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



Adding Elements

```
public boolean add(E element)
{
    Node<E> newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```

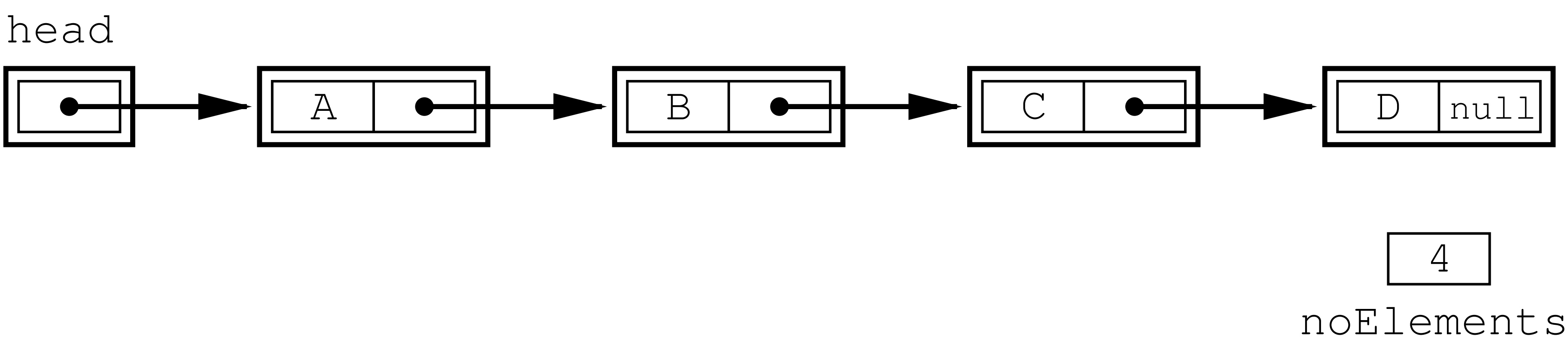


Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```

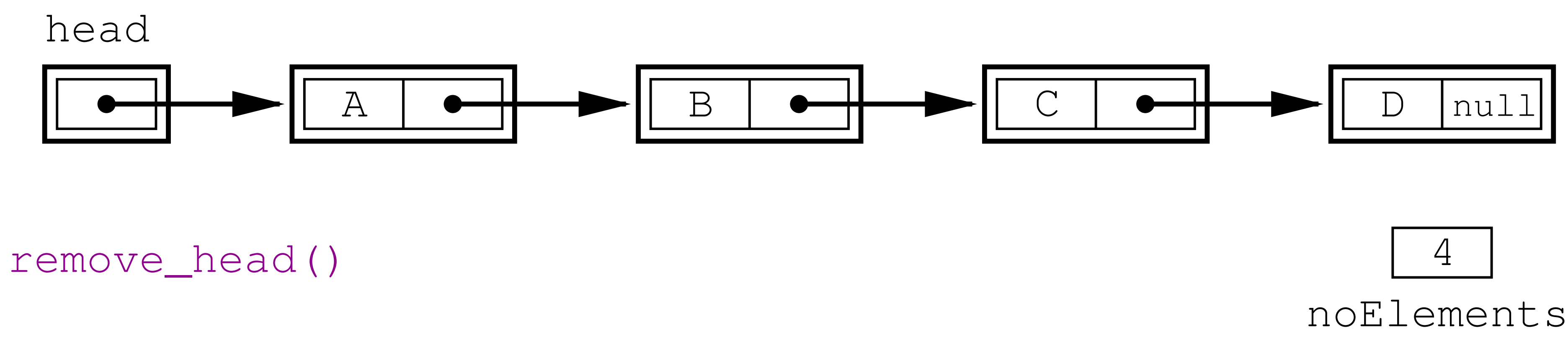
Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```



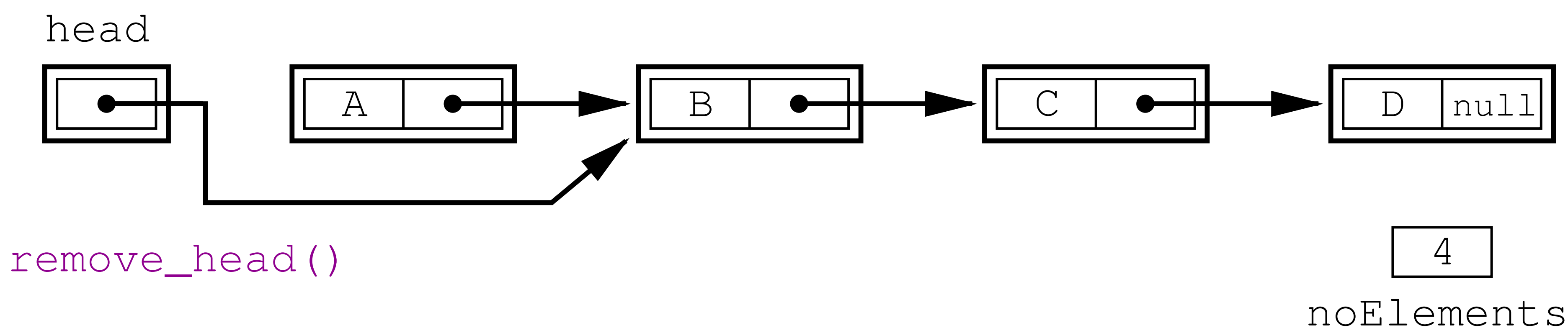
Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```



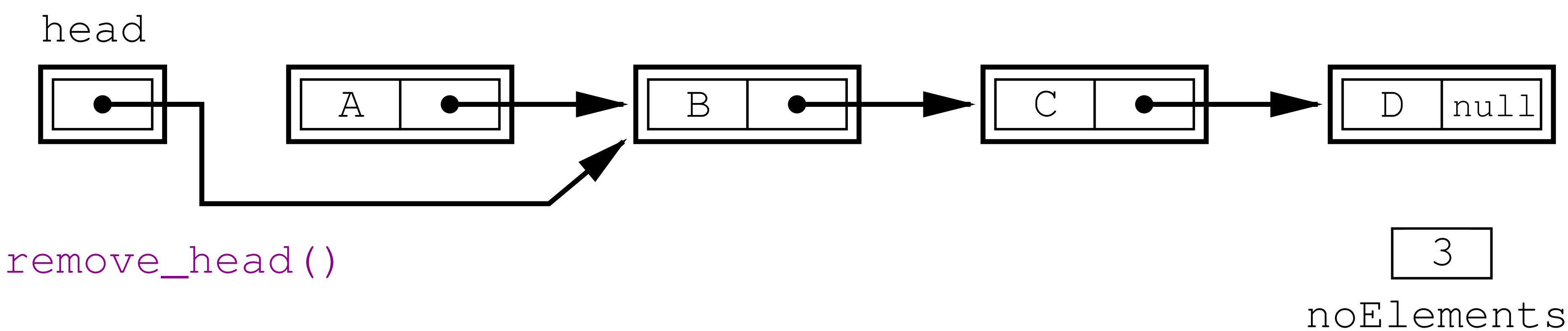
Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```



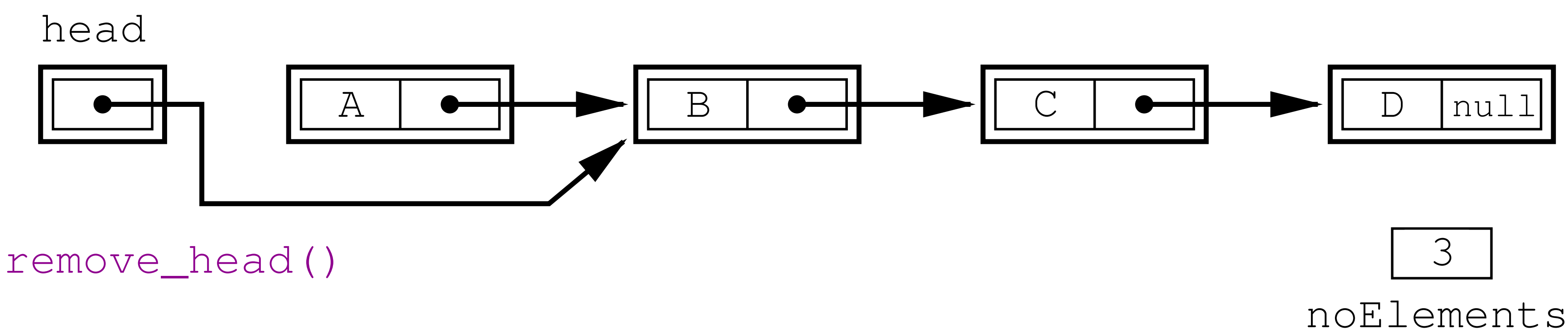
Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```



Removing the List Head

```
public boolean remove_head()
{
    if (!isEmpty()) {
        head = head.next;
        noElements--;
        return true;
    }
    return false;
}
```



Node A is removed by garbage collection

Contains

- Does the list contain `obj`?

```
public boolean contains(E obj)
{
  for (Node<E> current=head; current!=null; current=current.next)
        if (obj.equals(current.element))
            return true;
  return false;
}
```

- `current` iterates over nodes in list
- end when `current==null`
- All classes have `equals` method

Other Methods

- We can easily implement many other methods
  - ⋆ `get_head()`—return element at head of list
  - ⋆ `get(int i)`—return i^{th} item in list
  - ⋆ `remove(T obj)`-remove `obj` from list
- Note that `get(int i)` requires moving down the list so is $O(n)$ (i.e. not random access)

Outline

1. Arrays
2. Non-contiguous Data Structures
3. Singly Linked List
4. **Implementing Stacks and Queues**
5. Java Linked List Class
6. Using Linked Lists
7. Skip Lists

Stack Implementation

- It is easy to implement a stack using a linked list

```
public class LinkedListStack<E>
{
  private MyLinkedList<E> list = new MyLinkedList<E>();

  boolean push(E obj) {list.add(obj);}

  E peek() {return list.get_head();}

  E pop() {
    if (isEmpty()) throw EmptyStackException;
    T elem=list.get_head();
    list.remove_head();
    return elem;
  }

  boolean isEmpty() {return list.isEmpty();}
}
```

Complexity of Stack Implementation

- All stack operations take constant time, i.e. $\Theta(1)$
  - ⋆ hidden cost of creating and destroying `Node` objects

Complexity of Stack Implementation

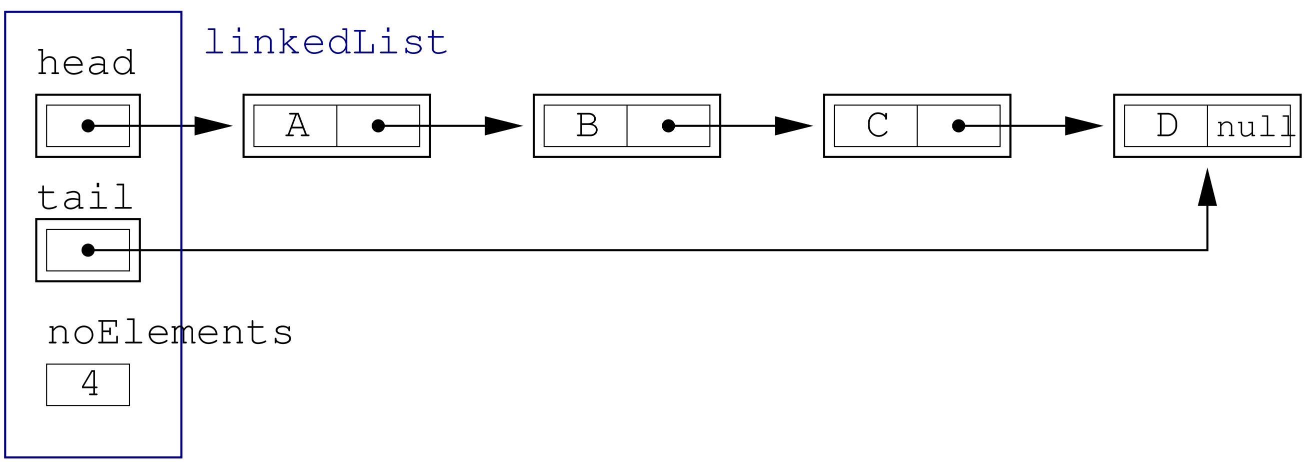
- All stack operations take constant time, i.e. $\Theta(1)$
  - ⋆ hidden cost of creating and destroying `Node` objects
- Memory requirement is $\Theta(n)$
  - ⋆ (approximately $2 \times n$ references and n objects)

Complexity of Stack Implementation

- All stack operations take constant time, i.e. $\Theta(1)$
  - ⋆ hidden cost of creating and destroying `Node` objects
- Memory requirement is $\Theta(n)$
  - ⋆ (approximately $2 \times n$ references and n objects)
- An array implementation is therefore slightly more efficient in practice

Point to the Back

- To find the end of the list takes n jumps
- Thus our linked list isn't the right data structure to implement a queue
- However, we could include a pointer to the end of the list

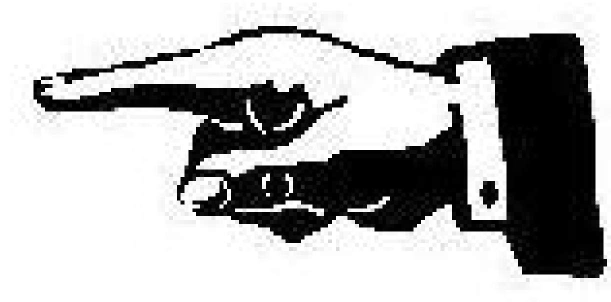


Implementing a Queue

- We can then add elements to the tail in constant time
- We can the implement a queue in $O(1)$ time by
  - `enqueue`ing at the back
  - `dequeue`ing at the head
- Note that although adding an element to the tail is constant time, removing an element from the tail is $O(n)$ as we have to find the new tail

Outline

1. Arrays
2. Non-contiguous Data Structures
3. Singly Linked List
4. Implementing Stacks and Queues
5. **Java Linked List Class**
6. Using Linked Lists
7. Skip Lists



Doubly Linked List

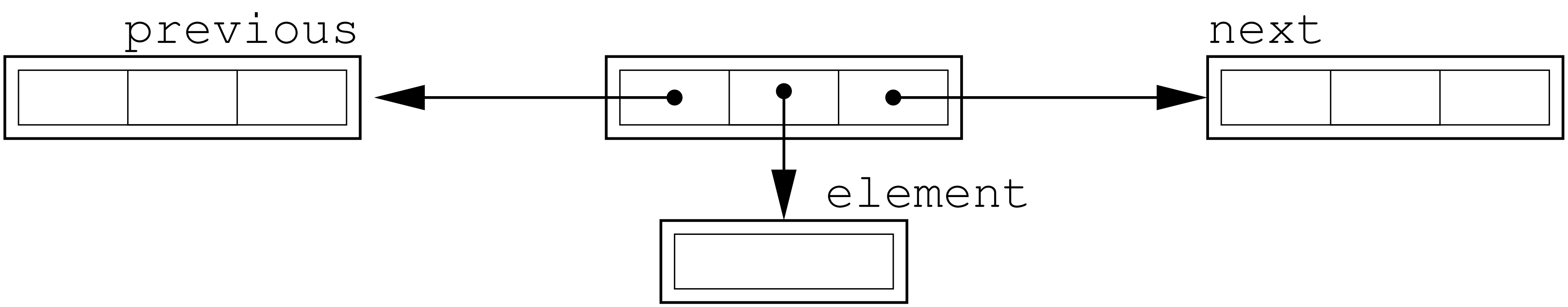
- Java provides a linked list class `LinkedList<T>` which allows $O(1)$ add and remove to both ends of the list
- To achieve this it uses a doubly-linked list with pointers to next and previous elements

```java
private static class Node<T>
{
  T element;
  Node<T> next;
  Node<T> previous;
}
```

Doubly Linked List

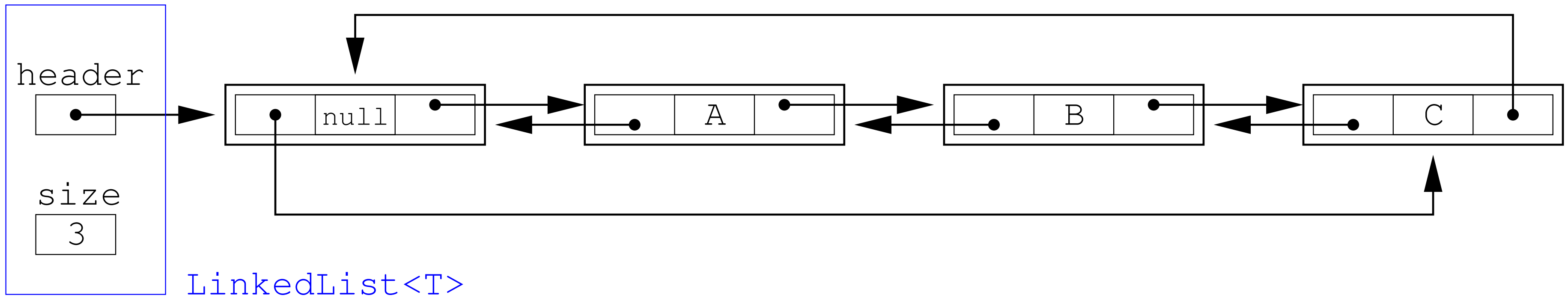
- Java provides a linked list class `LinkedList<T>` which allows $O(1)$ add and remove to both ends of the list
- To achieve this it uses a doubly-linked list with pointers to next and previous elements

```
private static class Node<T>
{
  T element;
  Node<T> next;
  Node<T> previous;
}
```



Dummy Node

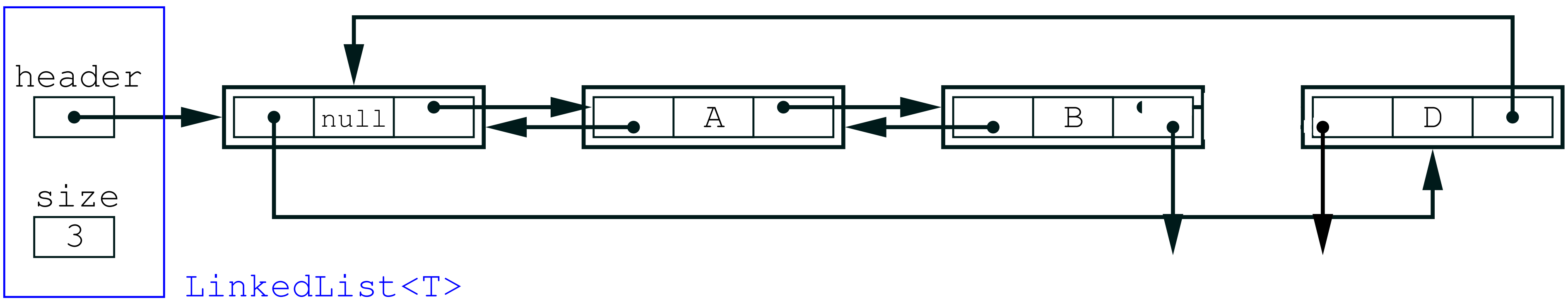
- dummy node used to make the implementation slicker



- Symmetric data structure so processing head and tail is equally efficient

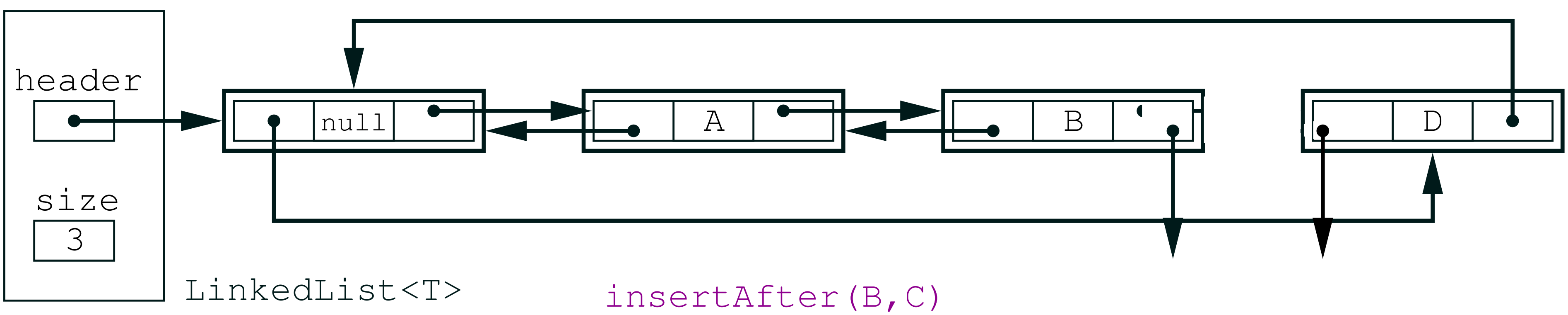
Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



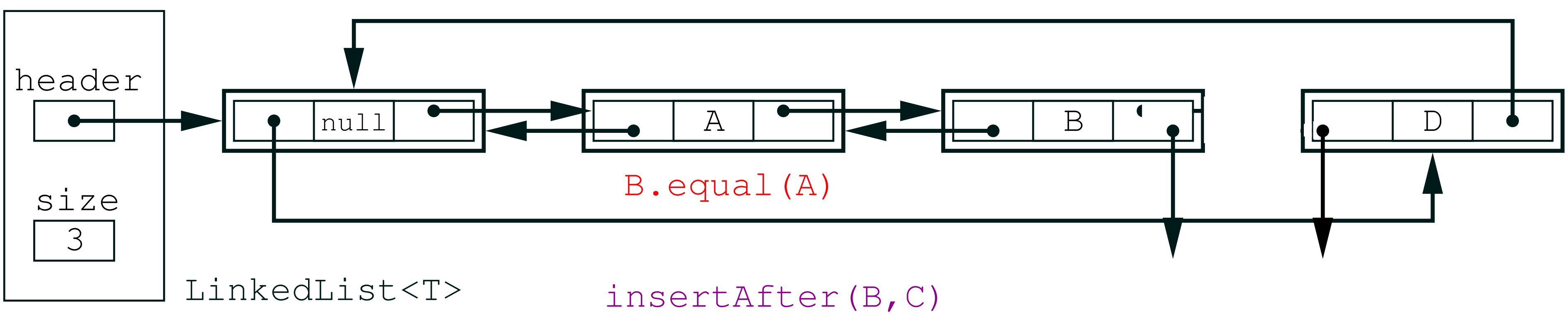
Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



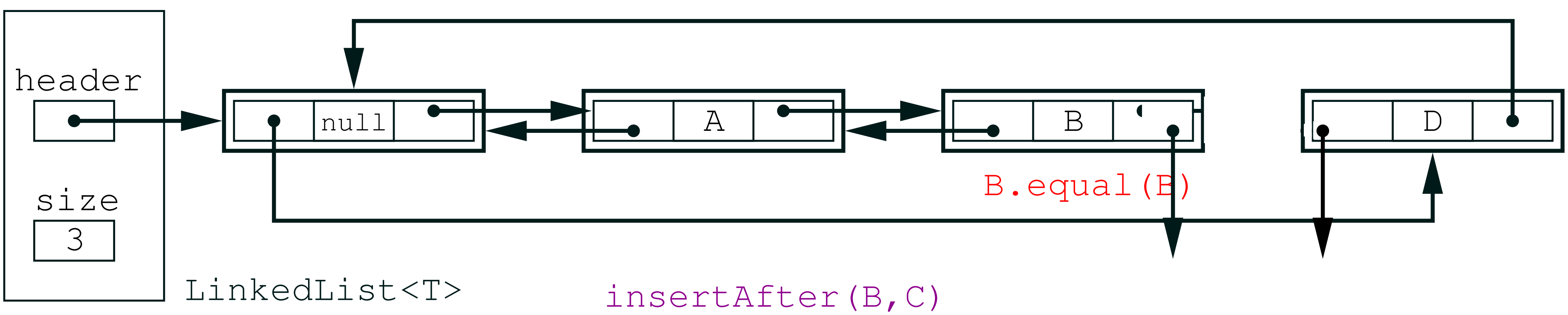
Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



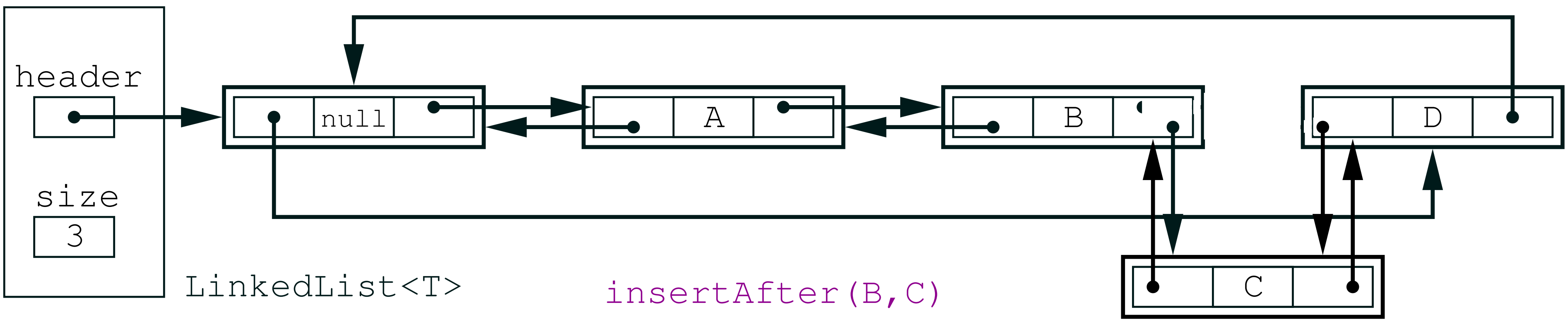
Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



Outline

1. Arrays
2. Non-contiguous Data Structures
3. Singly Linked List
4. Implementing Stacks and Queues
5. Java Linked List Class
6. **Using Linked Lists**
7. Skip Lists

When To Use Linked Lists

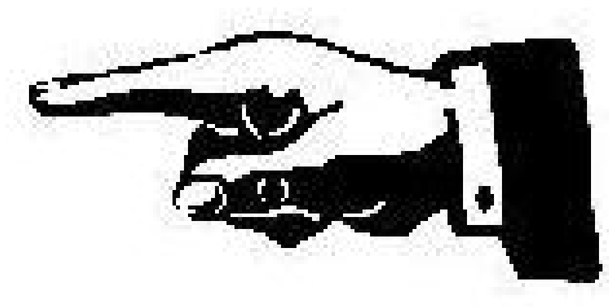
- linked lists have efficient insertion and deletion . . .
- . . . but it is difficult to think of applications where they are the best data structure to use
  - ⋆ lists – variable length arrays are usually better
  - ⋆ queues – linked list OK, but circular arrays are probably better
  - ⋆ sorted lists – binary trees much better

Line Editor

- One application where efficient insertion and deletion matters is a line editor
- We are usually working at a particular location in the text
- We often want to add or delete whole lines
- Storing the lines as strings in a linked list would allow a fairly efficient implementation

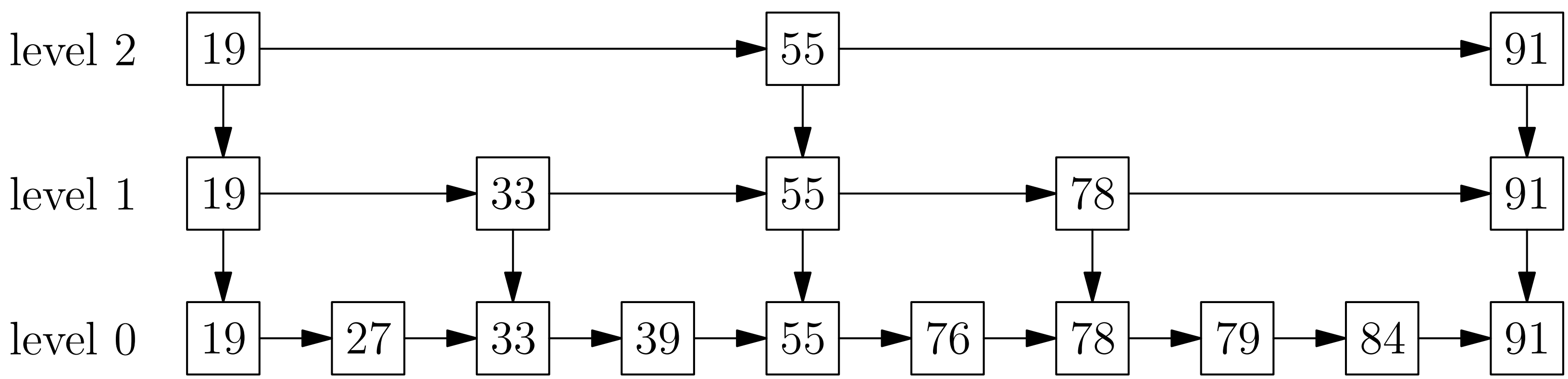
Outline

1. Arrays
2. Non-contiguous Data Structures
3. Singly Linked List
4. Implementing Stacks and Queues
5. Java Linked List Class
6. Using Linked Lists
7. **Skip Lists**



Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps
- Even if you kept an ordered list you still need to traverse it
- Skip lists are hierarchies of linked lists which support binary search

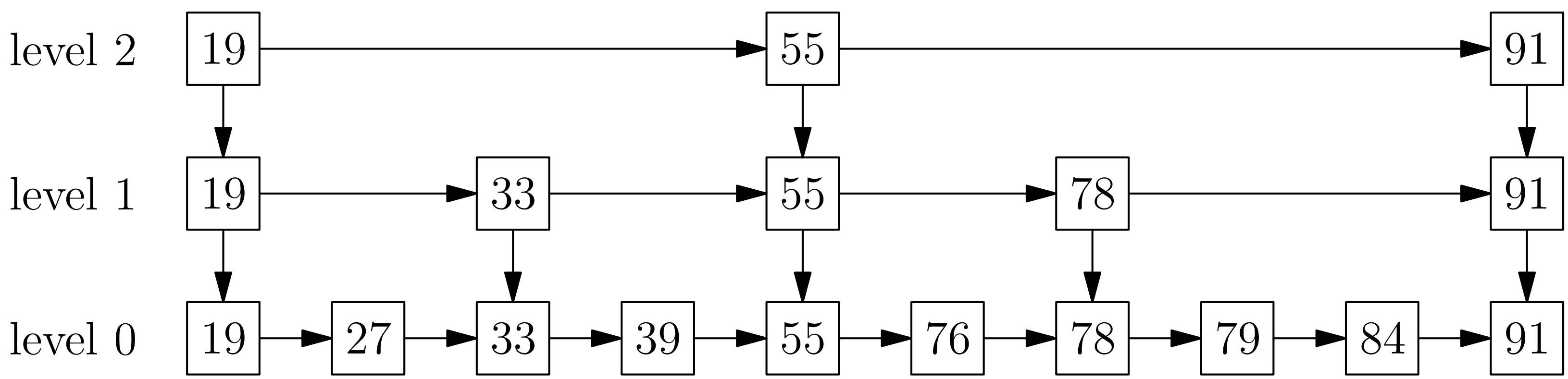


Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps

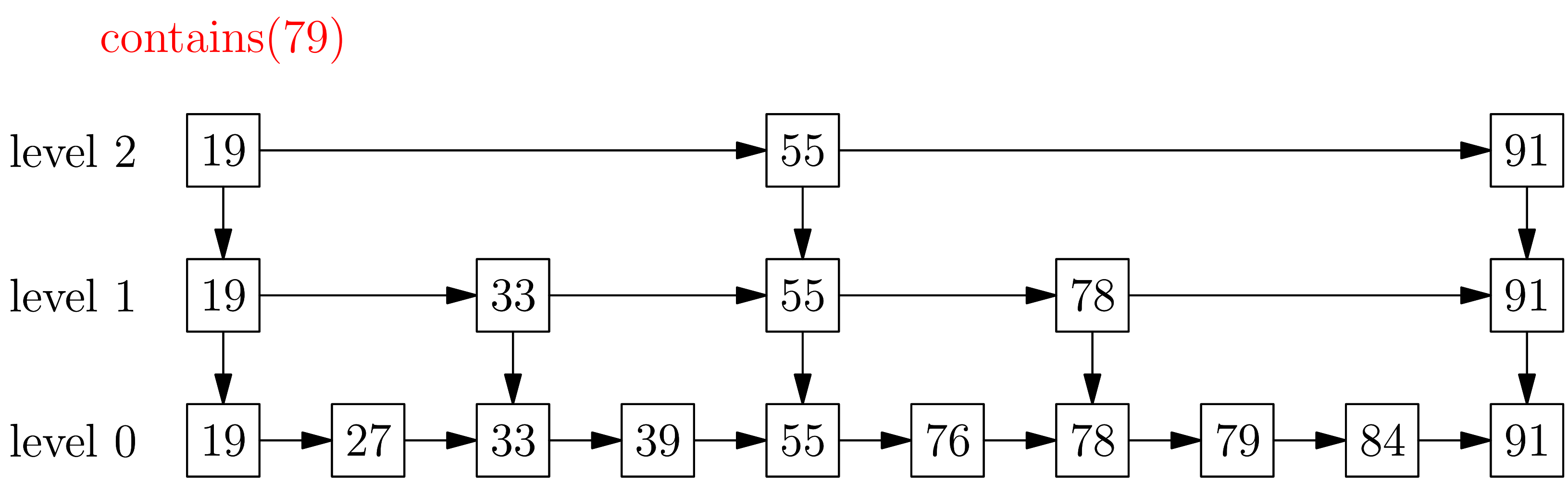
- Even if you kept an ordered list you still need to traverse it

- Skip lists are hierarchies of linked lists which support binary search



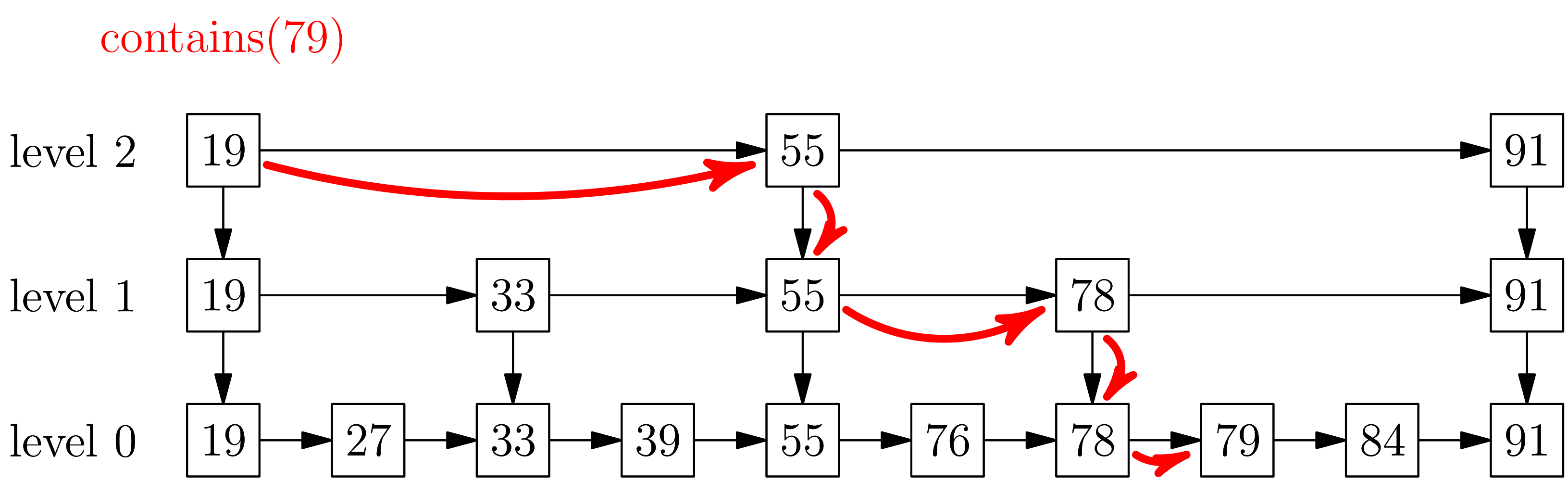
Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps
- Even if you kept an ordered list you still need to traverse it
- Skip lists are hierarchies of linked lists which support binary search



Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps
- Even if you kept an ordered list you still need to traverse it
- Skip lists are hierarchies of linked lists which support binary search



Efficiency of Skip Lists

- Skip lists provide $\Theta(\log(n))$ search as opposed to $\Theta(n)$
- They have similar time complexity to binary trees, although binary trees are slightly faster
- They have one advantage over binary trees – they allow efficient concurrent access
- Java 6 provides a `ConcurrentSkipListSet<T>` class

Lessons

- Node structures that point to other Node structures are used in many important data structures
- Linked lists are the simplest examples of this kind of structure and consequently have a dominant position in most DSA books
- In practice linked lists are seldom the data structure of choice – before choosing to use a linked list consider the alternatives
- There are some important uses for linked lists, e.g. skip lists and hash tables (see lecture on hashing)