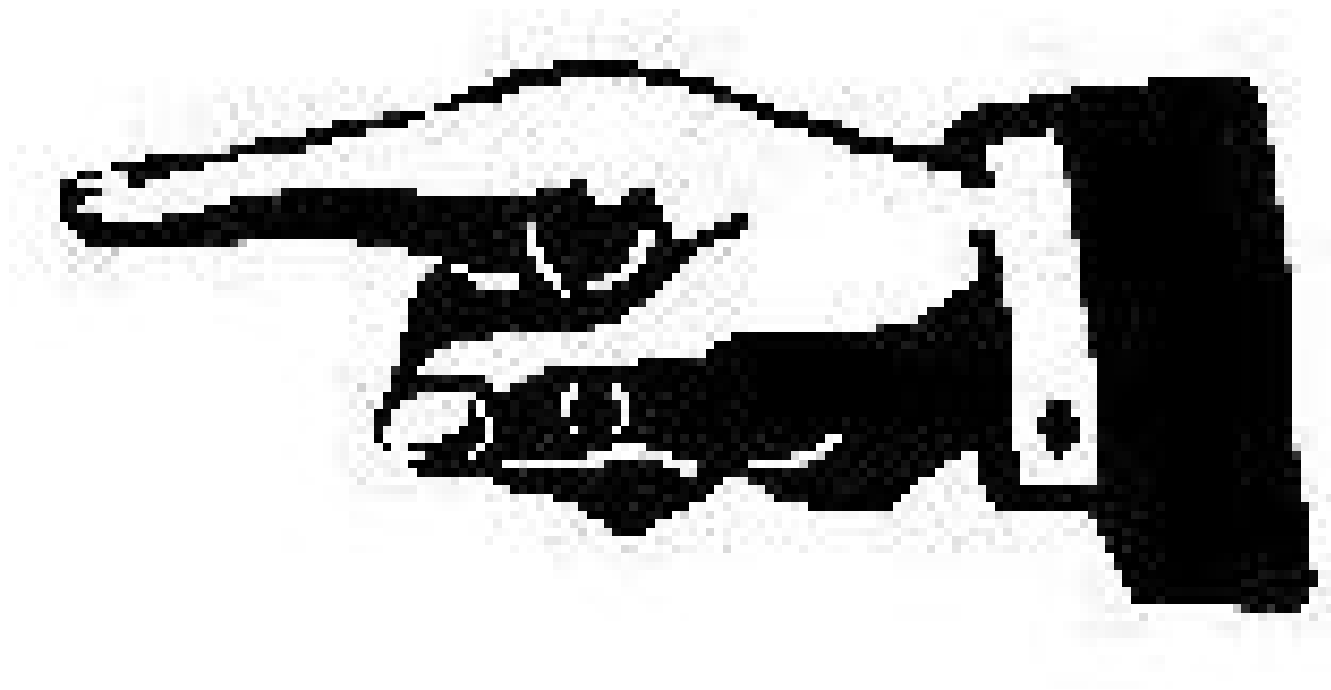


# Algorithms and Analysis

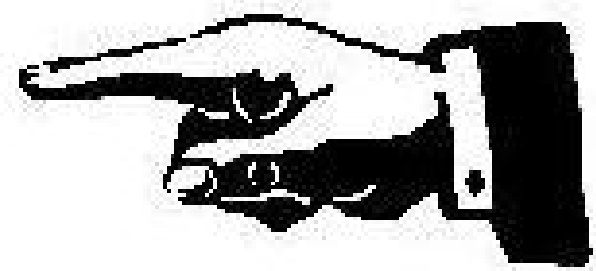
## Lesson 5: *Point to where you are going: links*



*Linked lists*

# Outline

1. **References**
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. Skip Lists



# Non-Contiguous Data

- So far we have considered arrays where the data is stored in a contiguous chunk of memory■
- This 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 using 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

```
struct Node<T>
{
    Node(U value, Node<U> *node): value(value), next(node) {}
    T element;
    Node<T> *next;
}
```

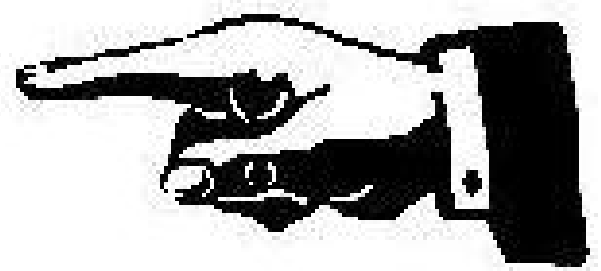
- We create new nodes

```
Node<int> *node = new Node<int>(10, pt_to_next)
```

- Note that node is the address of this node
- I make it a struct as this is a class where I want public access to the element and next
- I can make this class a private class of my linked list

# Outline

1. References
2. **Singly Linked List**
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. 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.■

# Implementation

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

```
class MyList {  
private:  
    template <typename U>  
    struct Node{  
        Node(U value, Node<U> *node): value(value), next(node) {}  
        U value;  
        Node<U> *next;  
    };  
    Node<T> *head;  
    unsigned noElements;
```



# Simple Methods

- The constructor is simple (and not strictly necessary)

```
MyList(): n(0), head(0) {}
```

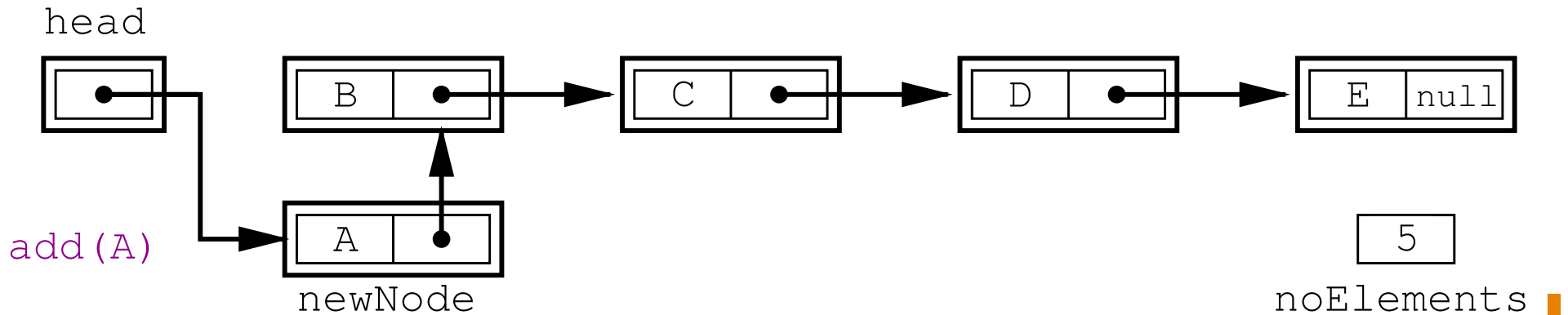
- Other simple methods are

```
unsigned size() const {return noElements;}
```

```
bool empty() const {  
    return head == 0;  
}
```

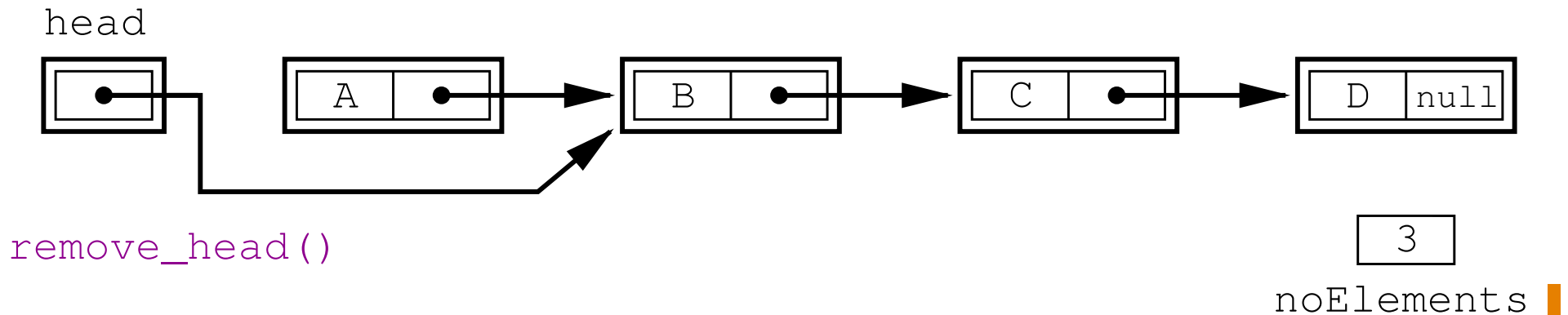
# Adding elements

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



# Remove Head of List

```
void remove_head()
{
    Node<T>* dead = head;
    head = head->next;
    noElements--;
    delete dead;
}
```

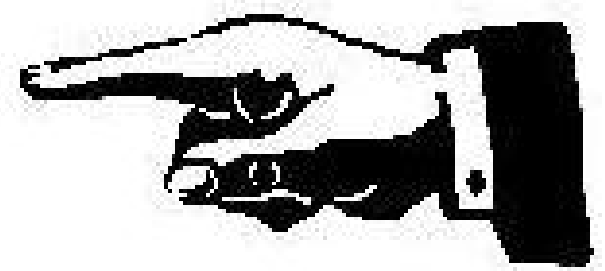


# Other Methods

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

# Outline

1. References
2. Singly Linked List
3. **Stacks and Queues**
4. Doubly Linked List
5. Using Linked Lists
6. Skip Lists



# Stack

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

```
template <typename T>
class Stack<E>
{
    private Mylist<T> list = new mylist<T>();

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

    E top() {return list.get_head();} // throw exception

    E pop() {
        T tmp = list.get_head();
        list.remove_head();
        return tmp;
    }

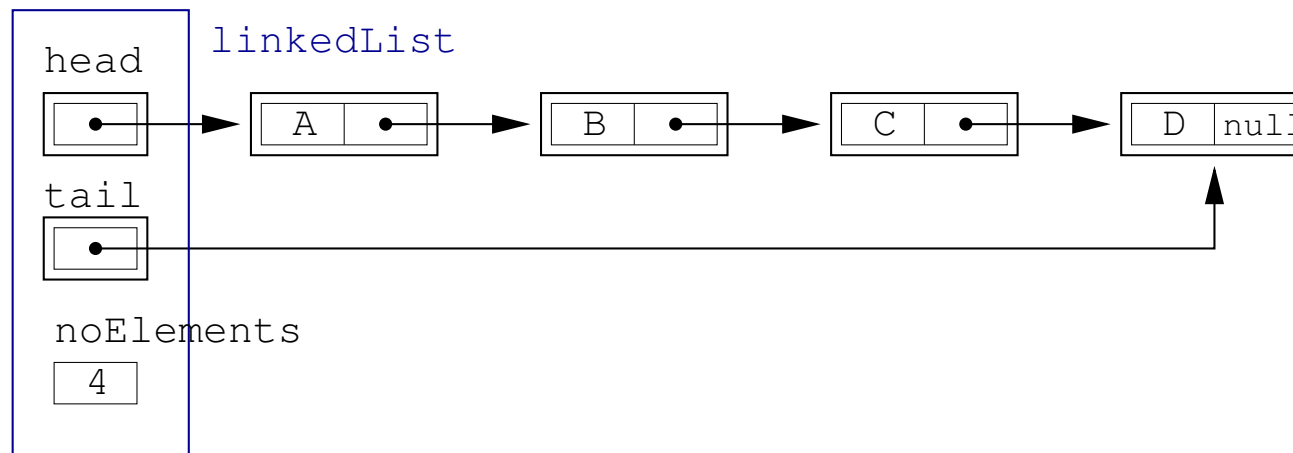
    boolean empty() {return list.empty();}
}
```

# Complexity of Stack

- All operations of the stack is constant time, i.e.  $O(1)$ ■
- This is the same time complexity as an array implementation■
- Memory requirement is approximately  $2 \times n$  reference and  $n$  objects■—same as worst case for an array■
- However, hidden cost of creating and destroying Node objects■
- The array implementation is therefore slightly faster■

# Point to the Back

- To find the end of the queue 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 queue



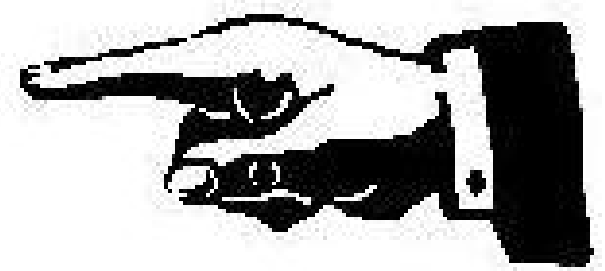


# Implementing a Queue

- We can then add elements to the tail in constant time■
- We can then implement a queue in  $O(1)$  time by
  - ★ enqueueing at the back■
  - ★ dequeueing at the head■
- I leave the implementation details as an exercise for you■
- 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

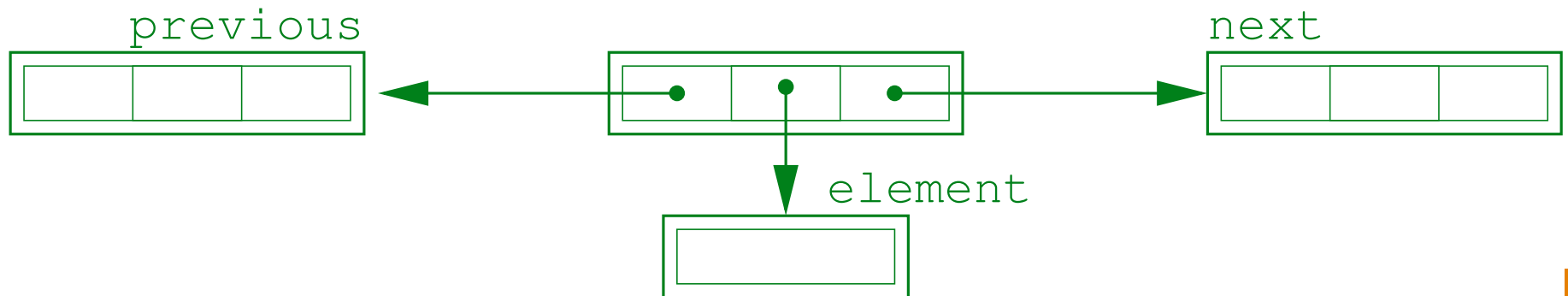
1. References
2. Singly Linked List
3. Stacks and Queues
4. **Doubly Linked List**
5. Using Linked Lists
6. Skip Lists



# Doubly linked list

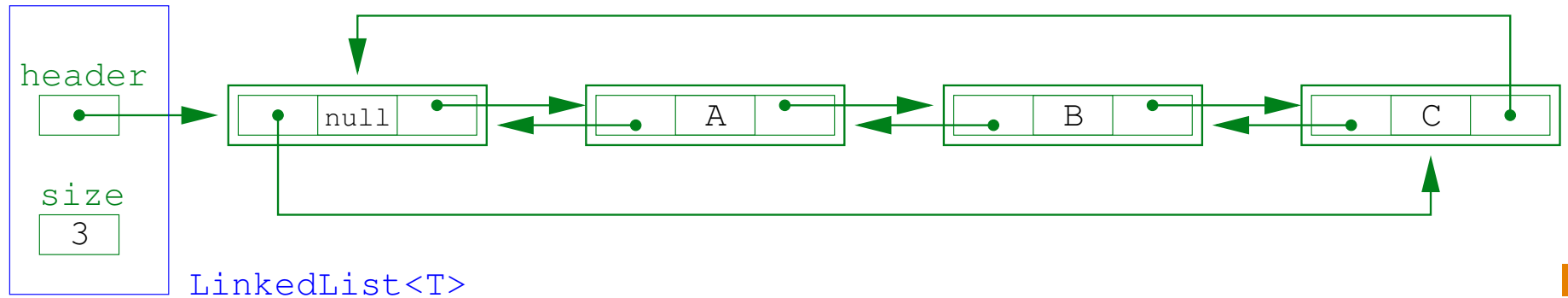
- In a more powerful linked list we would like to navigate the list in either direction
- To achieve this it uses a doubly-linked lists with elements to next and previous

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



# Dummy Node

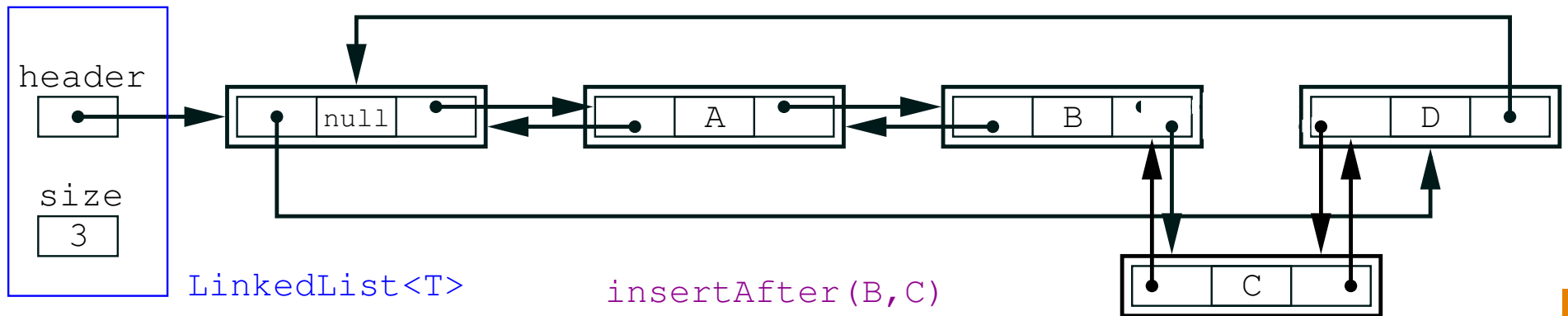
- List includes a dummy node—this make the implementations slicker



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

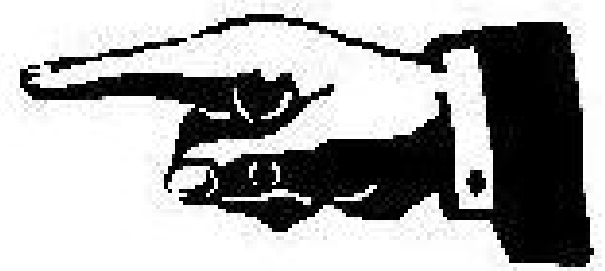
# Time Complexity

- 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



# Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. **Using Linked Lists**
6. Skip Lists



# When To Use Linked Lists

- It is difficult to think of applications where linked lists are the best data structure■
- lists—variable length arrays are usually better■
- queues—linked list OK, but circular arrays are probably better■
- sorted lists—binary trees much better■
- linked lists have efficient insertion and deletion but it is difficult to think of an application where this matters■

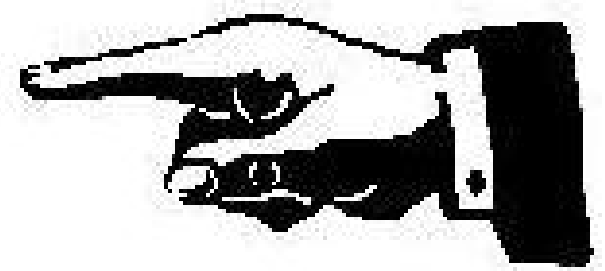
# Line Editor

- 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■



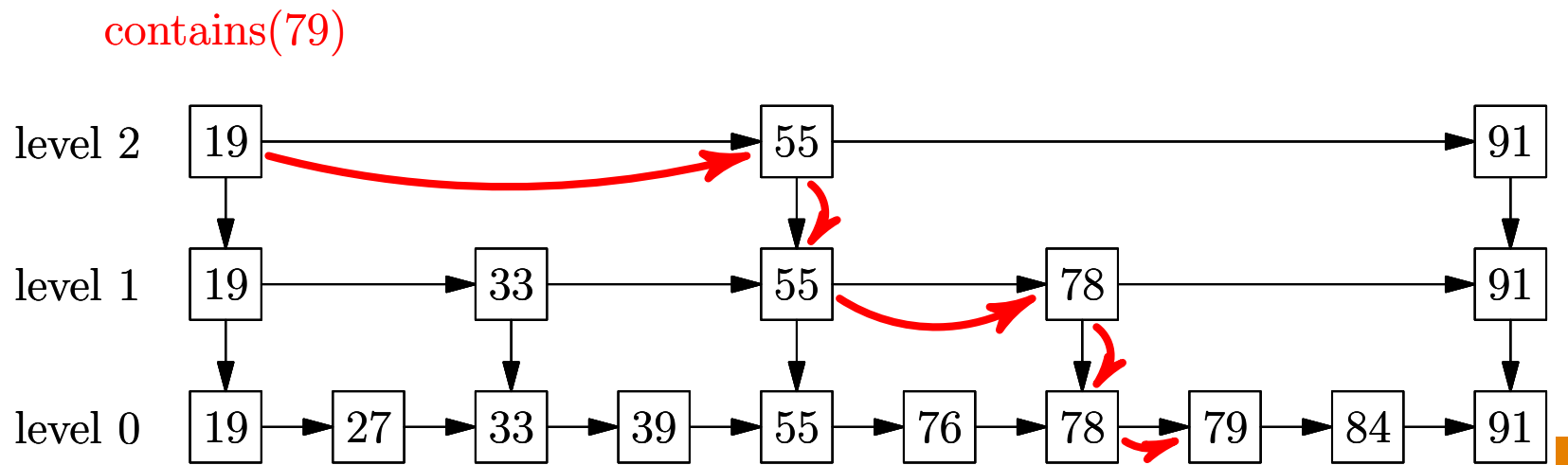
# Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. **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 allow binary search



# Efficiency of Skip Lists

- Skip lists provide  $\Theta(\log_2(n))$  search as opposed to  $\Theta(n)$ ■
- They have the similar time complexity to binary trees, although binary trees are slightly faster■
- They have one advantage over binary trees—they allow efficient concurrent access■
- The standard template library provides a doubly linked list, `list<T>`■, as well as a singly linked-list `slist<T>`■

# 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 has 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)■