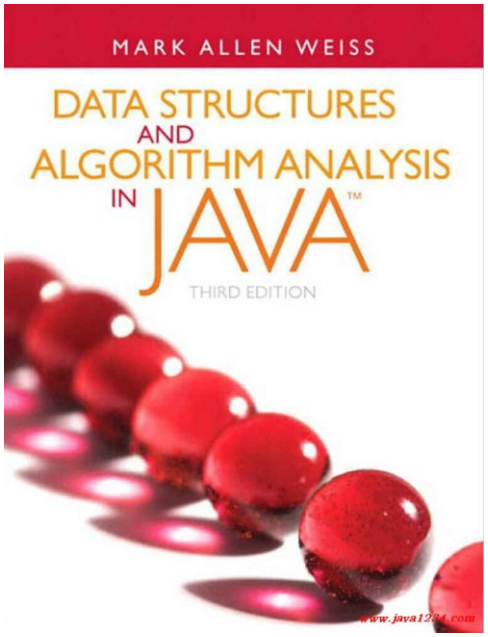
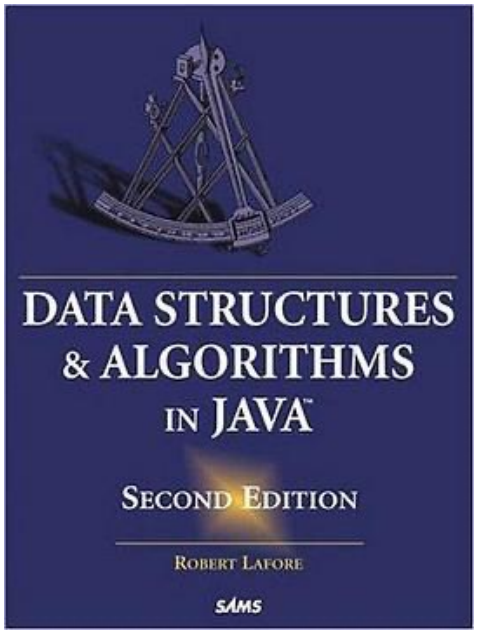
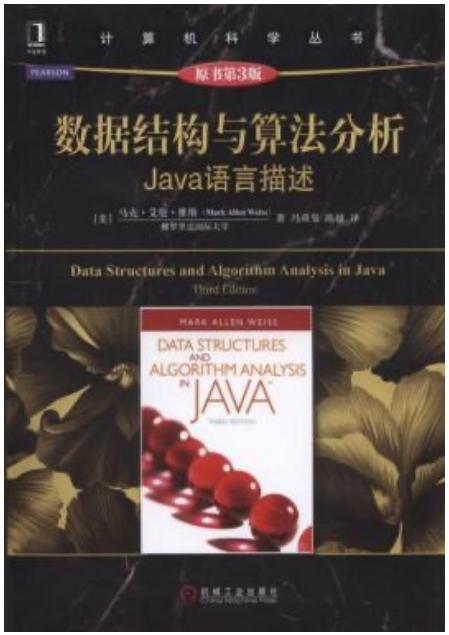
Topic 8 – Linked Lists







Topics

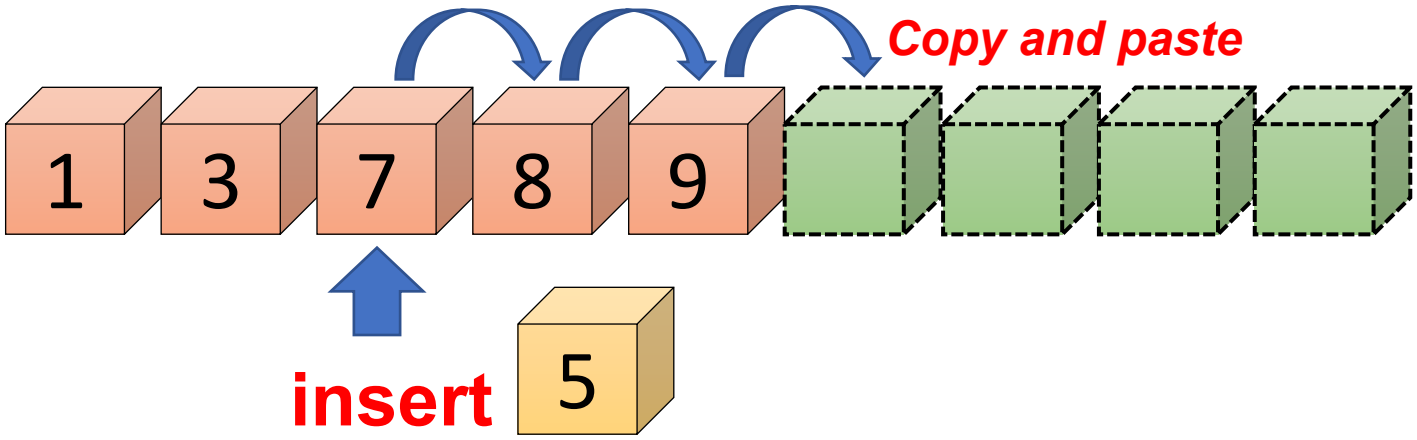
- Introduction
- Programming Revision
- Methods and Objects
- Arrays and Array Algorithms
- Big O Notation
- Sorting Algorithms
- Stacks and Queues
- **Linked Lists**
- Recursion
- Bit Manipulation

Content

- **Singly Linked List**
- Doubly
- Iterators
- Stacks using Linked List

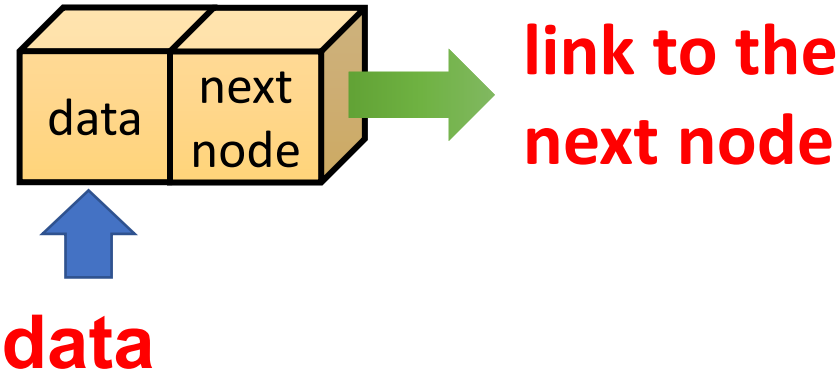
Why are arrays limited?

- Every data structure we have considered thus far has involved the use of arrays
- The main problem with arrays is that in order to move an item from one slot to another it has to be copied, pasted and deleted
- This is very time consuming – can we come up with an alternative that avoids this process?

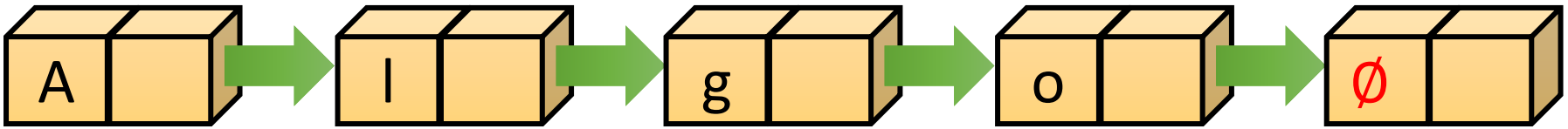


Linked List

- A linked list is an abstract data structure consisting of a sequence of links
- Each link (also called a node) stores
  - data
  - link to the next node



- Example.
  - “Algo”
  - **Stop at Ø**

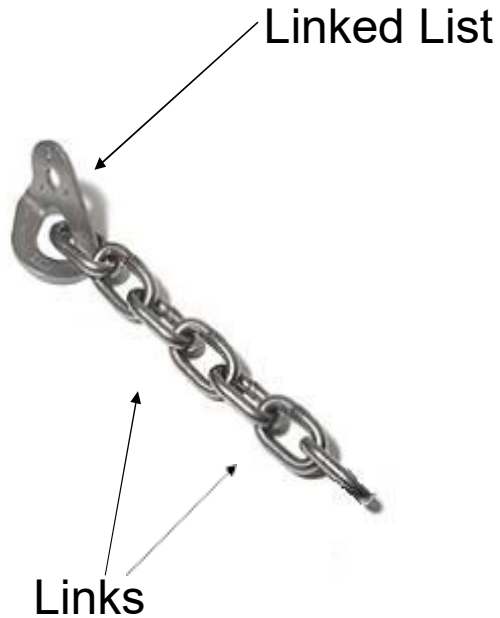


Advantages

- Why are linked lists better than arrays?
- Arrays waste space because they aren't always full
  - For example, you set "int arr[1000]" to store score of this class.
  - But we only have 50 students.
  - Then arr[50] to arr[999] will not use.
- When they get full it is not easy to extend them
- If you want to insert a new element in a particular slot you have to copy all the items that need to be moved
- Linked lists avoid all of these problems because they can adapt their ordering by changing the items they point to – no memory is wasted and extra links can easily be added

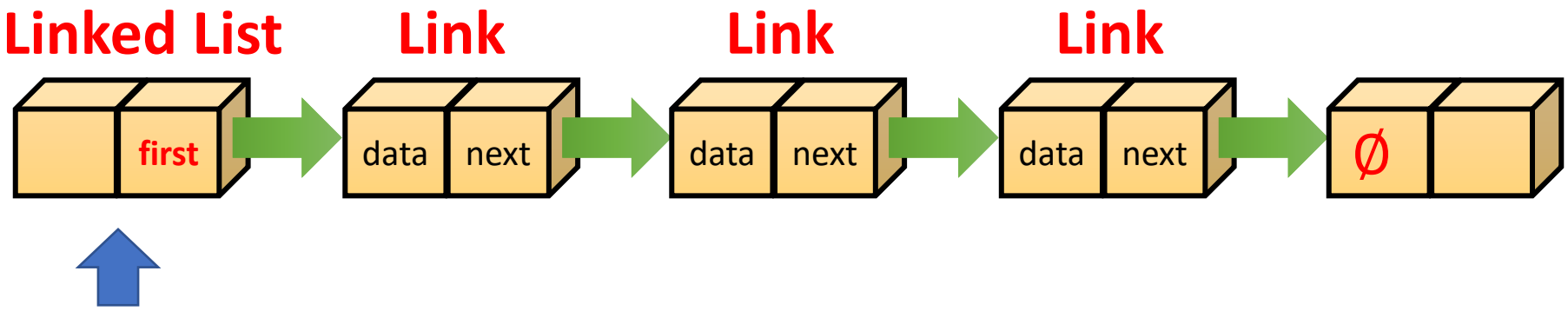
Structure

- One special type of class called **Linked List** stores the first or anchor link for all the subsequent Link objects
- A **Linked List** object is instantiated to point to the start of the list
- **Link** objects are created for each link in the list
- All contain references to the next link in the list



Structure

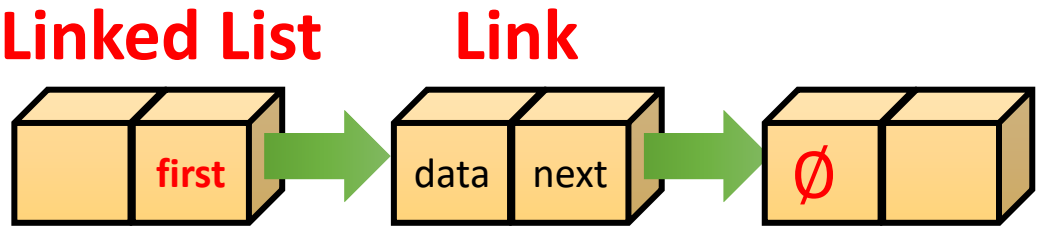
- The Linked List class stores a reference to the **first link**
- The Link class stores a bit of data and a reference to the next link




```
public LinkedList( ) {
    first = null;
}
```

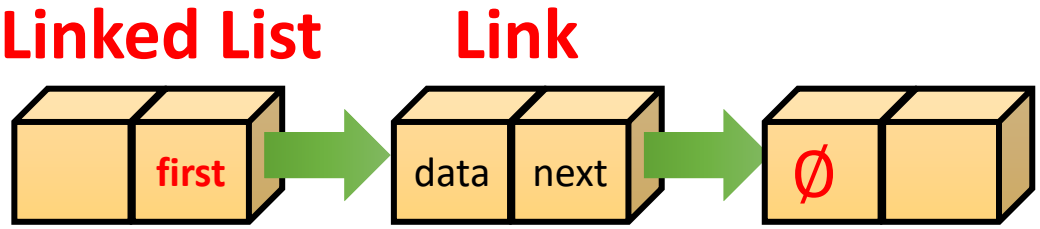
Java Implementation

```
class LinkedList {
    private Link first;
    public LinkedList( ){
        first = null;
    }
    public boolean isEmpty( ){
        return (first == null);
    }
    ……
}
```



- Linked list only contains a reference to the first link
- This is originally set to null

Java Implementation




```
public class Link {
    public int data;
    public Link next;
    public Link(int datain) { // constructor
    data = datain; // initialize data // 'next' is automatically set to null
    }
}
```

- Note that although the Link object contains another Link (next), this is only a reference to the next Link
- This is the same for any object, the label (e.g. next) is only a reference and does not represent the actual object itself
- next = new Link( ) : next acts as a reference to this object which is created somewhere else in memory

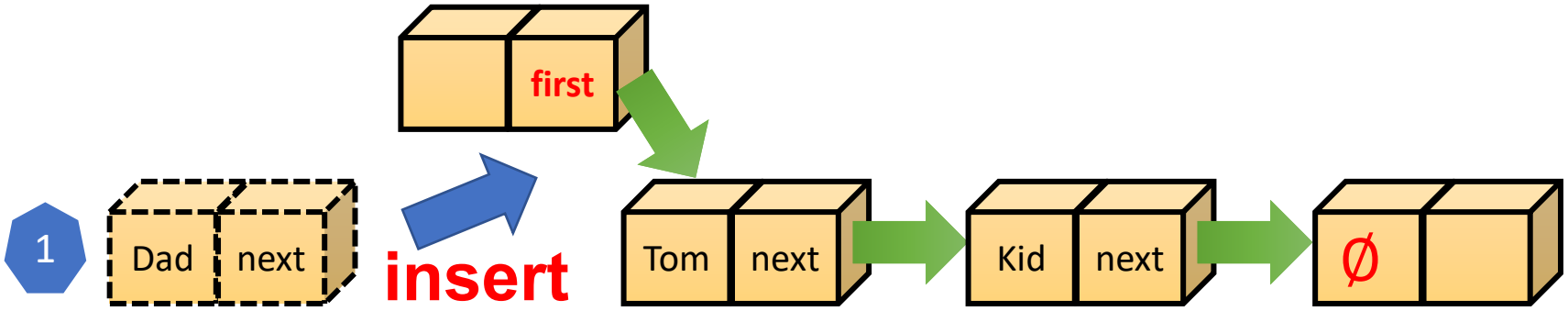
Inserting at the Head

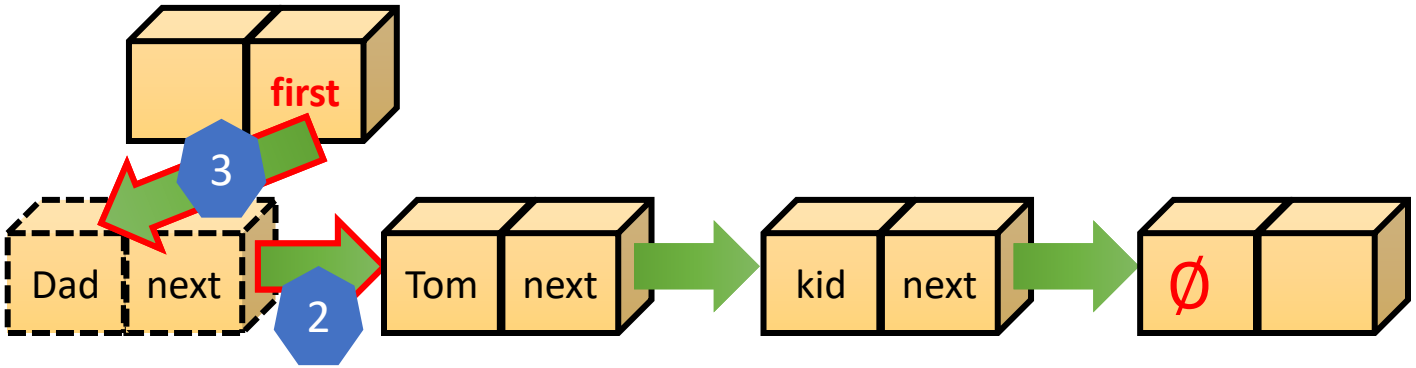
- The **head** is the **first link**
- The **tail** is the **last link**

- When inserting or removing links we always take care not to break the chain
- 1. Create a new link
- 2. Have new link point to old first link
- 3. Update Linked List to point to new link

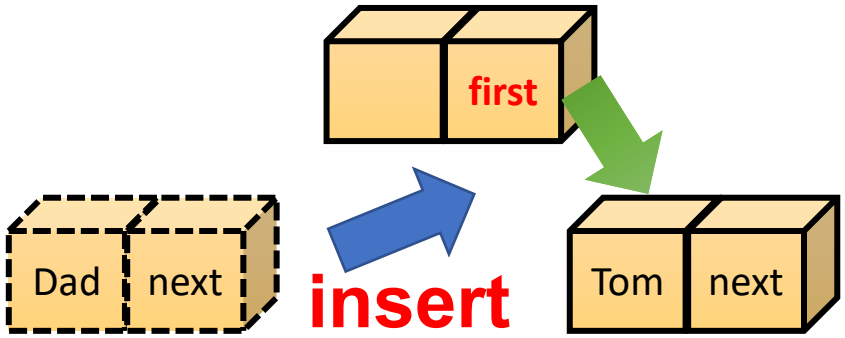
Inserting at the Head

- 1. Create a new link
- 2. Have new link point to old first link
- 3. Update Linked List to point to new link

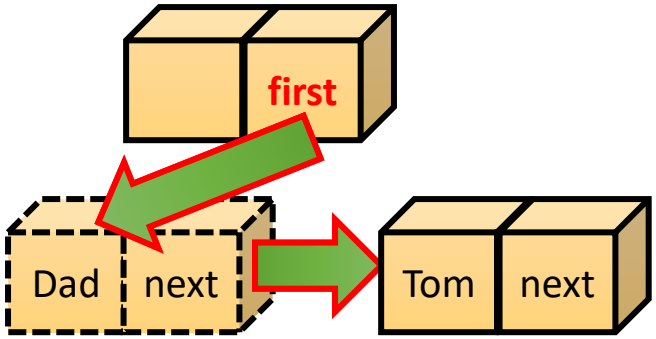




Inserting at Head



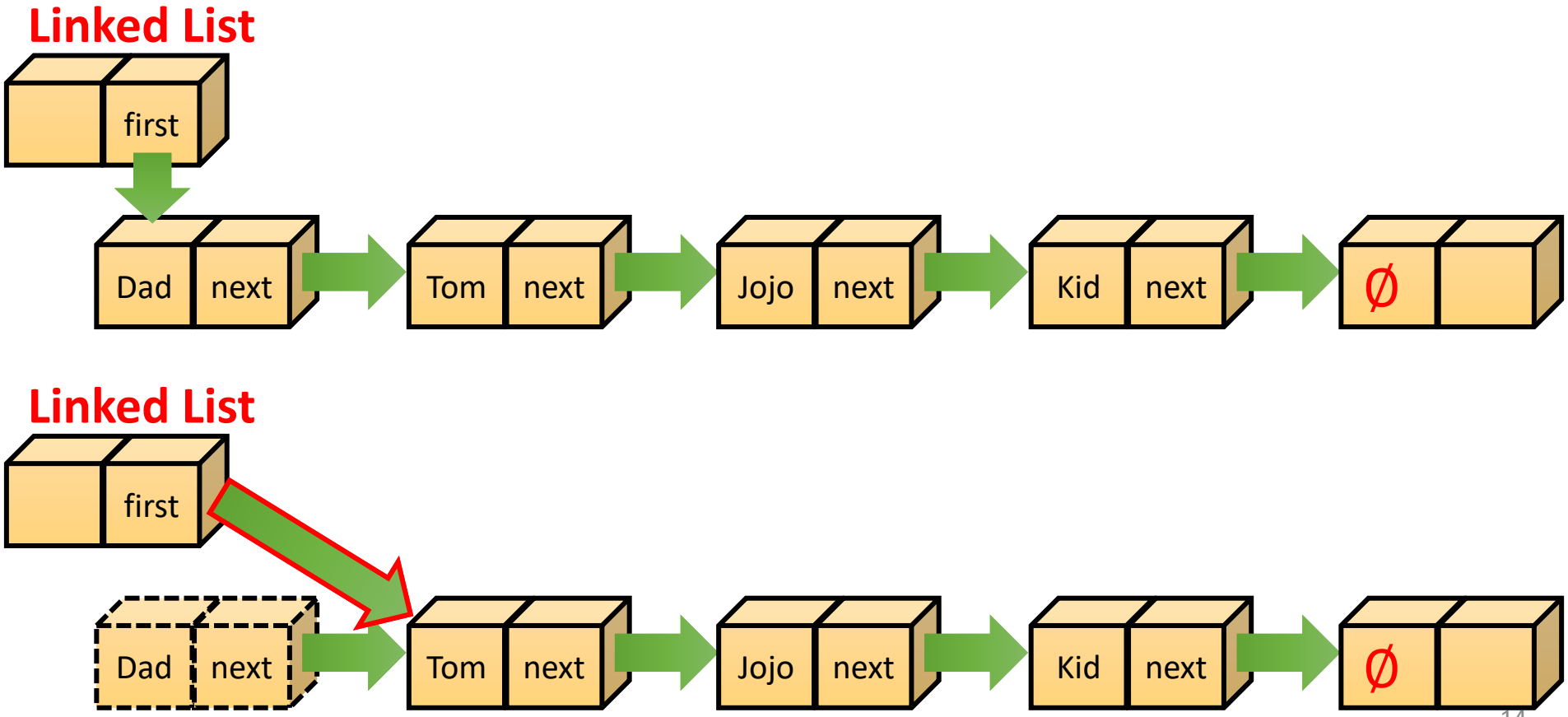

```
public void insertHead (int number) {
    Link newLink = new Link(number);
    newLink.next = first;
    first = newLink;
}
```



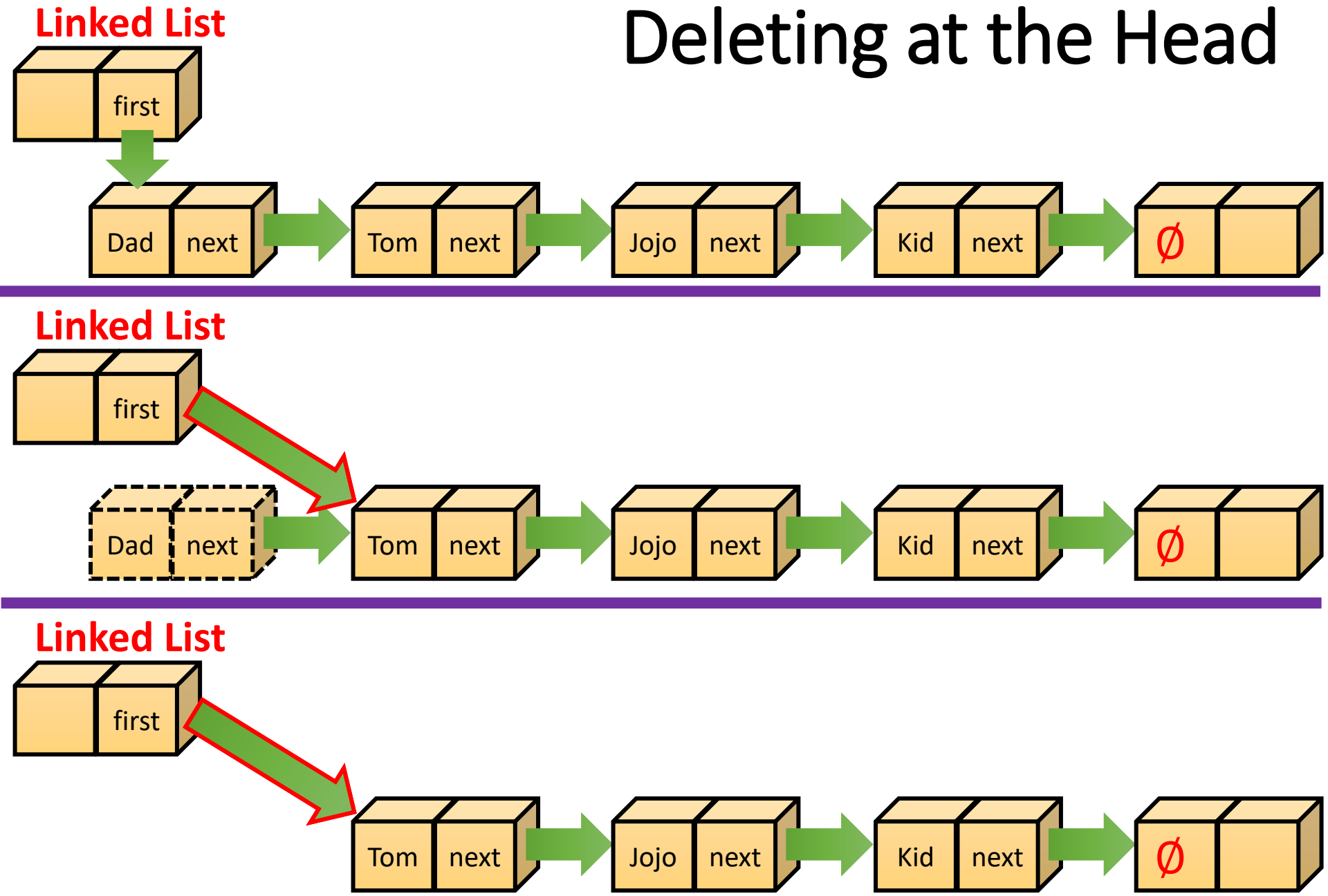
- This insertHead( ) method in LinkedList does the following
  - takes in a number
  - creates a new link with that piece of data
  - sets that link to point to the old first link
  - sets the first link to point to the new link

Deleting at the Head

- Update head to point to next node in the list
- Garbage collector will now reclaim the former first node

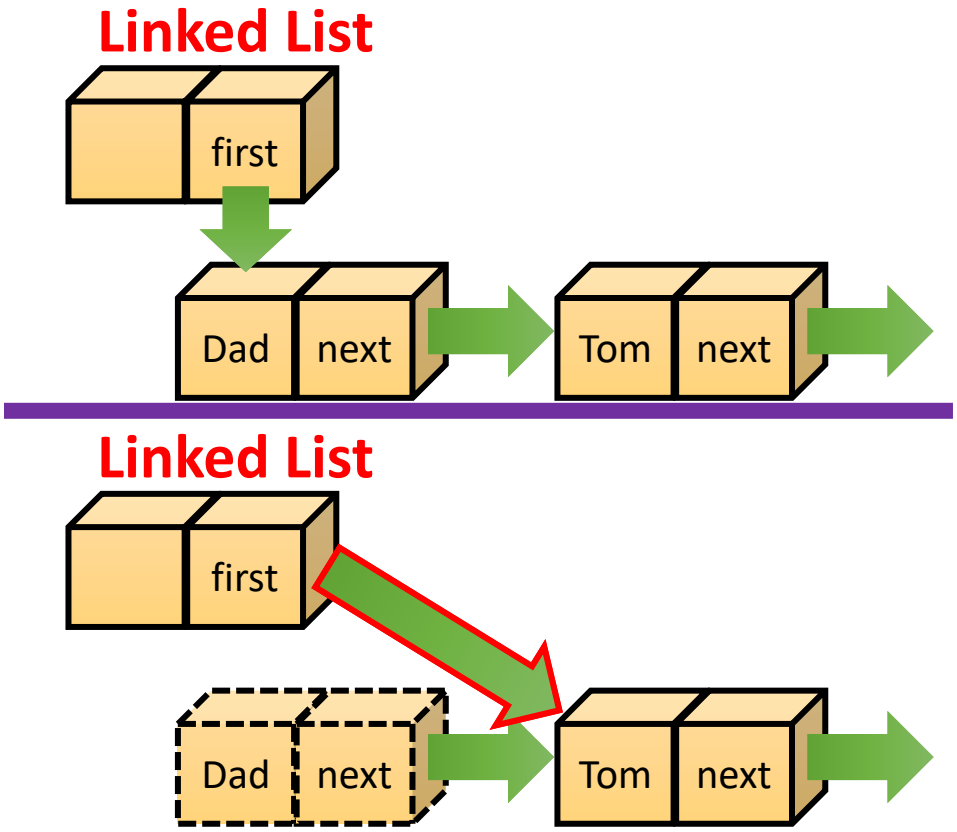


Deleting at the Head



Delete at Head

```
public Link deleteHead () {
    Link temp = first;
    first = first.next;
    return temp;
}
```



- This method in LinkedList does the following
  - Backs up the first link
  - Gets the link after the first link
  - Puts the old first link equal to that
  - Returns the link that has been ‘bypassed’

Traversing a linked list

- With arrays, each item occupies a particular slot, and we can index any slot we like
- With a linked list, you have to start at the beginning of the chain and work your way along in order to find an item
- Binary search not possible – big disadvantage of linked lists

```
public void display() {
    Link current = first; // start with first link
    while(current != null) {
        current.displayLink(); //print out the link
        current = current.next; //keep going until you come to the end
    }
}
```

```
public void displayLink() {
    System.out.println("{" + data + "}");
}
```

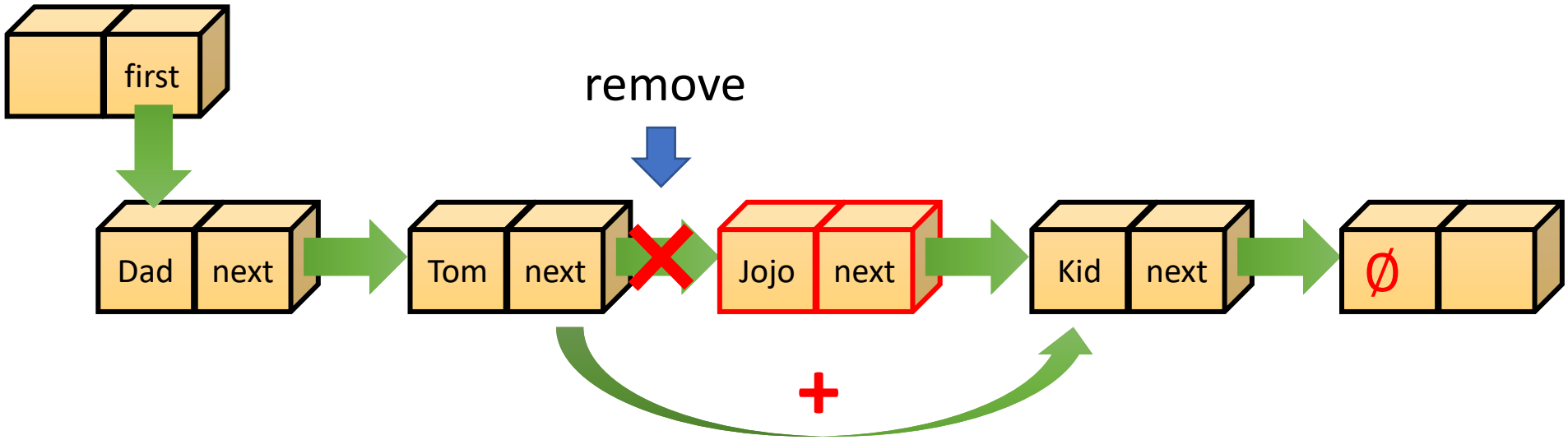
Finding a link

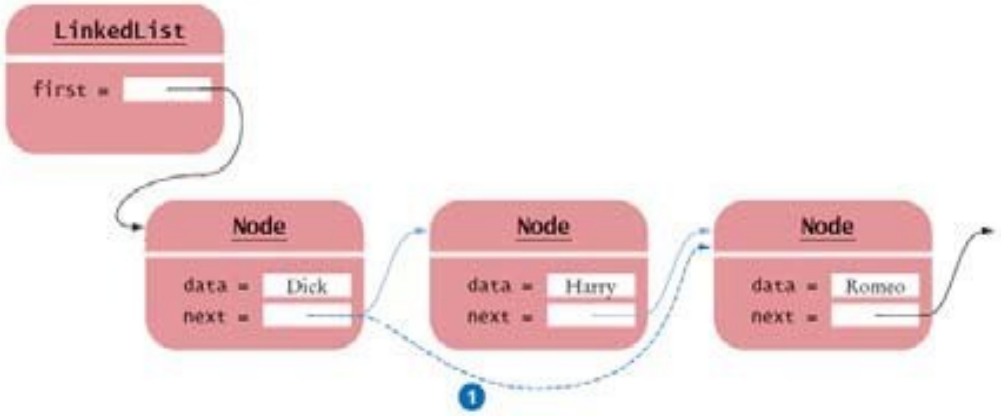
- What if we want to find or delete one particular node?

- The algorithm works like this:
  - Take in the value to be found or deleted
  - Start at the beginning of the list
  - Keep moving down the links until we find the correct one
  - All the while keep tracking the current link and the previous link (so we can join them up when required)
  - Now update the references to bypass the link to be deleted

Removing a Node From the Middle of a Linked List

Linked List





Removing a Node From the Middle of a Linked List

```
public Link delete(int key) { // delete link with given key
        Link current = first; // search for link
        Link previous = first; //put these equal to first Link
        while(current.data != key) {
                if(current.next == null) { return null; } // didn't find it
                else {
                        previous = current; // go to next link
                        current = current.next;
                }
        } // found it
        if(current == first){ // if first link,
                first = first.next; // change first
        } else{ // otherwise,
                previous.next = current.next; // bypass it
        }
        return current;
}
```

//NOTE: Assumes list is not empty and no duplicates

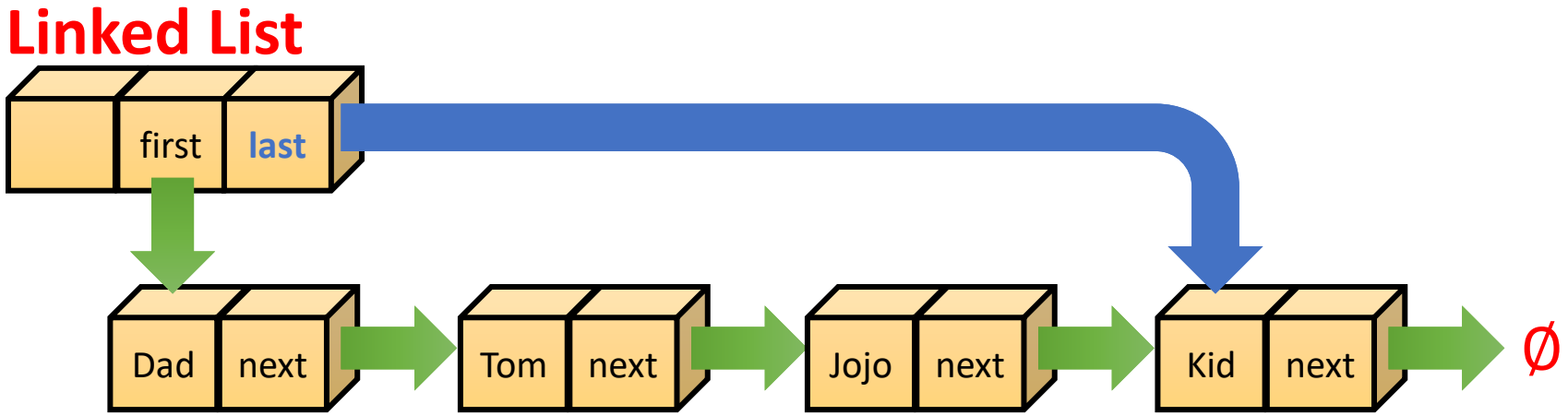
Exercise

- Write your Link and LinkedList classes with the following methods

- Link
  - displayLink()
- LinkedList
  - isEmpty()
  - insertHead()
  - deleteHead()
  - display()
  - delete(int key)

Double-Ended Linked Lists

- A double-ended list is similar to an ordinary linked list with one additional feature
- It has references to the last link as well as the first
- This allows a new link to be inserted or deleted at the end as well as the beginning
- Handy for implementing a queue (items arrive one end and leave at the other)

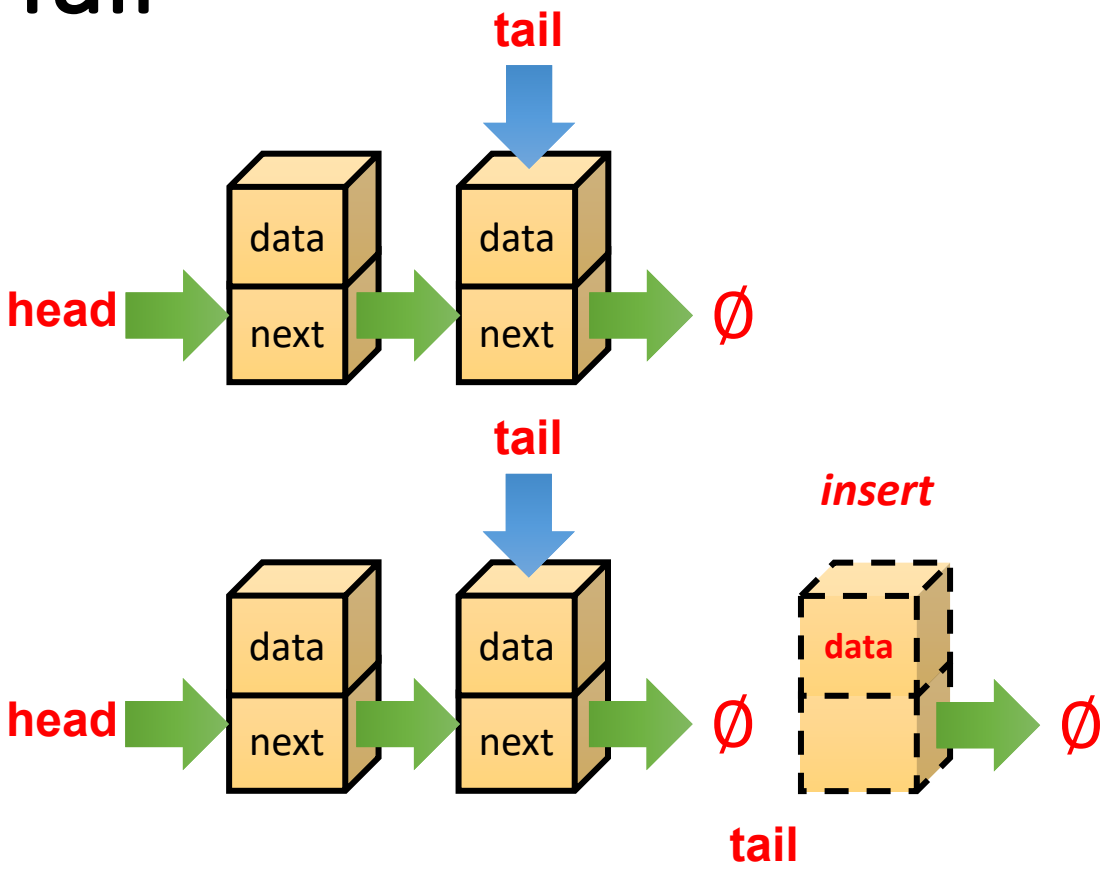


Double-Ended Linked Lists

- A double ended Linked List class will have the following variables
  - first/head
  - last/tail
- It will have the following methods
  - insertFirst(int data)
  - insertLast(int data)
  - deleteFirst( )
  - deleteLast( ) (requires a doubly-linked list)
- In these methods, the variables first and last are updated accordingly

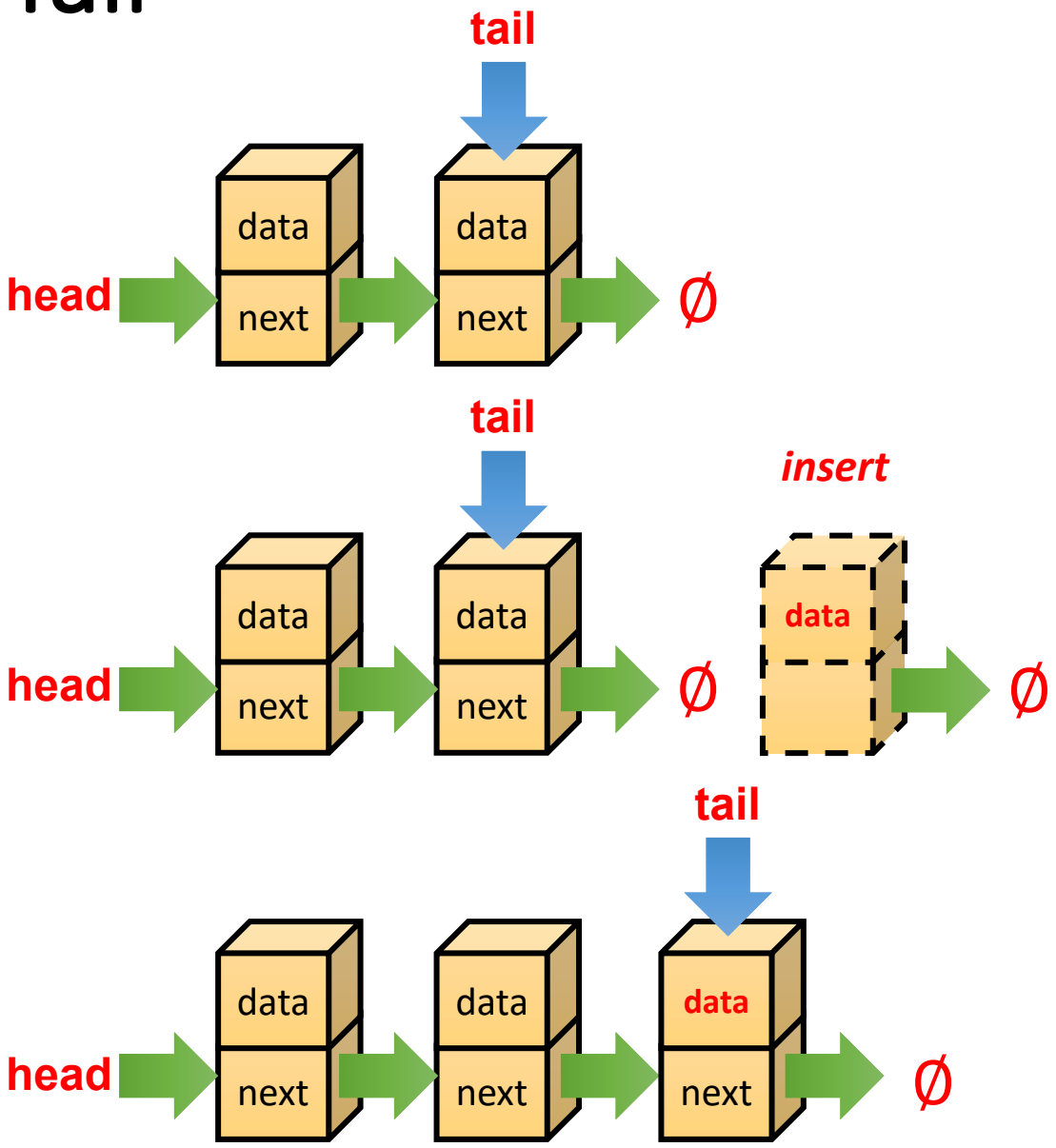
Inserting at the Tail

- Create a new link
- Have new link point to null



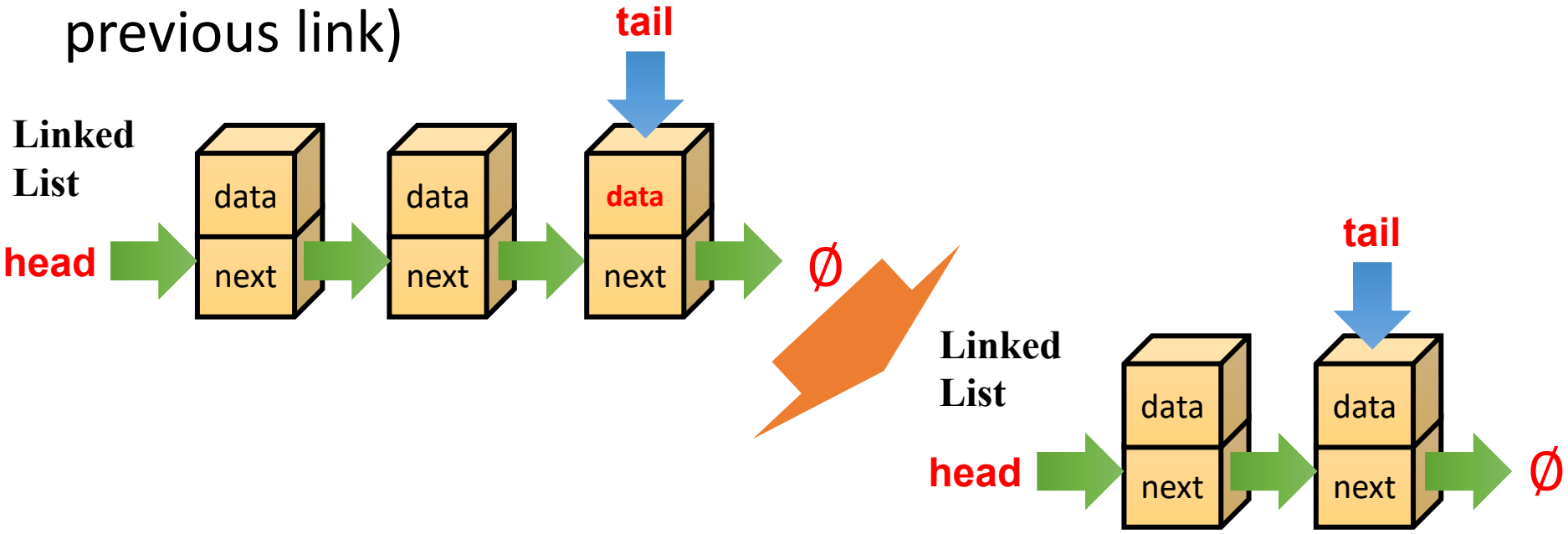
Inserting at the Tail

- Have old last link point to new node
- Update tail of Linked List object to point to new node



Removing at the Tail

- Removing at the tail of a singly-linked double ended list is not efficient!
- There is no constant time way to update the tail to point to the previous node
- We need to use double links (that point to both next and previous link)



Linked-List Efficiency

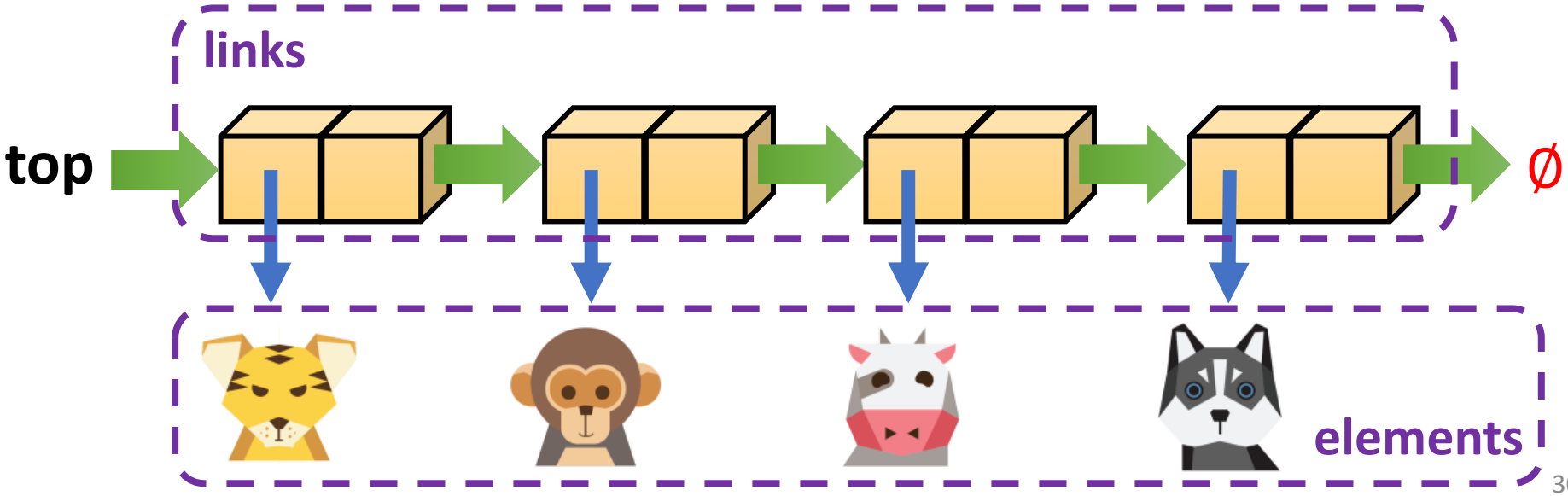
- Insertion and deletion at the beginning of a link list are very fast ➔ $\boldsymbol{O(1)}$
- Finding, deleting or inserting in a particular location requires traveling through an average of half the links ➔ $\boldsymbol{O(n)}$
- However, nothing needs to be copied, so it is a faster $O(n)$ than array insert – only references are updated
- Major advantage is that no memory is wasted
- Vectors in Java are expandable arrays but these usually expand more than they need to

Abstraction

- It doesn’t matter if we implement the stack or queue using an array or linked list
- In object-oriented programming an Abstract Data Type (ADT) is considered without regard to its implementation
- We only want to know how to use the methods, not how they carry out their tasks
- The ADT specification is called the **interface** (e.g. in a stack these would be **push( )** and **pop( )**

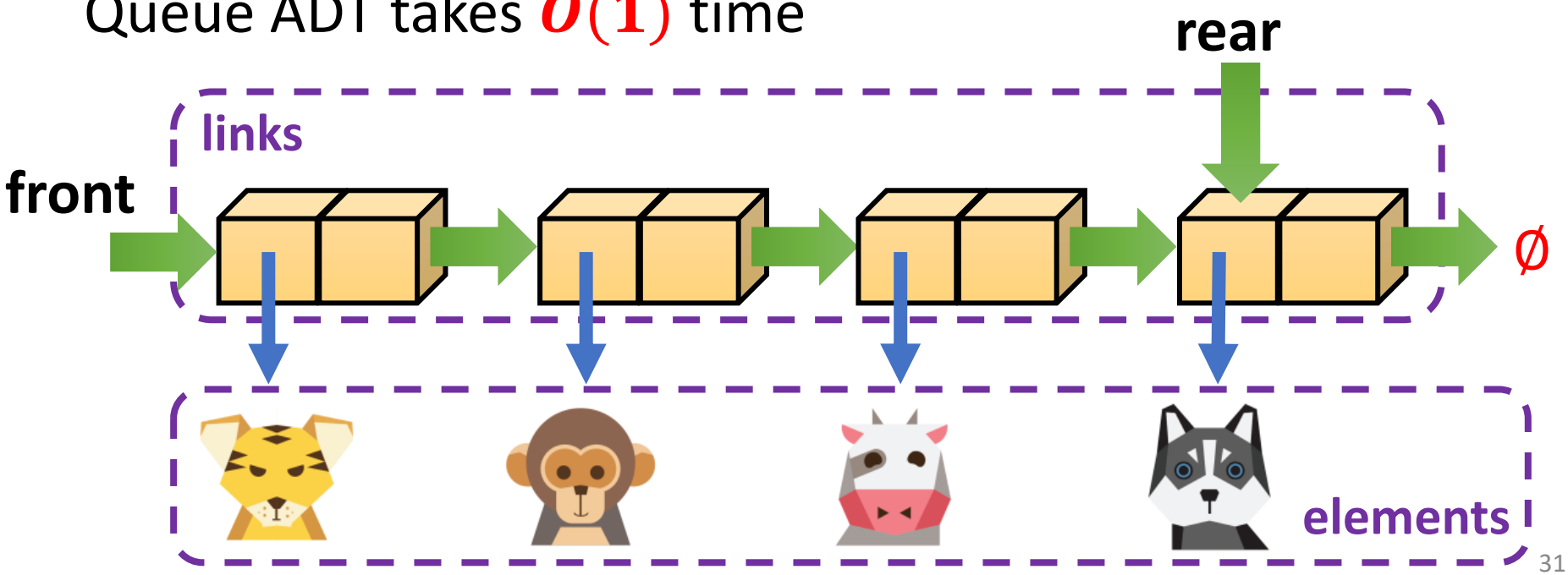
Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $\boldsymbol{O(n)}$ and each operation of the Stack ADT takes $\boldsymbol{O(1)}$ time



Queue with a Singly Linked List

- We can implement a queue with a double ended linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $\boldsymbol{O(n)}$ and each operation of the Queue ADT takes $\boldsymbol{O(1)}$ time

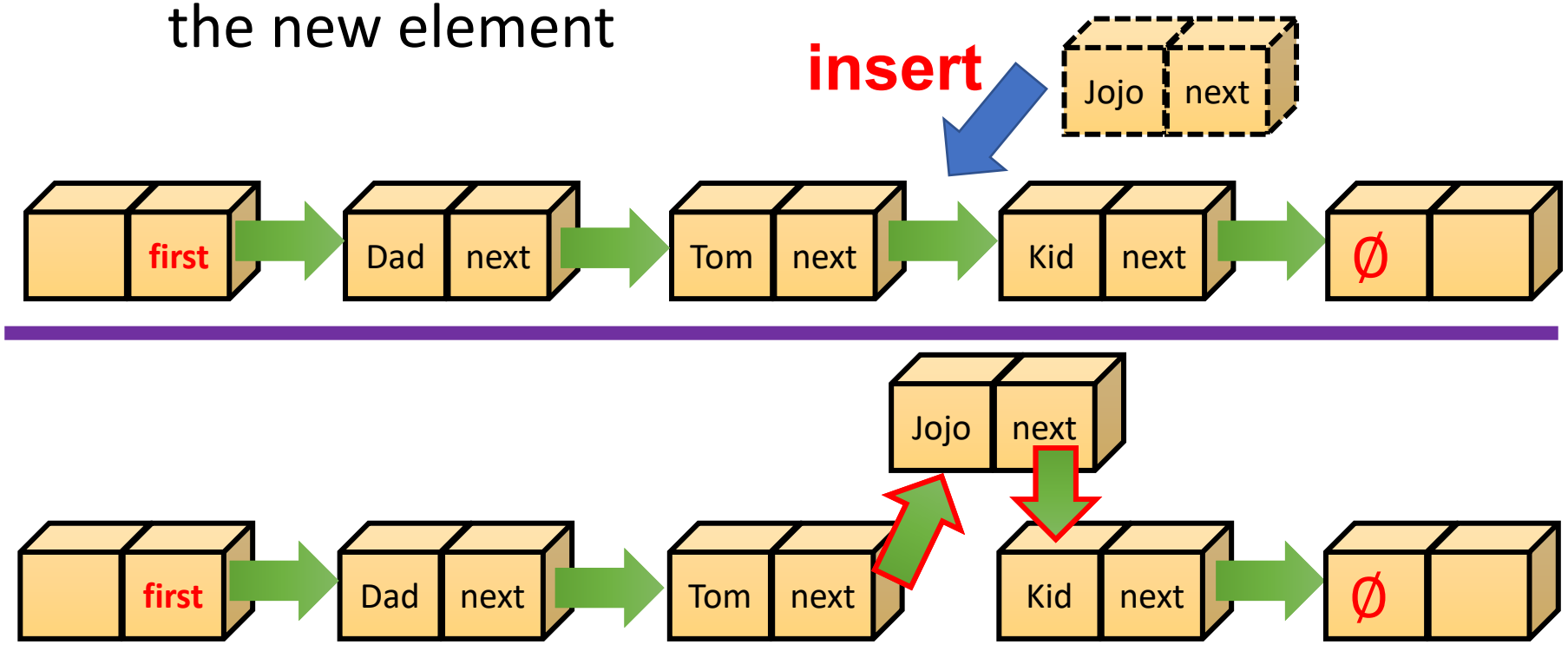


Sorted Lists

- Remember the priority queue – this required the items to be sorted
- In order to keep a linked list sorted, we have to insert at particular locations
- Only way to find the correct location is to search through the list – can't use binary search
- When you find the correct location (i.e. when you find a value bigger than the one you want to insert) update the pointers of the relevant links

Insertion

- After finding the correct location for the item we must update the pointers
  - The new element must be set to point to the next element in the linked list
  - The previous element must be updated to point to the new element



Insertion to ordered list

```
public void insertOrdered(Link newlink) { // insert (in order)
        Link previous = null; // start at first
        Link current = first; // until end of list
        while(current != null && newlink.data > current.data) { // while key > current
                previous = current;
                current = current.next; // go to next item
        }
        if(previous==null) { // at beginning of list
                newlink.next = first; // newlink -> old first
                first = newlink; // first --> newlink
        }
        else{ // not at beginning
                previous.next = newlink; // old prev --> newlink
                newlink.next = current; // newlink --> old current
        }
} // end insert()
```

New link needs to be inserted at the beginning

Efficiency of Sorted Linked Lists

- Insertion and deletion of items in the sorted linked lists requires $\boldsymbol{O(n)}$ comparisons ($\frac{\boldsymbol{n}}{\boldsymbol{2}}$ on average) because we have to step through half of the list
- The minimum (or max) can be found or deleted in $\boldsymbol{O(1)}$ time (will be at the top of the sorted linked list)
- If an application frequently accesses the minimum item and fast insertion isn't critical then this is a good choice

Using linked lists for array sorting

- Take the items from your unsorted array
- Put them into a sorted linked list one by one
- You can then put them back in the array and it will be sorted
- Actually turns out to be significantly more efficient than insertion sort because fewer copies are necessary!
- Still $\boldsymbol{O(n^2)}$ comparisons, however each item is only copied twice
  - Once from the array to the linked list
  - Once back from the linked list into the array
- $\boldsymbol{2n}$ copies is much better than the usual $\boldsymbol{O(n^2)}$ copies using Insertion Sort

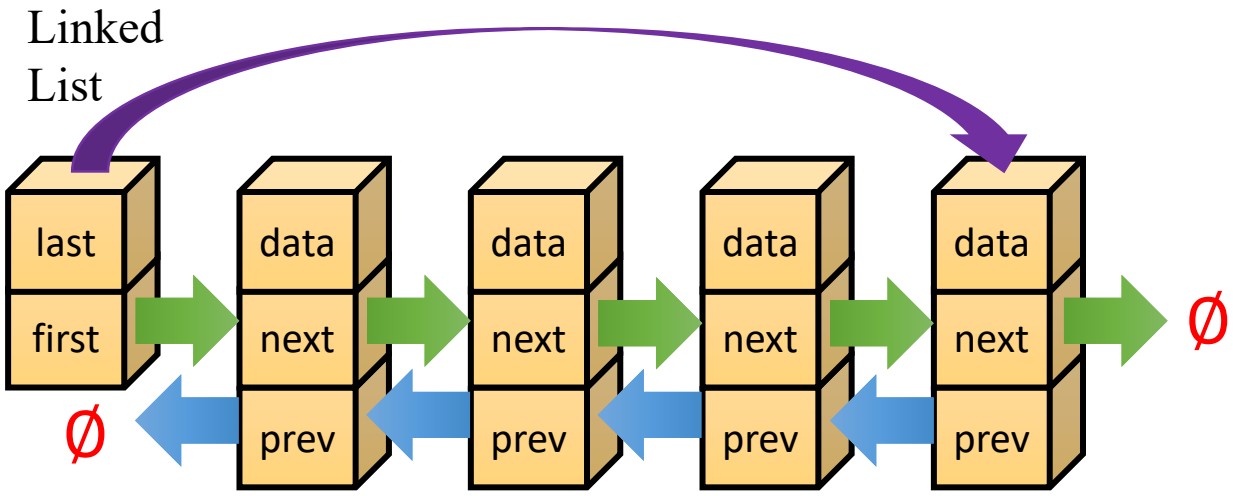
Content

- Singly Linked List
- **Doubly Linked List**
- Iterators
- Stacks using Linked List

Doubly Linked Lists

- Removing an element at the tail of a singly linked list is not easy since
  - **We can't move backwards!!!**
- Indeed, there are many times we need to know the **predecessor** as well as the successor of a particular node
- A linked list where you can traverse it forwards or backwards is a **doubly linked** list (references going both ways)

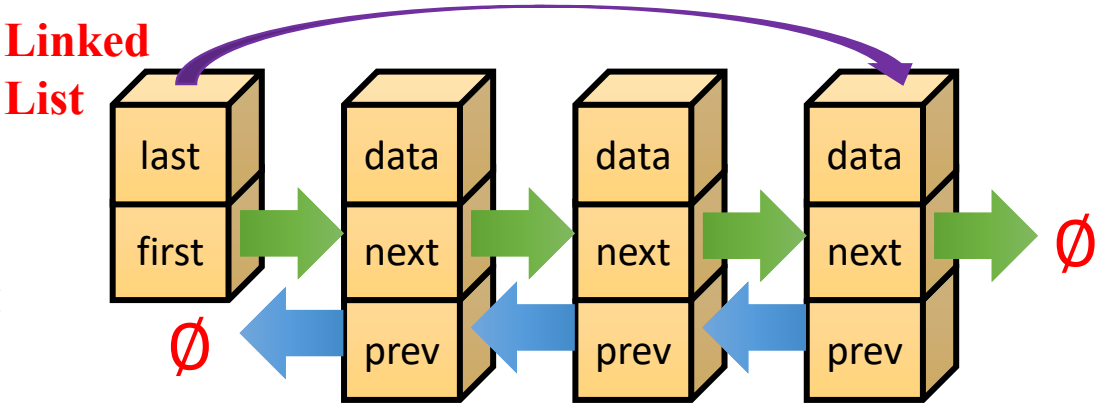
Double-Ended Doubly Linked Lists



Double-Ended Doubly Linked Lists

- The specification for the link class now looks like this:

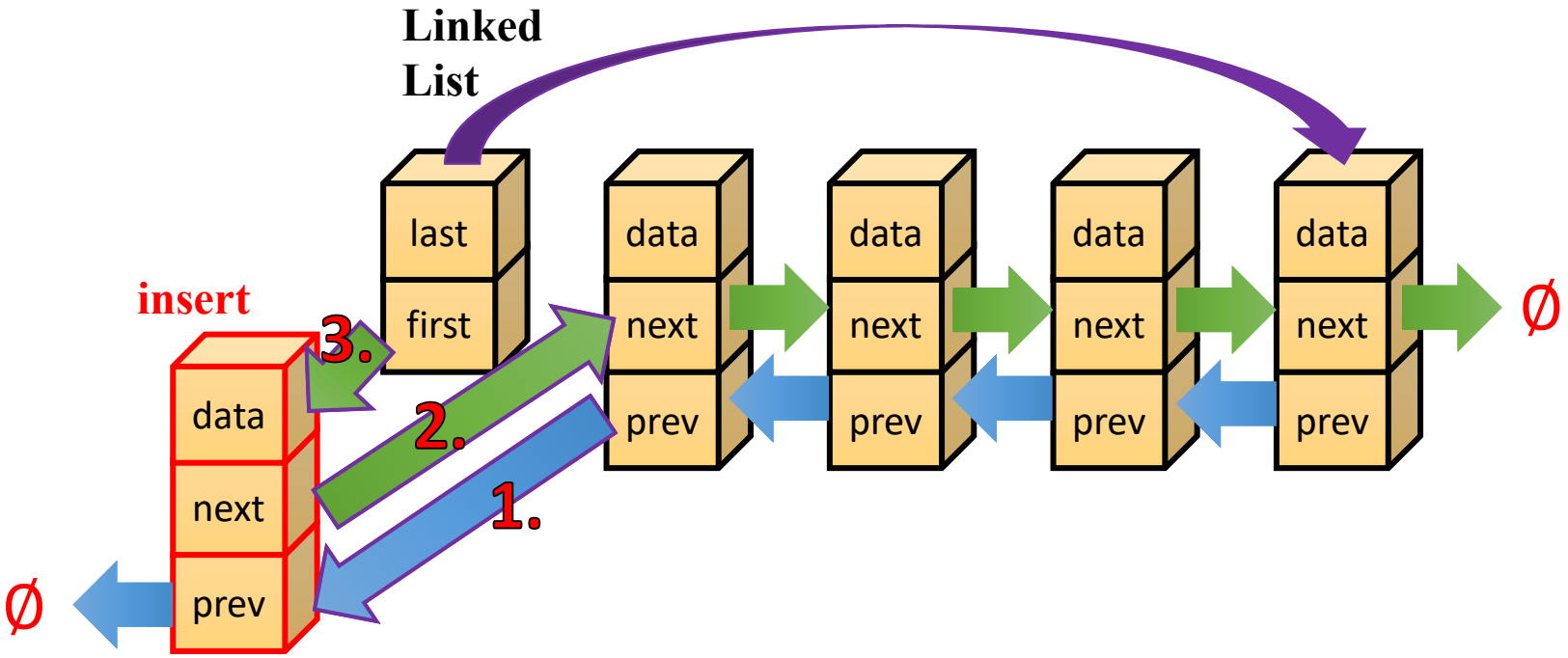
```
public class Link {
        public int data;
        public Link next;
        public Link previous;
}
```



- Now every time we insert or delete a link, we must deal with four links instead of two
  - Update the two attachments between the previous link and the new one
  - Update the two attachments between the next link and the new one
- Now we can move backwards or forwards through the list

Insertion at beginning

- **1.** – Change prev of first link
- **2.** – Change next of new link
- **3.** – Change first of Linked List
- Step 3 must be last or chain would be lost!!!



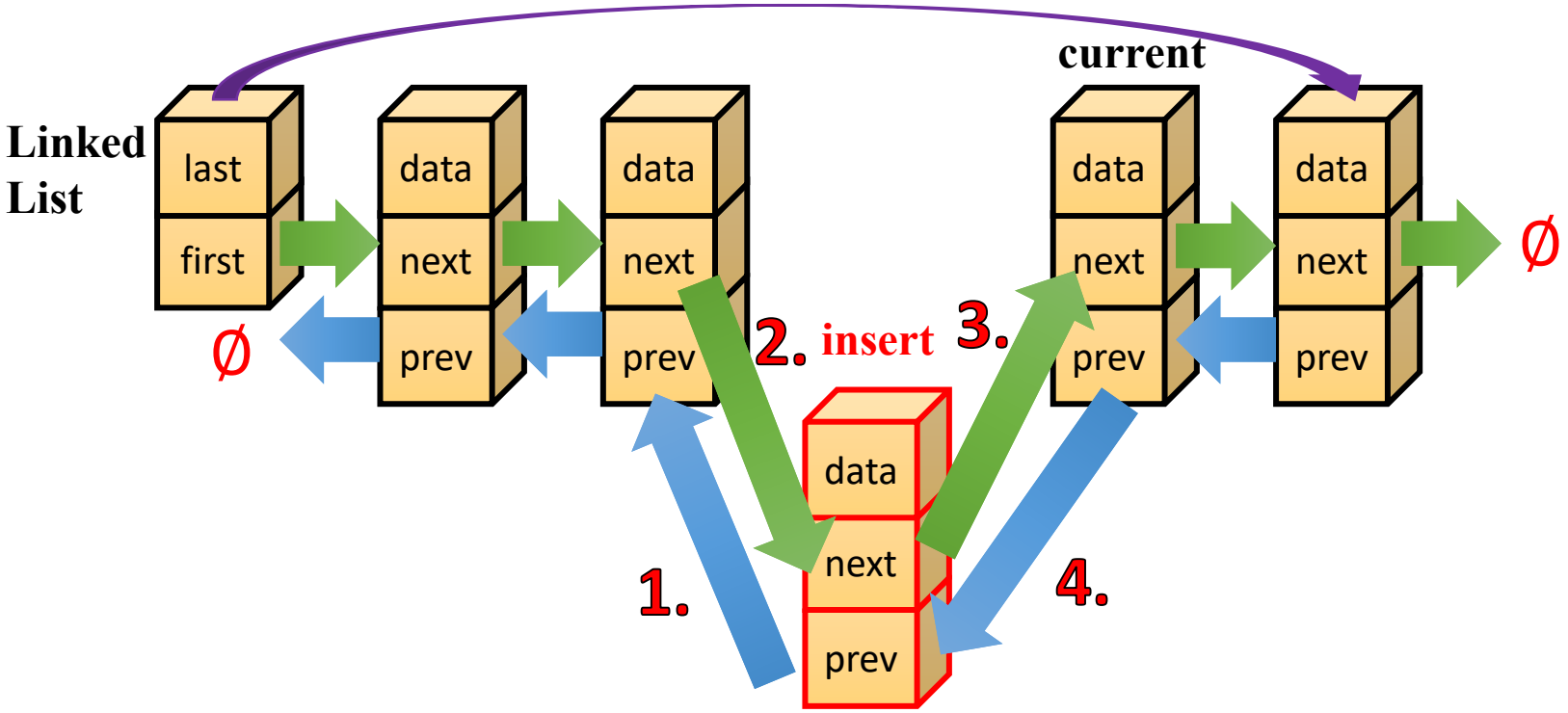
Java Implementation

```
public void insertHead(long data) { // insert at head
    Link newLink = new Link(data); // make new link

    if( isEmpty() ) { // if empty list,
        last = newLink; // newLink <-- last
    }
    else {
        first.previous = newLink; // 1. newLink <-- old first
    }
    newLink.next = first; // 2. newLink --> old first
    first = newLink; // 3. first --> newLink
}
```

Insertion in order

- **1.** – Change prev of new link
- **2.** – Change next of left link
- **3.** – Change next of new link
- **4.** – Change prev of right link

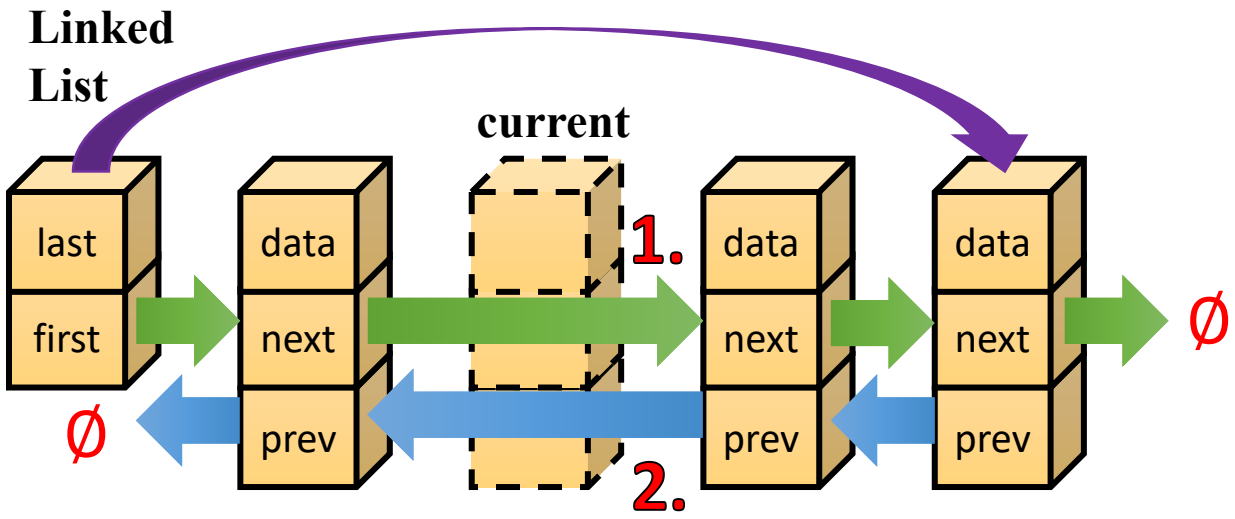


Java Implementation

```
public void insertOrdered(long data) { //inserts data in order
        Link current = first; // start at beginning
        while(current!=null && data > current.data) { current = current.next;
        } // move to next link
        Link newLink = new Link(data); // make new link
        if(current==first) { // if insertion at head
                insertHead(data);
        } else if(current==null){ //if insertion at tail
                insertTail(data);
        } else { // somewhere in middle
                newLink.previous = current.previous; // step 1
                current.previous.next = newLink; // step 2
                newLink.next = current; // step 3
                current.previous = newLink; // step 4
        }
}
```

Deletion

- **1.** – Change next of left link
- **2.** – Change prev of right link



Java Implementation

```
public Link delete (long key) { // delete item with given key
        Link current = first; // start at beginning
        while(current.data != key){ // until match is found
                current = current.next; // move to next link
        }
        if(current == null){ return null; } // didn't find it
        if(current == first){ // found it; first item?
                first = current.next; // first --> old next
        } else { // not first. // old previous --> old next
                current.previous.next = current.next;
        }
        if(current==last){ // last item?
                last = current.previous; // old previous <-- last
        } else { // not last item  // old previous <-- old next
                current.next.previous = current.previous;
        }
        return current; // return value
}
```

Exercise

- Write your DoublyLinkedList class

- Attributes:
  - first Link
  - last Link

- Functions:
  - isEmpty()
  - insertFirst(), insertLast()
  - deleteFirst(), deleteLast()
  - insertAfter()
  - deleteKey()
  - insertOrdered()

Content

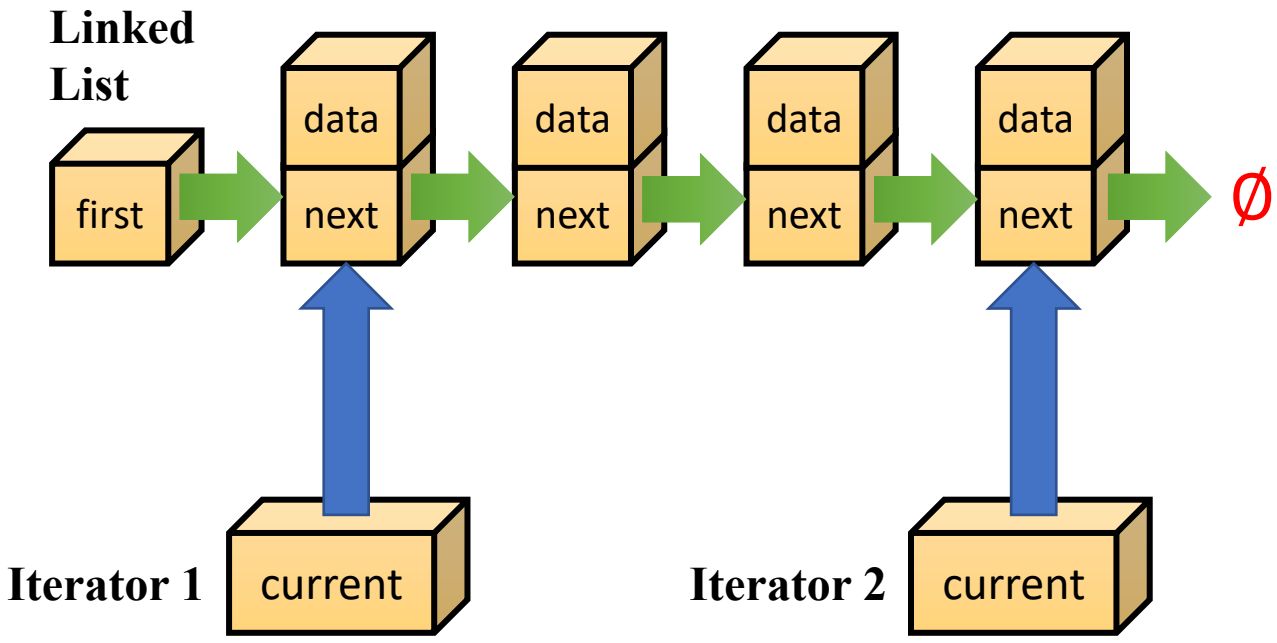
- Singly Linked List
- Doubly Linked List
- **Iterators**
- Stacks using Linked List

Iterators

- In order to find a link, we started at the beginning and searched all the way through
- This doesn't give the user any control over the items passed over
- Suppose you want to traverse a list and perform some operation on certain link, e.g.,
  - Increase the wages of all employees being paid the minimum without affecting anyone else's
  - Delete all customers in a database who haven't ordered anything for the past six months
- This is easy in an array because every element has a fixed index
- We need a method that can step from link to link performing an action on each one

List Iterator

- Iterators always point to some link in the list
- They are associated with the list but are not part of it



Iterator Class

- Objects containing references to items in data structures, used to traverse these structures, are commonly called ***iterators***
- The main advantage of using an object is that we can create as many references as we want

```
class ListIterator() {
        private Link current;
        …
}
```

- It is easier to get the linked list class to create the iterator, so we add a getIterator( ) method to it

```
LinkedList theList = new LinkedList();
ListIterator iter1 = theList.getIterator();
Link aLink = iter1.getCurrent(); //access link at iterator
iter1.nextLink(); // move iterator to next Link
```

Additional Iterator Features

- We've seen that's it's handy to store a reference to the **previous** link
- If our linked list isn't doubly-linked then the iterator should store both **current** and **previous** so that it can delete links
- It is also handy for the iterator to store a reference to the **linked list** class so it can access the first element of the list

Iterator Code

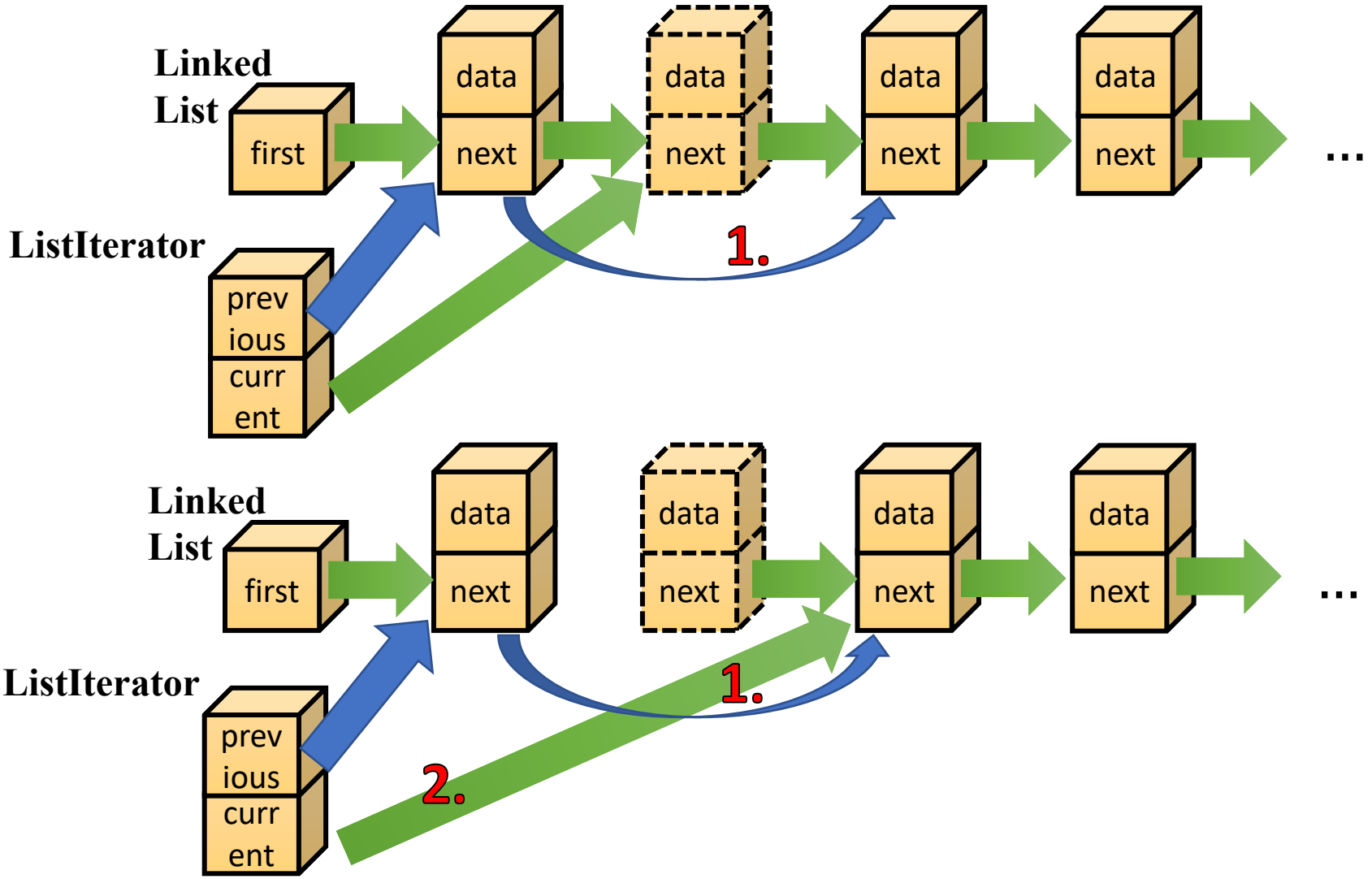
```
class ListIterator {
    private Link current; // current link
    private Link previous; // previous link
    private LinkList ourList; // our linked list
    public ListIterator(LinkList list) { // constructor
        ourList = list;
        reset();
    }
    public void reset() { // start at 'first'
        current = ourList.getFirst();
        previous = null;
    }
    public void nextLink() {
        previous = current; //set previous to this
        current = current.next; //set this to next
    }
```

Iterator Methods

- Additional methods can make the iterator a flexible and powerful class
- All operations that involve iterating through the list are more naturally performed by the iterator
  - reset() – Sets the iterator to the start of the list
  - nextLink() – moves the iterator to the next link
  - getCurrent() – returns the link at the iterator
  - atEnd() – returns true if iterator is at end of list
  - insertAfter() – inserts a new link after the iterator
  - insertBefore() – inserts a new link before the iterator
  - deleteCurrent() – deletes the link at the iterator

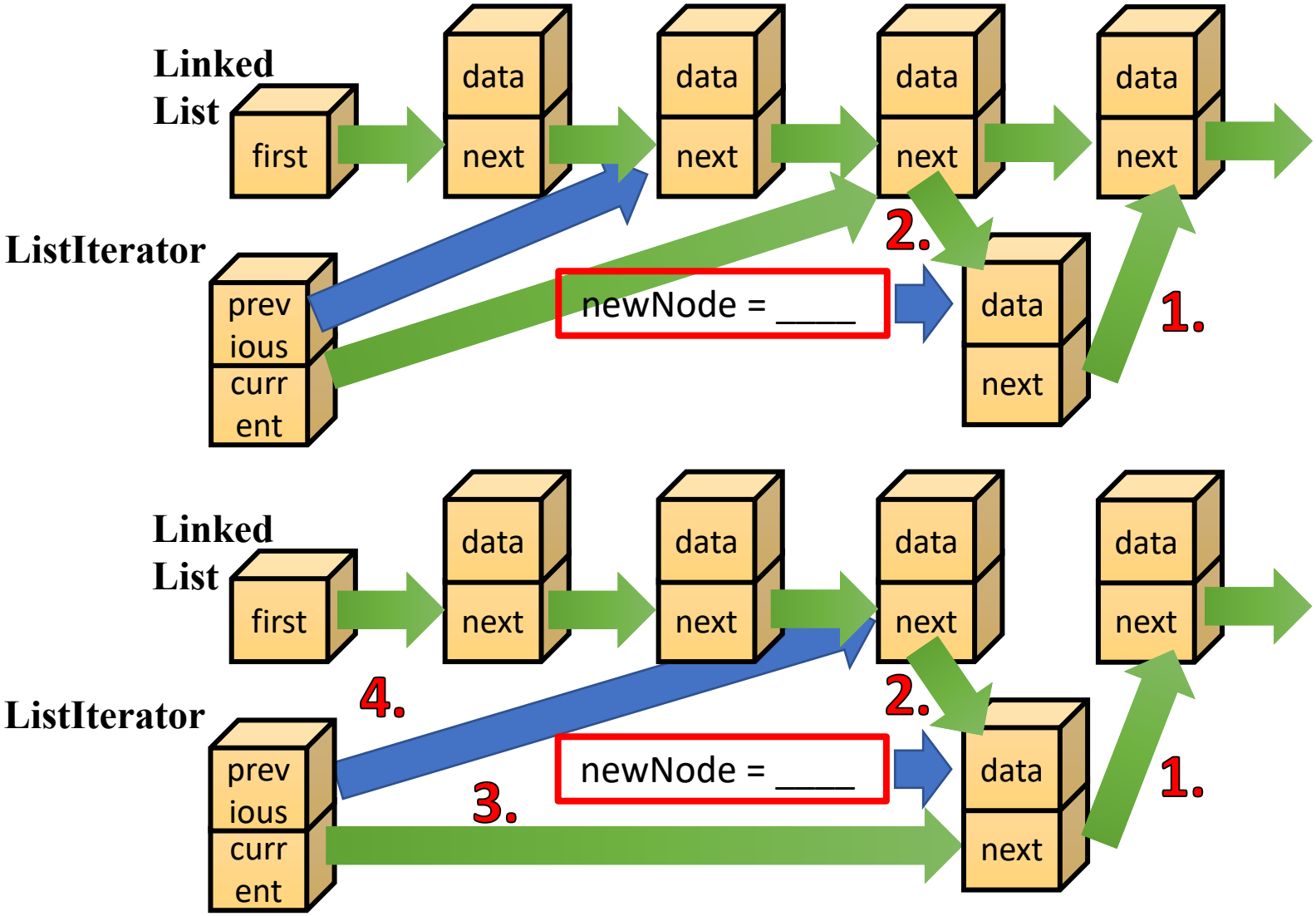
Removing a Link using the Iterator

- Removing a Link from the middle of a Linked List



Adding a Link Using an Iterator

- Adding a Link to the middle of a Linked List after current



Where does the Iterator point?

- When you delete an item, should the iterator point to the next item, previous item, or back to the beginning of the list?
- Keeping it in the vicinity is usually convenient
- However, can't set it to the previous item because there's no way to reset the iterator's previous field to the one before that (unless it's doubly linked)
- Move it to the next item, and back to the start if you've just deleted the last item

The atEnd() Method

- Does this return true when the iterator points to the last valid link or when the iterator points **past** the last link?
- If it points past the last link, then you can't do anything with that link
- You also can't back up an iterator in a singly linked-list (e.g. you couldn't search for the last link and then delete it)
- The better approach is to make sure that the iterator always points to a valid link – should return true when it points to the last valid link (has to always check if next is null)

Code Overview

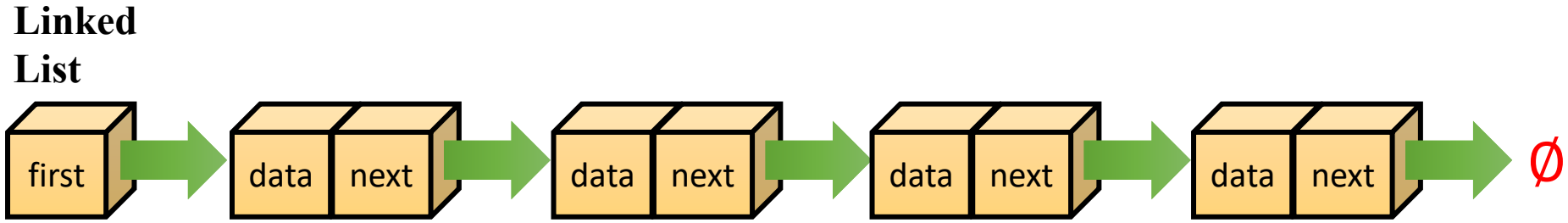
- ListIterator

Content

- Singly Linked List
- Doubly Linked List
- Iterators
- **Stacks using Linked List**

Stacks using Linked Lists

- Implement a stack structure for reversing a word using a single-ended singly-linked list
- We need a single-ended singly-linked list where we can insert and delete at the head

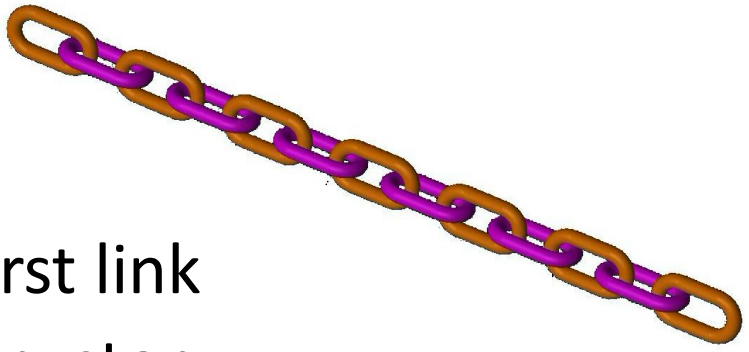


Link

- First, create your link

```
class Link {
    public char data; // data item
    public Link next; // next link in list
    public Link(char data) { // constructor
        this.data = data; // initialize data
    }
}
```

Linked List



- Now, make your list

```
public class LinkedList {
    private Link first; // ref to first link
    public LinkedList(){ // constructor
        first = null; // no links on list yet
    }
    public boolean isEmpty(){ // true if list is empty
        return (first==null);
    }
…
…add in the insertHead() and removeHead() methods…
```

Methods

- We need insertHead( ) and deleteHead( )

```
public void insertHead(char data){ // make new link
    Link newLink = new Link(data);
    newLink.next = first; // newLink --> old first
    first = newLink; // first --> newLink
}
public Link removeHead() { // delete first item
    // (assumes list not empty)
    Link temp = first; // save reference to link
    first = first.next; // delete it: first-->old next
    return temp; // return deleted link
}
```

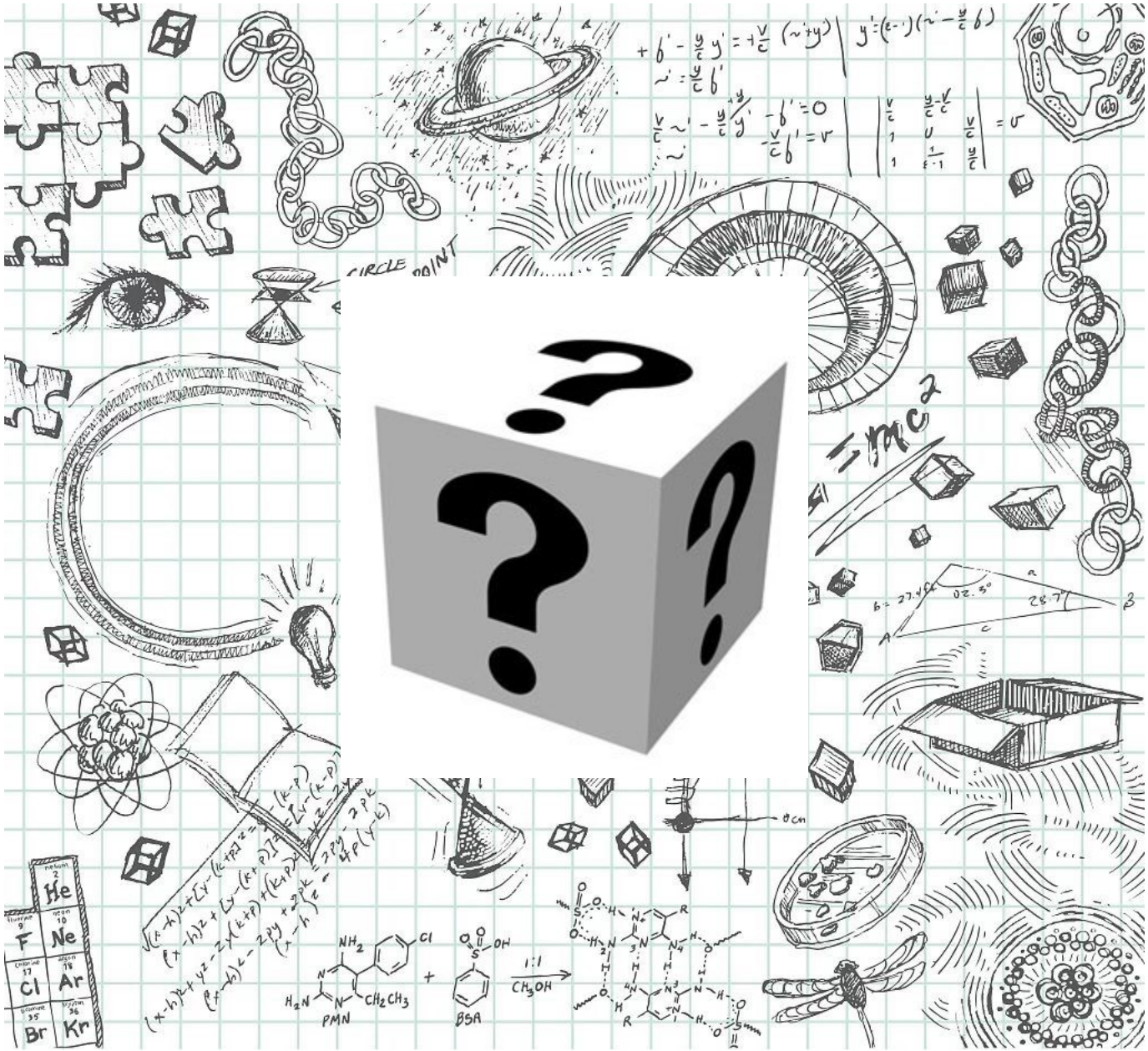
Stack

- Wrap it all up in a stack - abstraction

```
public class Stack {
    private LinkedList list;
    public Stack() { // constructor
        list = new LinkedList();
    }
    public void push(char data) { // put item on top of stack
        list.insertHead(data);
    }
    public char pop() { // take item from top of stack
        return list.removeHead().data;
    }
}
```

Exercise

- Write your Stack class based on LinkedList


?
?
?
E=mc²
PMN
BSA
1:1
CH₃OH