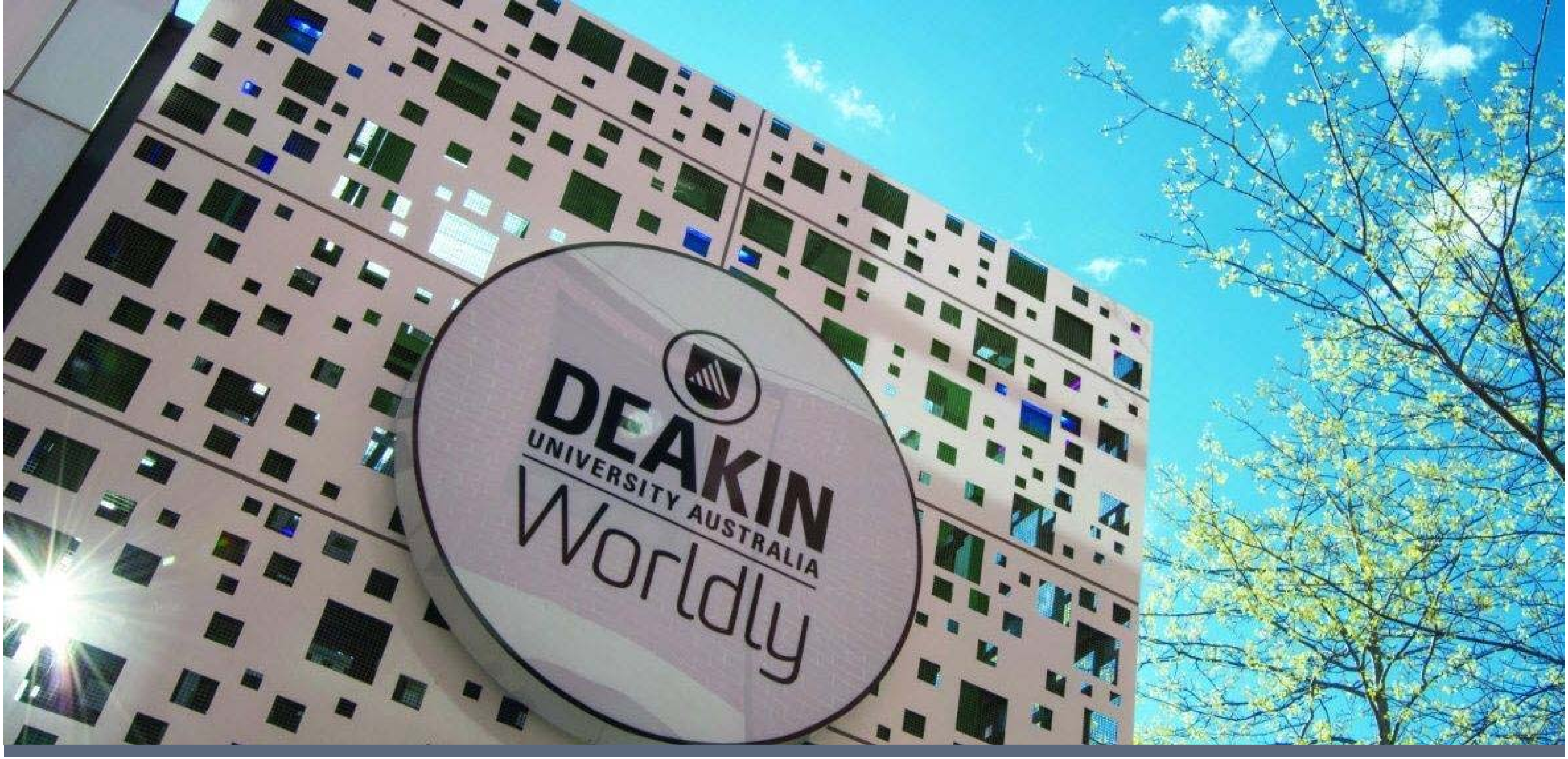


Lecture 4. Single and Double Linked Lists

SIT221 Data Structures and Algorithms

Linked Lists: Appetizer

By now, we are familiar with arrays, dynamic arrays and vectors.

We like arrays because accessing elements by index takes constant time, i.e. $\theta(1)$.

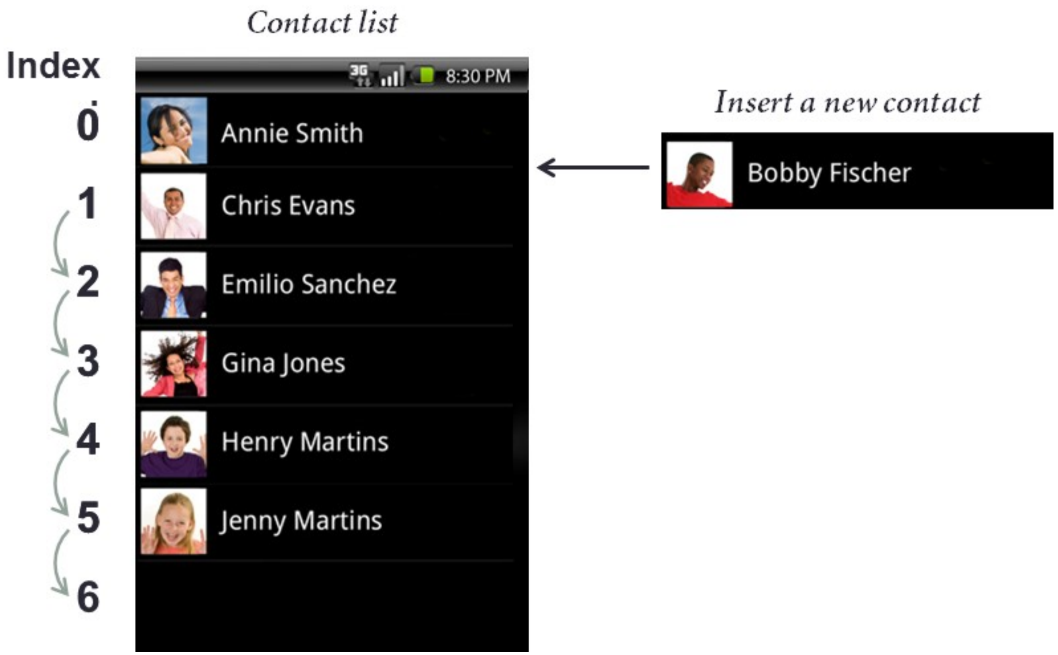
However…

- What if we want to add an element at the end of array?
- What if we want to add an element at the beginning of array?
- What if we have a really large number of items?
- How does it deal with computer memory?

Linked Lists: Appetizer

Problem: Array stores elements contiguously.

$\Rightarrow$ Insertion/deletion of an element takes linear time, i.e. $\theta(n)$.



For example, adding a new contact to a sorted array requires to add an empty element at the end of the array and shift all the items forward in the sequence starting from item 0.

Linked Lists: Appetizer

Major problems with arrays:

- Often need to shuffle elements when inserting or deleting an element.
- Cannot change size, can only reallocate and copy all elements from old to a new array.
- Or you pre-allocate and waste extra space.
- Many applications require resizing! Required size not always immediately available.

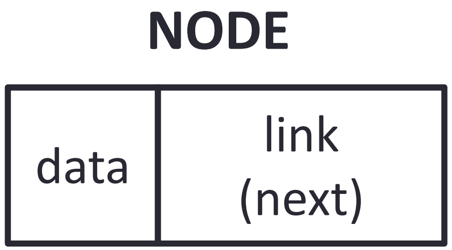
Linked lists solve these problems

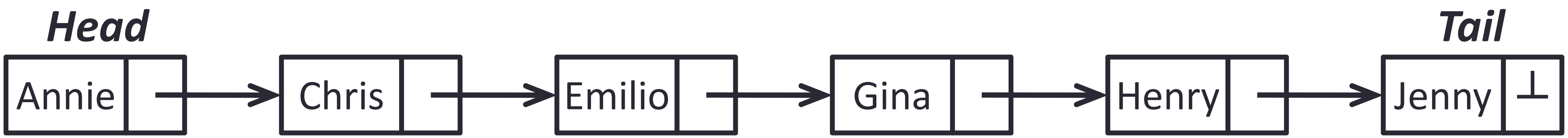
A linked list is a data structure that is composed of nodes that have:

- values
- a pointer to another node

Singly Linked List: The Idea

- Store elements discretely in separate objects called nodes.
- Let each of them know the next element (via references).



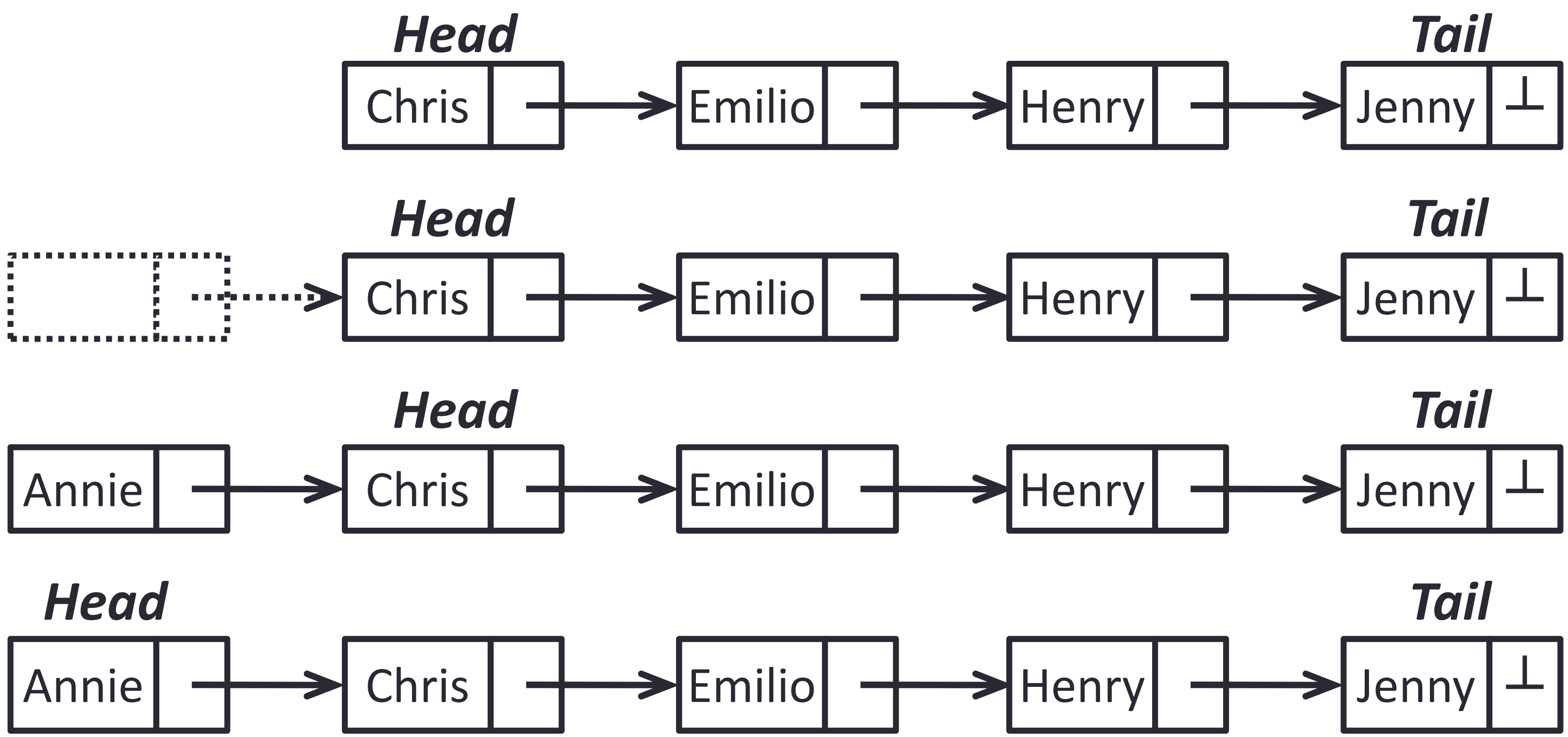


- Keeping track of our linked list requires that we know where it starts. Then we can follow the trail of links to traverse the whole list.
- So we are going to use the Head (First) node to keep track of it.
- We may also record the Tail to introduce some extra operations.

Singly Linked List: Operations

Inserting at the Head: $\theta(1)$ runtime complexity

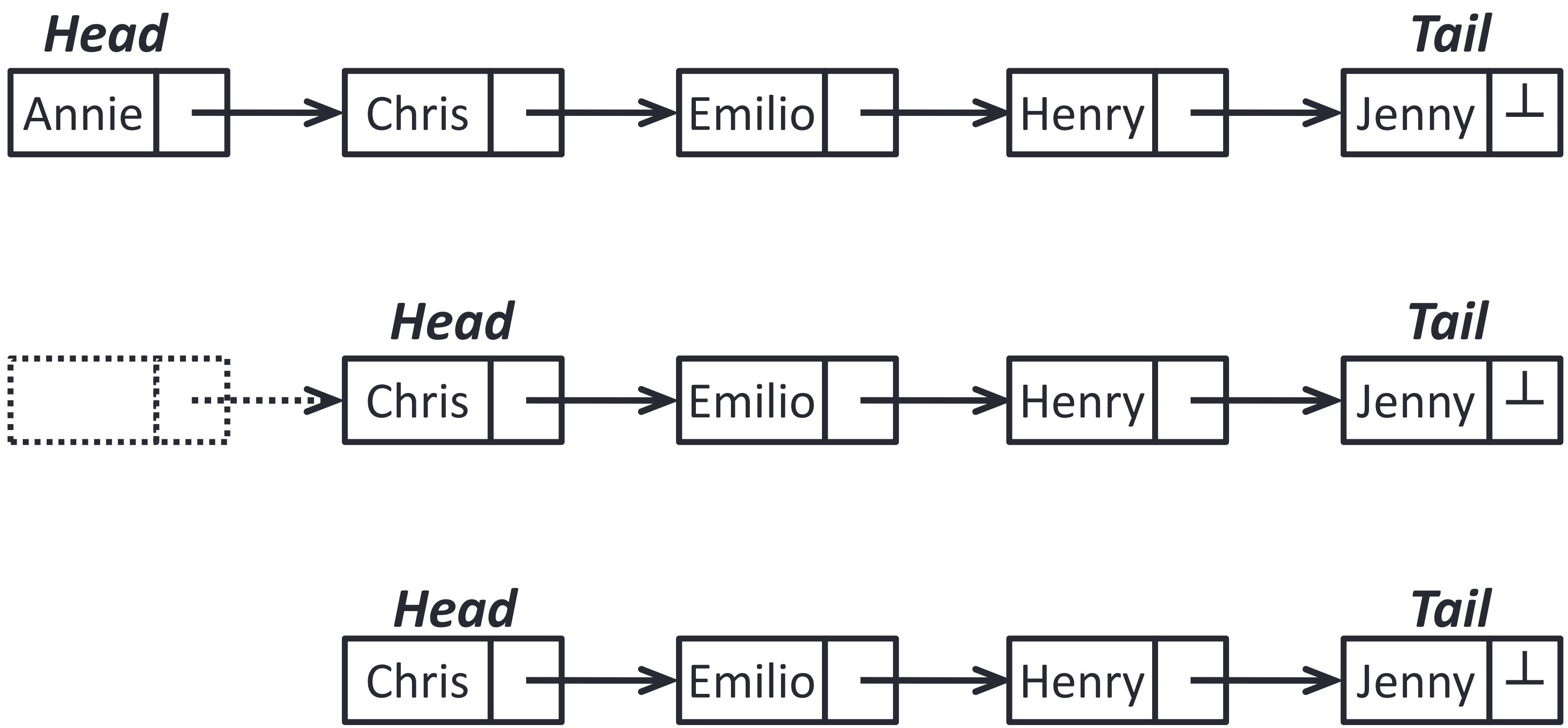
1. Allocate a new node
2. Insert a new value
3. Make the new node pointing to the old Head
4. Update Head to point to the new node



Singly Linked List: Operations

Removing at the Head: $\theta(1)$ runtime complexity

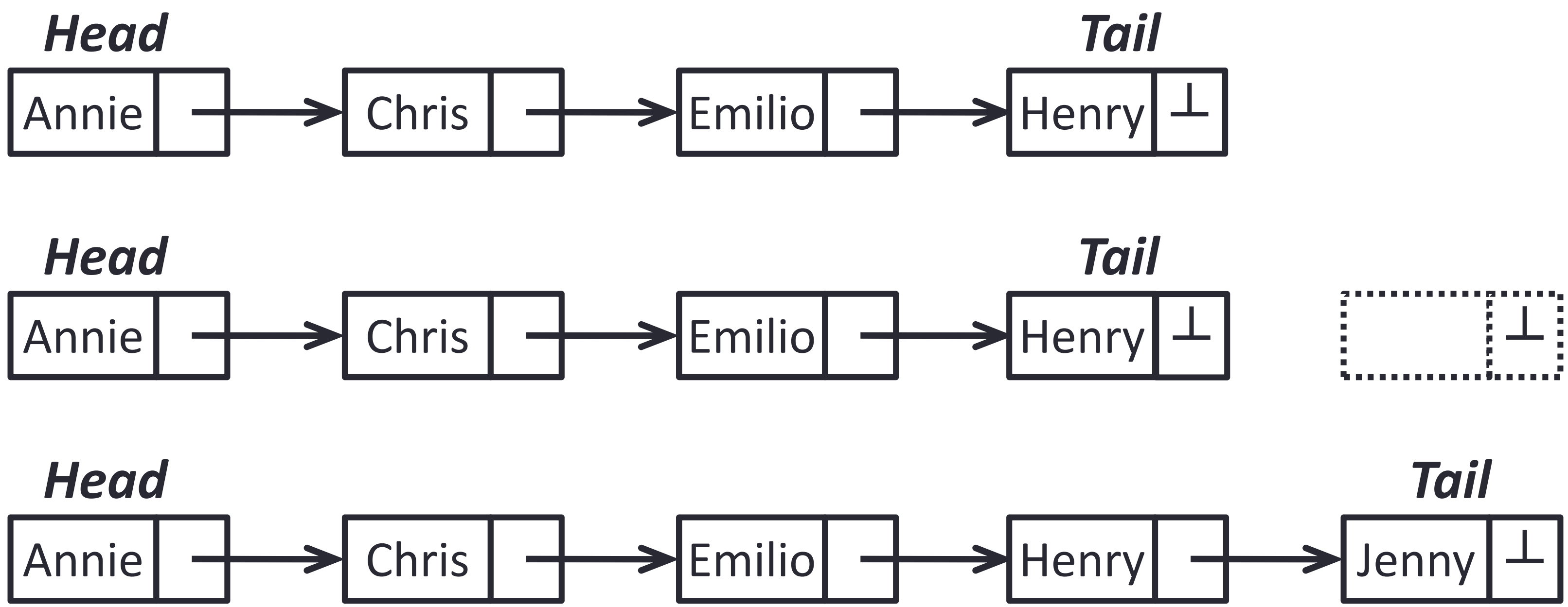
1. Update Head to refer to the next node in the list
2. Allow garbage collector to reclaim the former first node



Singly Linked List: Operations

Inserting at the Tail: $\theta(1)$ runtime complexity

1. Allocate a new node
2. Enter a new value
3. Make the new node pointing to null
4. Make the old last node pointing to the new node
5. Update Tail to point to the new node



Singly Linked List: Operations

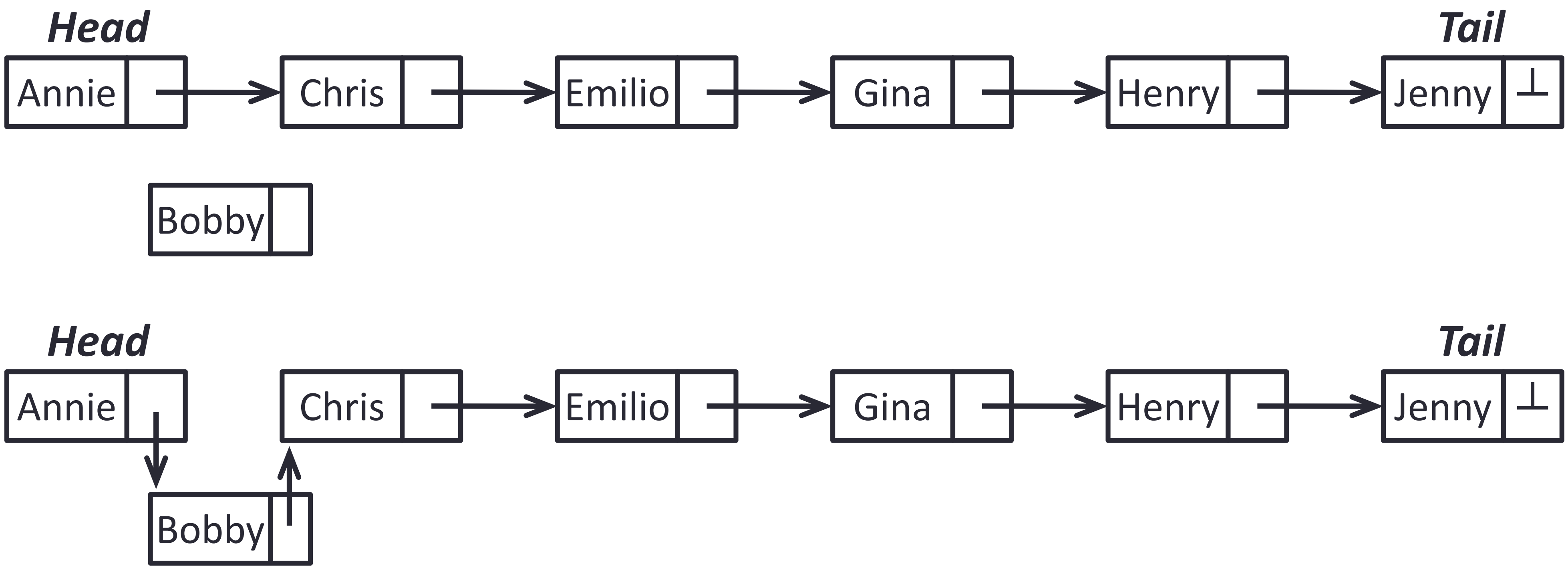
Removing at the Tail: $\theta(n)$ runtime complexity

1. Removing at the tail of a singly linked list cannot be efficient!
2. There is no constant-time way to update Tail to point to the previous node.

Singly Linked List: Operations

Inserting after a node*: $\theta(1)$ runtime complexity

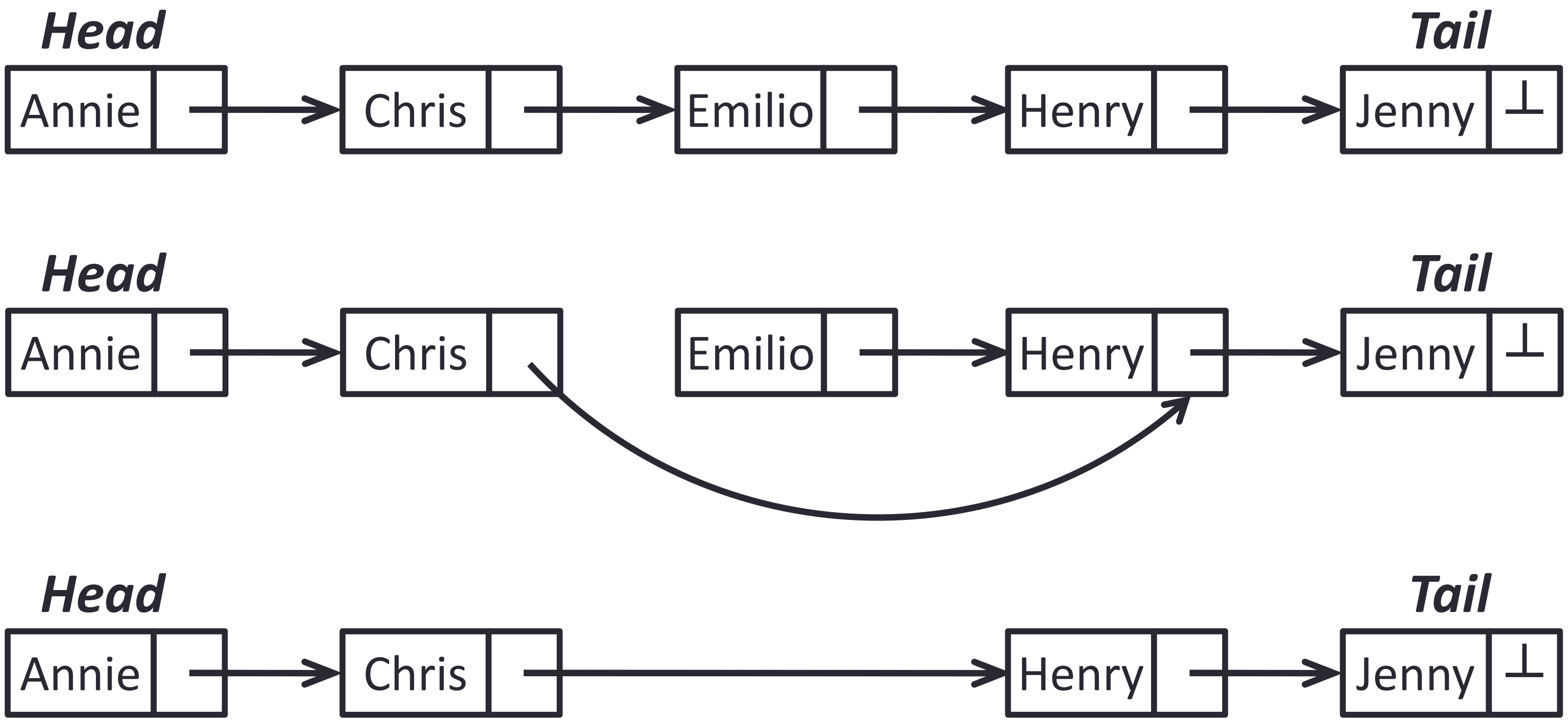
1. Allocate a new node
2. Enter a new value
3. Make the new node pointing to the Next node of node*
4. Make the node* pointing to the new node



Singly Linked List: Operations

Removing after a node*: $\theta(1)$ runtime complexity

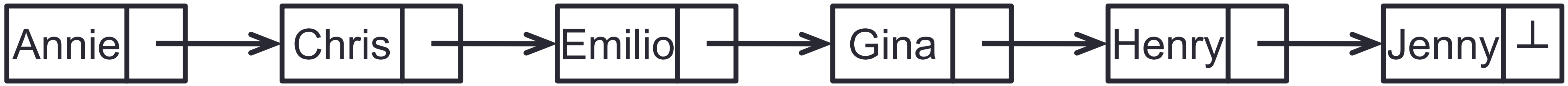
1. Update node* to point to the Next of the Next node in the list
2. Allow garbage collector to reclaim the Next node of node*



Singly Linked List: Operations

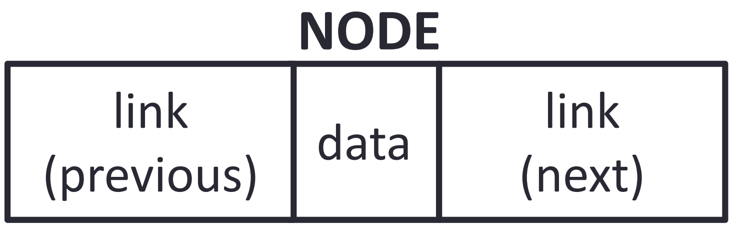
Removing a <u>node</u>: $\theta(n)$ runtime complexity

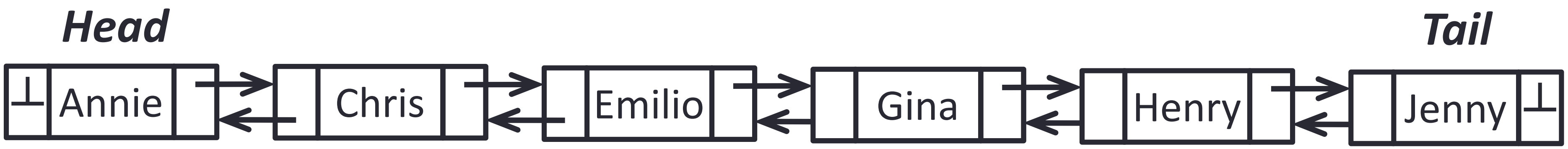
We cannot delete “Gina” if the reference to “Emilio” is not given.



Doubly Linked List

Let each of elements know the next and the previous element.

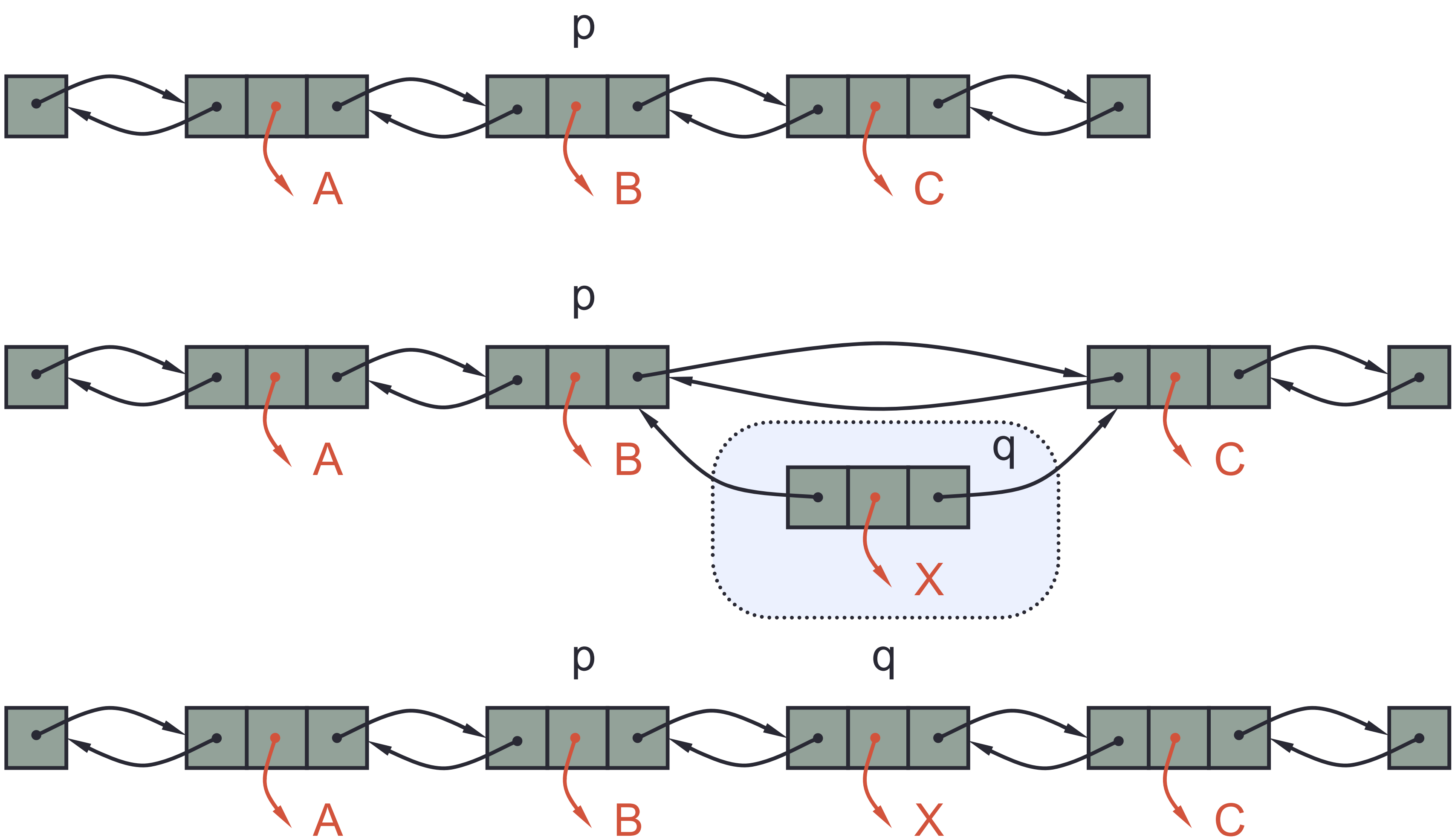




Now we can insert or delete a node in $\theta(1)$ given only that node's memory address (reference).

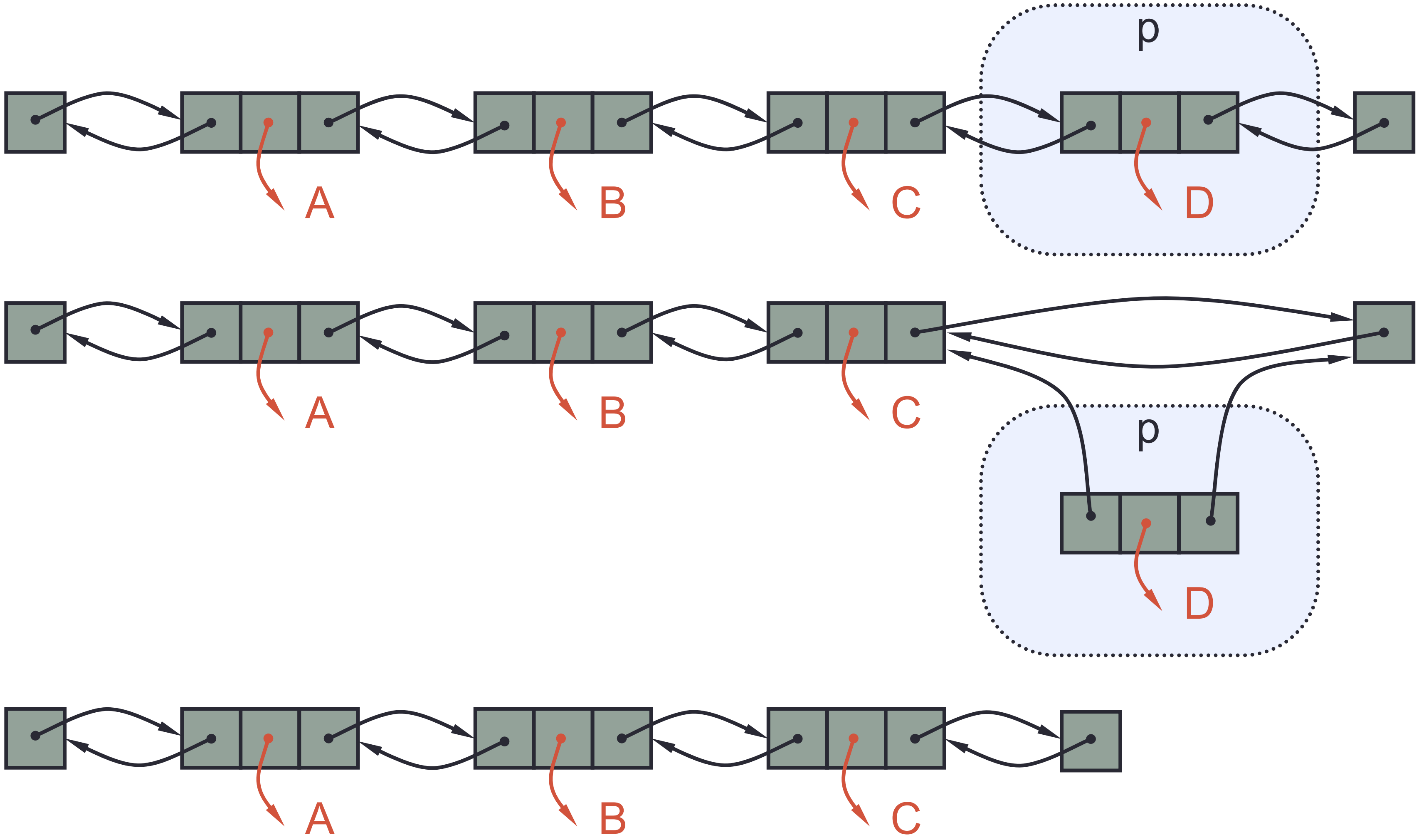
Doubly Linked List: Operations

Inserting after a node: $\theta(1)$ runtime complexity



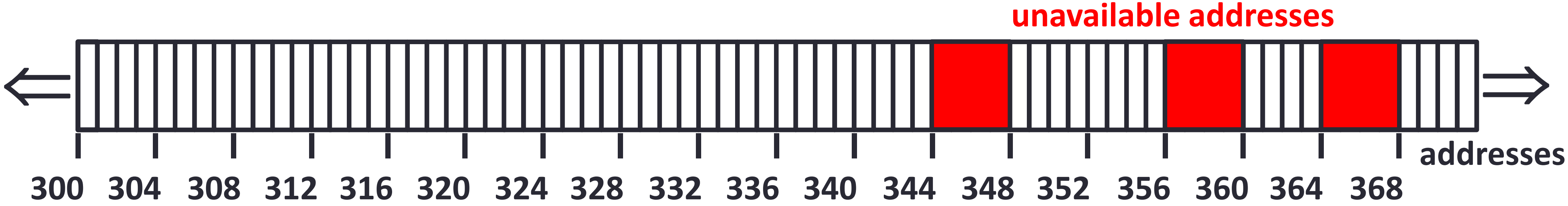
Doubly Linked List: Operations

Removing a <u>node</u>: $\theta(1)$ runtime complexity



Linked Lists vs. Array: Memory Issues

Memory. Array representation



Linked Lists vs. Array: Memory Issues

Memory. Array representation

int x = 5; *: needs 4 bytes*

Linked Lists vs. Array: Memory Issues

int x = 5; *: needs 4 bytes*

int y[] = new int[3] {2,7,4}; *: needs 12 bytes*

add element {1} to y $\rightarrow$ extend the size of y by two, thus *20 bytes* in total

Linked Lists vs. Array: Memory Issues

int x = 5; *: needs 4 bytes*

int y[] = new int[3] {2,7,4}; *: needs 12 bytes*

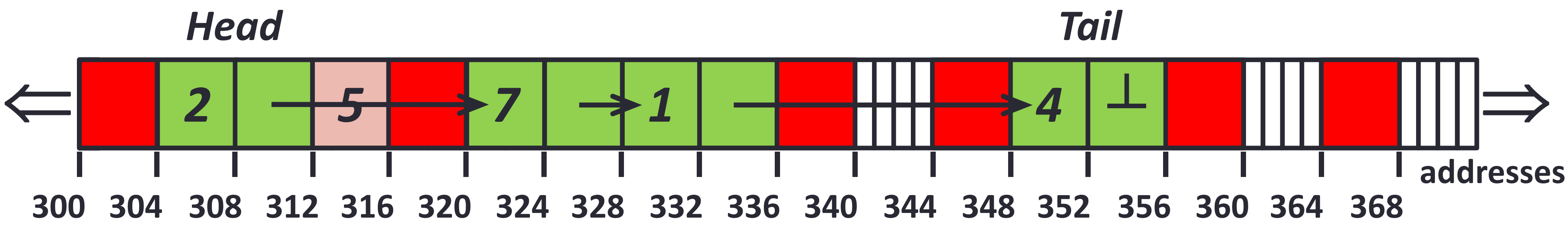
add element {1} to y $\rightarrow$ extend the size of y by two, thus *20 bytes* in total

copy y to the new place

Problems: copying, fragmentation

Linked Lists vs. Array: Memory Issues

Memory. Singly linked list representation



- No copying required
- Less affected by fragmentation

Linked Lists vs. Array: Complexity Summary

Operation	Singly Linked List	Doubly Linked List	Dynamic Array
Access by index	$\theta(n)$	$\theta(n)$	$\theta(1)$
Insert	$\theta(1)^{*}$	$\theta(1)$	$\theta(n)$
Remove	$\theta(1)^{*}$	$\theta(1)$	$\theta(n)$
First	$\theta(1)$	$\theta(1)$	$\theta(n)$
Last	$\theta(n)^{**}$	$\theta(1)$	$\theta(1)$
Concatenation	$\theta(n)^{**}$	$\theta(1)$	$\theta(n)$
Count	$\theta(1)^{***}$	$\theta(1)^{***}$	$\theta(1)$

* only as Insert After and Remove After
** if the Last element is not tracked
*** only if an additional counter for the number of elements is used

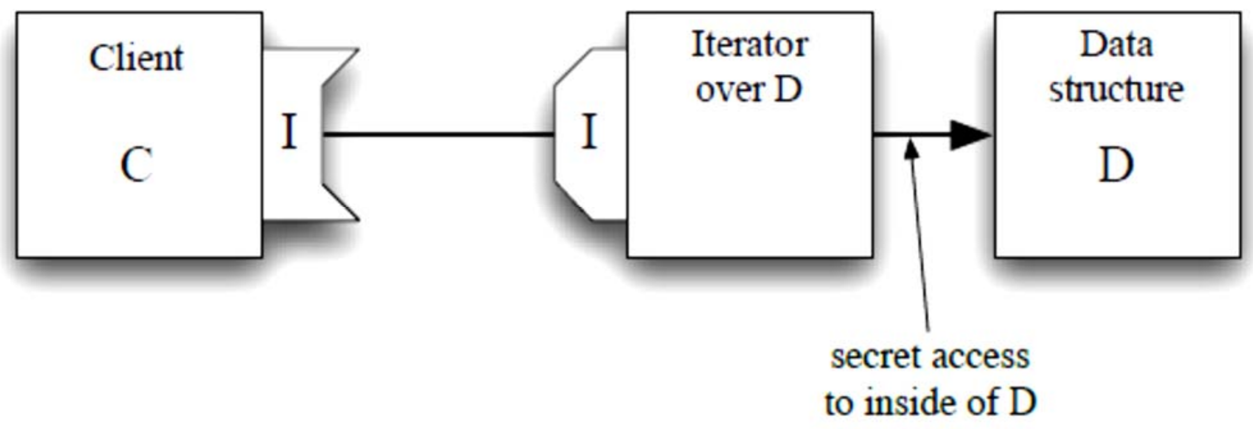
Auxiliary Memory requirements:

- Array stores only elements, i.e. $\theta(1)$
- Singly linked list stores the successor of each element, i.e. $\theta(n)$
- Doubly linked list stores the predecessor and successor of each element, i.e. $\theta(2n)$

Enumeration interface to traverse data structures

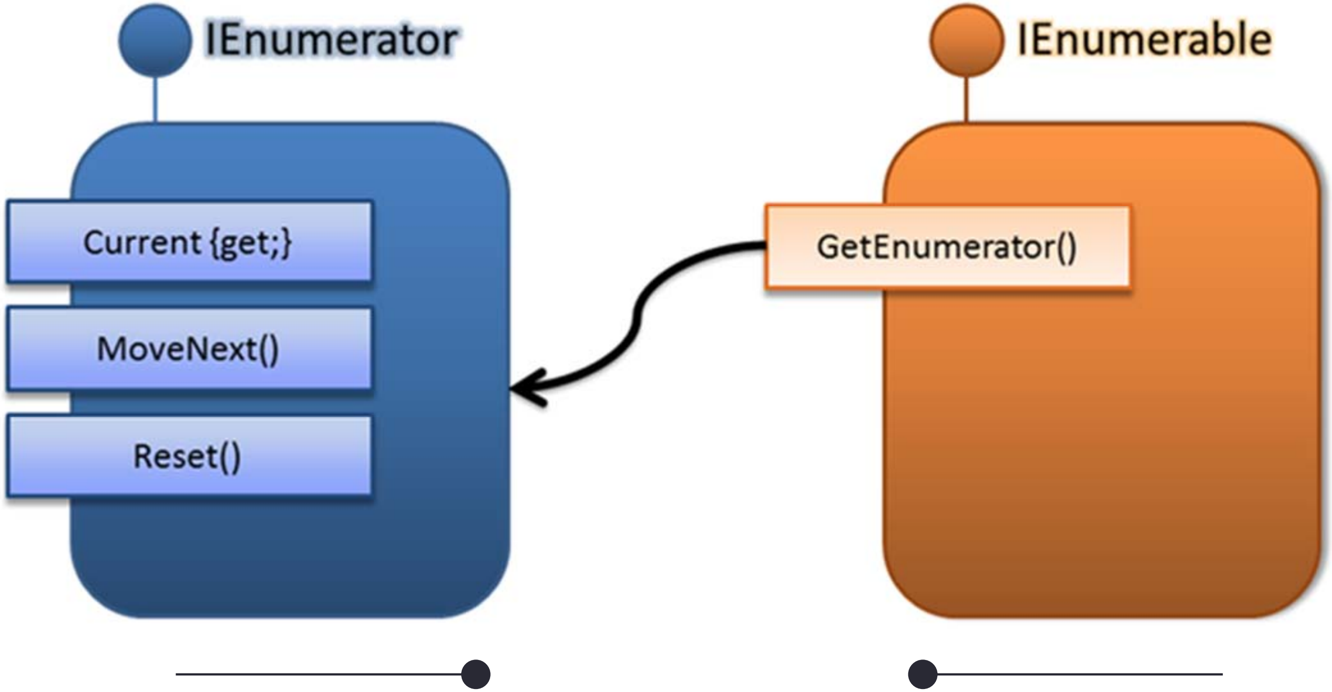
Enumeration of elements in a generic data structure is possible through implementation of so-called **Iterator** object-oriented programming design pattern.

Iterator provides a simple way for a program to access all of the components of a data structure **without knowing the representation of that data structure**.



Enumeration interface to traverse data structures

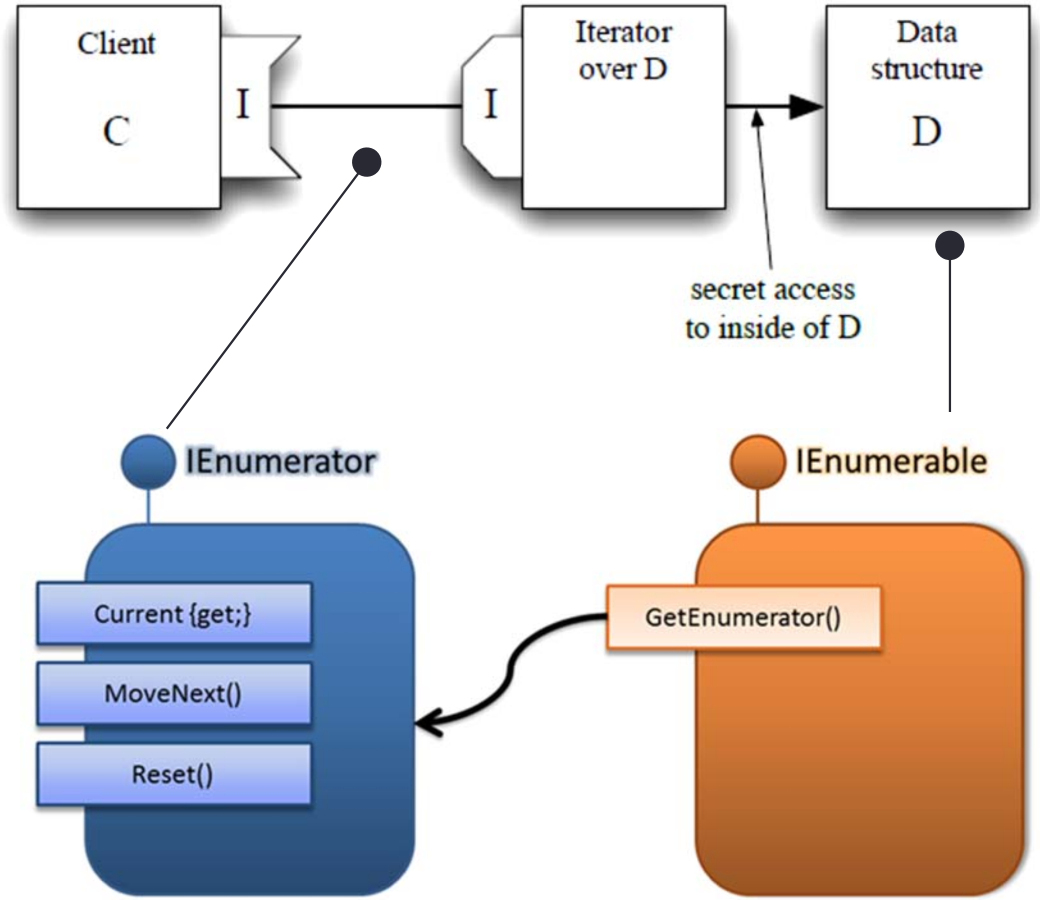
IEnumerator and IEnumerable are two interfaces to support enumeration of elements in a generic data structure in C#.



Implemented by a special Iterator class providing interface-specified access to the elements of the data structure

Implemented by a data structure

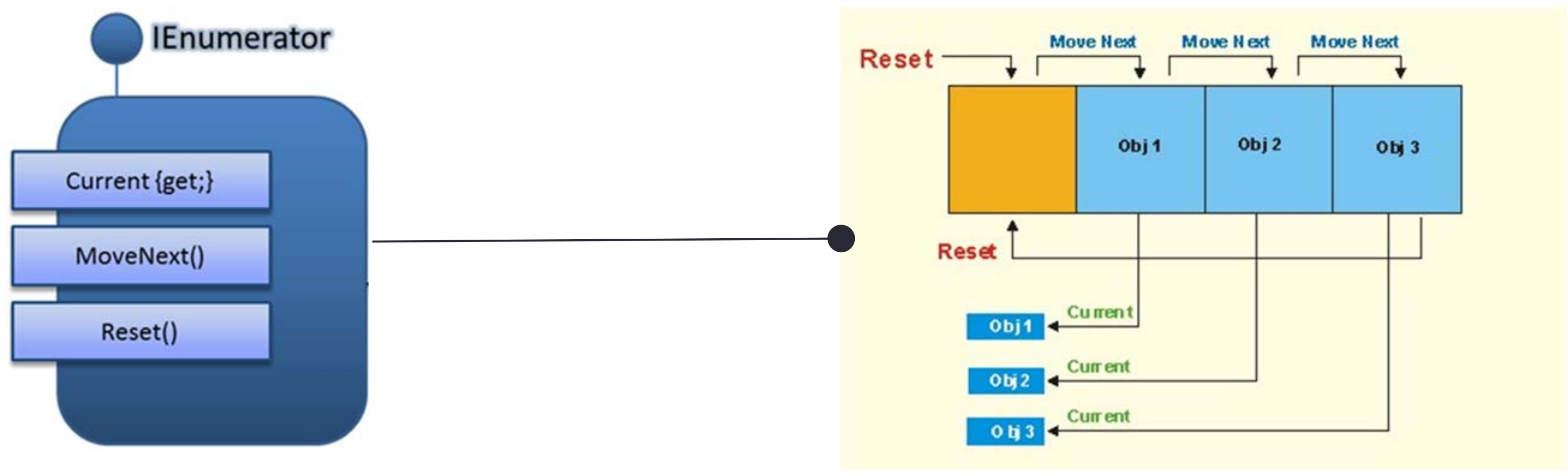
Enumeration interface to traverse data structures



– The data structure creates and returns an instance of Iterator via GetEnumerator() method inherited from the IEnumerable interface.

– This happens every time when the client needs to traverse the elements of the data structure.

Enumeration interface to traverse data structures



Current	Property. Gets the element in the collection at the current position of the enumerator.
bool MoveNext()	Advances the enumerator to the next element of the collection.
void Reset()	Sets the enumerator to its initial position, which is before the first element in the collection.
void Dispose()	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.

Summary

- Divide and conquer is an important concept in algorithmics.
- Master theorem is a general tool for solving standard recursive formulas.
- Invariants are an important tool to show correctness of algorithms/programs.
- Binary Search is effective to locate elements in a sorted array.
  - Algorithm maintains two invariants.
  - It halves the problem size in each iteration.
  - This implies $O(\log n)$ comparisons.

Other references and things to do

- Have a look at the attached references in CloudDeakin.
- Read chapters 3.2 and 3.4 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.