1501118 Data Structures and Algorithms

Chapter 3: Linked Lists 2

+

Chapter 4: Algorithm Analysis

2nd semester AY2022

School of Information Technology

Outline

- Quiz
- Circular Linked Lists
- Doubly Linked Lists (Self Study)
- Algorithm Analysis
- Summary

Chapter 3: Linked Lists

QUIZ

1501118
Data Structures and Algorithms
Chapter 3: Linked Lists 2

2nd semester AY2022

School of Information Technology

Circular Linked Lists (CLL)

- Round-Robin Scheduling
- Process management by Operating System

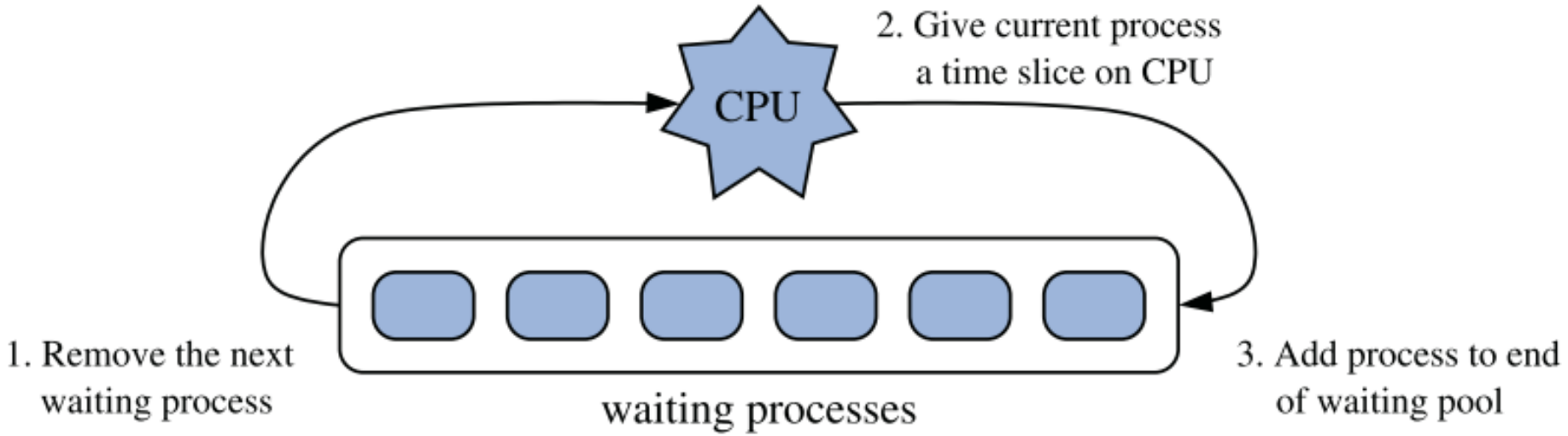
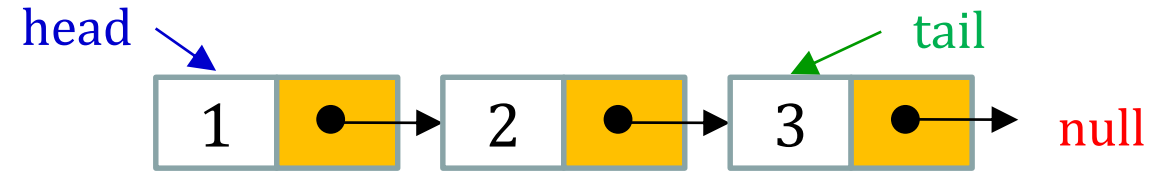


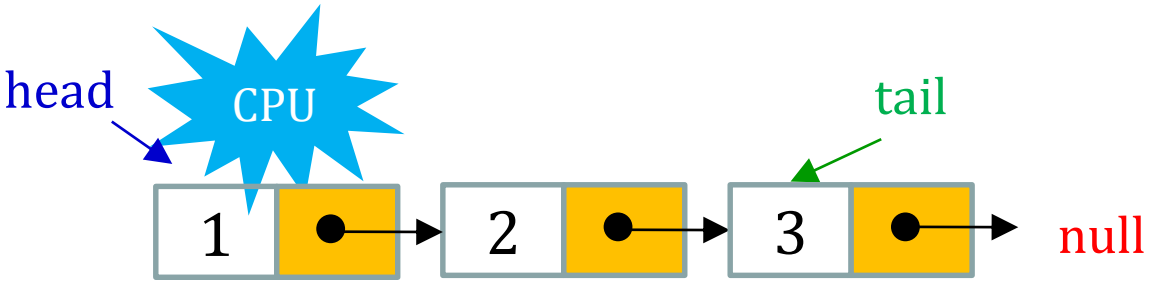
Image source: M. T. Goodrich et al., Data Structures and Algorithms in Java

Round-Robin Scheduling

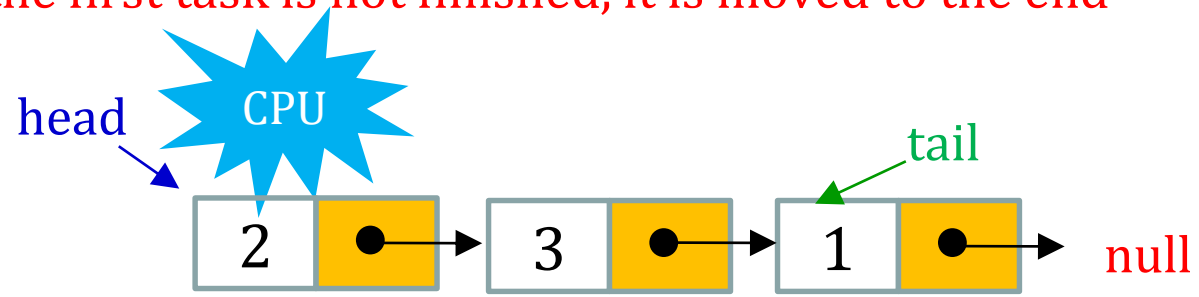
- OS has to manage multiple tasks (processes)
- The processes are queuing
- OS gets the first task and assigns it to CPU to process in a limited time (time slice)
- If that process is not finished, OS moves it to the end of queue and manage the second task
- This continues until all processes are done

Round-Robin Scheduling and SLL





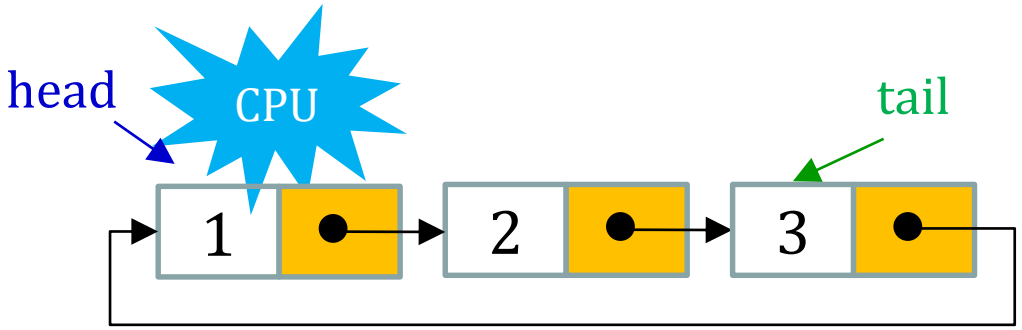
Assume that the first task is not finished, it is moved to the end



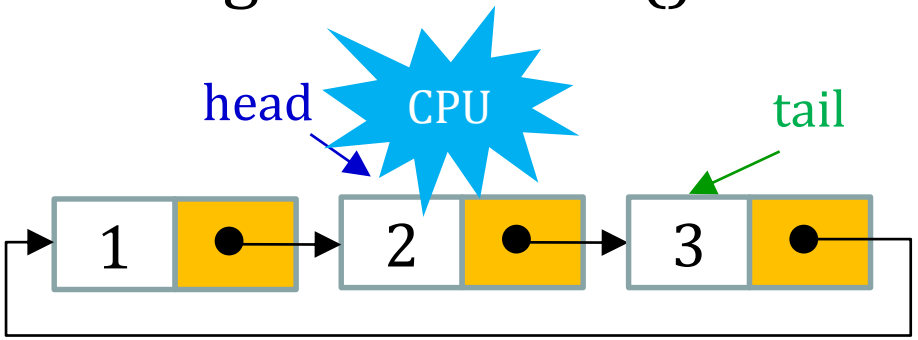
It is not efficient to change head and tail pointers every time including to insert a new node at the tail.

Round-Robin Scheduling and SLL

- If we use SLL, how many steps do we need to move the first node to the last node?
  1. tail.link(head)



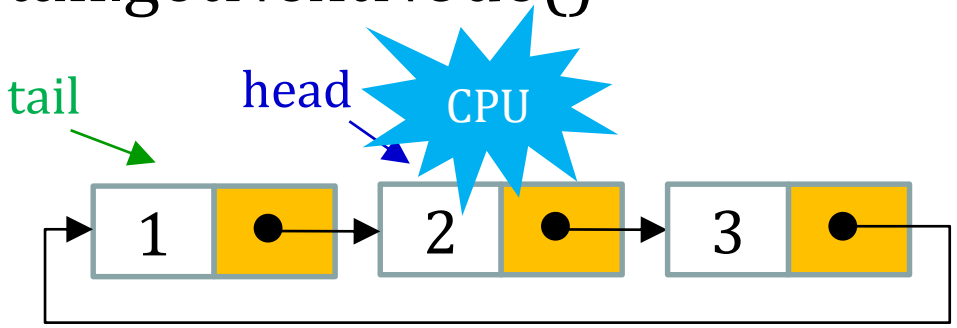
  2. head = head.getNextNode()



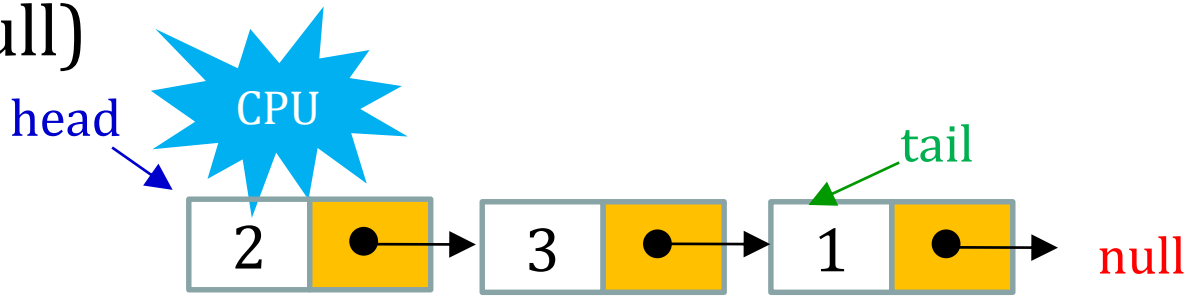
Round-Robin Scheduling and SLL

- If we use SLL, how many steps do we need to move the first node to the last node?

3. tail=tail.getNextNode()

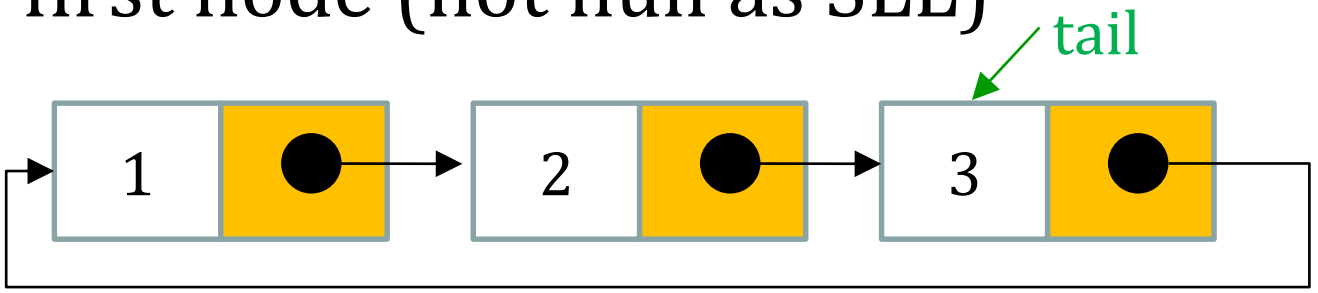


4. tail.link(null)



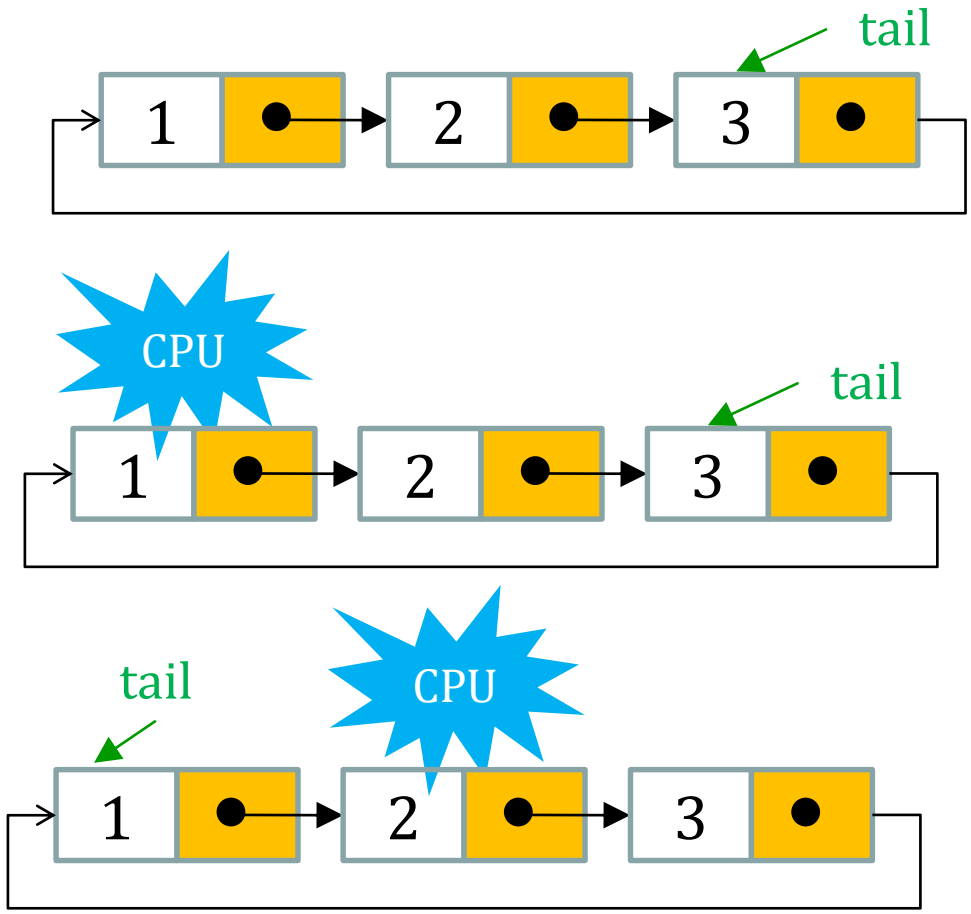
CLL

- CLL is a LL without a head pointer (no header node) but the last link points back to the first node (not null as SLL)



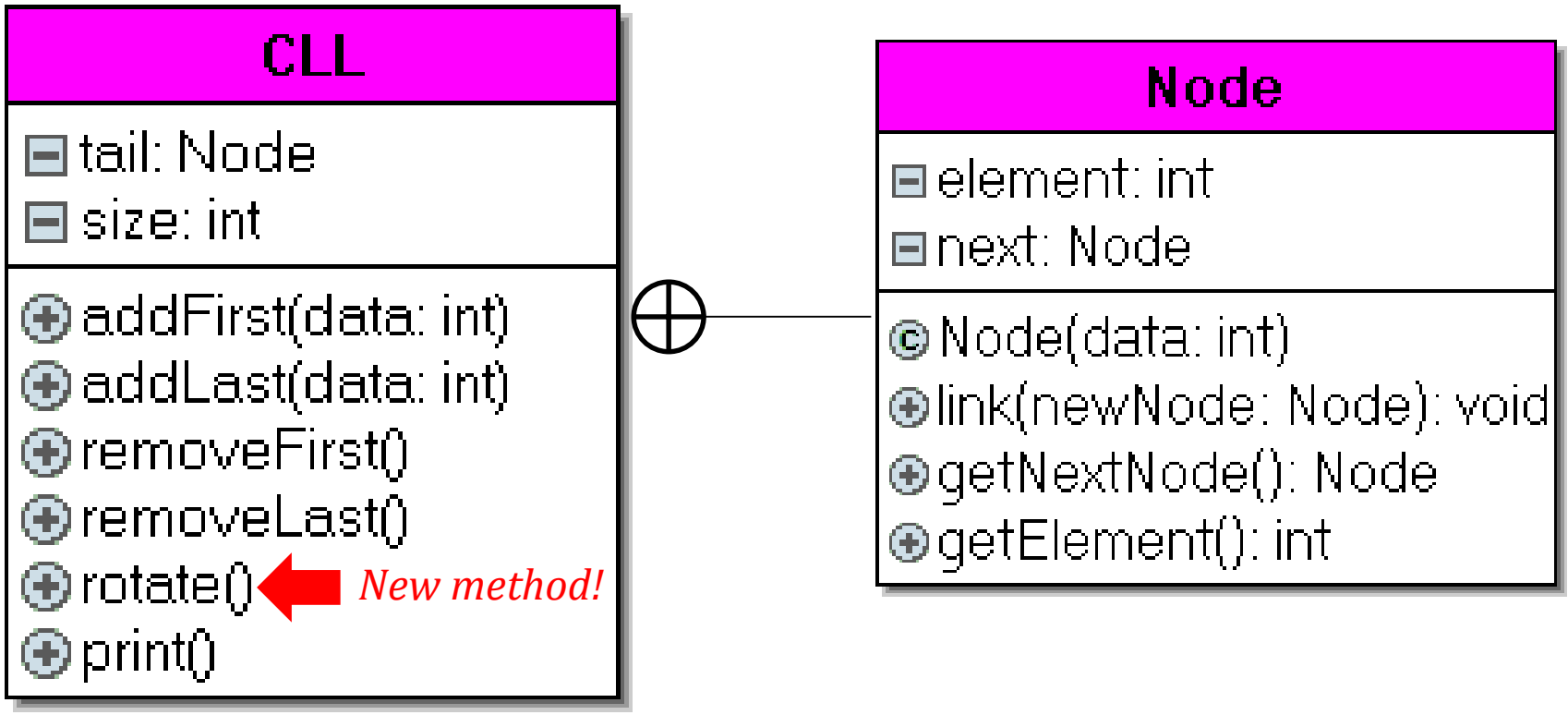
- The advantage over SLL is that we can **rotate** the first node to be the last node easily by changing only the tail pointer

Round-Robin Scheduling and CLL



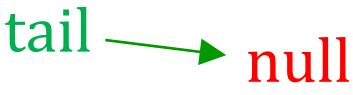
After time slice of process 1 is depleted but the process is not finished, we rotate the list so that this process becomes the end node without extracting and inserting the node.

CLL Class Diagram

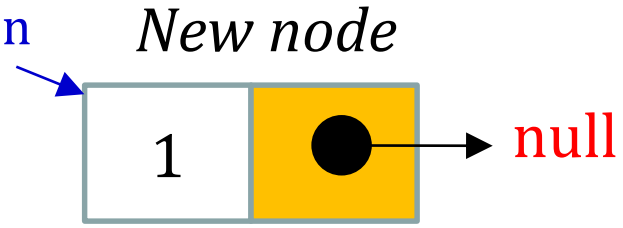


CLL: Add New Node

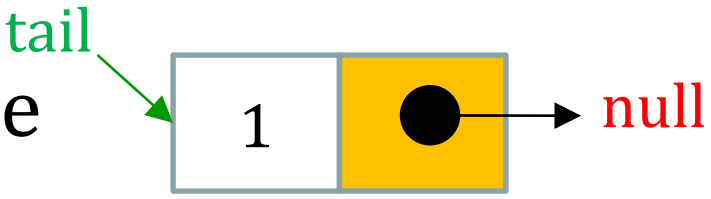
- Assume there is no node
- tail points at null



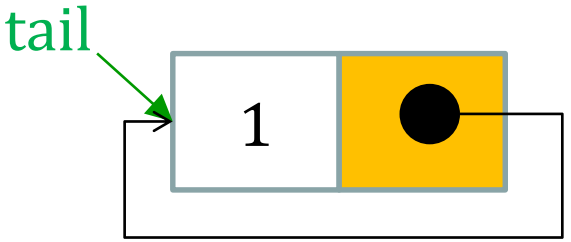
- Create a new node



- Set tail to point at new node

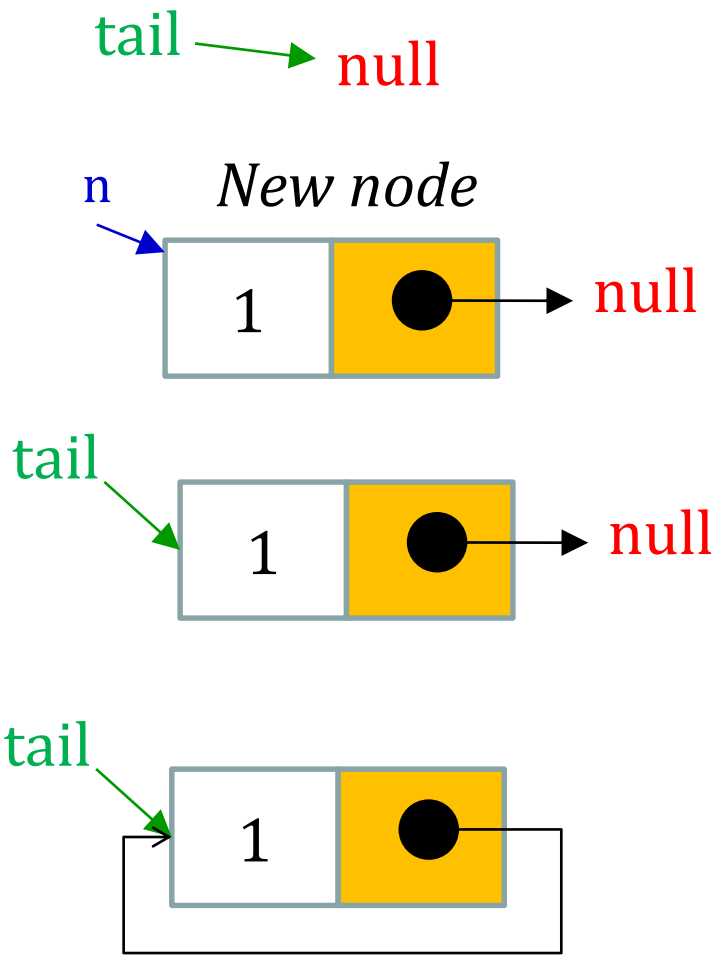


- Set link of tail to itself



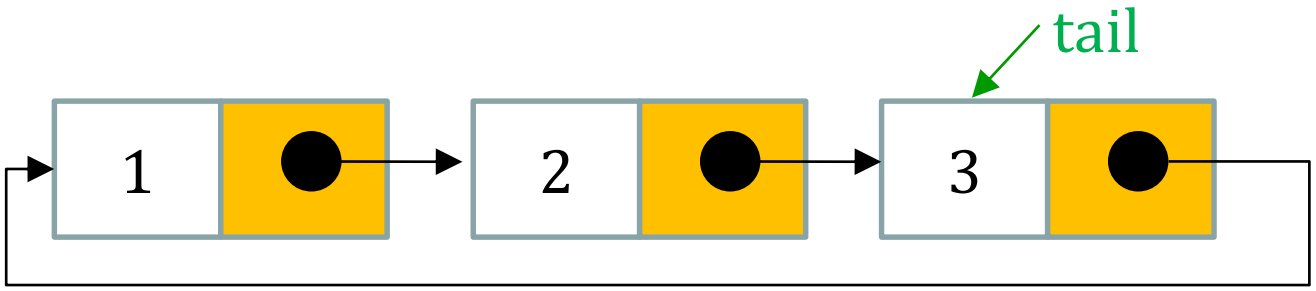
CLL: Add New Node

- `Node tail = null;`
- `Node n = new Node(1);`
- `tail = n;`
- `tail.link(n);`

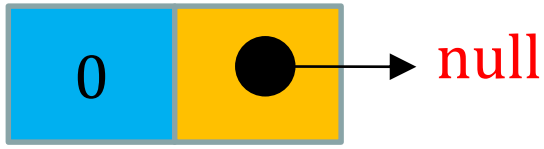


CLL: Add First Node

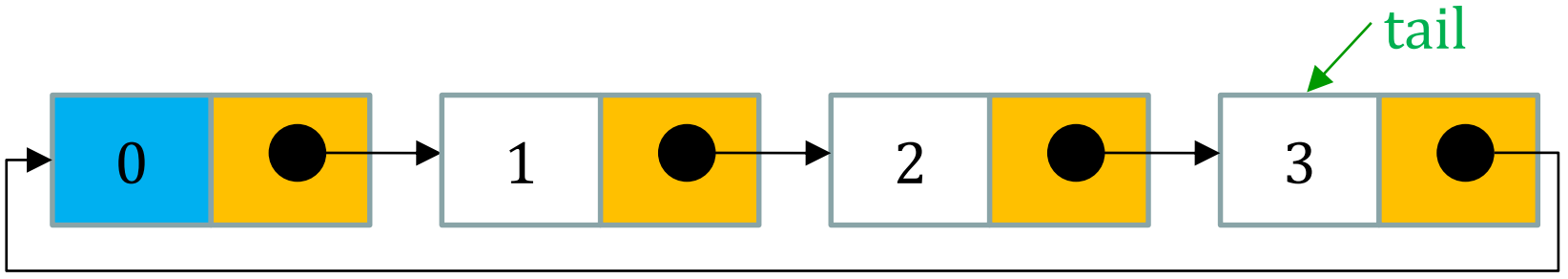
- Assume there are some nodes



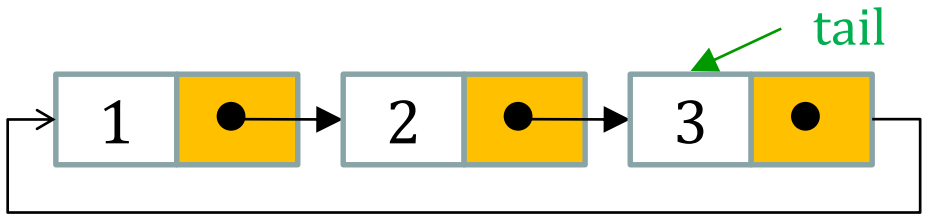
- New node



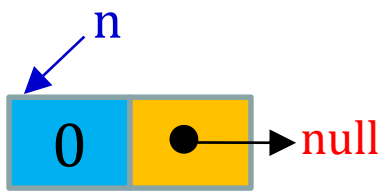
- Result



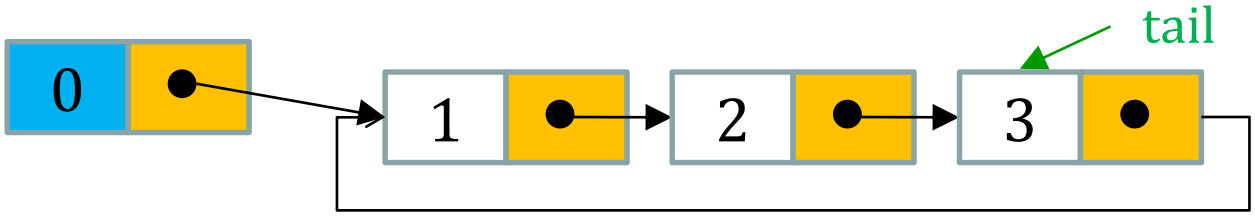
CLL: Add First Node (steps)



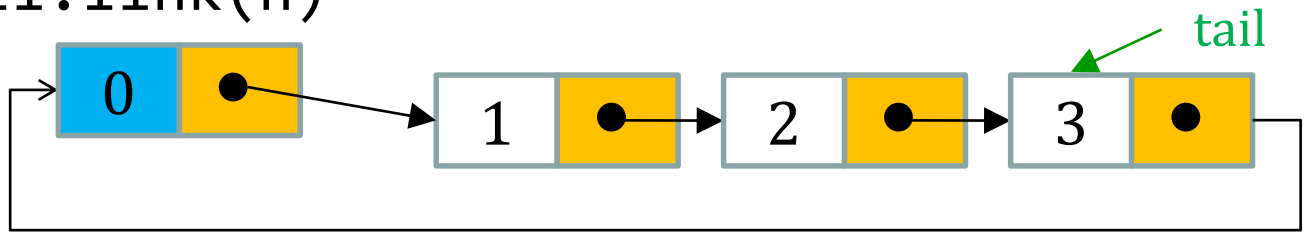
- `Node n = new Node(0);`



- `n.link(tail.getNextNode());`

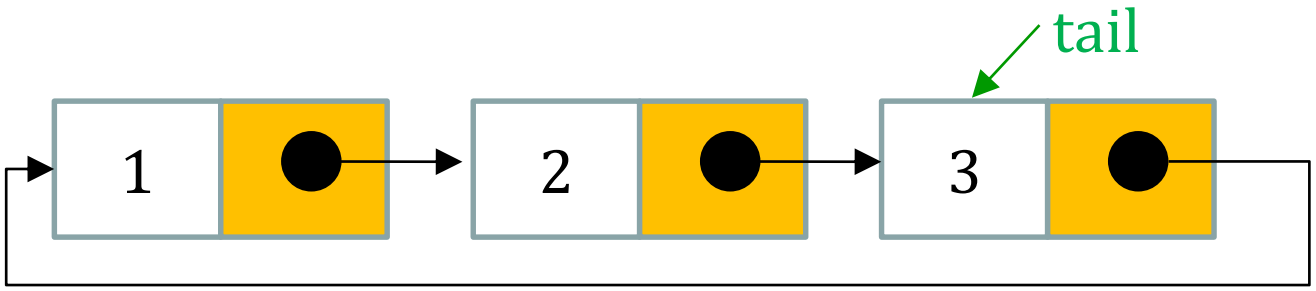


- `tail.link(n)`

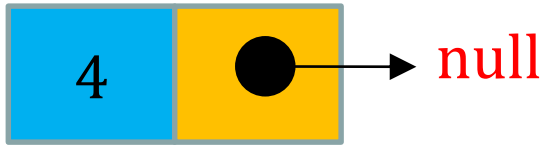


CLL: Add Last Node

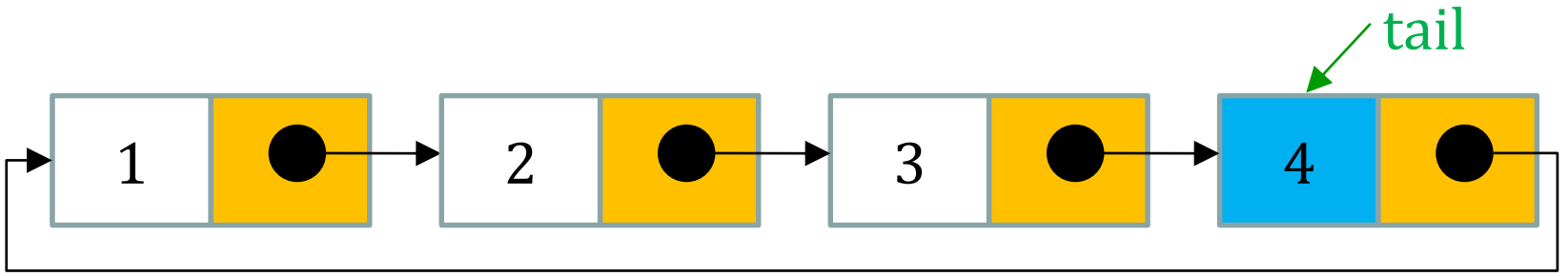
- Assume there are some nodes



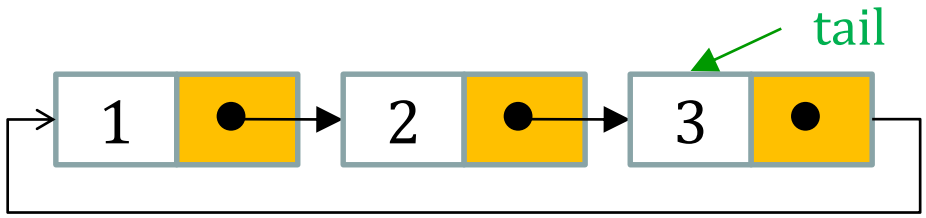
- New node



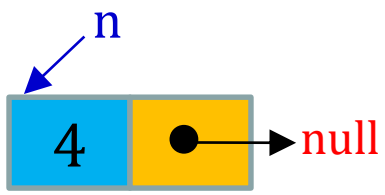
- Result



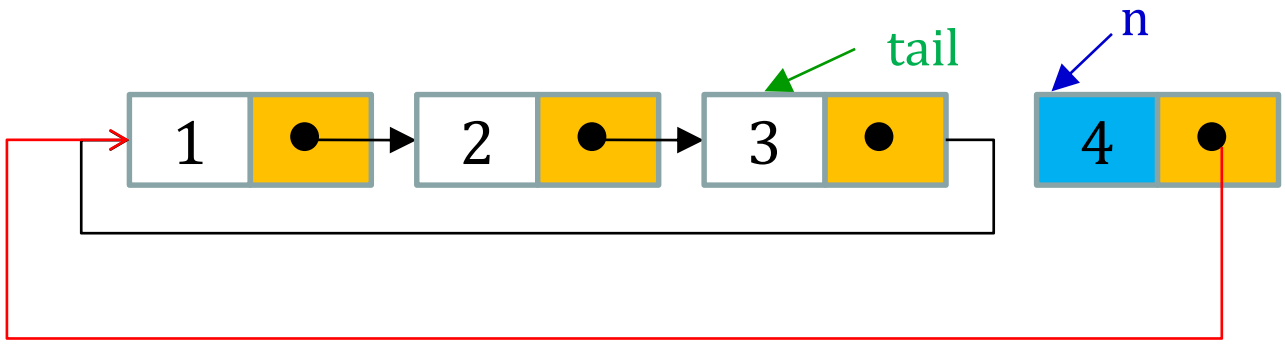
CLL: Add Last Node (steps)



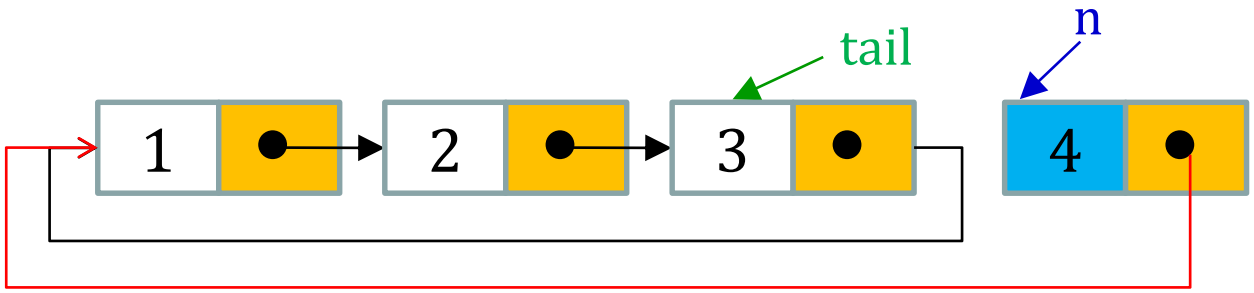
- `Node n = new Node(4);`



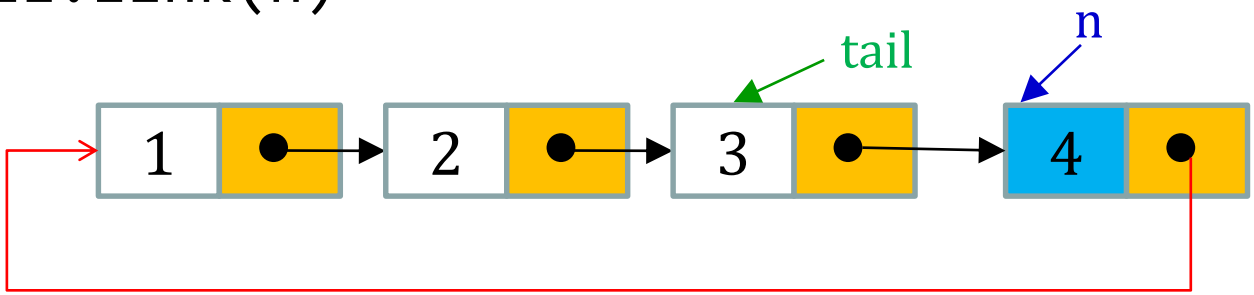
- `n.link(tail.getNextNode())`



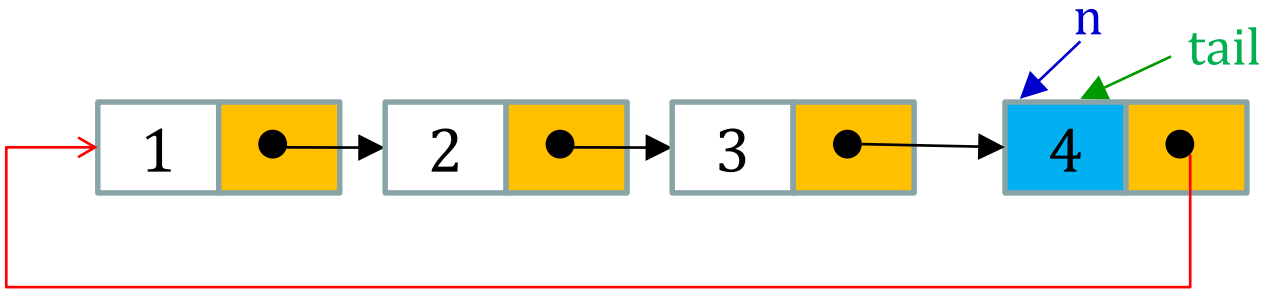
CLL: Add Last Node (steps)



- `tail.link(n)`



- `tail = n OR tail = tail.getNextNode();`

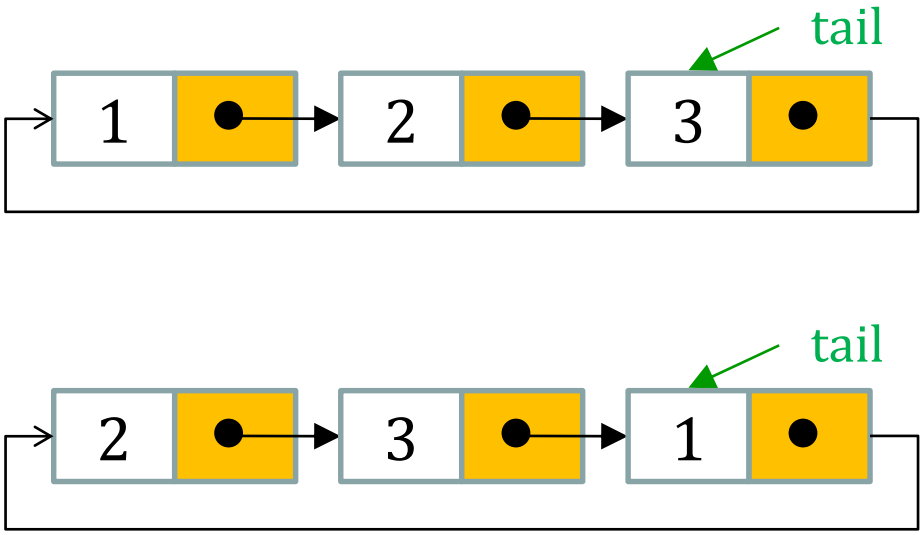


CLL: Methods

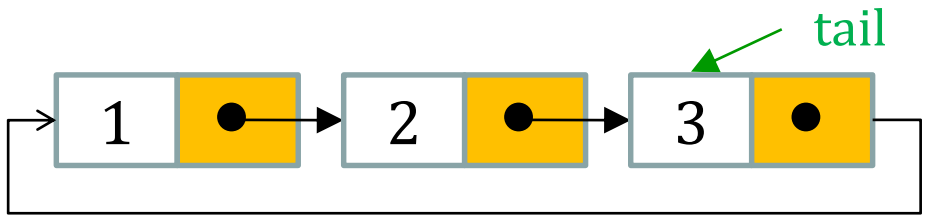
- addFirst();
- addLast();
- removeFirst();
- removeLast();
- print();
- rotate();

CLL: Rotate

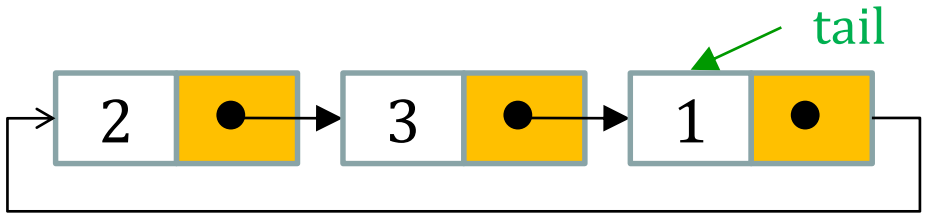
- Move the first node to the end
- How to do that?



CLL: Rotate (Solution)




```
if(size>1) {
        tail = tail.getNextNode()
}
```

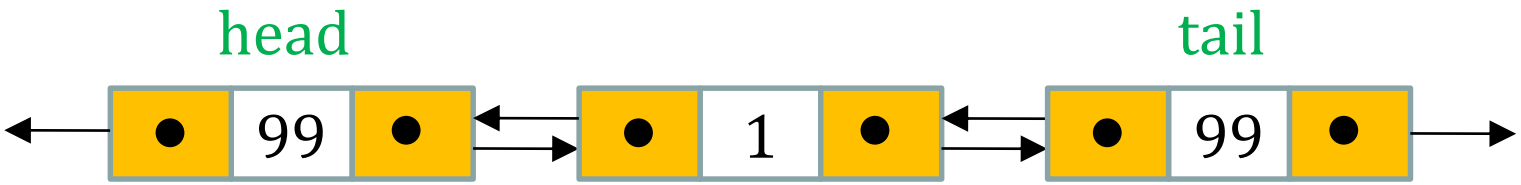


Outline

- Introduction
- Circular Linked Lists
- Doubly Linked Lists (Self Study)
- Algorithm Analysis
- Summary

Doubly Liked Lists (DLL)

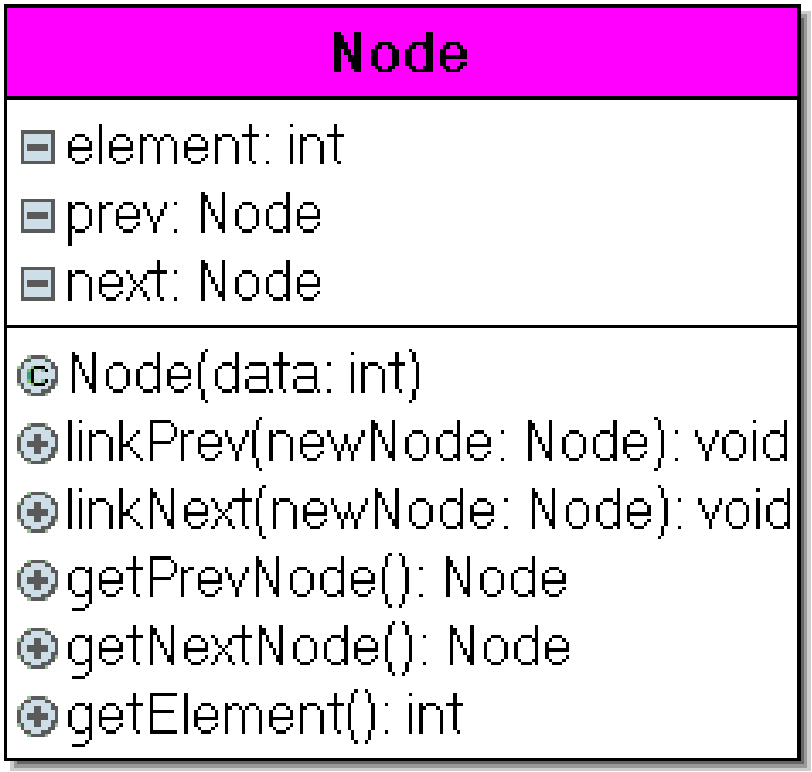
- DLL is a linked list that each node has a next link and a previous link
- Two dummy nodes (sentinels) are used to guard as head and tail of DLL



- DLL is superior in terms of speed thanks to the two-way links. However it requires more storage space

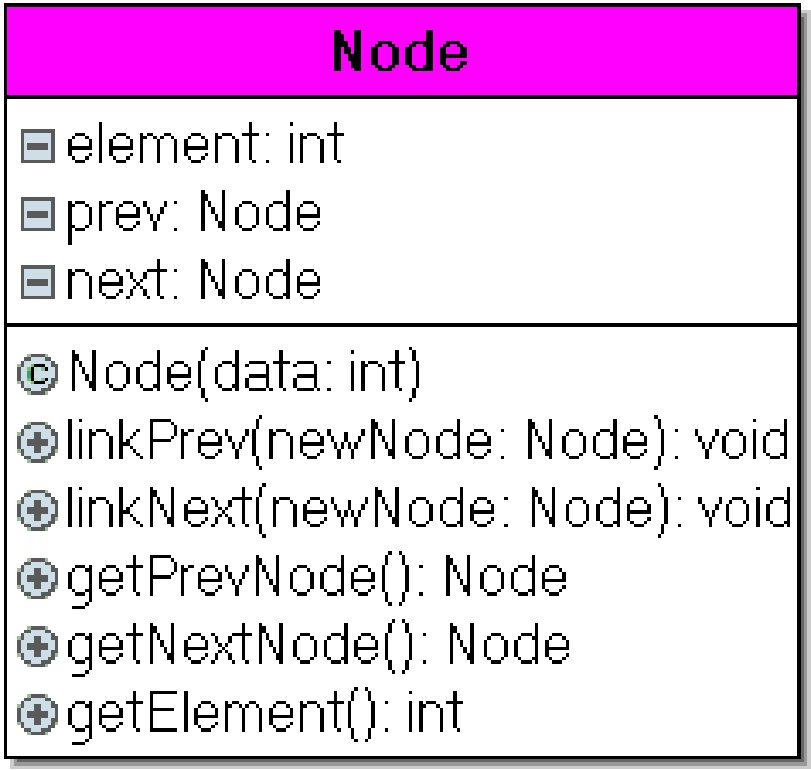
Node of DLL

- Must contain an element and two pointers: previous and next pointers



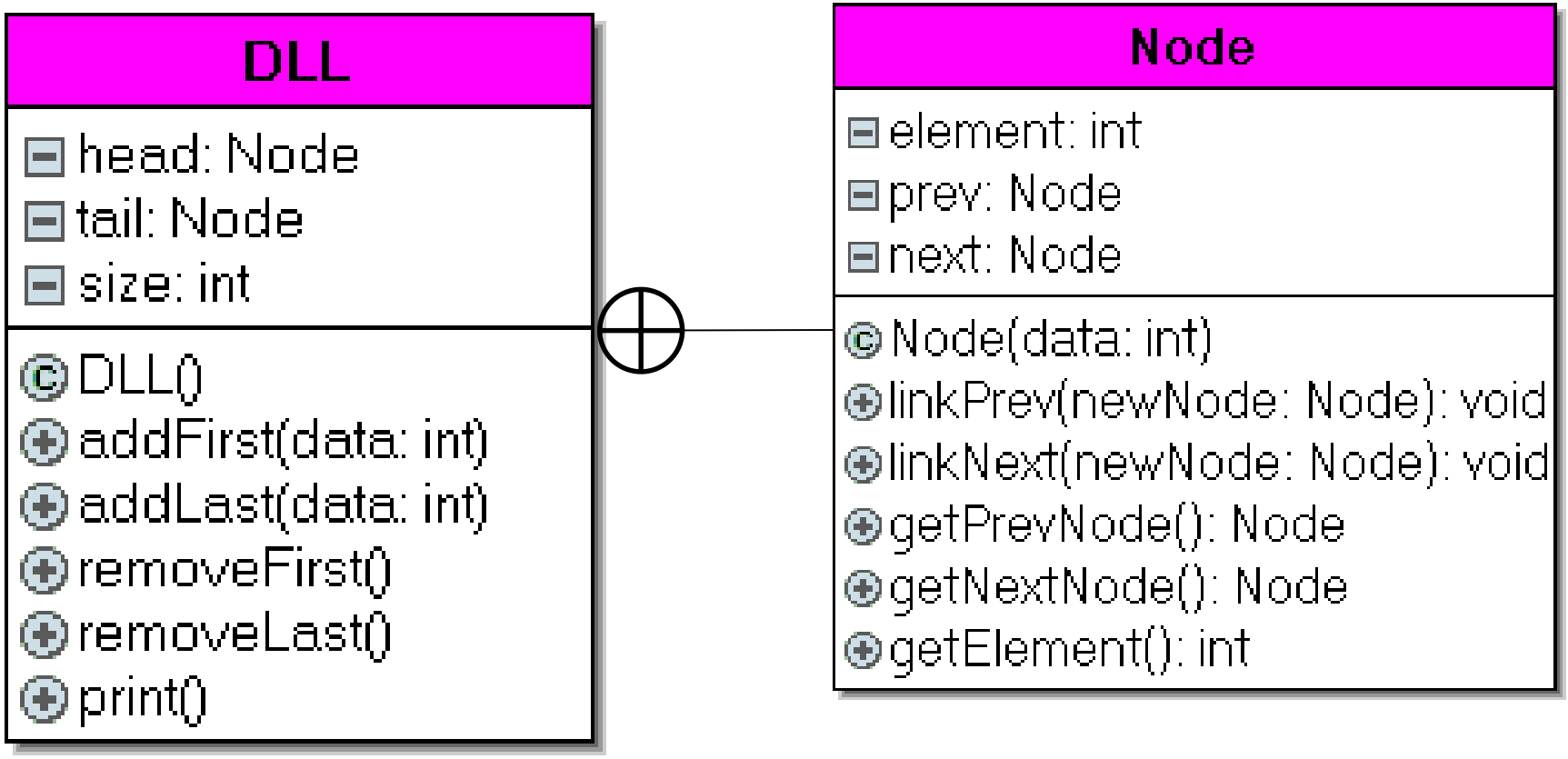
Node of DLL: methods

- Node()
  - Constructor to assign data to element
- linkPrev() and linkNext()
  - to link current node to new node
- getPrevNode() and getNextNode()
  - To return previous and next nodes
- getElement()
  - To return element of current node



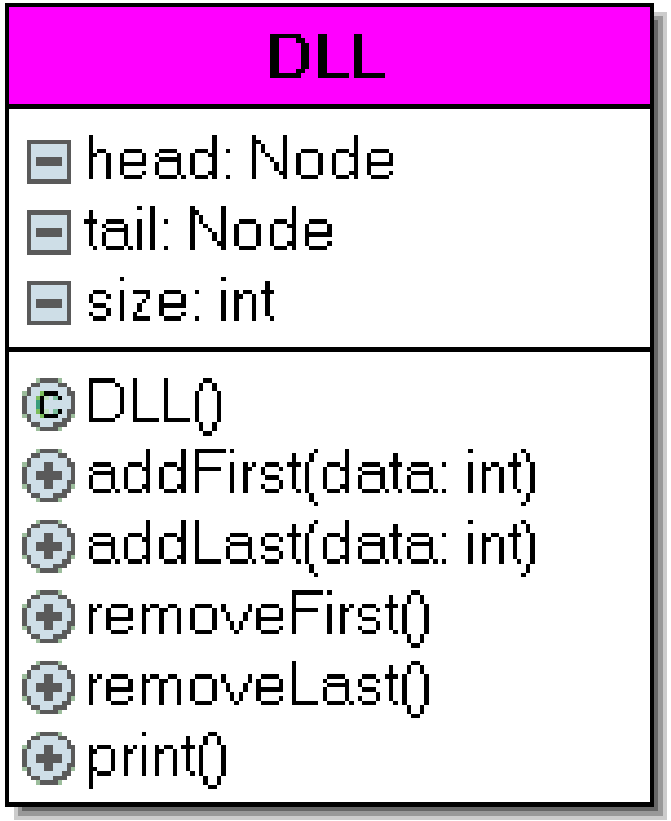
DLL Class

- Nested class



DLL Methods

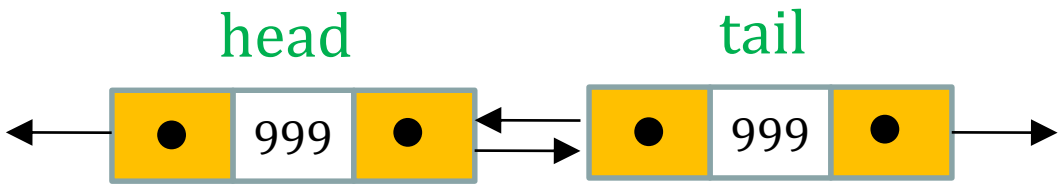
- DLL()
  - Constructor to create head and tail and link them together
- addFirst() and addLast()
  - Add new node at the front or the end
- removeFirst() and removeLast()
  - Remove first or last node
- print()
  - Show elements of DLL



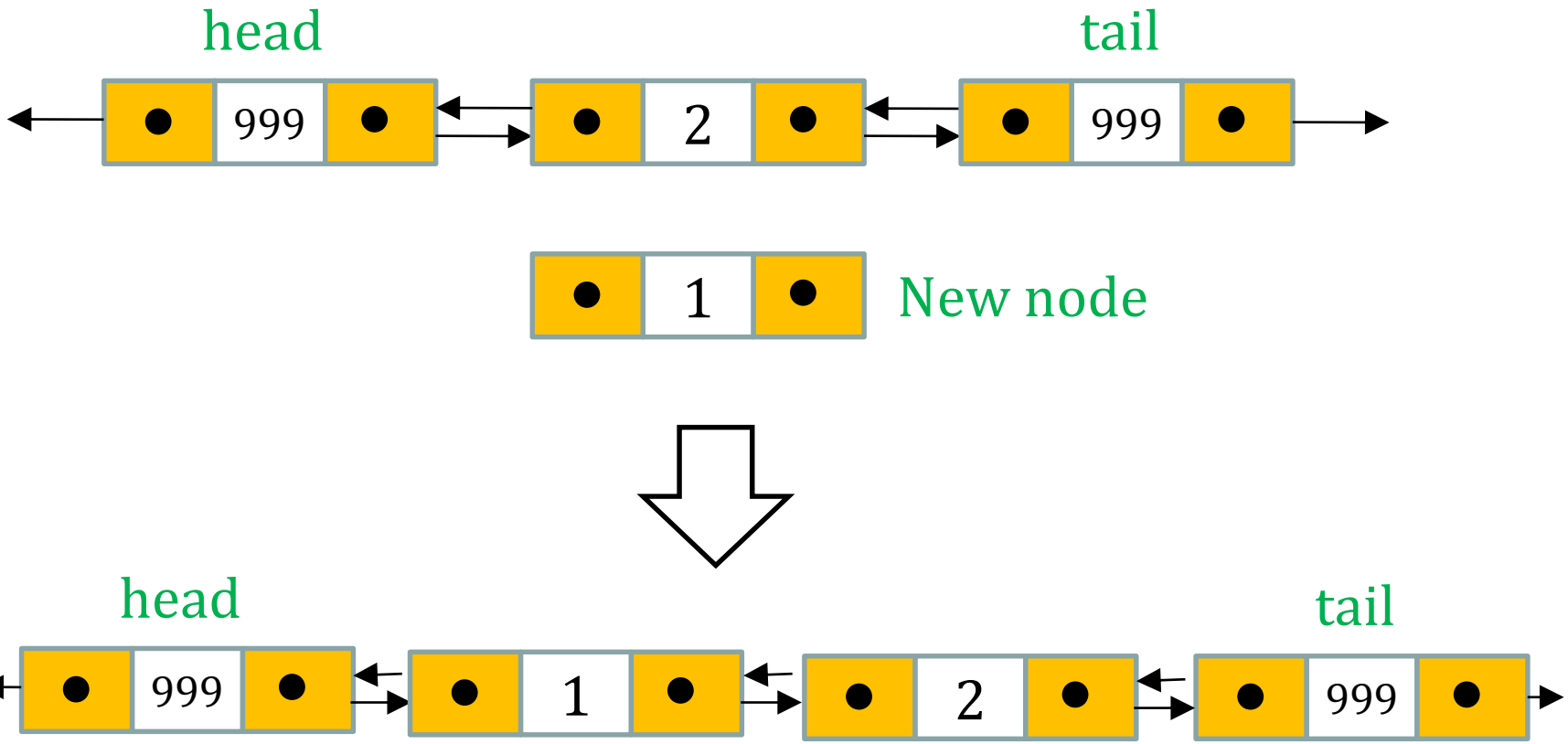
DLL: Constructor

- Create head and tail and link them together

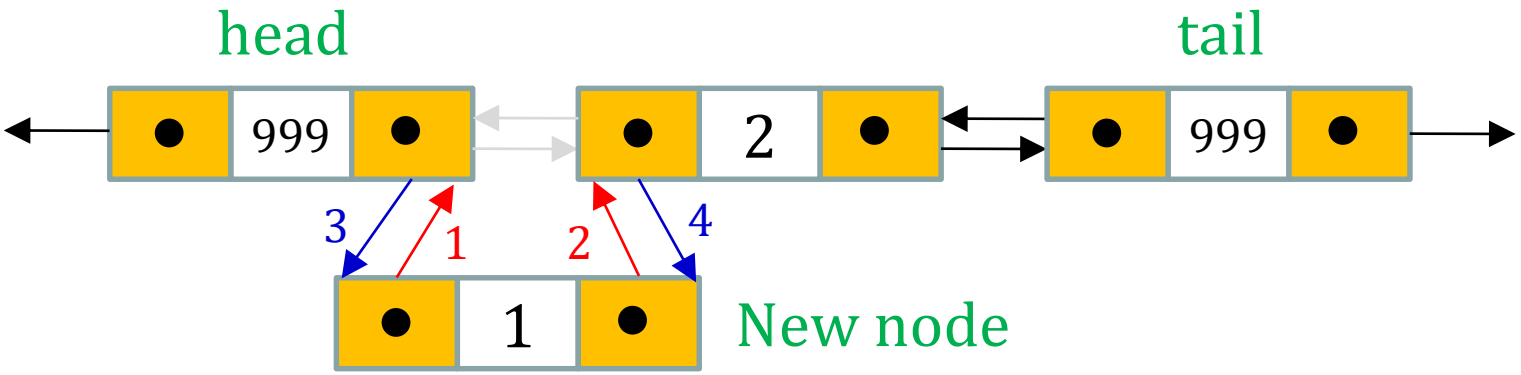
```
private Node head=null;
private Node tail=null;
private int size=0;      //SLL's size

public DLL(){
  //build an empty list
  head = new Node(999);
  tail = new Node(999);
  head.linkNext(tail);
  tail.linkPrev(head);
}
```



DLL: Add First Node

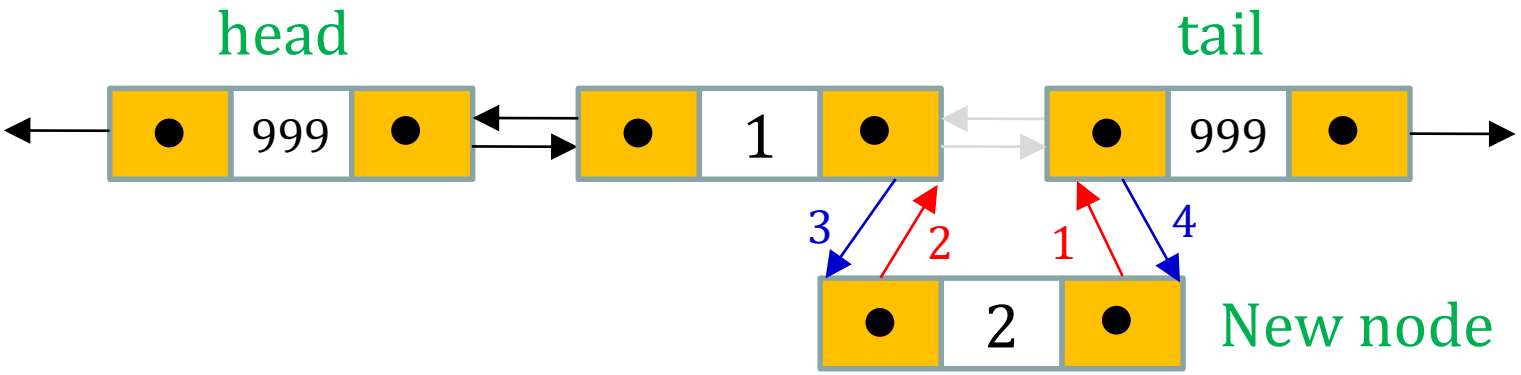


DLL: Add First Node



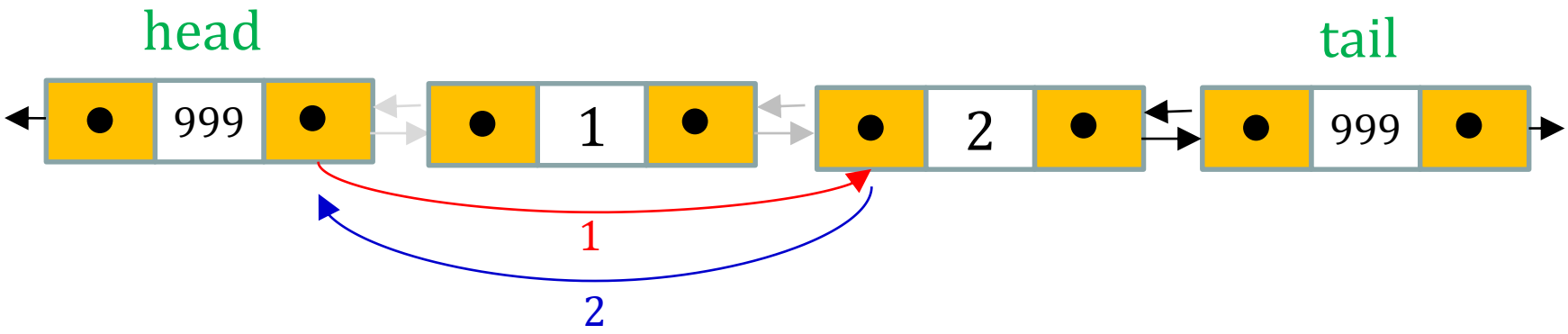

```
public void addFirst(int data){
  //create new node
  Node newNode = new Node(data);
1 newNode.linkPrev(head);
2 newNode.linkNext(head.getNextNode());
3 head.getNextNode().linkPrev(newNode);
4 head.linkNext(newNode);
  size++;
}
```

DLL: Add Last Node



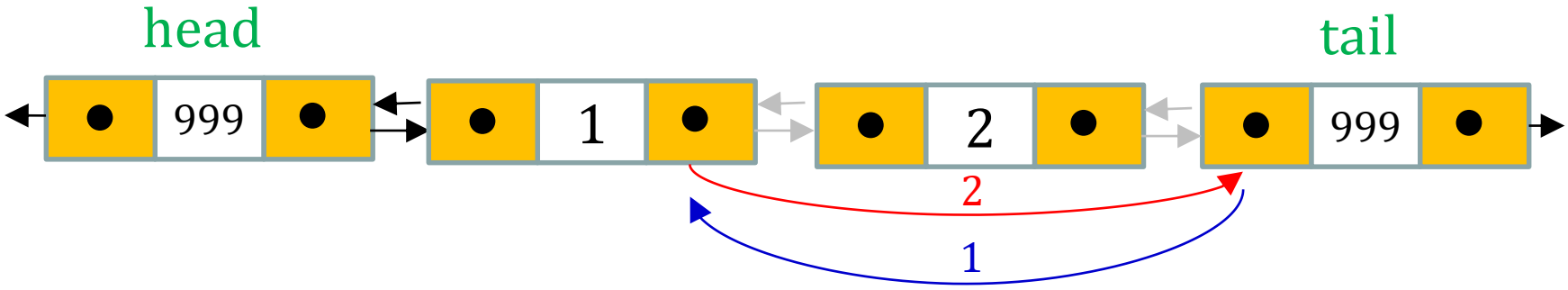

```
public void addLast(int data){
  Node newNode = new Node(data);
1 newNode.linkNext(tail);
2 newNode.linkPrev(tail.getPrevNode());
3 tail.getPrevNode().linkNext(newNode);
4 tail.linkPrev(newNode);
  size++;
}
```

DLL: Remove First Node




```
public void removeFirst(){
   //if(size==0) do nothing
   if(size==1){
      head.linkNext(tail);
      tail.linkPrev(head);
      size--;
   }
   else if(size>1){
1     head.linkNext(head.getNextNode().getNextNode());
2     head.getNextNode().linkPrev(head);
      size--;
   }
}
```

DLL: Remove Last Node




```
public void removeLast(){
  //if(size==0) do nothing
  if(size==1){
    head.linkNext(tail);
    tail.linkPrev(head);
    size--;
  }
  else if(size>1){
1   tail.linkPrev(tail.getPrevNode().getPrevNode());
2   tail.getPrevNode().linkNext(tail);
    size--;
  }
}
```

1501118
Data Structures and Algorithms

Chapter 4: Algorithm Analysis

2nd semester AY2022

School of Information Technology

Outline

- Introduction
- Circular Linked Lists
- Doubly Linked Lists
- CH4: Algorithm Analysis
- Summary

Efficient Algorithms

1. Give correct results
2. Run fast
3. Use less resources

- The last two items are dependent to an input size
- This course will focus only on "running time" (assume the algorithm is correct and neglect the resources used)

Time Measurement

- Experimental analysis
- Asymptotic analysis

Experimental Measurement

- Find elapsed time of running an algorithm

> Start time

>> Algorithm

> End time

- Elapsed time = End time – Start time
- Then test with different number of inputs

Example 1

```
public class TimeMeasure {

    public static void main(String[] args) {
        int n = 10000;
        int result = 0;
        long startTime = System.currentTimeMillis();
        for(int i=1;i<=n;i++) {
            for(int j=1;j<=n;j++) {
                result = i+j;
            }
        }
        long endTime = System.currentTimeMillis();
        long elapsedTime = endTime - startTime;
        System.out.println(elapsedTime);
    }
}
```

Start time

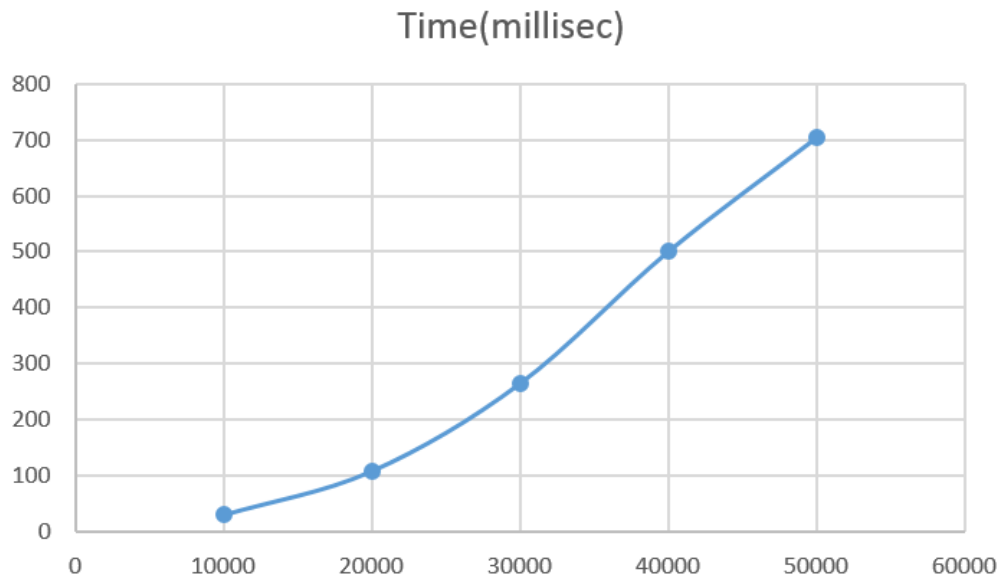
Algorithm

End time

Data Size VS Time

- From the exercise 1, assume that we vary data size (n) from 10000 to 50000

n	10K	20K	30K	40K	50K
Time(millisec)	31	109	265	500	703

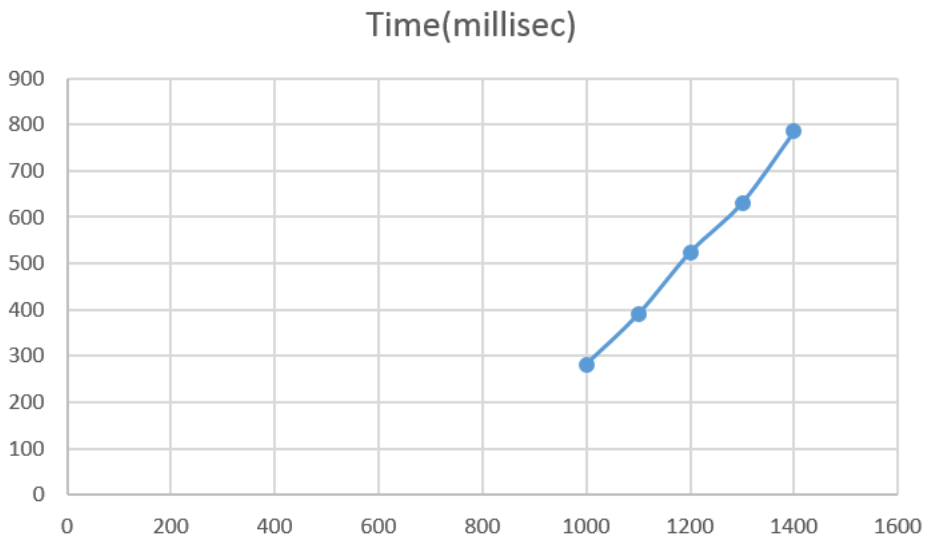


Example 2

```
for(int i=1;i<=n;i++) {
    for(int j=1;j<=n;j++) {
        for(int k=1;k<=n;k++) {
         result = i+j+k;
        }
    }
}
```

Assume that n is 1000 to 1400

n	1000	1100	1200	1300	1400
Time(millisec)	282	390	524	631	785



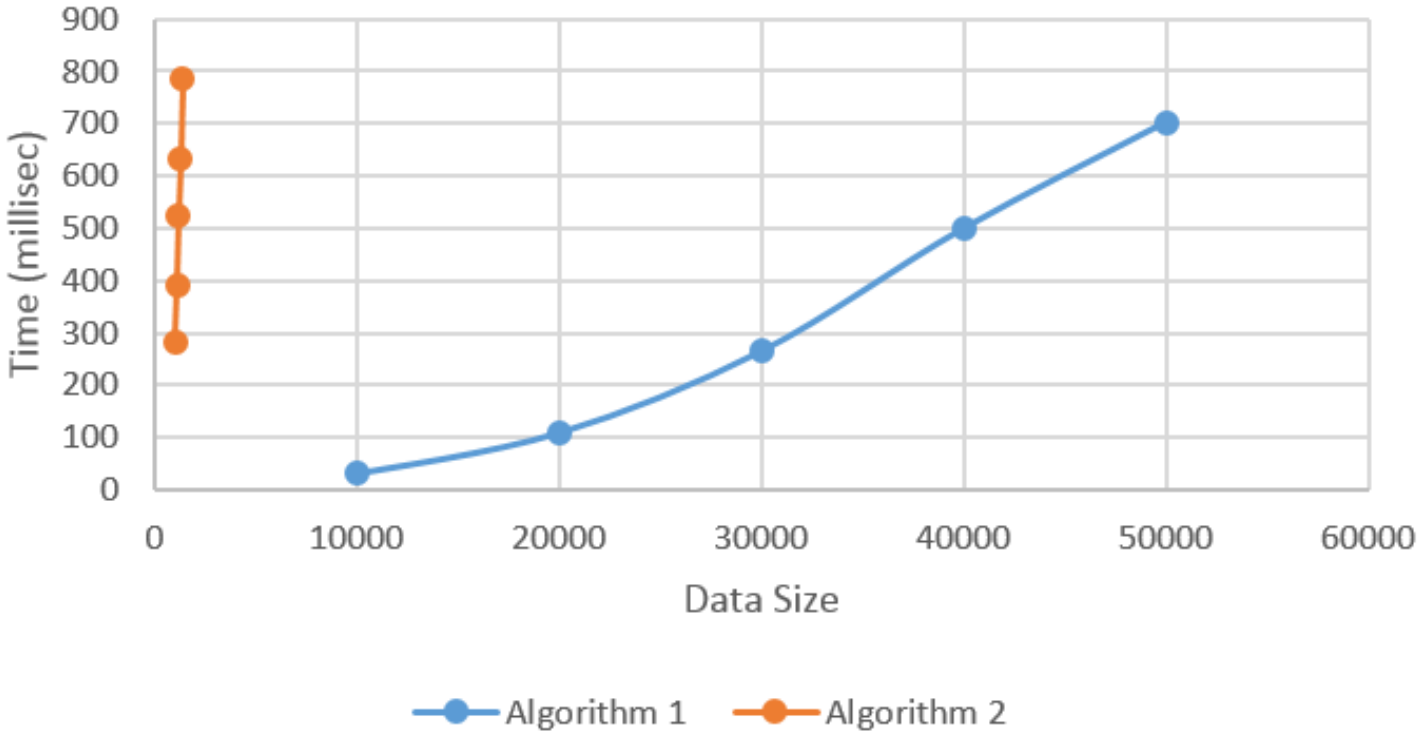
Which one is Faster?

- Algorithm 1

```
for(int i=1;i<=n;i++) {
    for(int j=1;j<=n;j++) {
        result = i+j;
    }
}
```

- Algorithm 2

```
for(int i=1;i<=n;i++) {
    for(int j=1;j<=n;j++) {
        for(int k=1;k<=n;k++) {
            result = i+j+k;
```



Limitations

- Must tested on the same hardware and software environments
- Limited sets of test inputs. Testing data may not cover all possibilities.
- Algorithms must be implemented to programming codes. Sometimes it will be too difficult to do or it takes too much time.
- Have to wait (sometimes very long) until the end of algorithm to get elapsed time.

Time Measurement

ASYMPTOTIC ANALYSIS

Primitive Operations

- All primitive operations take constant running time (not change with number of inputs)

- Declare variables or assign values to variables
  - int a = 1;

Primitive Operations

- Arithmetic operations
  - a = b + c;
- Compare two numbers
  - if(a>b)
- Access array's element by index
  - a[0]
- Call or get values from methods*
  - Math.sqrt(4)

**Just the call, not including the method processes*

Growth Rate of Functions

- Let n be a number of inputs

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

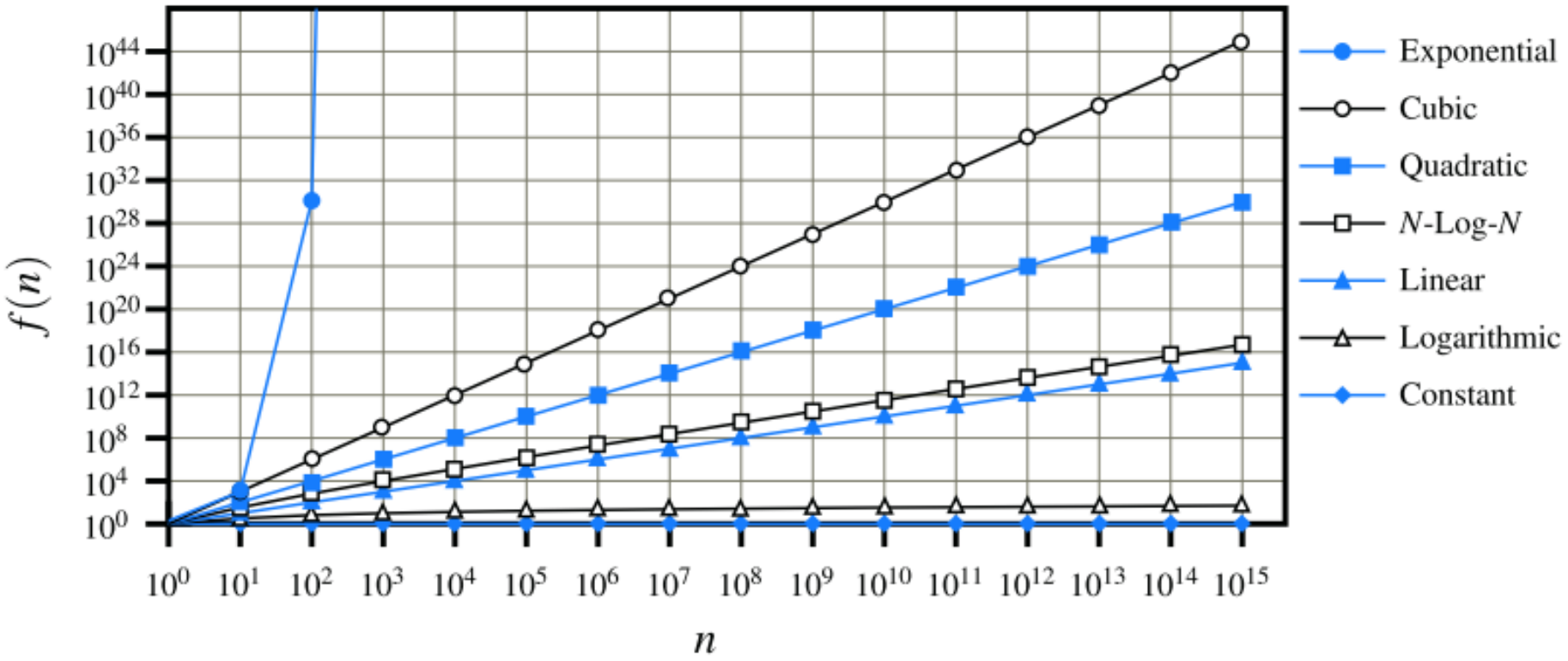
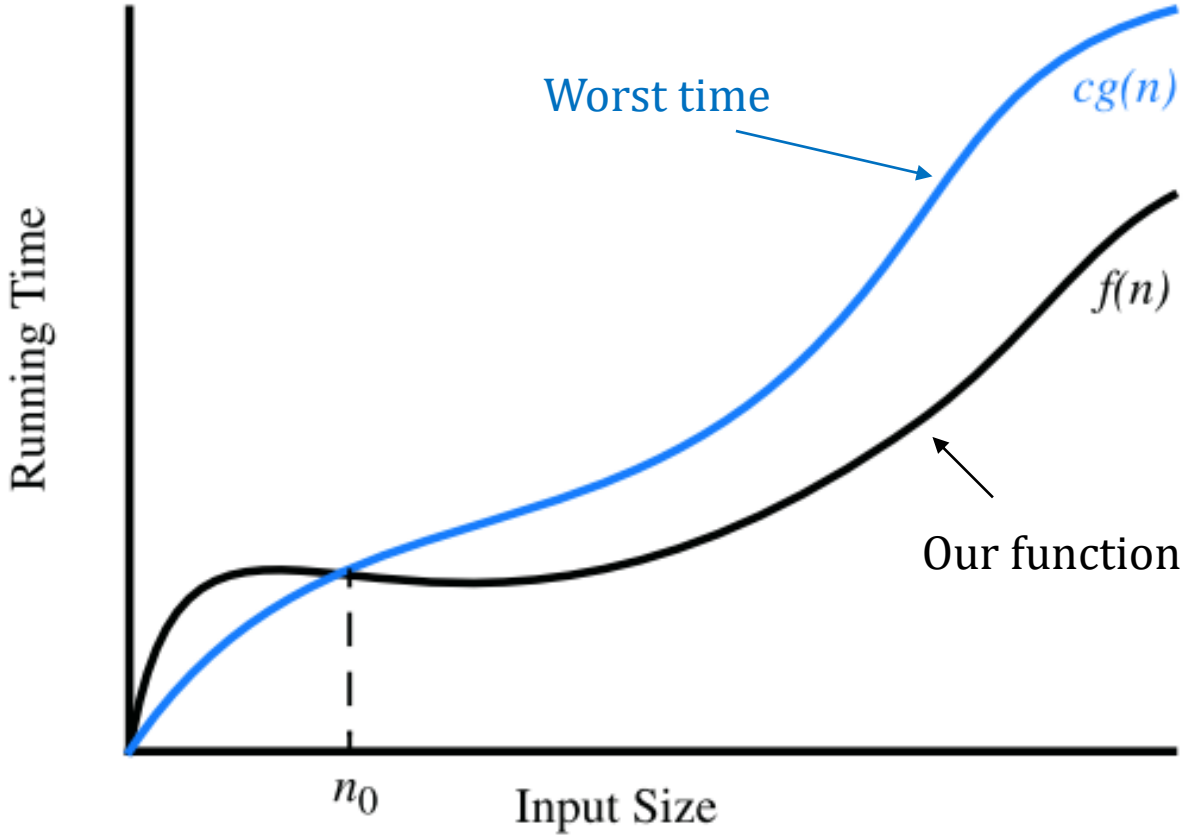


Image source: M. T. Goodrich et al., Data Structures and Algorithms in Java

Big-Oh Notation

- f(n) is O(g(n)) → f(n) <= c g(n) when $n>n_0$



Big-Oh Notation

- Tells "upper bound" or "worst case" of a function or an algorithm
- Means "no longer than" or "less than or equal"
- Example
  - O(1) tells that this function takes constant time to run
  - O(n) → linear time
  - $O(n^2)$ → quadratic time
  - Algorithm of O(n) is faster than of $O(n^2)$

Big-Oh Properties

- Ignore constant factors and lower-order terms
  - $3n^2 + n$ is $O(n^2)$
  - $2n + 1 + n^4$ is $O(n^4)$

- How about?
  - $5n^2 + 3n \log n + 2n + 5$
  - $5 \log n + 1 + 2^{n+2}$
  - $100 \log n + 3n$

Other Notations

- Big Omega Ω
  - Greater than or equal → Best case
- Big Theta Θ
  - Equal
- Little or small o
  - Less than
- Little omega ω
  - Greater than

Exercise A

Codes		Time
int sum = 0, i, j;		1
for(i=1; i<=n ; i ++) {		n
sum++;	//for each i	1
}		

Because the loop repeats "n" times

Constant time

Total number of operations is 1+(n*1) = **O(n)**

Exercise B

Codes		Time
int sum = 0, i, j;		1
for(i=1; i<=n ; i ++) {		n
for(j=1; j<=n; j++) {	//for each i	n
sum++;	//for each j	1
}		
}		

Total number of operations is 1+(n*n*1) = **$O(n^2)$**

Exercise B (Explained)

```
for(i=1; i<=n ; i++){
        for(j=1; j<=n; j++){
                sum++;
```

i	1			2			3		
j	1	2	3	1	2	3	1	2	3
Total j	3			3			3		

Assume that n = 3

for loop “j”, each “i” causes “n” operations.
So the average operations for each “i” in loop “j” is
(n*n)/n = n

Exercise C

Codes		Time
int sum = 0, i, j;		1
for(i=1; i<=n ; i ++) {		n
for(j=1; j<=n*n; j++) {	//for each i	n*n
sum++;	//for each j	1
}		
}		

Total number of operations is ?

Exercise D

Codes		Time
`int sum = 0, i, j;`		1
`for(i=1; i<=n ; i ++) {`		n
`    for(j=1; j<=i; j++) {`	//for each i	n/2
`        sum++;`	//for each j	1
`    }`		
`}`		

Total number of operations is 1+(n * n/2 * 1) = **$O(n^2)$**

Exercise D (Explained)

```
for(i=1;i<=n;i++){
        for(j=1;j<=i;j++){
                sum++;
```

i	1	2		3		
j	1	1	2	1	2	3
Total j	1	2		3		

Assume that n = 3
for loop "j", each "i" causes different number of operations.

The average operations for each "i" in loop "j" is
(1+2+3+…+n)/n = (n/2)(n+1)/n = (n+1)/2 ~ n/2

Exercise D (Another Explanation)

```
for(i=1; i<=n ; i++){
        for(j=1; j<=i; j++){
                sum++;
```

i	1	2		3		
j	1	1	2	1	2	3

Assume that n=3
Total operations = 1+2+3

Thus for "n" values
Total operations = $1+2+3+\ldots+n = (n/2)(n+1) = n^2/2 + n/2$
which is $O(n^2)$

Exercise E

Codes		Time
int sum = 0, i, j, k;		1
for(i=1; i<=n ; i ++) {		n
for(j=1; j<=i; j++) {	//for each i	n/2
for(k=1; k<=j; k++){	//for each j	n/4*
sum++;	//for each k	1
}		
}		
}		

Total is 1+(n * n/2 * n/4 * 1) = **$O(n^3)$**

Exercise E (Explained)

i	**1**	**2**			**3**					
j	1	1	2		1	2		3		
Total j for i	1	2			3					
k	1	1	1	2	1	1	2	1	2	3
Total k for i	1	3			6					

Average k for j = [1+3+6+…] / [1+2+3+…] =
[n(n+1)(n+2)/6] / [(n/2) (n+1)] = (n+2)/3 ~ n/3
But usually n/4 is preferred instead!

Exercise E (Another Explanation)

i	**1**	**2**			**3**					
j	1	1	2		1	2		3		
Total j for i	1	2			3					
k	1	1	1	2	1	1	2	1	2	3
Total k for i	1	3			6					

Assume that n=3, total operations = 1+3+6 = 10
Thus for "n" values
Total operations = 1+3+6+…+ (n/2)(n+1) =
$n(n+1)(n+2)/6$ which is $O(n^3)$

Exercise F

Codes	Time
int sum = 0, i, j, k;	1
for(i=1; i<=n ; i ++) {	n
for(j=1; j<=i*i; j++) {	$n^2/2$
for(k=1; k<=j; k++){	$n^2/4$
sum++;	1
}	
}	
}	

Total is $1+(n * n^2/2 * n^2/4 * 1) = \mathbf{O(n^5)}$

Exercise G

Codes	Time
int sum = 0, i, j, k;	1
for(i=1; i<=n ; i ++) {	n
for(j=1; j<=i*i; j++) {	$n^2/2$
if(j%i == 0) {	1
for(k=1; k<=j; k++){	$n^2/4$
sum++;	1
}	
}	
}	
} Total is $1+(n * n^2/2) + (n^2/2 * 1 * n^2/4 *1) =$ **$O(n^4)$**	

Exercise G (Explain)

i	1	2			
j<=i*i	1	1	2	3	4
j%i==0	yes	no	yes	no	yes
k<=j	1	-	1 2		1 2 3 4

Not all i and j process k.
Therefore, we split the algorithm to two parts
-i and j → $O(n^3)$
-j and k → $O(n^2 \times n^2) = O(n^4)$
Max is $O(n^4)$

Exercise H

Codes		Time
int sum = 0, i, j, k;		1
for(i=1; i<=n ; i ++) {		n
for(j=1; j<=n; j++) {	//for each i	n
for(k=1; k<=n; k++){	//for each j	n
if(i==j && j==k)	//for each k	1
sum++;		1
}		
}		
}	Total is 1+[n * n * n * (1+1)] = **O(n^3)**	

Exercise I

Codes	Time
int sum = 0, i, j, k;	1
for(i=1; i<=n ; i ++) {	n
for(j=1; j<=n; j++) {	n
if(i==j){	1
for(k=1; k<=n; k++){	n
if(j==k)	1
sum++;	1
}	
}	
}	
} Total is 1+[(n * n) + (n * 1 * n * (1+1))] = **$O(n^2)$**	

Summary of Analysis Rules

- Rule 1: A single loop is O(n)
- Rule 2: General **k** nested loop is $O(n^k)$
- Rule 3: If nested loop is broken by "if" statement, split loops and find Big O for each part. Then choose max.

Summary of Analysis Rules

- Rule 4: If there are two or more consecutive algorithms, choose largest Big O

- Rule 5:

	Running time
if(condition)	A
statement 1;	B
else	
statement 2;	C

Total time = A + max(B, C)

Summary

- What are differences between each type of link lists?
- Explain operations on linked lists?
- How to measure efficiency of algorithms?
- What is Big O?
- How to evaluate algorithms in terms of Big O?