LISTS

CS210 – Data Structures and Algorithms

Dr. Basit Qureshi

جامعة الامير سلطان
PRINCE SULTAN
UNIVERSITY

https://www.drbasit.org/

TOPICS

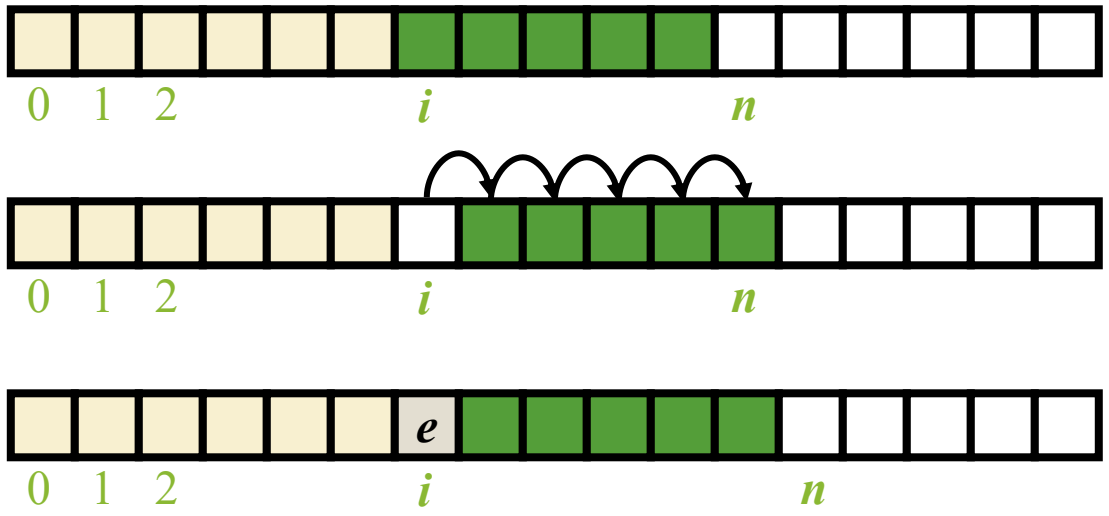
- Arrays and Array Lists
- Singly Linked Lists
  - Insert – Search – Remove
  - Other methods
- Doubly Linked Lists
  - Insert – Search - Remove
- Circular Linked List
  - Insert – Search - Remove

ARRAYS

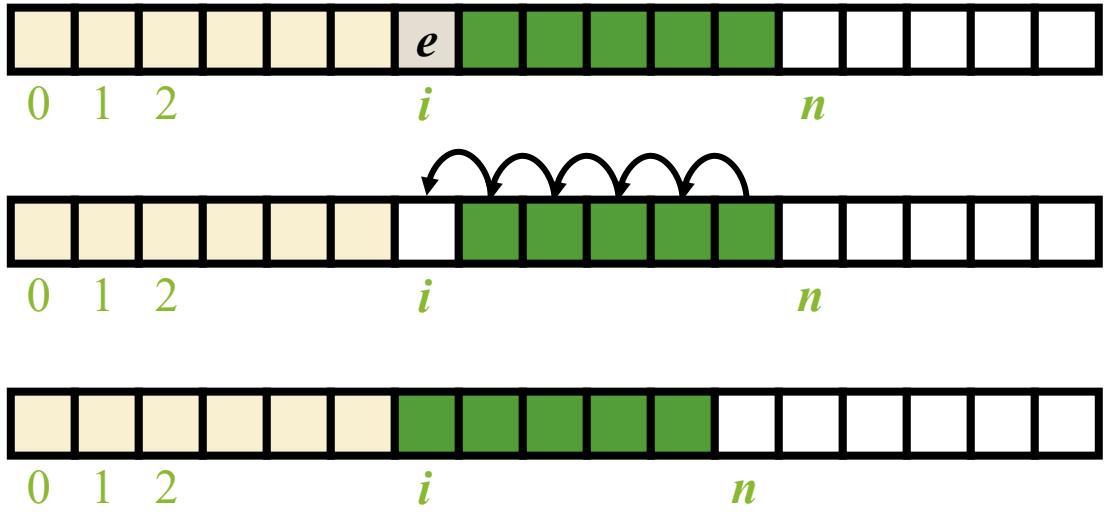
- An ***array*** is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
- Goods: Random access; Easy sorting, iteration; replacement of multiple variables.
- Bads: Fixed size, non-variable length / capacity; Costly to insert/delete; wasted resources, re-sizing is costly.

ARRAYS

- Problems!
  - Fixed size, non-variable length / capacity.
  - Adding removing entries is expensive. Requires Shifting.



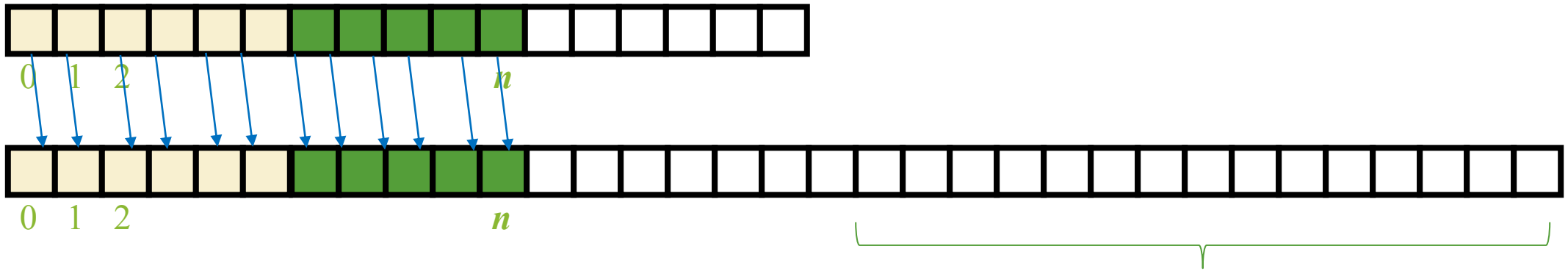
To add an entry e into array board at index i, we need to make room for it by shifting forward the $n - i$ entries



To remove the entry e at index i, we need to fill the gap left by e by shifting backward the $n - i - 1$ elements

ARRAYS

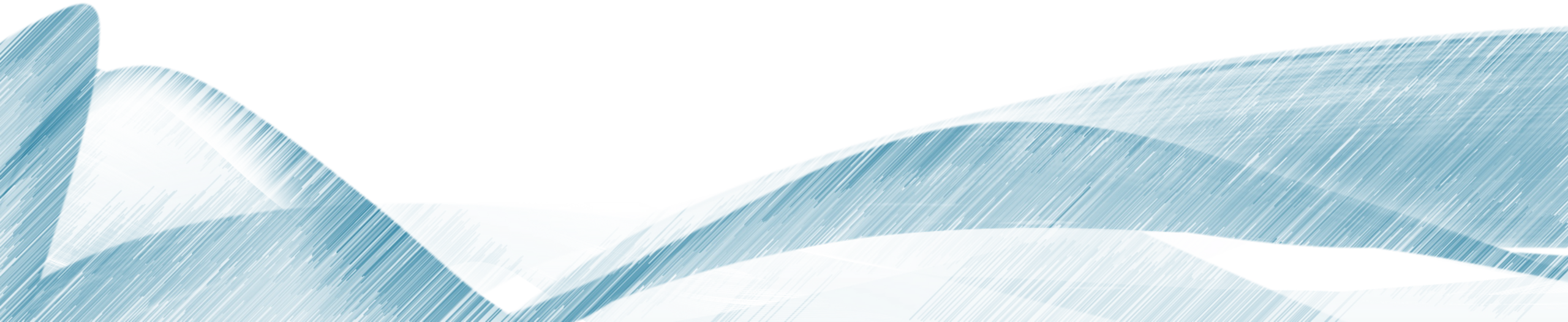
- Problems!
  - Fixed size, non-variable length / capacity.
  - Adding removing entries is expensive. Requires Shifting.
  - wasted resources, re-sizing is costly



1. Create a new array double the size
2. Copy "EACH" element from the old array to new
3. Discard the old array

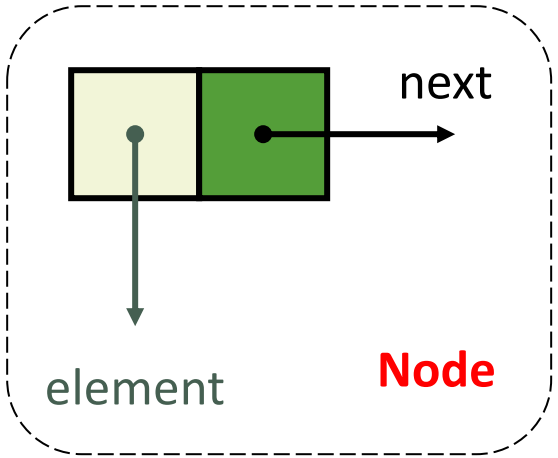
`java.util.ArrayList`

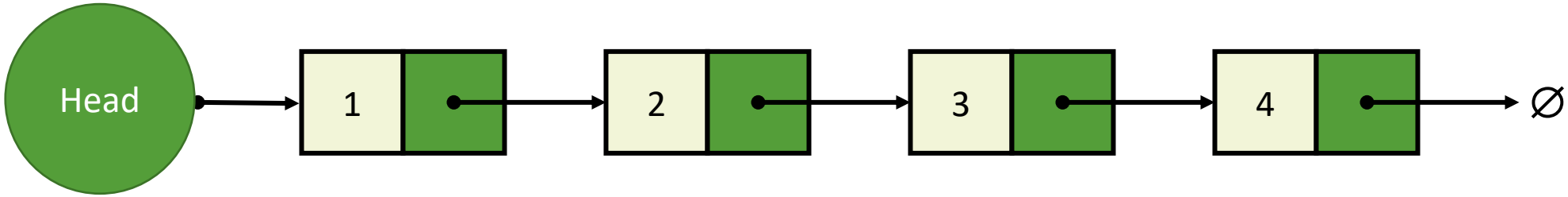
SINGLY LINKED LISTS



SINGLY LINKED LISTS (SLL)

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each Node stores
  - Element value
  - link to the next node





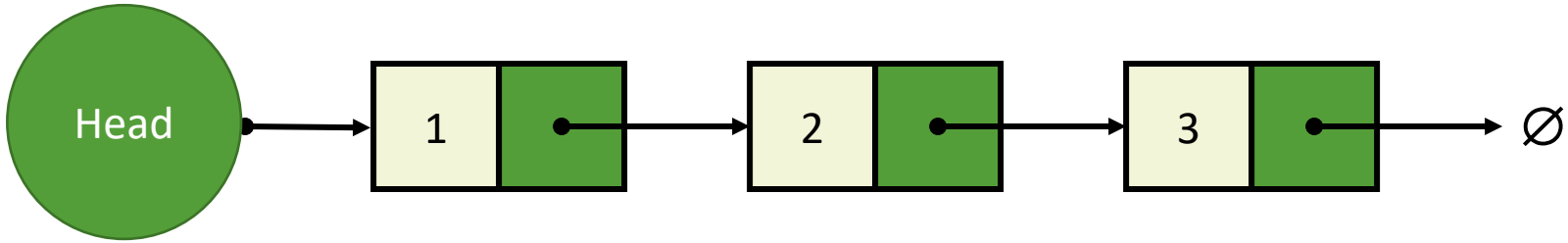
SINGLY LINKED LISTS (SLL) API

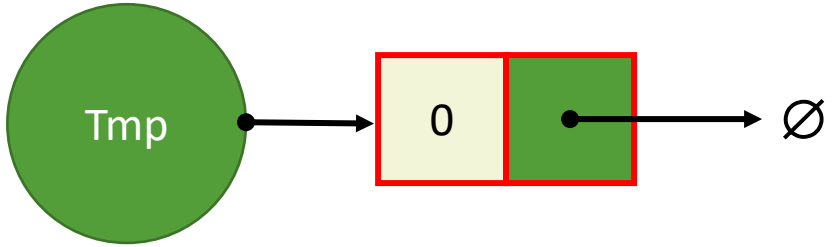
SLL
Node Head; int size;
void **insert**(int x); Node **remove**(int x); Node **search**(int x); boolean **isEmpty**(); String **toString**();

Node
int val; Node next;
Node(); **Node**(,);

SINGLY LINKED LISTS (SLL) API

- Insert
  - **Beginning**, end, middle;

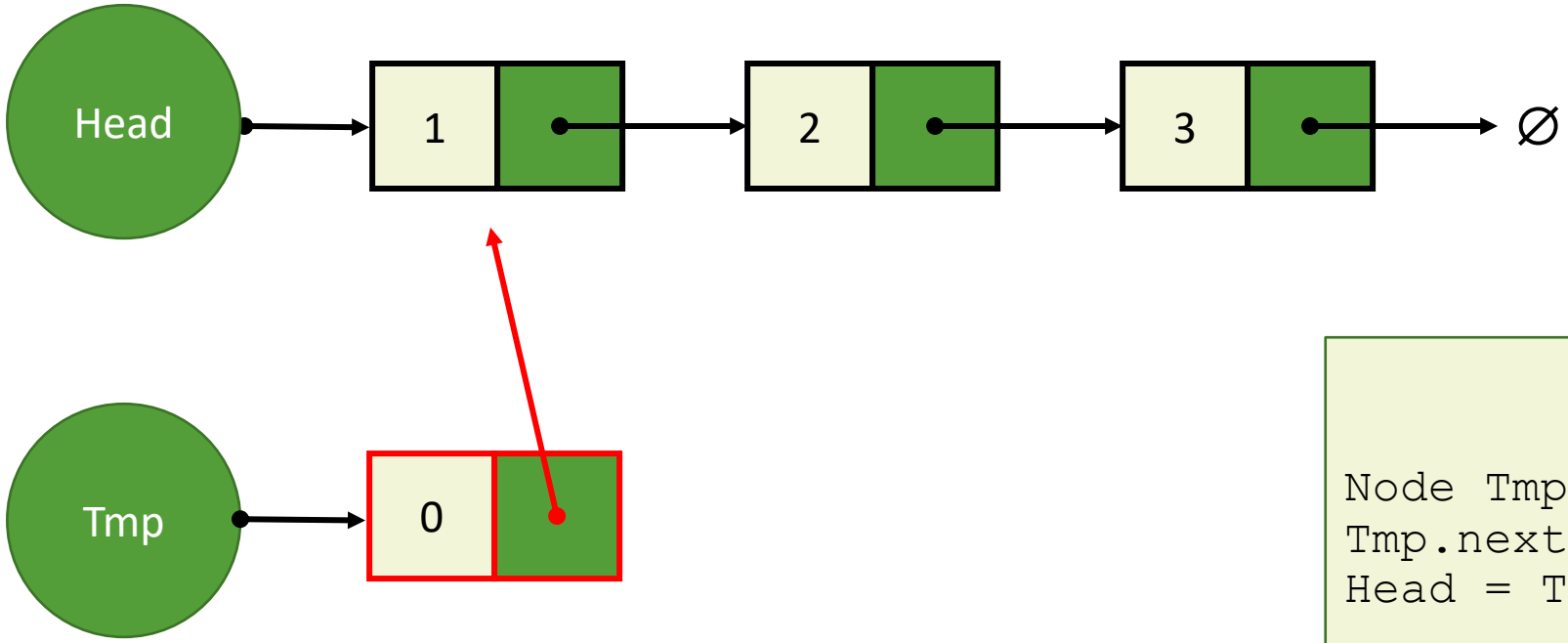





```
Node Tmp = new Node(0);
Tmp.next = Head;
Head = Tmp;
```

SINGLY LINKED LISTS (SLL) API

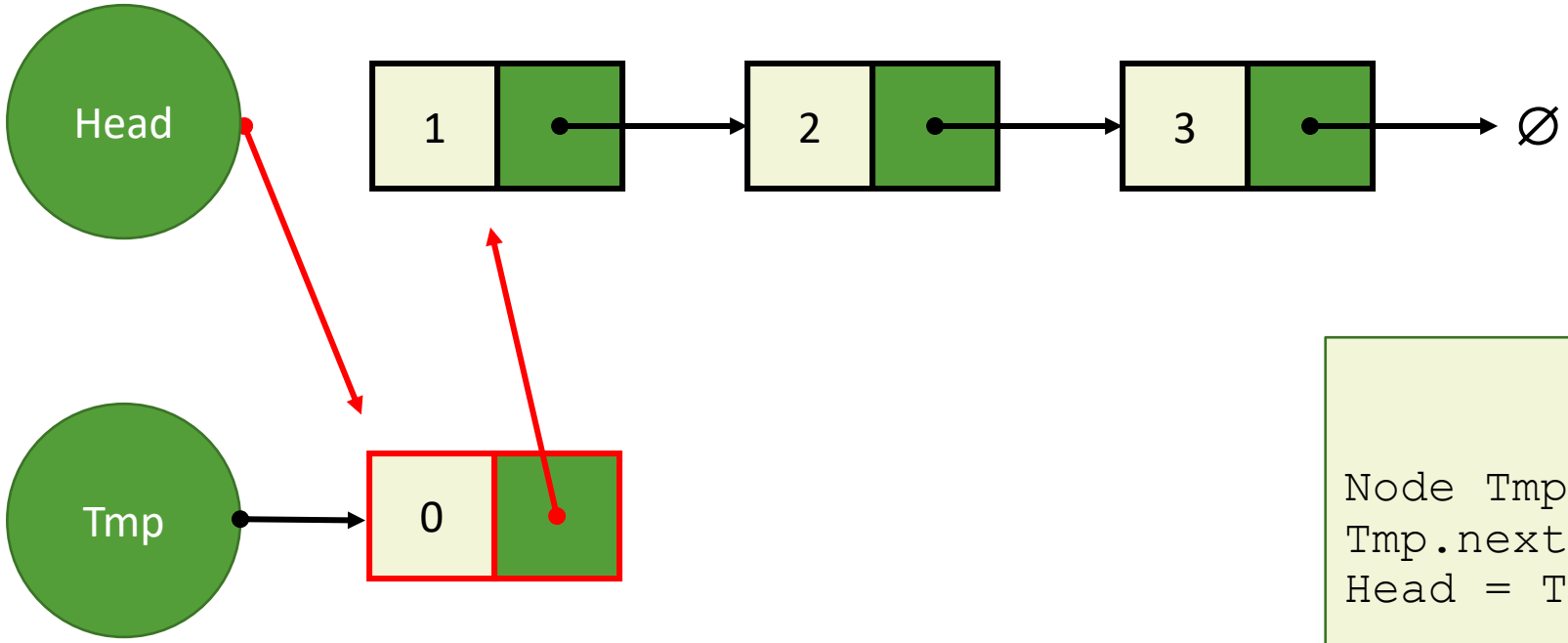
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
Head = Tmp;
```

SINGLY LINKED LISTS (SLL) API

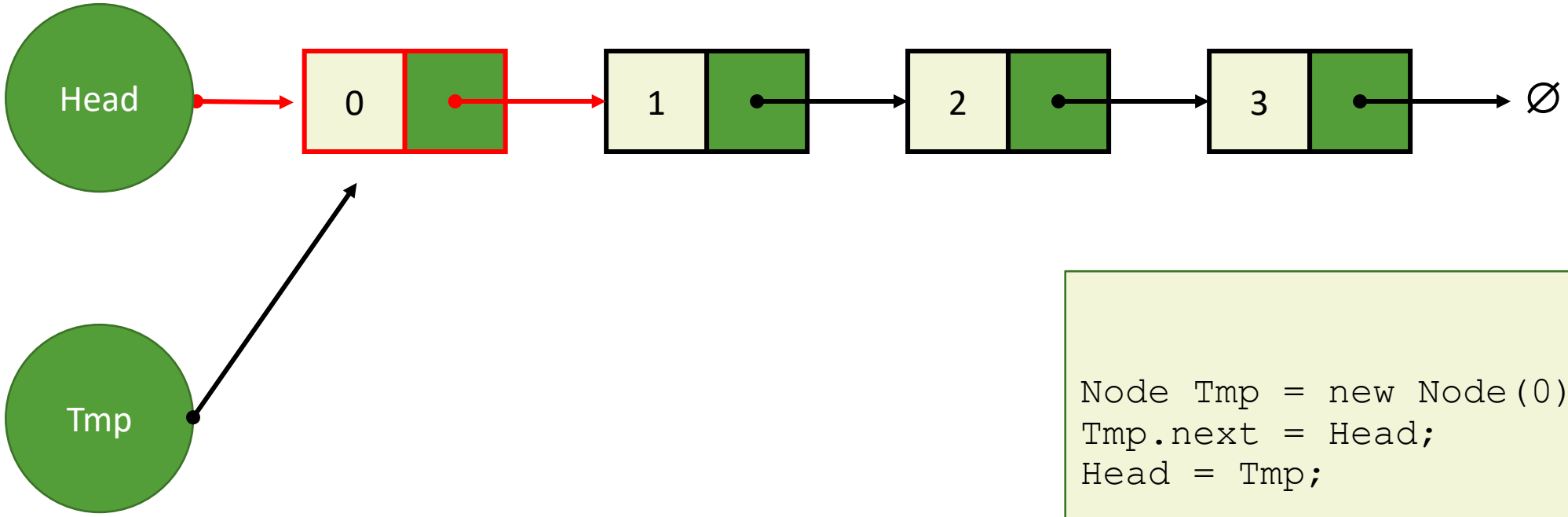
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
Head = Tmp;
```

SINGLY LINKED LISTS (SLL) API

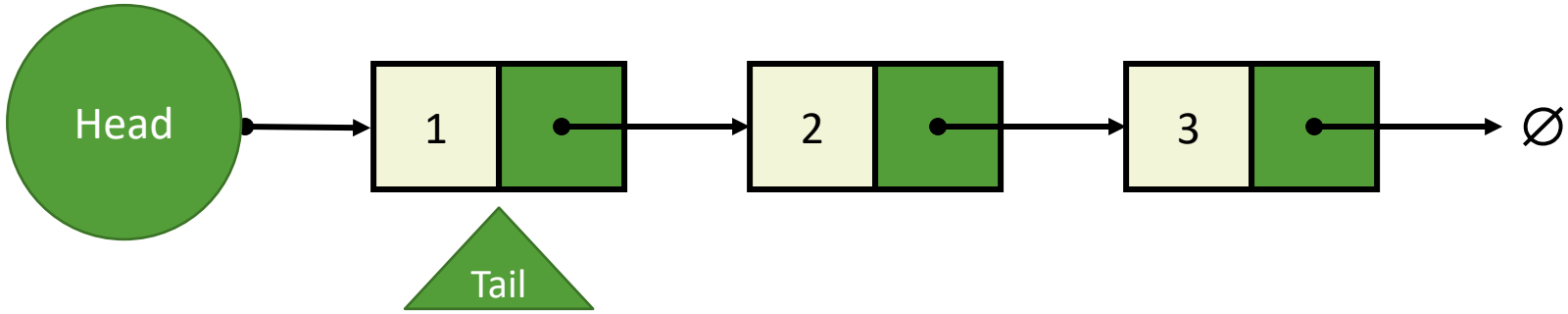
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
Head = Tmp;
```

SINGLY LINKED LISTS (SLL) API

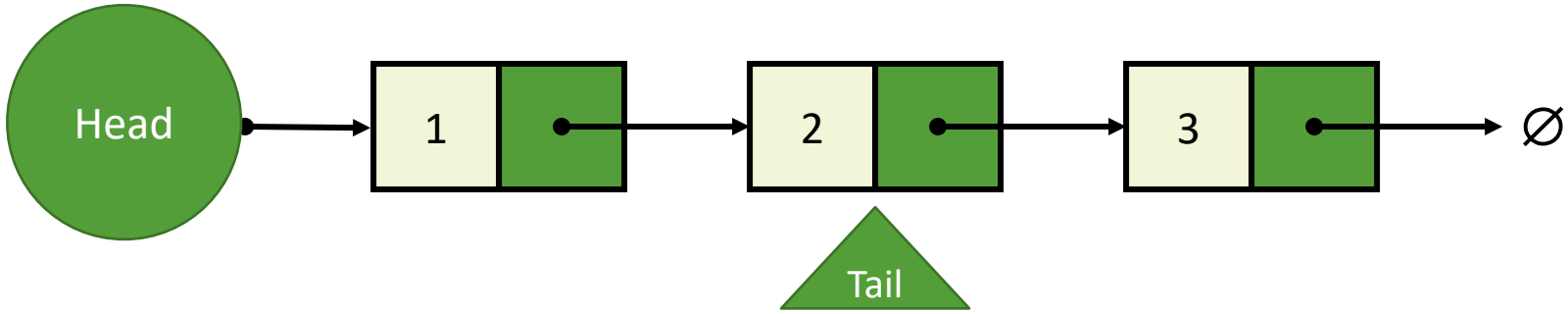
- Insert
  - Beginning, **end**, middle;




```
Node Tail = Head;
while(Tail.next!=null)
  Tail = Tail.next;
```

SINGLY LINKED LISTS (SLL) API

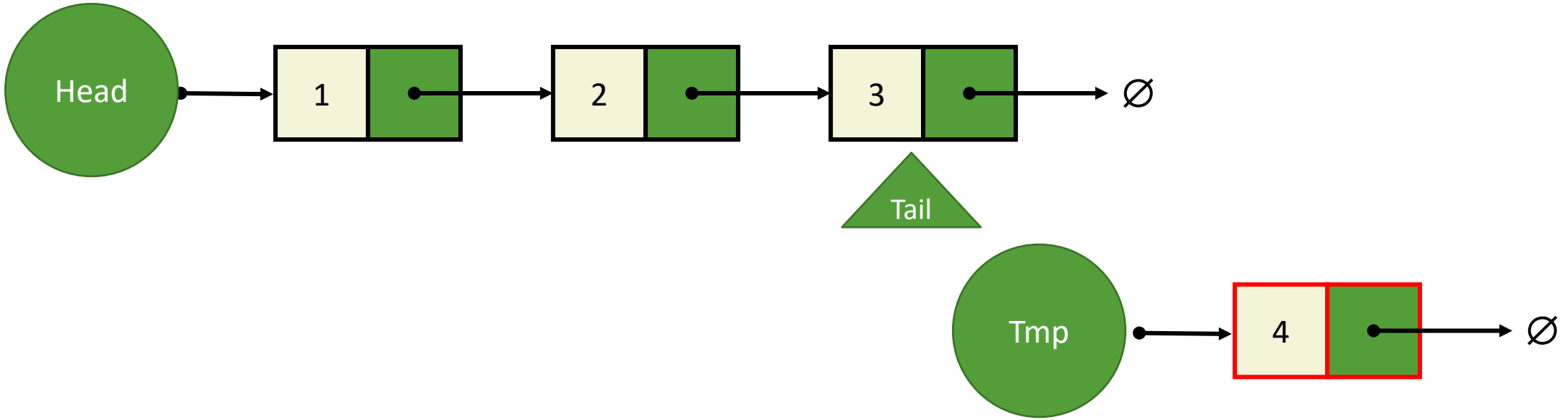
- Insert
  - Beginning, **end**, middle;




```
Node Tail = Head;
while(Tail.next!=null)
  Tail = Tail.next;
```

SINGLY LINKED LISTS (SLL) API

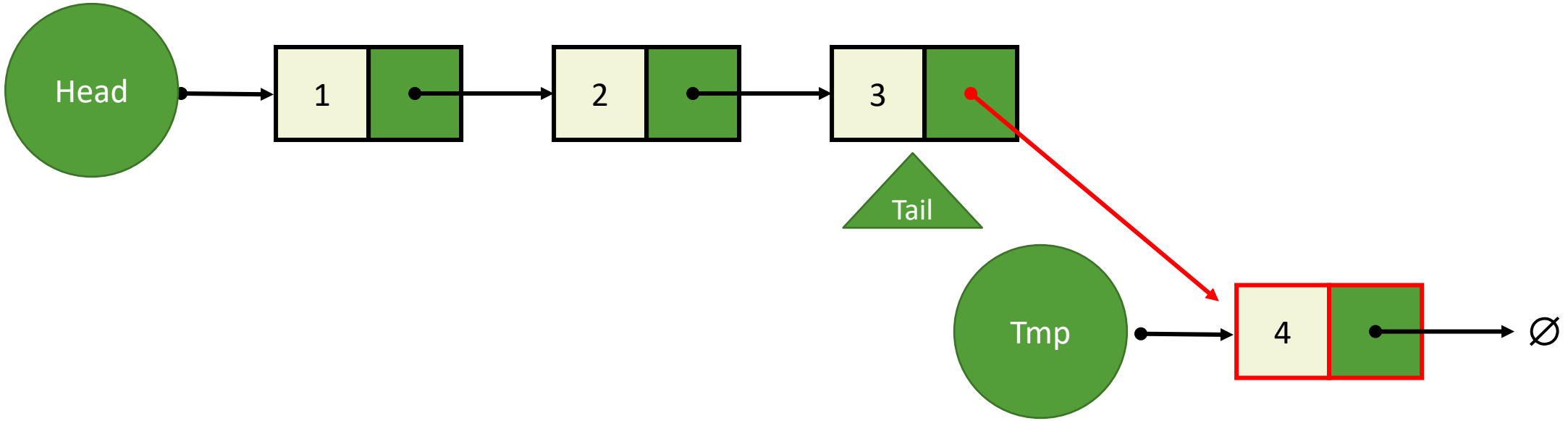
- Insert
  - Beginning, **end**, middle;




```
Node Tail = Head;
while(Tail.next!=null)
  Tail = Tail.next;

Node Tmp = new Node(4);
Tmp.next = null;
```

SINGLY LINKED LISTS (SLL) API

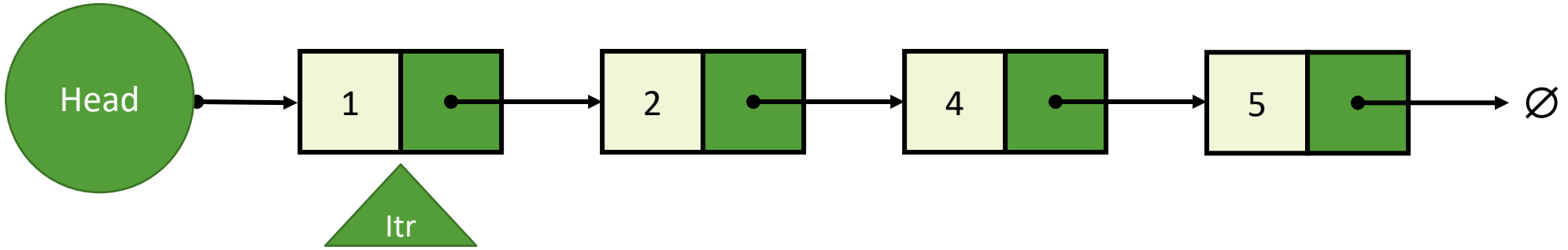
- Insert
  - Beginning, **end**, middle;




```
Node Tail = Head;
while(Tail.next!=null)
  Tail = Tail.next;

Node Tmp = new Node(4);
Tmp.next = null;
Tail.next = Tmp;
```

SINGLY LINKED LISTS (SLL) API

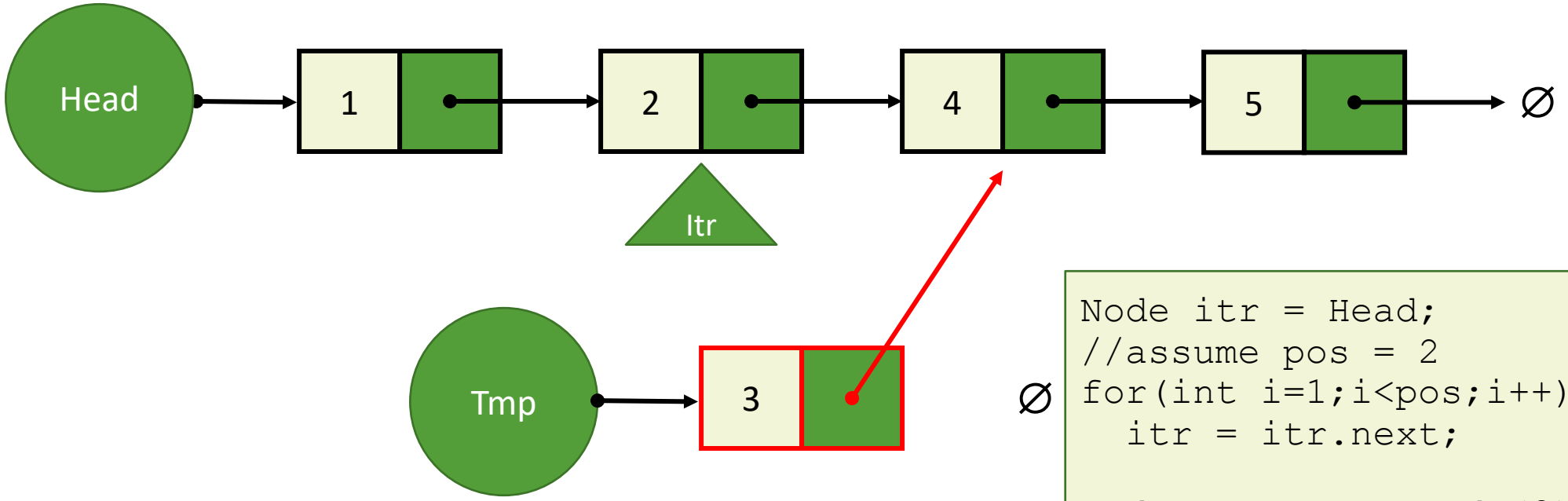
- Insert
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 2
for(int i=1;i<pos;i++)
  itr = itr.next;
```

SINGLY LINKED LISTS (SLL) API

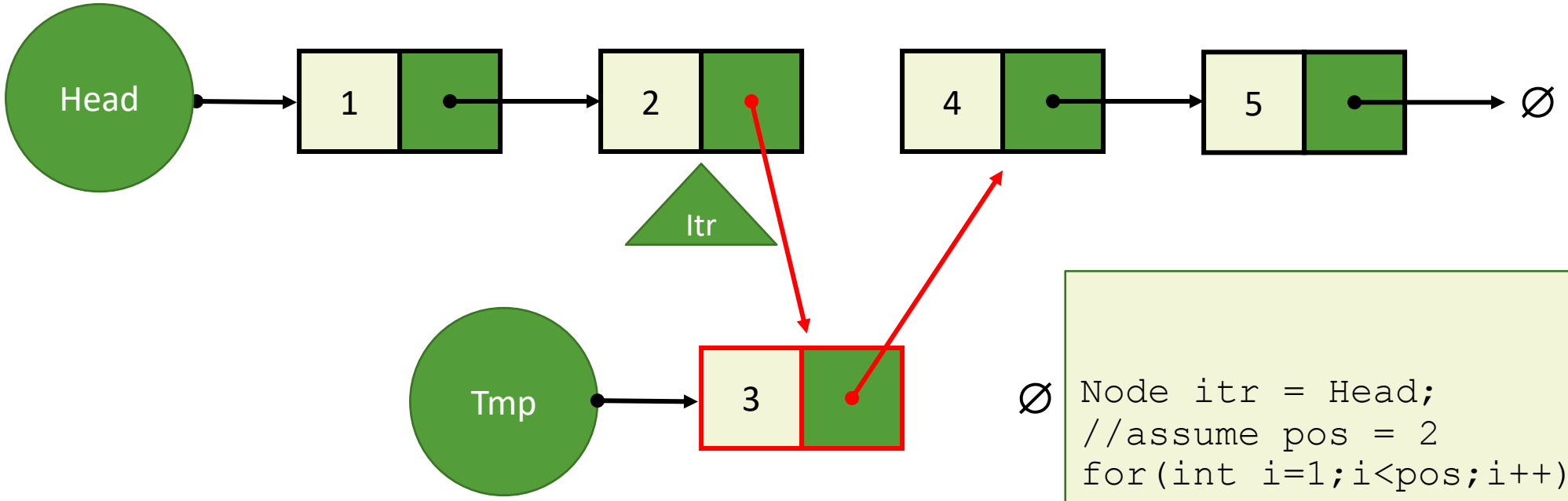
- Insert
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 2
for(int i=1;i<pos;i++)
  itr = itr.next;

Node Tmp = new Node(3);
Tmp.next = itr.next;
itr.next = Tmp;
```

SINGLY LINKED LISTS (SLL) API

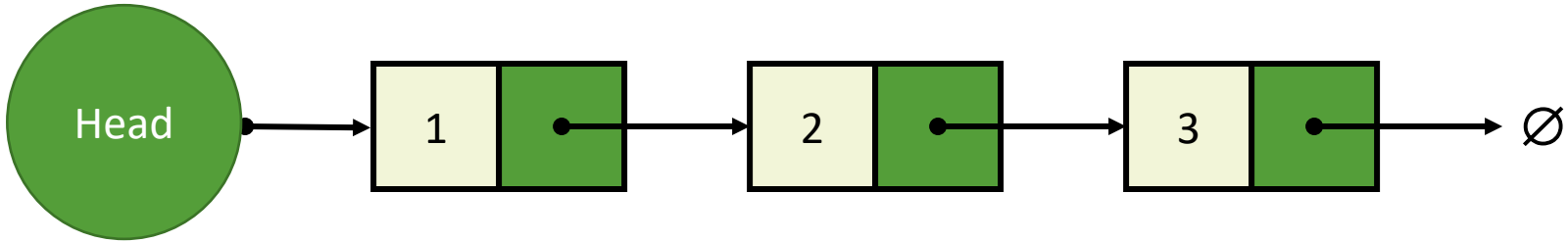
- Insert
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 2
for(int i=1;i<pos;i++)
  itr = itr.next;

Node Tmp = new Node(3);
Tmp.next = itr.next;
itr.next = Tmp;
```

SINGLY LINKED LISTS (SLL) API

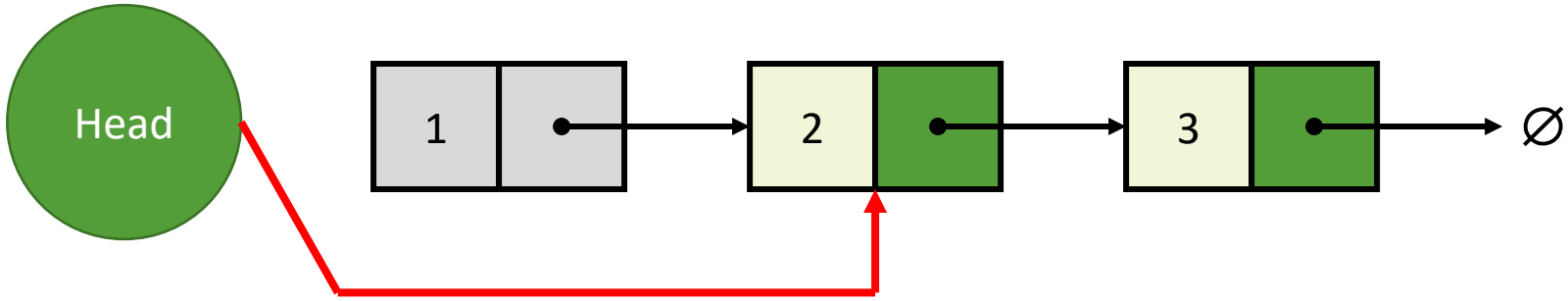
- Remove
  - **Beginning**, end, middle;




```
Head = Head.next;
```

SINGLY LINKED LISTS (SLL) API

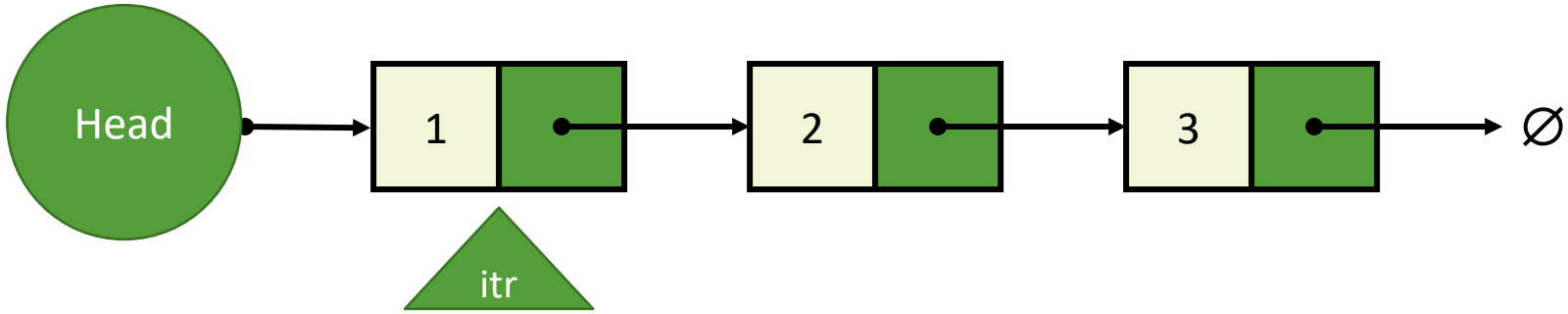
- Remove
  - **Beginning**, end, middle;




```
Head = Head.next;
```

SINGLY LINKED LISTS (SLL) API

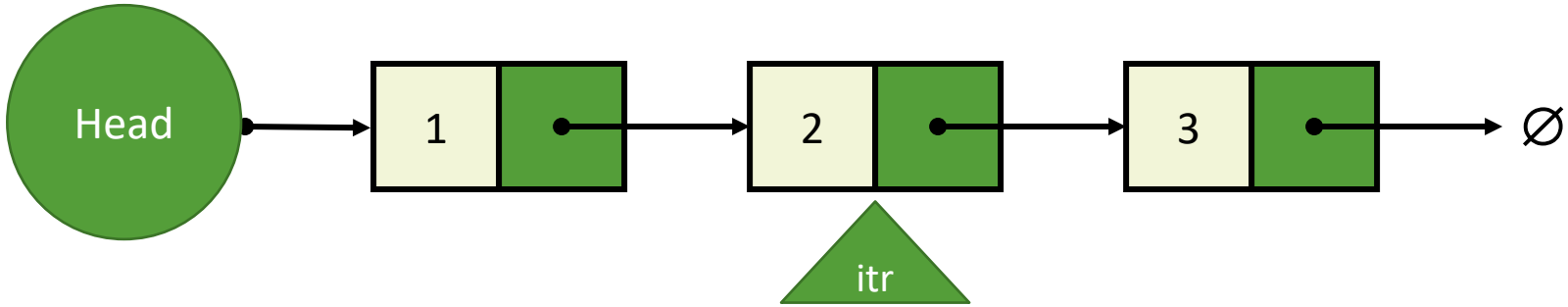
- Remove
  - Beginning, **end**, middle;




```
Node itr = Head;
for(int i=1;i<size - 1;i++)
  itr = itr.next;
```

SINGLY LINKED LISTS (SLL) API

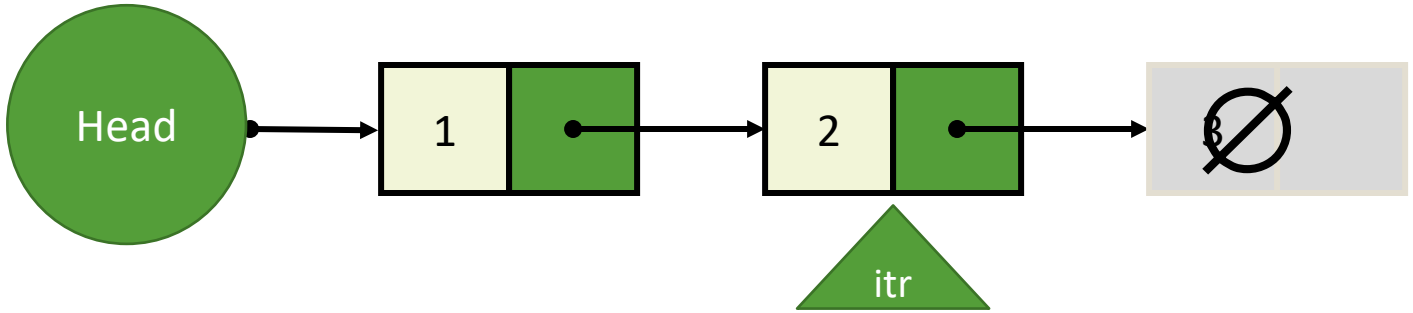
- Remove
  - Beginning, **end**, middle;




```
Node itr = Head;
for(int i=1;i<size - 1;i++)
  itr = itr.next;
```

SINGLY LINKED LISTS (SLL) API

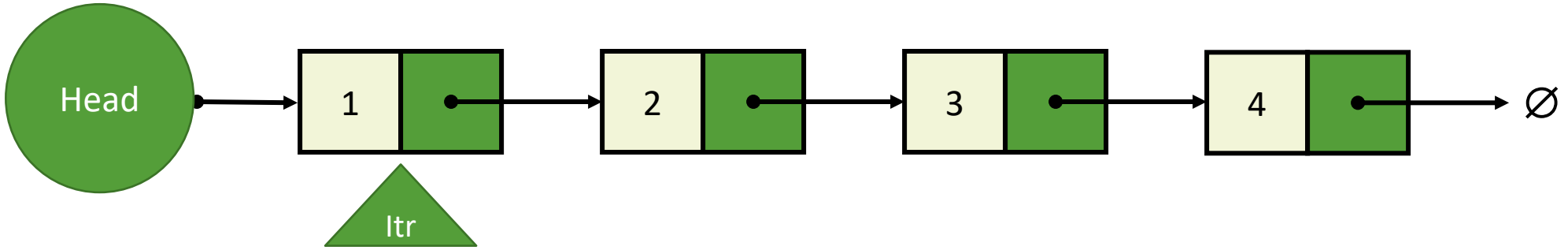
- Remove
  - Beginning, **end**, middle;




```
Node itr = Head;
for(int i=1;i<size - 1;i++)
  itr = itr.next;
itr.next = null;
```

SINGLY LINKED LISTS (SLL) API

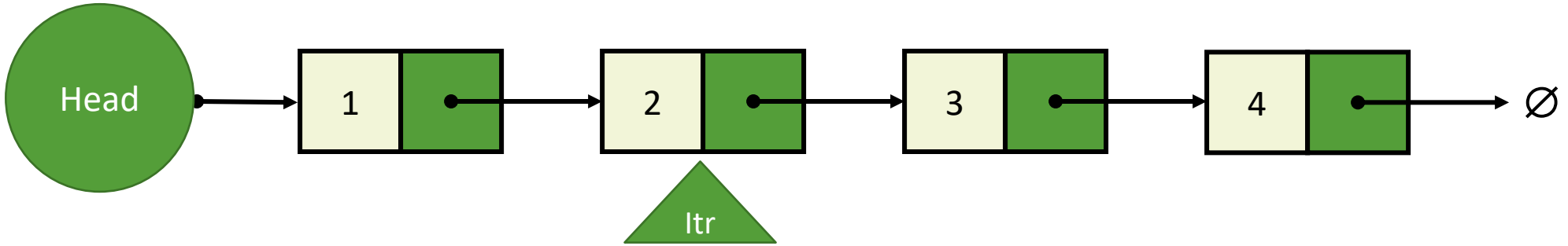
- Remove
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 3
for(int i=1;i<pos - 1;i++)
  itr = itr.next;
```

SINGLY LINKED LISTS (SLL) API

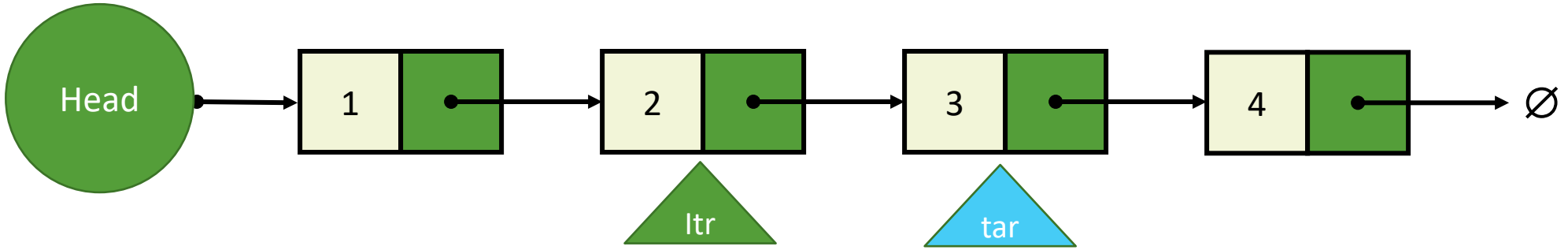
- Remove
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 3
for(int i=1;i<pos - 1;i++)
  itr = itr.next;
```

SINGLY LINKED LISTS (SLL) API

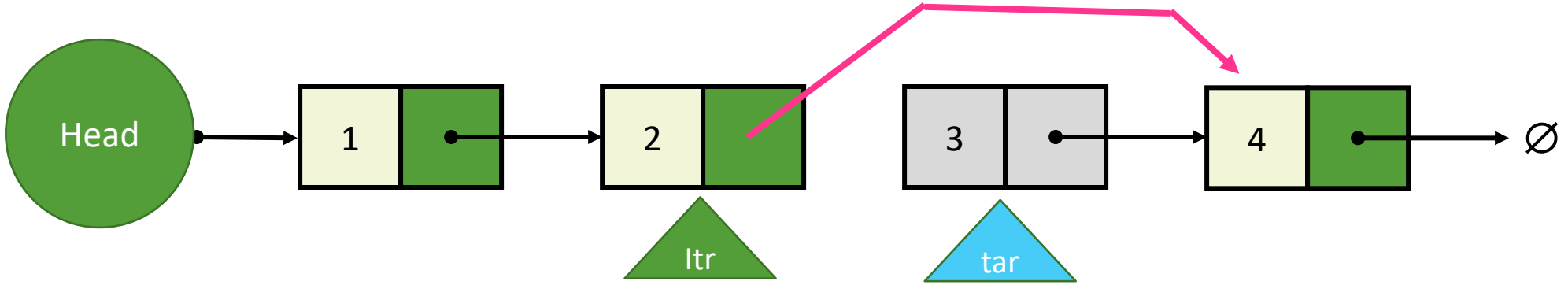
- Remove
  - Beginning, end, **middle**;




```
Node itr = Head;
//assume pos = 3
for(int i=1;i<pos - 1;i++)
  itr = itr.next;

Node tar = itr.next;
itr.next = tar.next;
```

SINGLY LINKED LISTS (SLL) API

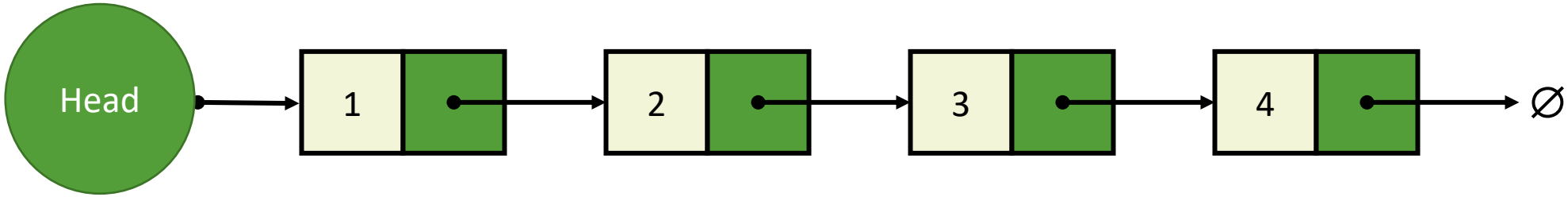
- Remove
  - Beginning, end, **middle**;



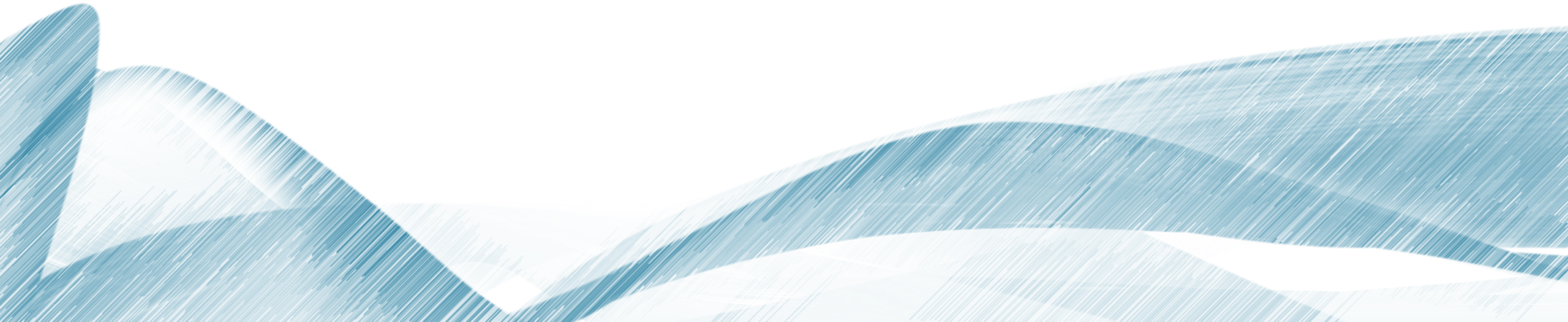

```
Node itr = Head;
//assume pos = 3
for(int i=1;i<pos - 1;i++)
  itr = itr.next;

Node tar = itr.next;
itr.next = tar.next;
```

SINGLY LINKED LISTS (SLL)

- Insert at the beginning is fast; middle and end is not efficient.
- Remove from the beginning is fast; middle and end is also not efficient.
- Once the Iterator moves forward, there is no way to take it back.

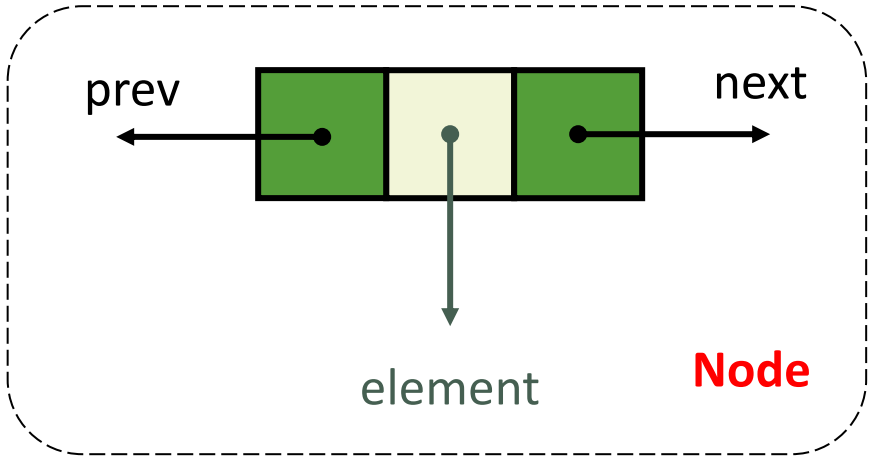


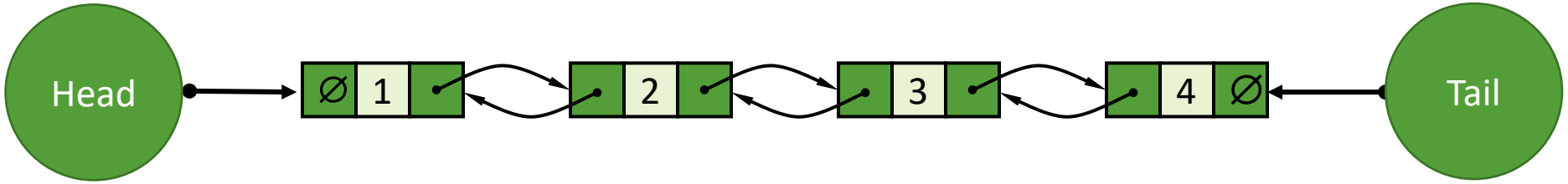
DOUBLY LINKED LISTS



DOUBLY LINKED LISTS (DLL)

- A doubly linked list can be traversed forward and backward
- Nodes store:
  - Element value
  - link to the previous node
  - link to the next node





DOUBLY LINKED LISTS (DLL) API

DLL

`Node Head;`
`Node Tail;`
`int size;`

`void insert(int x);`
`Node remove(int x);`
`Node search(int x);`
`boolean isEmpty();`
`String toString();`

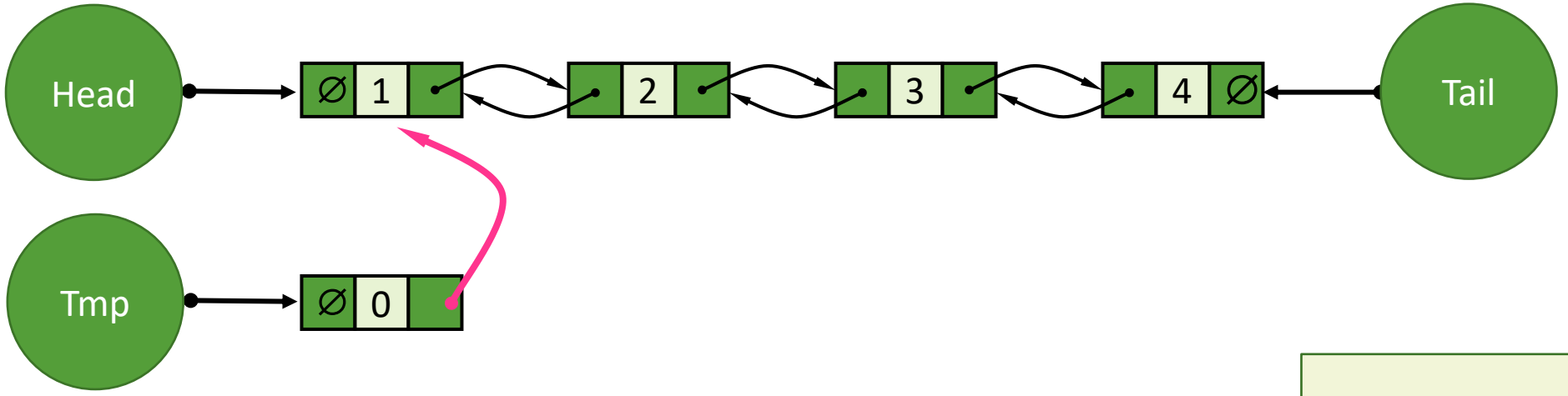
Node

`int val;`
`Node next;`
`Node prev;`

`Node();` `Node(,);`

DOUBLY LINKED LISTS (DLL)

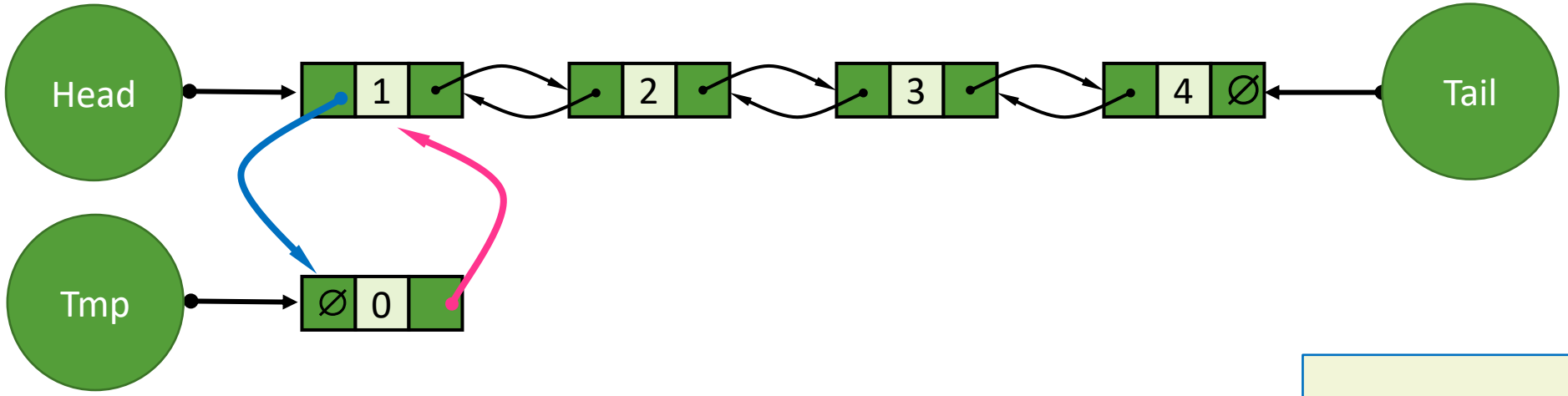
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
```

DOUBLY LINKED LISTS (DLL)

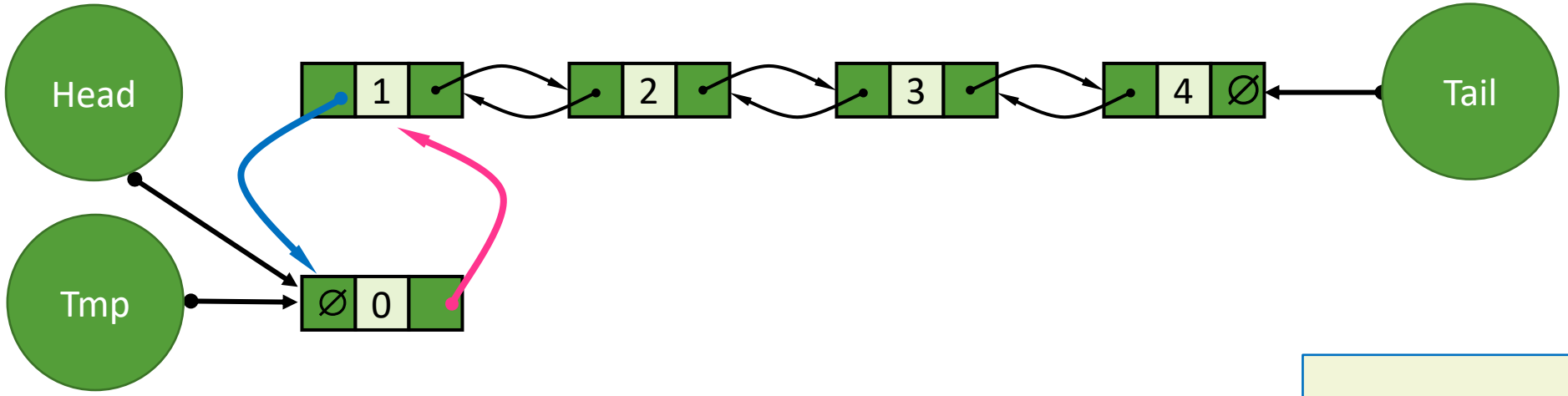
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
Head.prev = Tmp;
```

DOUBLY LINKED LISTS (DLL)

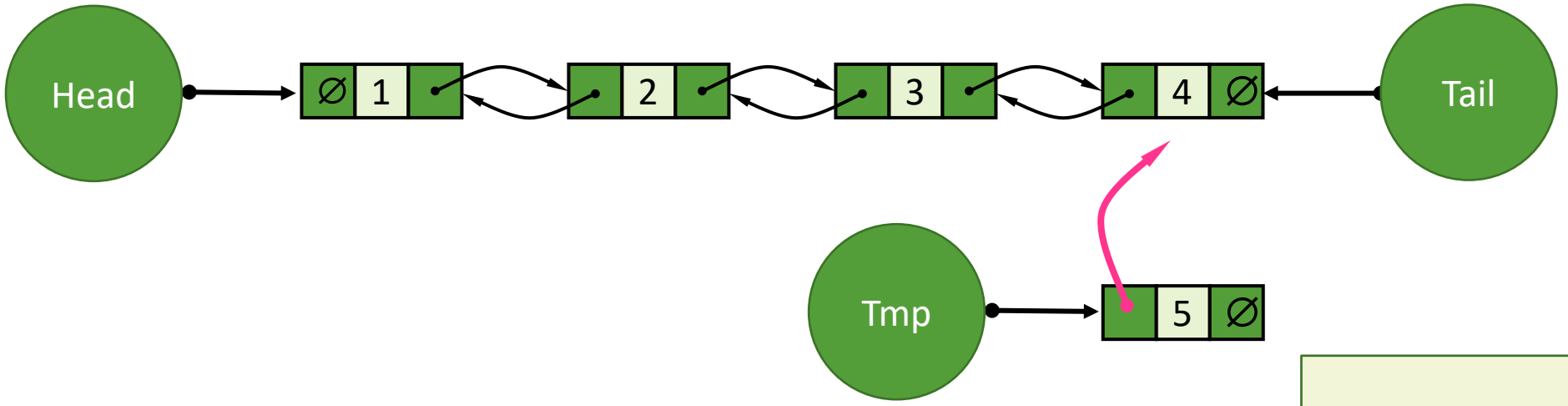
- Insert
  - **Beginning**, end, middle;




```
Node Tmp = new Node(0);
Tmp.next = Head;
Head.prev = Tmp;
Head = Tmp;
```

DOUBLY LINKED LISTS (DLL)

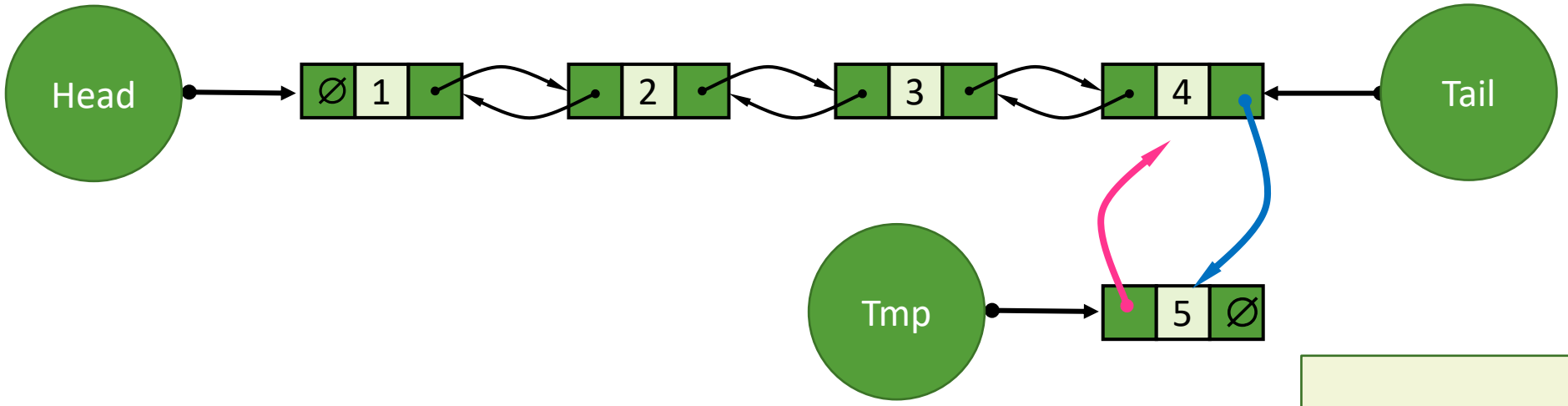
- Insert
  - Beginning, **end**, middle;




```
Node Tmp = new Node(5);
Tmp.prev = Tail;
```

DOUBLY LINKED LISTS (DLL)

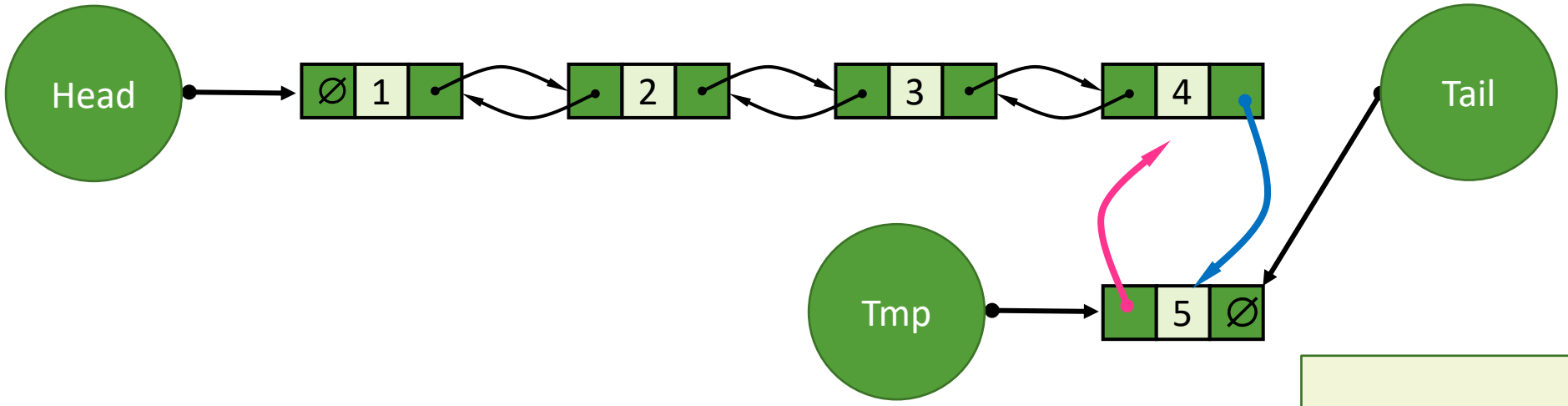
- Insert
  - Beginning, **end**, middle;




```
Node Tmp = new Node(5);
Tmp.prev = Tail;
Tail.next = Tmp;
```

DOUBLY LINKED LISTS (DLL)

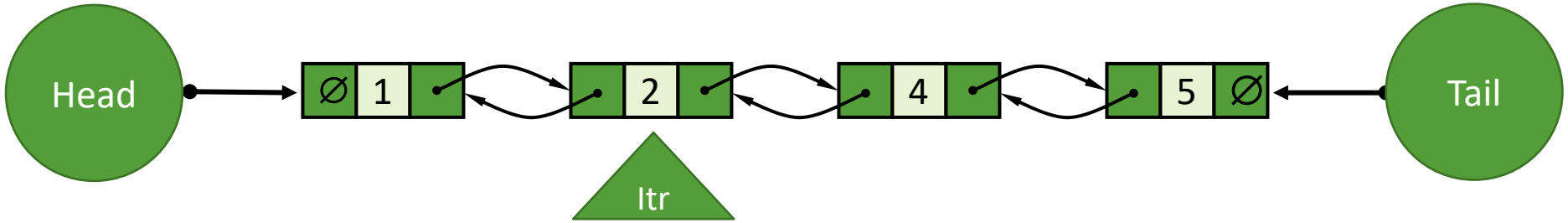
- Insert
  - Beginning, **end**, middle;

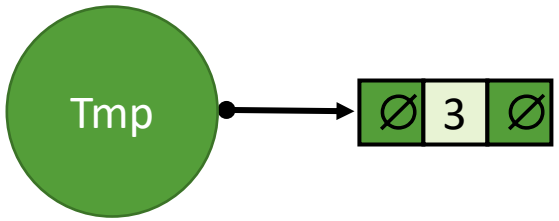



```
Node Tmp = new Node(5);
Tmp.prev = Tail;
Tail.next = Tmp;
Tail = Tmp;
```

DOUBLY LINKED LISTS (DLL)

- Insert
  - Beginning, end, **middle**;

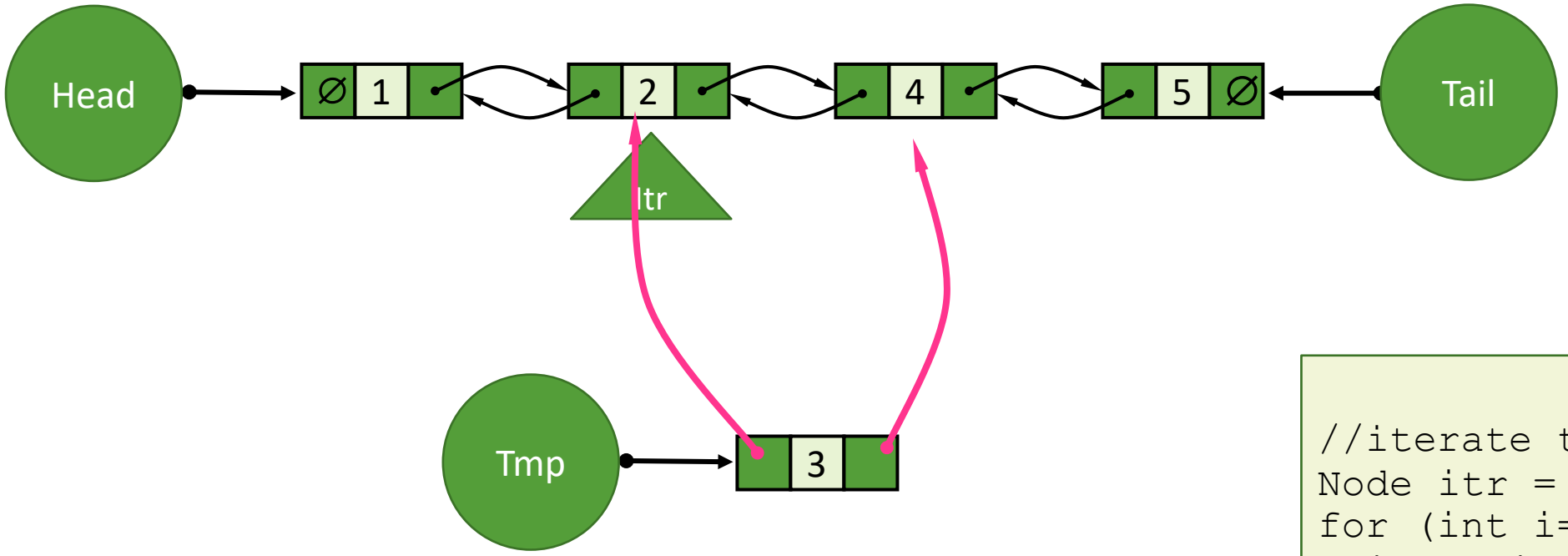





```
//iterate to the pos = 2;
Node itr = Head;
for (int i=1;i<pos;i++)
  itr = itr.next;

Node Tmp = new Node(3);
```

DOUBLY LINKED LISTS (DLL)

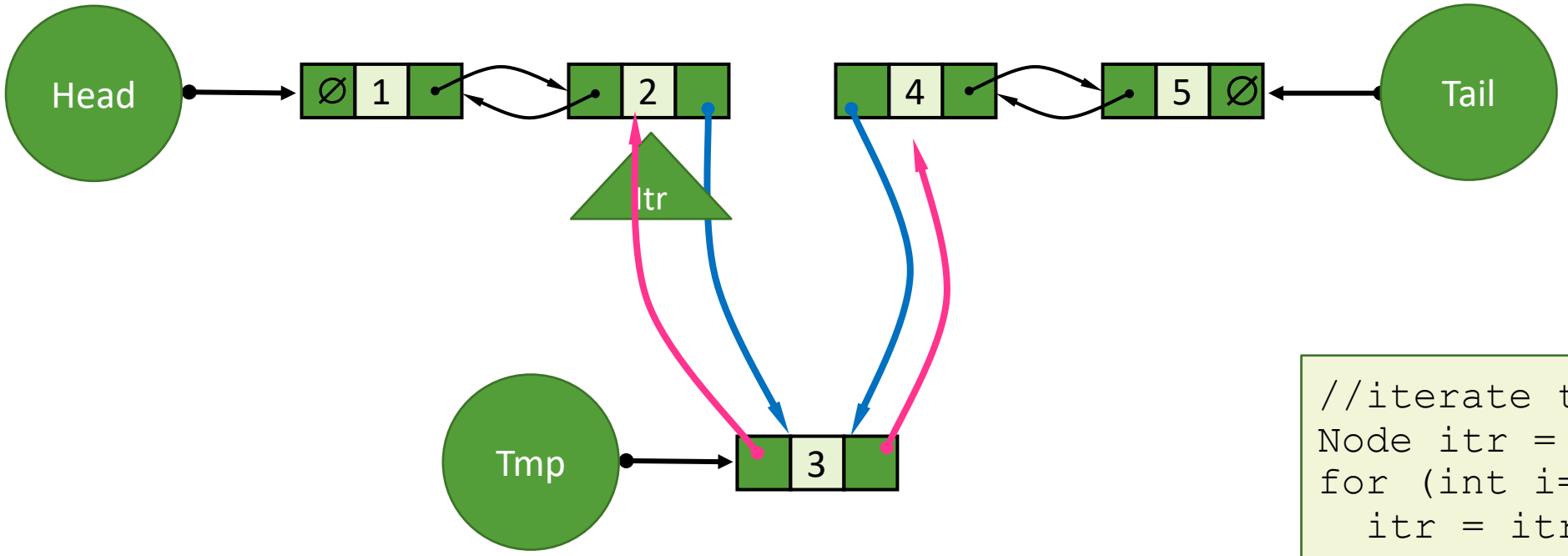
- Insert
  - Beginning, end, **middle**;




```
//iterate to the pos = 2;
Node itr = Head;
for (int i=1;i<pos;i++)
  itr = itr.next;

Node Tmp = new Node(3);
Tmp.next = Itr.next;
Tmp.prev = Itr;
```

DOUBLY LINKED LISTS (DLL)

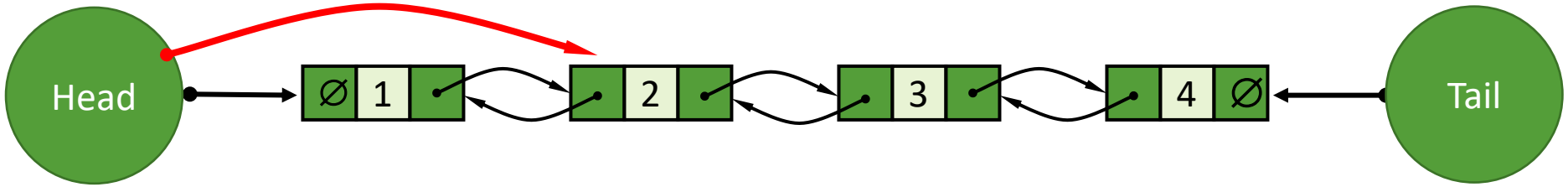
- Insert
  - Beginning, end, **middle**;




```
//iterate to the pos = 2;
Node itr = Head;
for (int i=1;i<pos;i++)
  itr = itr.next;

Node Tmp = new Node(3);
Tmp.next = Itr.next;
Tmp.prev = Itr;
Itr.next = Tmp;
Tmp.next.prev = Tmp;
```

DOUBLY LINKED LISTS (DLL)

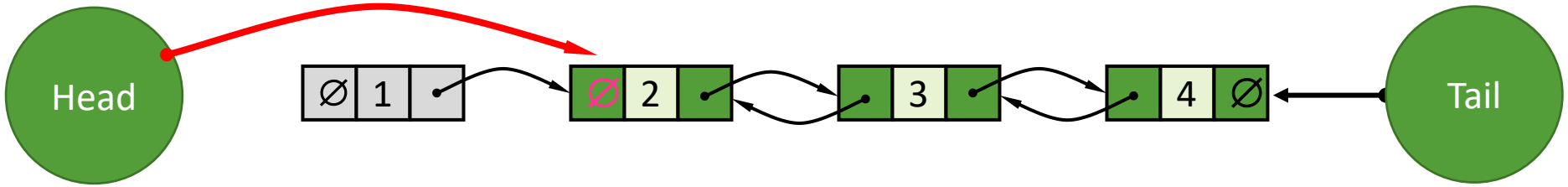
- Remove
  - **Beginning**, end, middle;




```
Head = Head.next;
```

DOUBLY LINKED LISTS (DLL)

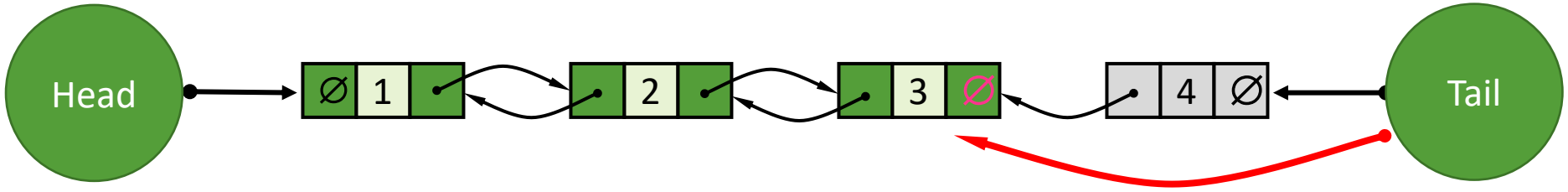
- Remove
  - **Beginning**, end, middle;




```
Head = Head.next;
Head.prev = null;
```

DOUBLY LINKED LISTS (DLL)

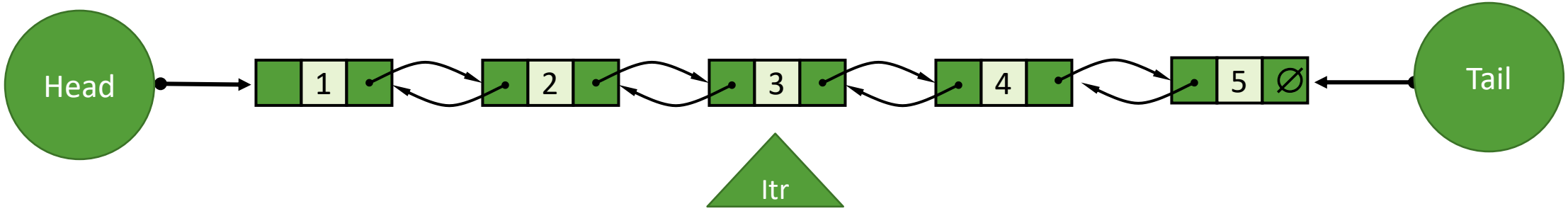
- Remove
  - Beginning, **end**, middle;




```
Tail = Tail.prev;
Tail.next = null;
```

DOUBLY LINKED LISTS (DLL)

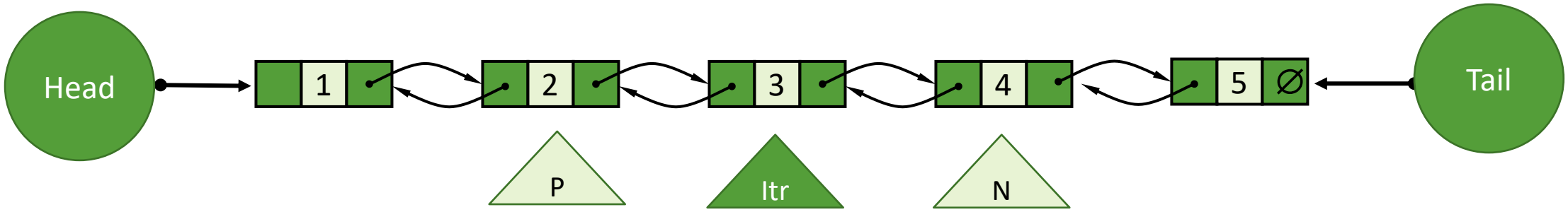
- Remove
  - Beginning, end, **middle**;




```
Node itr=Head;
//assume pos = 3
for(int i=1;i<pos;i++)
   itr = itr.next;
```

DOUBLY LINKED LISTS (DLL)

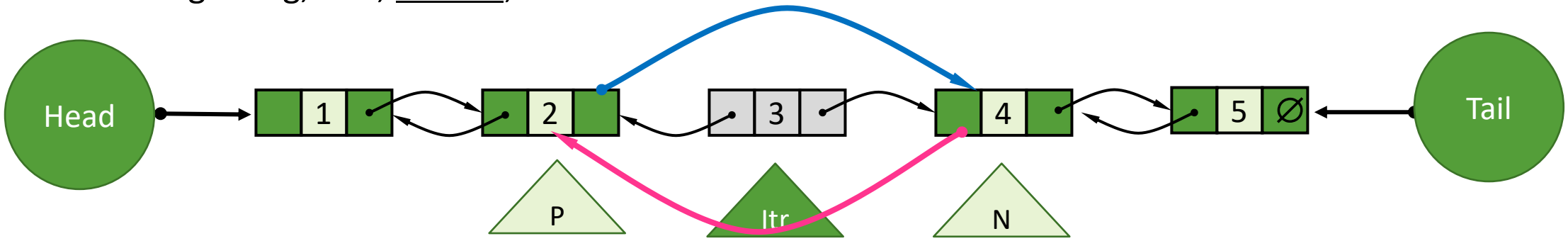
- Remove
  - Beginning, end, **middle**;




```
Node itr=Head;
//assume pos = 3
for(int i=1;i<pos;i++)
   itr = itr.next;
Node P = itr.prev;
Node N = itr.next;
```

DOUBLY LINKED LISTS (DLL)

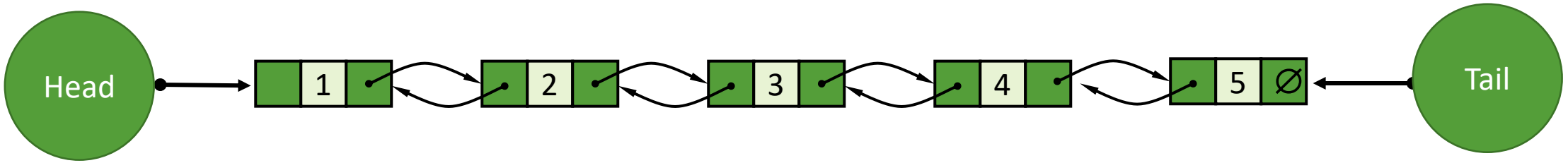
- Remove
  - Beginning, end, **middle**;



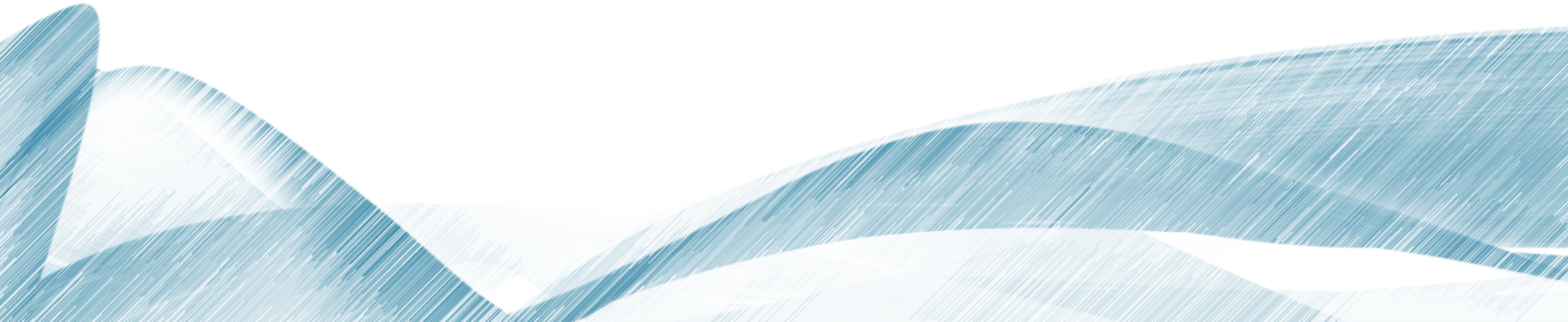

```
Node itr=Head;
//assume pos = 3
for(int i=1;i<pos;i++)
   itr = itr.next;
Node P = itr.prev;
Node N = itr.next;
P.next = N;
N.prev = P;
```

DOUBLY LINKED LISTS (DLL)

- Insert at the beginning **and End** is fast; middle is not efficient.
- Remove from the beginning **and End** is fast; middle is also not efficient.
- We can move the Iterator forward or backward.
- May reduce the time **by half** for searching/traversing if list is sorted.
- Faster than SLL?
- Takes more memory compared to SLL.

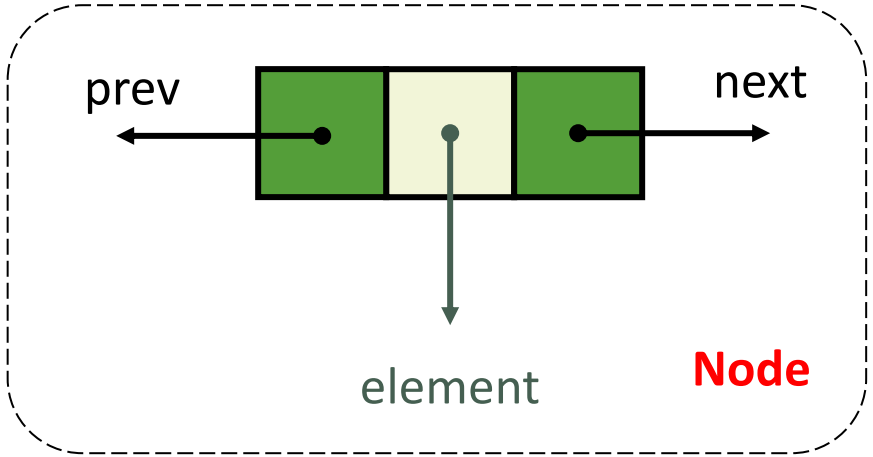


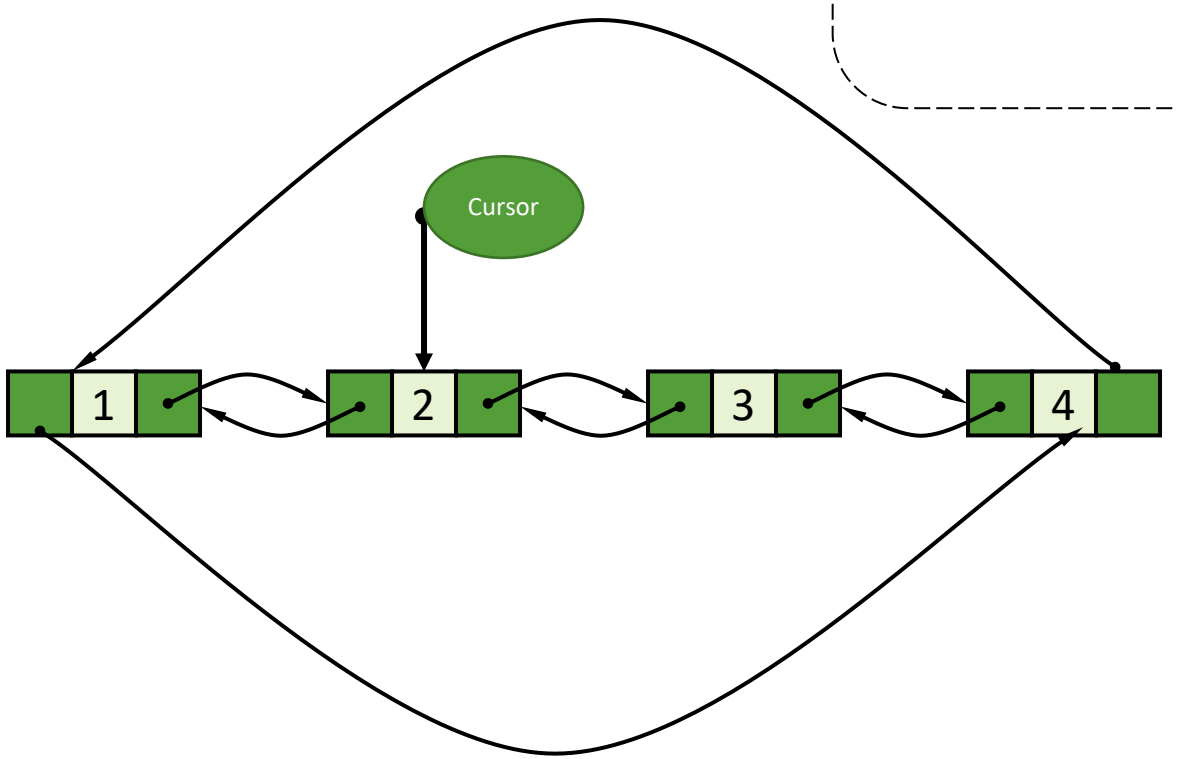
CIRCULAR LINKED LISTS



CIRCULAR LINKED LISTS (CLL)

- A circular linked list can be traversed forward or backward (or both). There is no Head or Tail, rather a moveable Cursor is used to access the list.
- Nodes store:
  - Element value
  - link to the previous node
  - link to the next node





CIRCULAR LINKED LISTS (CLL) API

CLL
`Node Cursor;` `int size;`

`void insert(int x);`
`Node remove(int x);`
`Node search(int x);`
`boolean isEmpty();`
`String toString();`

Node

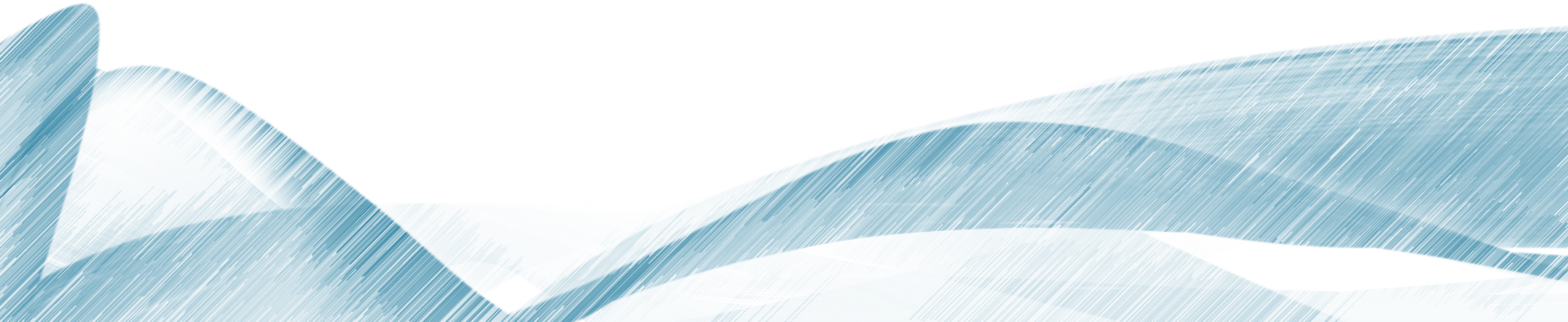
`int val;`
`Node next;`
`Node prev;`

`Node();` `Node(,);`

CIRCULAR LINKED LISTS (CLL)

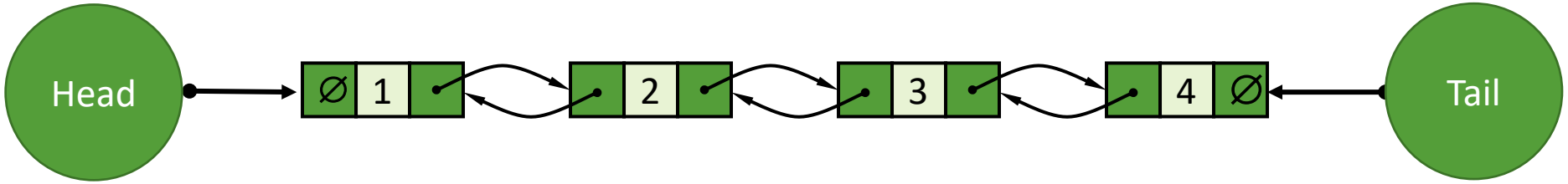
- Insert
  - Since there is no Beginning or end of the list; the only case that applies is middle. This is similar to Double Linked List middle insertion.
- Remove
  - Similar to removal from Doubly linked list middle case. Since there is no head or tail, the beginning and end case do not apply.

OTHER LINKED LISTS

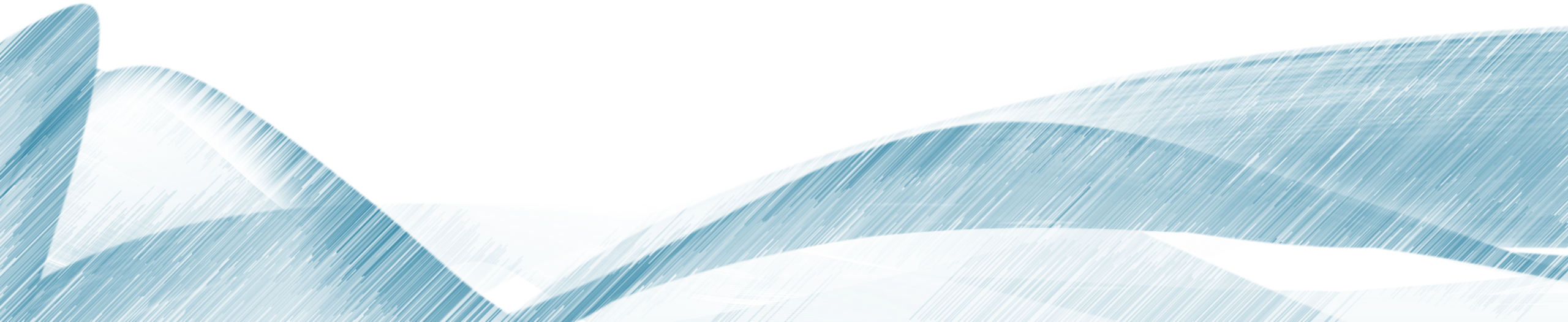


OTHER LINKED LISTS

- Positional Lists
  - Extension of Doubly Linked List
  - Maintains position of elements in the DLL
  - AddFirst, AddLast, AddBefore, AddAfter; RemoveBefore, RemoveAfter
- Sorted/Ordered Lists
  - AddOrdered, RemoveOrdered

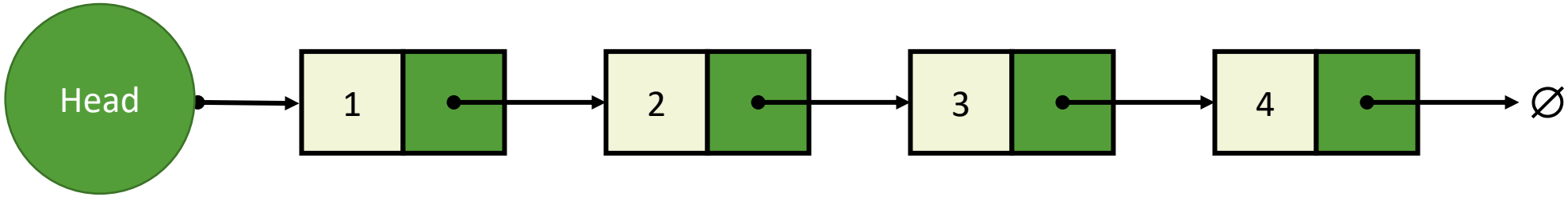


PERFORMANCE



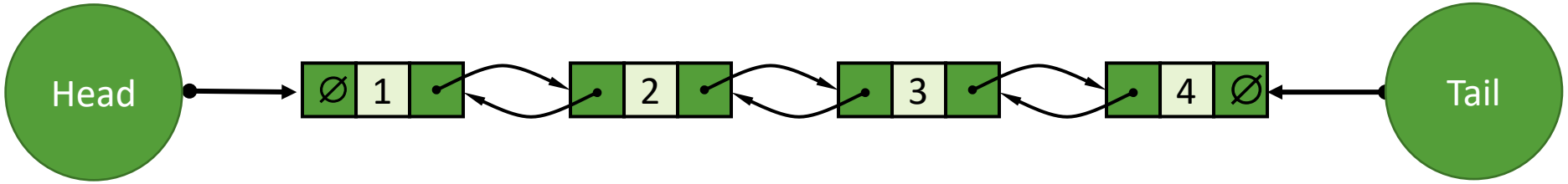
PERFORMANCE

- Singly Linked List (common operations)
  - Insert: [beginning] Constant time
  - Insert: [end] Linear time
  - Insert: [middle] Linear time
  - Remove: [beginning] Constant time
  - Remove: [end] Linear time
  - Remove: [middle] Linear time
  - Search: Linear time
- Overall: Takes Linear time and space



PERFORMANCE

- Doubly Linked List (common operations)
  - Insert: [beginning] Constant time
  - Insert: [end] Constant time
  - Insert: [middle] Linear time
  - Remove: [beginning] Constant time
  - Remove: [end] Constant time
  - Remove: [middle] Linear time
  - Search: Linear time
  - Cost: Additional space needed compared to SLL.
- Overall: Takes Linear time and space



PERFORMANCE

- Circular Linked List (common operations)
  - Insert: [middle] Linear time
  - Remove: [middle] Linear time
  - Search: Linear time
  - Cost: Additional space needed compared to SLL.
- Overall: Takes Linear time and space