Ch.04

Linked List

(kmitl) cs-department

Outline

- What a Linked List is
- Linked List Operations
  - Random access (retrieve/update)
  - Add/Insert
  - Search
  - Does order / unordered matter?
- More (than singly) linked list
  - Circular Linked List
  - Doubly Linked List
- Example interview questions

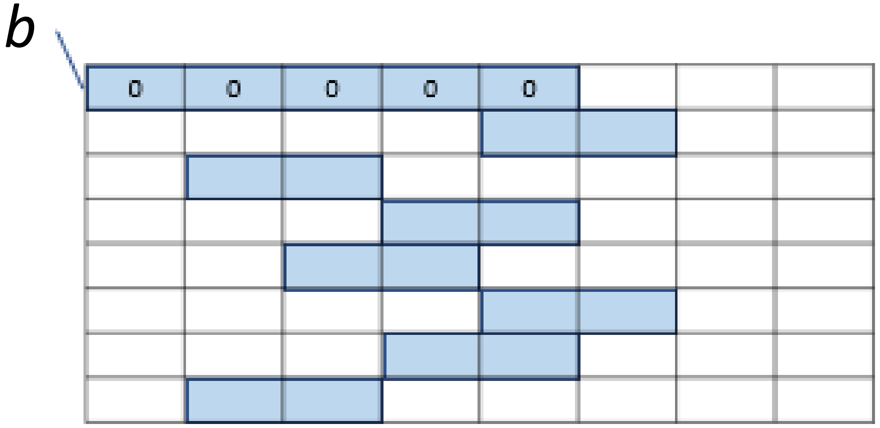
Linked List

var	addr
b	0

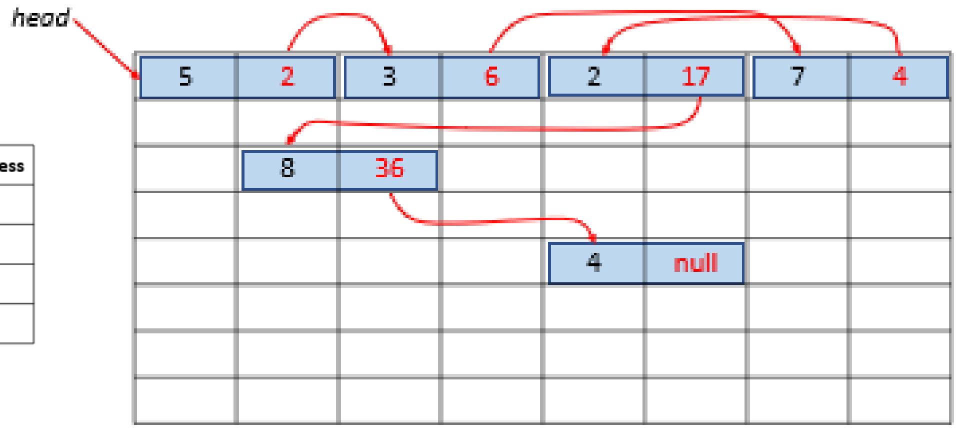
- Limitation of Array
  - We have 45 bytes of memory but we **cannot** allocate an array of size 10.
  - Now, how can we make a collection of data of size 10?
- Linked List
- A data structure where a unit of data contain the data itself and a reference to the next data.
  - Made possible by class in JAVA

var	addr
b	0

revisited



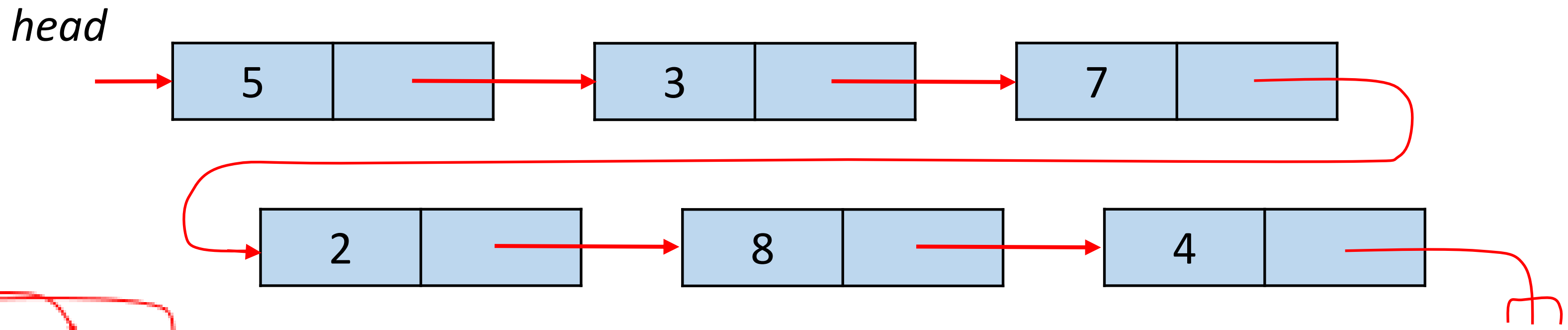
var	address
head	0

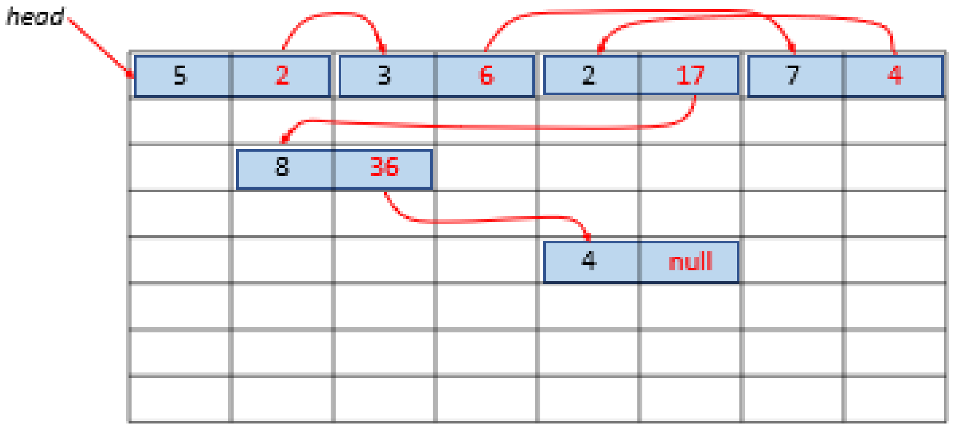


Visualizing Linked List

- Each set of data and address of the next data is called **a** **node**.
- Assume we want to store the following data:
  5 3 7 2 8 4
- Only variable required is **head**

```
public class Node {
     int data;
     Node next;
}
```





var	address
head	0

(List) Chained Object

```java
class Province {
    String name;
    Province next;
    Province(String n) {
        name = n;
    }
}
```

```java
public static void main(String[] args) {
    //from Bangkok to Petchburi
    Province bangkok = new Province("Bangkok");
    Province sakorn = new Province("Samutsakorn");
    Province songkram = new Province("Samutsongkram");
    bangkok.next = sakorn;
    sakorn.next = songkram;
    songkram.next = new Province("Petchburi");

    Province curCity = bangkok;
    System.out.println("start (" + curCity.name + ")");
    while (curCity.next != null) {
        curCity = curCity.next;
        System.out.println("visiting " + curCity.name);
    } //patch.next is null
    System.out.println("enjoy!");
}
```

```
start (Bangkok)
visiting Samutsakorn
visiting Samutsongkram
visiting Petchburi
enjoy!
```

MyLinkedList.java

LinkedListTester.java

```
public class LinkedListTester {
   public static void main(String[] args) {
      MyLinkedList mList = new MyLinkedList();

      // your code here

      System.out.println(mList.toString());
   }
}
```

MyLinkedList.java

```
public class MyLinkedList {
   public class Node {
      int data;
      Node next;
      public Node(int d) {
         data = d;
      }
   }
   Node head = null;

   // your code here

   public String toString() {
      StringBuffer sb = new StringBuffer("head ");
      Node p = head;
      while(p!=null) {
         sb.append("--> [");
         sb.append(p.data);
         sb.append("] ");
         p = p.next;
      }
      sb.append("-> null");
      return new String(sb);
   }
}
```

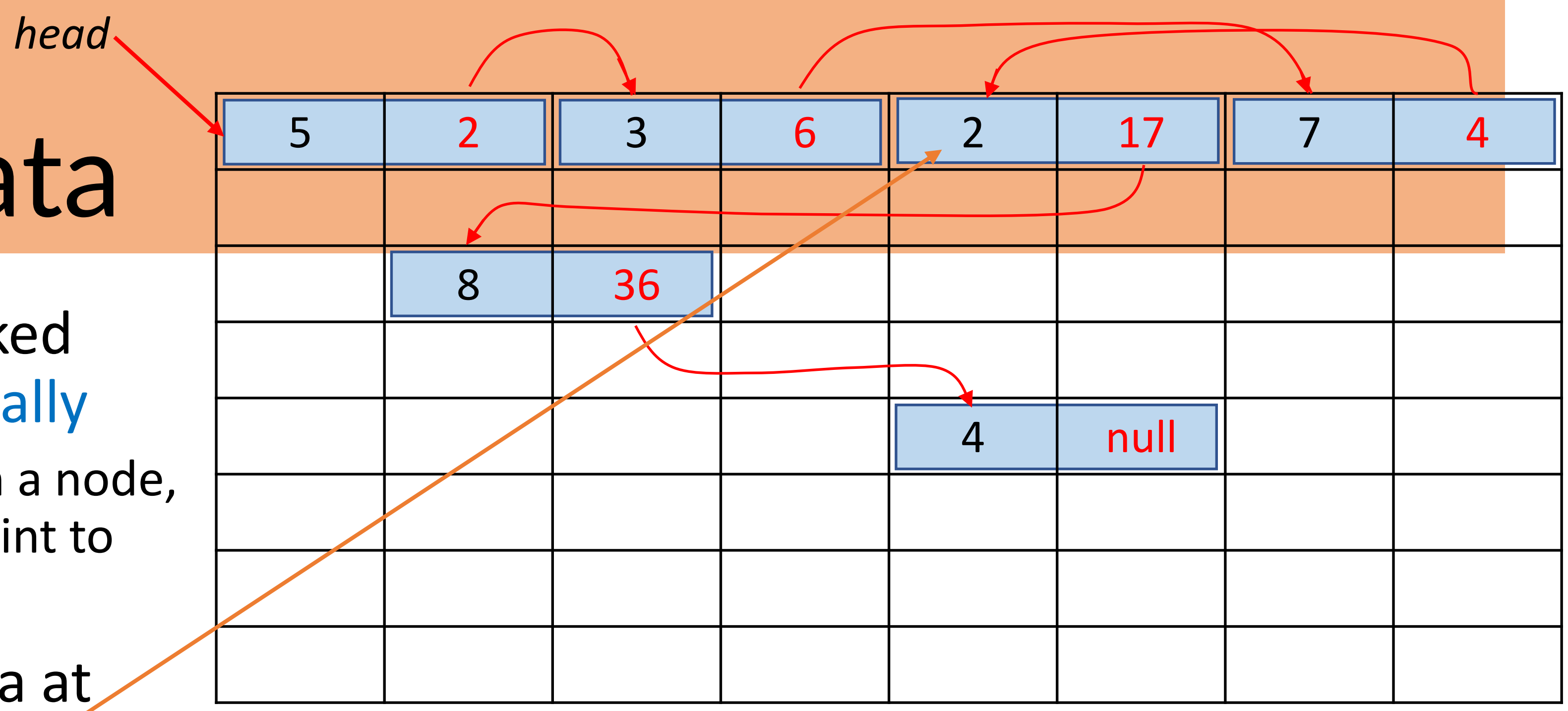
JAVA Inner class
Refer to as MyLinkedList.Node

Linked List Operations

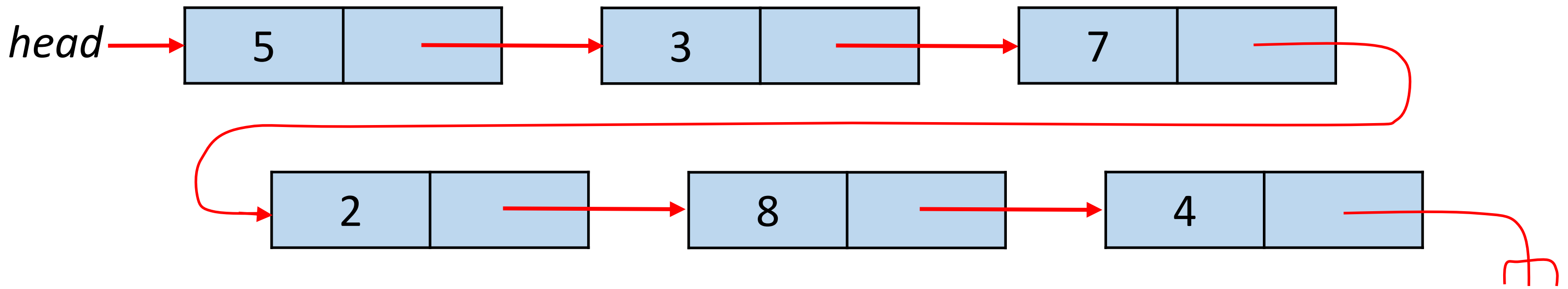
- We are to analyze the following Linked List operations
  - Random access (retrieve/update)
  - Add/Insert
  - Search
  - Delete
  - Does ordered / unordered matter?

- Important remark -> classical linked list does not involve index, yet admittedly we understand the semantic of .get(index)

Accessing Linked List Data



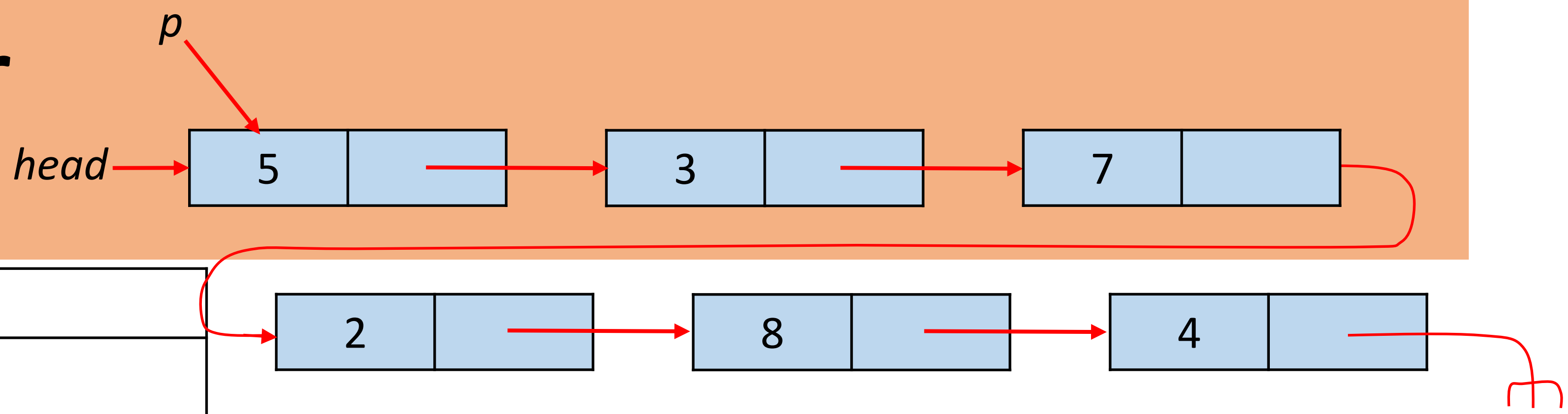
- Random access data in linked list must be done sequentially
  - Before we do anything with a node, we must have a variable point to that node, typically call **p**.
- For example, to access data at index=3, we must go from 0 to 2 to 6 to 4 to access 2
- Thus, getAt()'s big-O is O(1), O(n), and O(n) respectively



var	address
head	0

Getter/Setter

getAt()/setAt()




```
getAt() & setAt()
```

```
public int getAt(int i) {
    Node p = head;
    while(i>0) {
        p = p.next;
        i--;
    }
    return p.data;
}

public void setAt(int d, int i) {
    Node p = head;
    while(i>0) {
        p = p.next;
        i--;
    }
    p.data = d;
}
```

- Both methods are O(1), O(n), O(n)
- What happen when head is null?
- We cannot test these methods right now.
    - Let wait until we implement **add()**
- Typically, we will not use linked list this way.

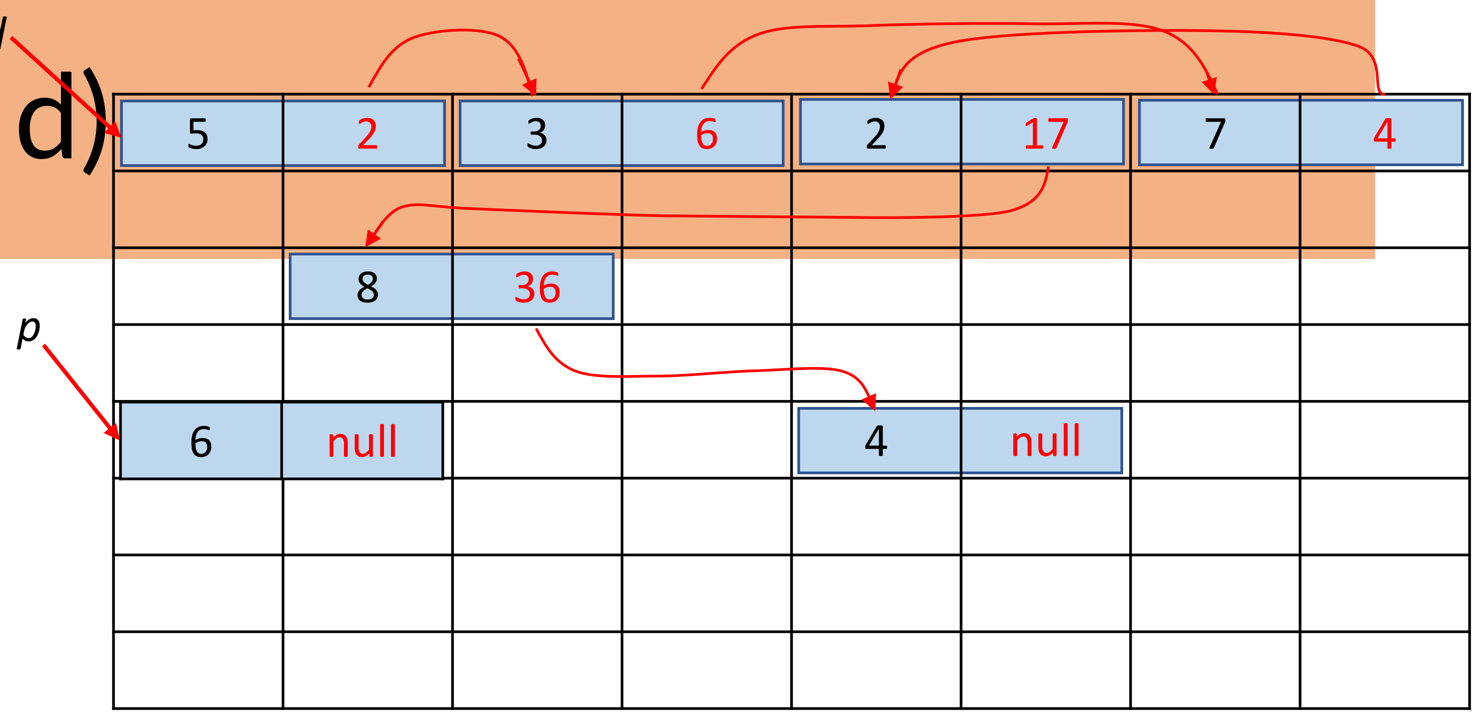
Add opeartion

- Where should we put a new data?
  - Two choices: at the head, or at the tail.
  - One of them is O(1), the other is O(n)
- The correct choice is at the head
  - because to get to the tail, we need O(n)
- The steps are simple:
  - Create a new **node**, put the **data** in
  - Point **next** of that node to **head**
  - Point **head** to that **node**

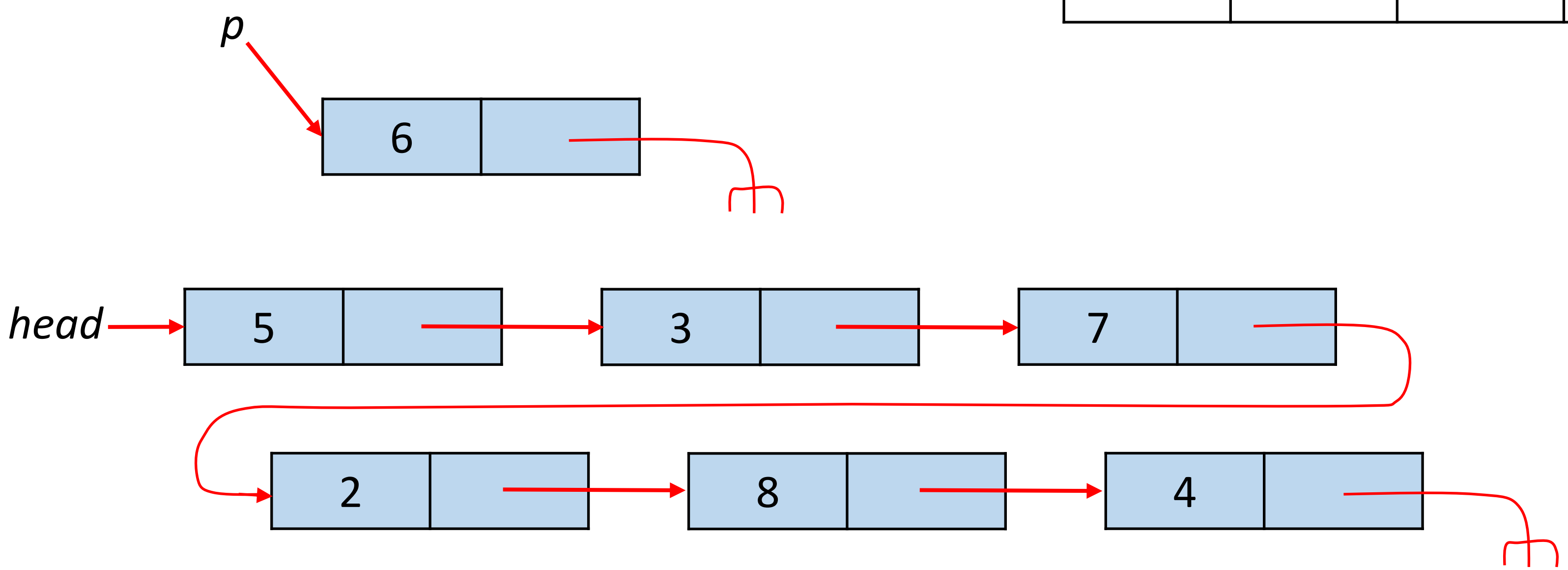
- Let's implement it

```
public void add(int d) {
    Node p = new Node(d);
    p.next = head;
    head = p;
}
```

Method add(int d)



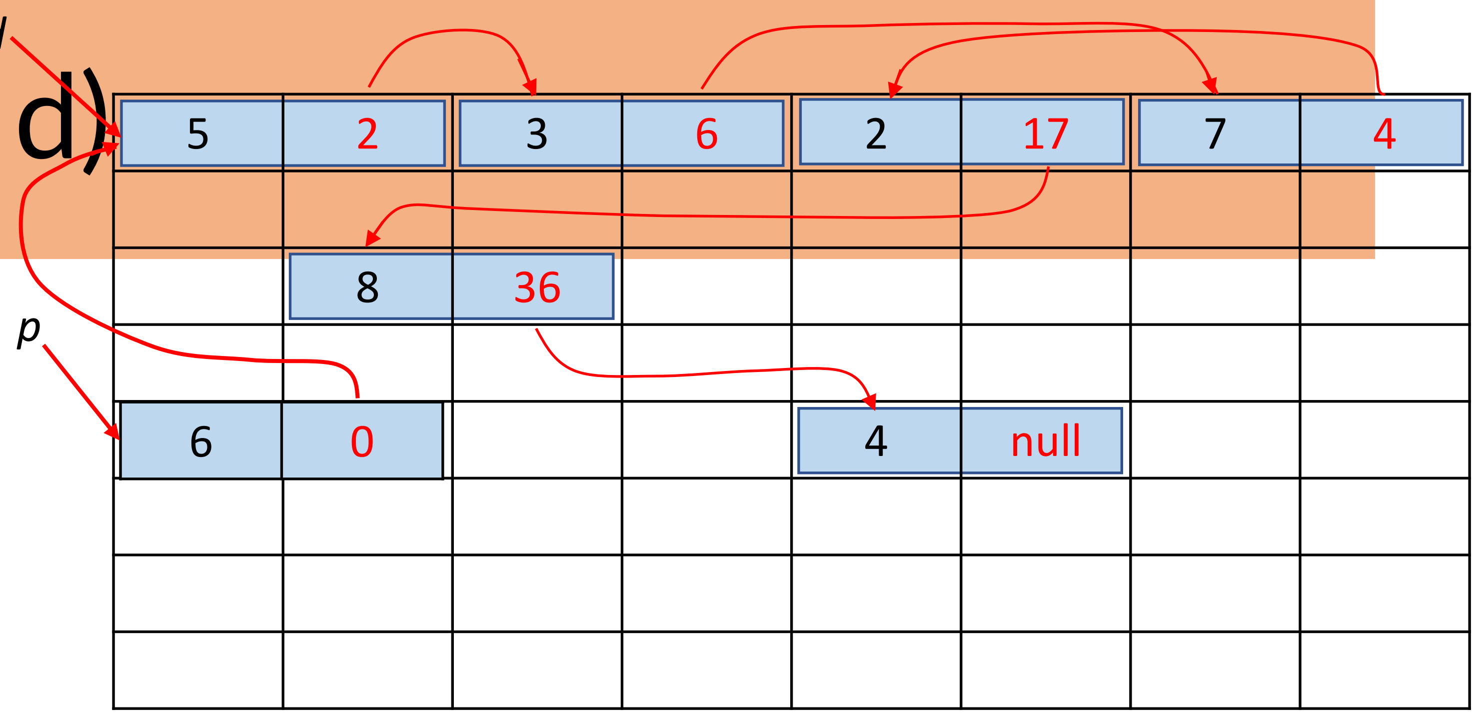

```
public void add(int d) {
    Node p = new Node(d);
    p.next = head;
    head = p;
}
```

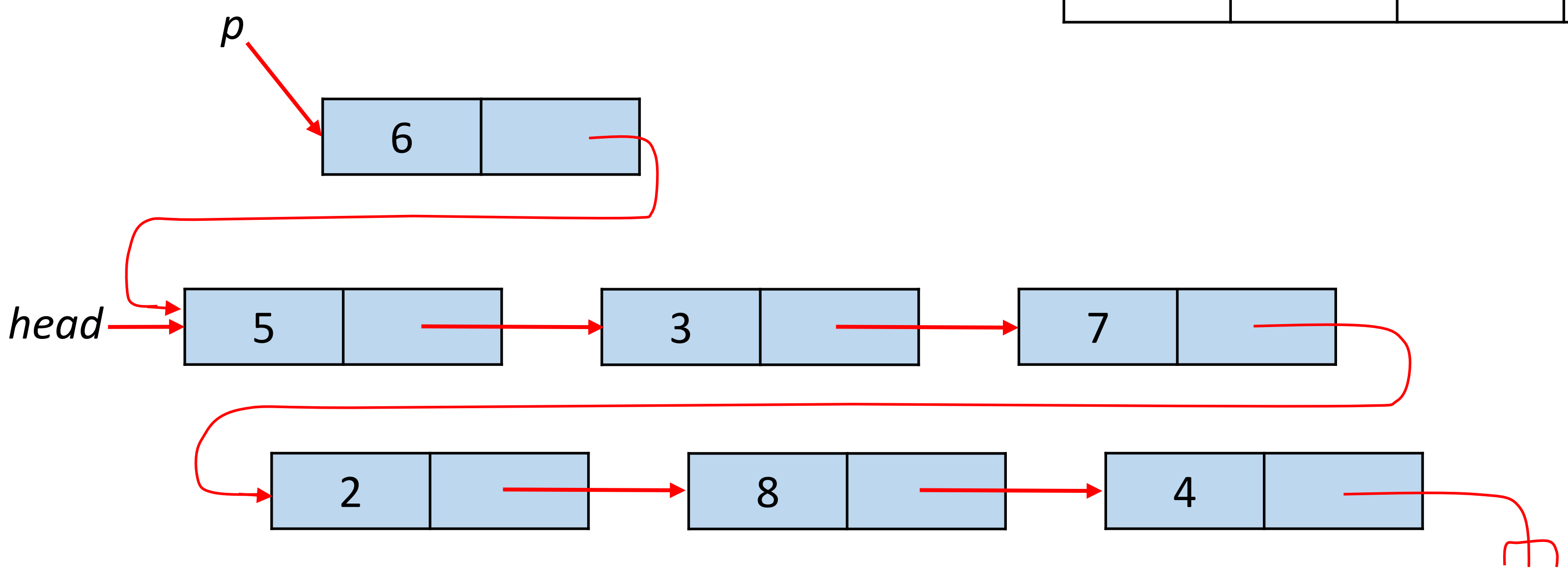


var	address
head	0
p	32

Method add(int d)

```
public void add(int d) {
     Node p = new Node(d);
     p.next = head;
     head = p;
}
```

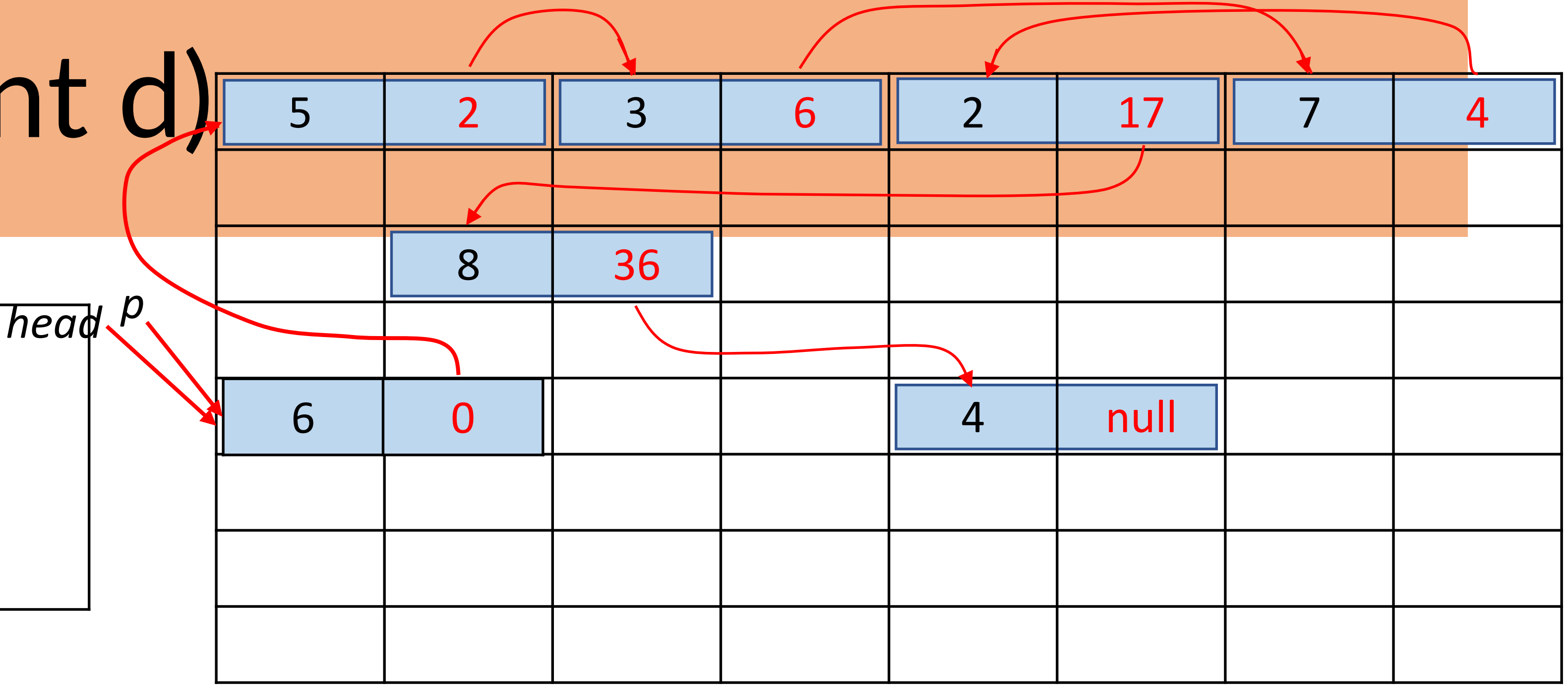


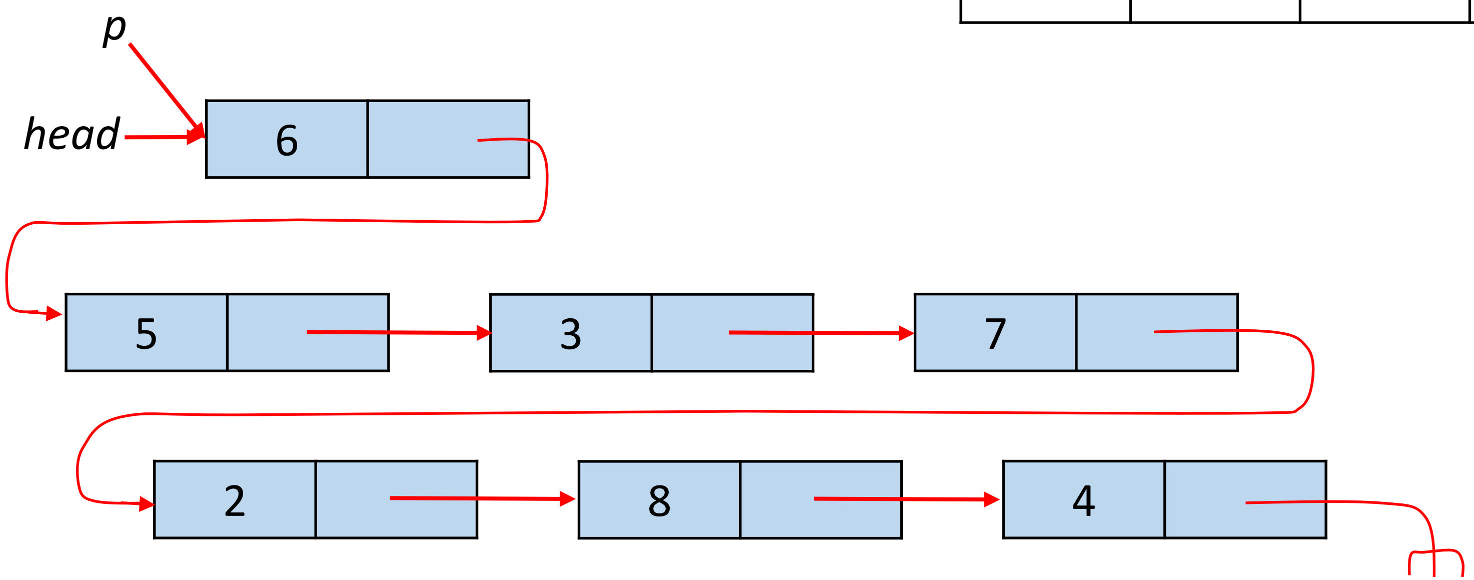


var	address
head	0
p	32

Method add(int d)

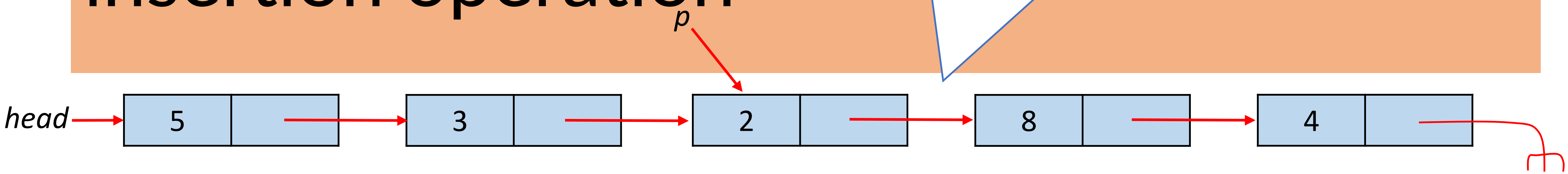
```
public void add(int d) {
     Node p = new Node(d);
     p.next = head;
     head = p;
}
```





var	address
head	32
p	32

insertion operation

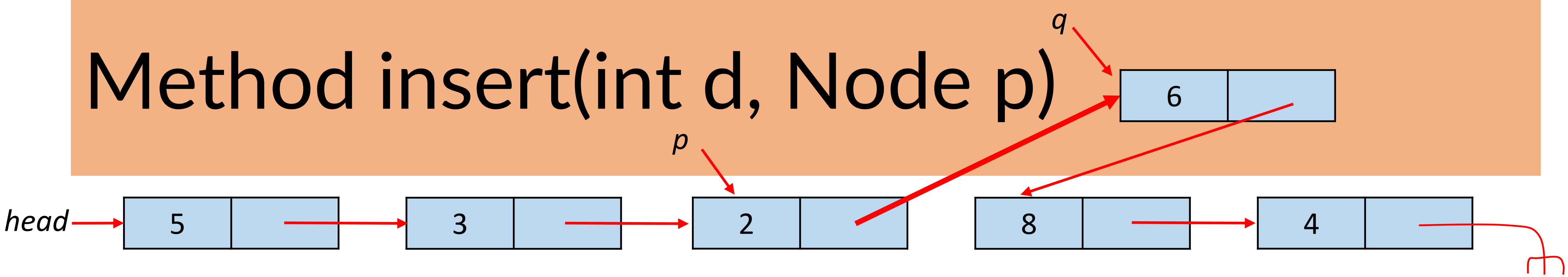


- To insert, we must have a pointer point to the node before the position to insert
  - For example, to insert between 2 and 8 we need a pointer point at 2.
- Steps are:
  - Create a new node q with the new data
  - Point next of q to next of p
  - Point next of p to q

- The implementation is simple:

```
public void insert(int d, Node p) {
    Node q = new Node(d);
    q.next = p.next;
    p.next = q;
}
```

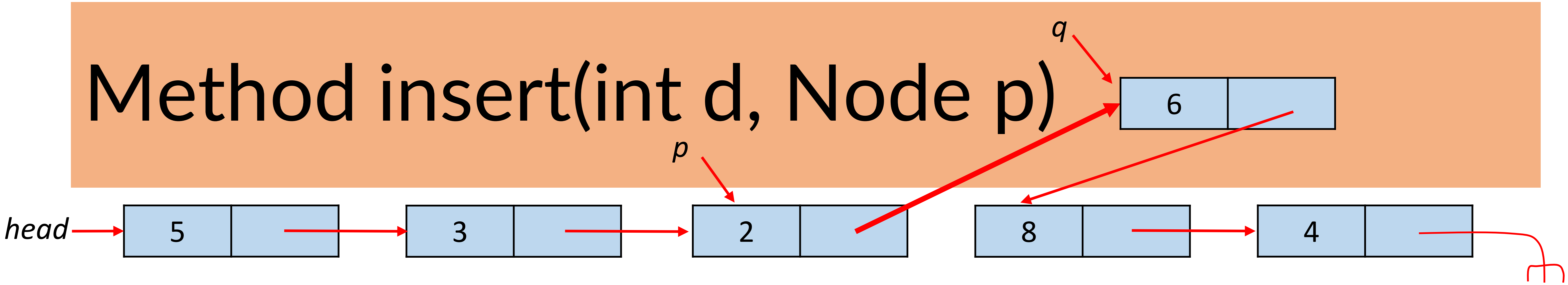
Method insert(int d, Node p)



- To insert, we must have a pointer point to the node before the position to insert
  - For example, to insert between 2 and 8 we need a pointer point at 2.
    - Cost of locating p = ?
- Steps are:
  - **Create a new node q with the new data**
    - Big-O is O(1)
  - Point next of q to next of p
  - Point next of p to q

```
public void insert(int d, Node p) {
    Node q = new Node(d);
    q.next = p.next;
    p.next = q;
}
```

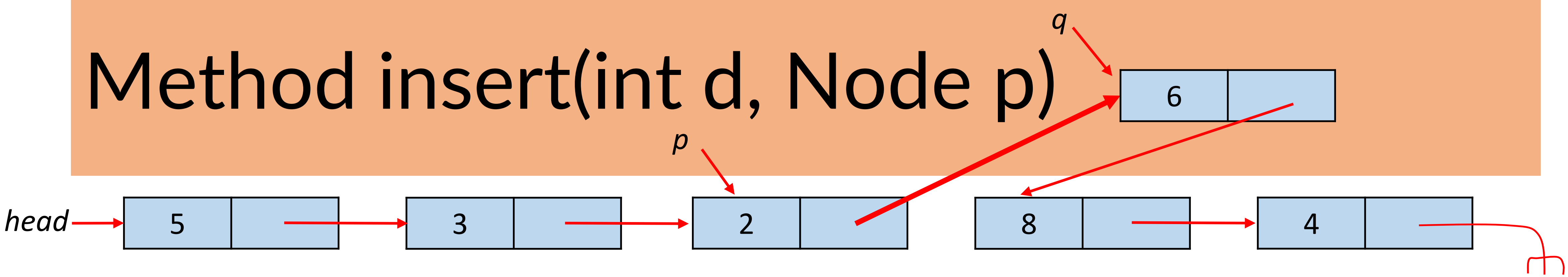
Method insert(int d, Node p)



- To insert, we must have a pointer point to the node before the position to insert
  - For example, to insert between 2 and 8 we need a pointer point at 2.
    - Cost of locating p = ?
- Steps are:
  - Create a new node q with the new data
    - Big-O is O(1)
  - **Point next of q to next of p**
  - Point next of p to q

```
public void insert(int d, Node p) {
    Node q = new Node(d);
    q.next = p.next;
    p.next = q;
}
```

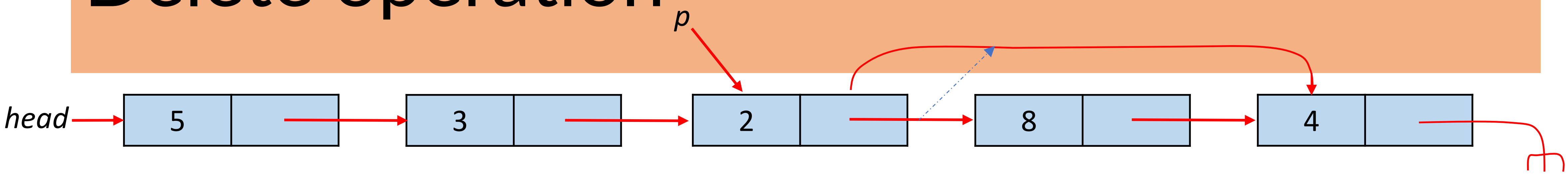
Method insert(int d, Node p)



- To insert, we must have a pointer point to the node before the position to insert
  - For example, to insert between 2 and 8 we need a pointer point at 2.
    - Cost of locating p = ?
- Steps are:
  - Create a new node q with the new data
    - Big-O is O(1)
  - Point next of q to next of p
  - **Point next of p to q**

```
public void insert(int d, Node p) {
    Node q = new Node(d);
    q.next = p.next;
    p.next = q;
}
```

Delete operation



- To delete, we must have a pointer point to the node before the position to delete
  - For example, to delete 8 we need a pointer point at 2.
- Steps is:
  - Point next of p to next of next of p
  - Big-O is O(1)
- The left-over data node will be handled by the garbage collector.

- The implementation is simple:

```
public void delete(Node p) {
    p.next = p.next.next;
}
```

Search operation

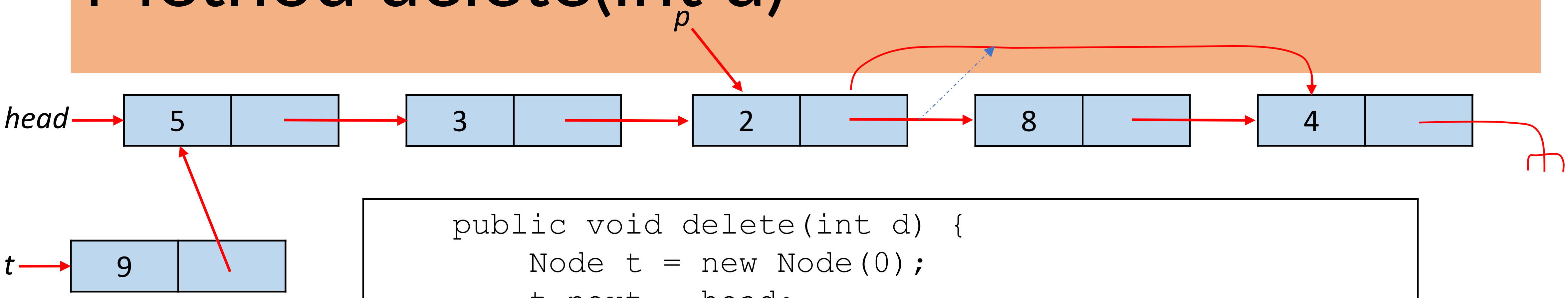
- Binary search is NOT possible due to random access is O(n) in average.
- Best we can do is sequential search which is O(n) in average.
- The implementation is as follow:
- Note that, in main(), we must call find() like this:

```
MyLinkedList.Node p = mList.find(4);
```

```
public Node find(int d) {
    Node p = head;
    while(p!=null) {
        if(p.data==d) return p;
        p = p.next;
    }
    return null;
}
```

The result of `find()` cannot be use as an input of `insert()` or `delete()`!

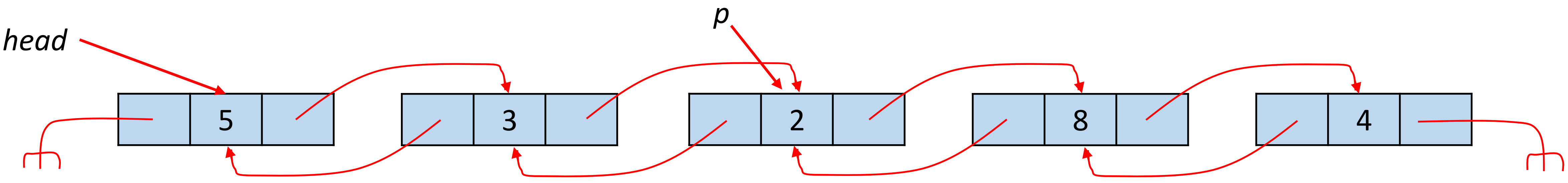
Method delete(int d)




```
public void delete(int d) {
    Node t = new Node(0);
    t.next = head;
    Node p = t;
    while( (p.next!=null) && (p.next.data!=d) ) {
        p = p.next;
    }
    if(p.next!=null) {
        p.next = p.next.next;
    }
    head = t.next;
}
```

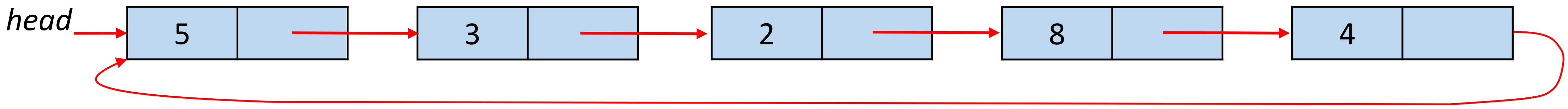
More on Linked List
Doubly Linked List

- Easy to go back and forward
  - Used to implement undo/redo functionality
- Easy to delete at the spot.
  - To delete at p, given p.previous and p.next
    - p.next.previous = p.previous
    - p.previous.next = p.next
    - //Now, no one point at p
  - Time complexity is still O(1).



More on Linked List
Circular Linked List

- Used in round-robin (time-sharing) mechanism.
- For deque
  - double end operations – push_front(), push_back(), pop_front(), pop_back()



More on Linked List
Circular Linked List

Doubly Linked List

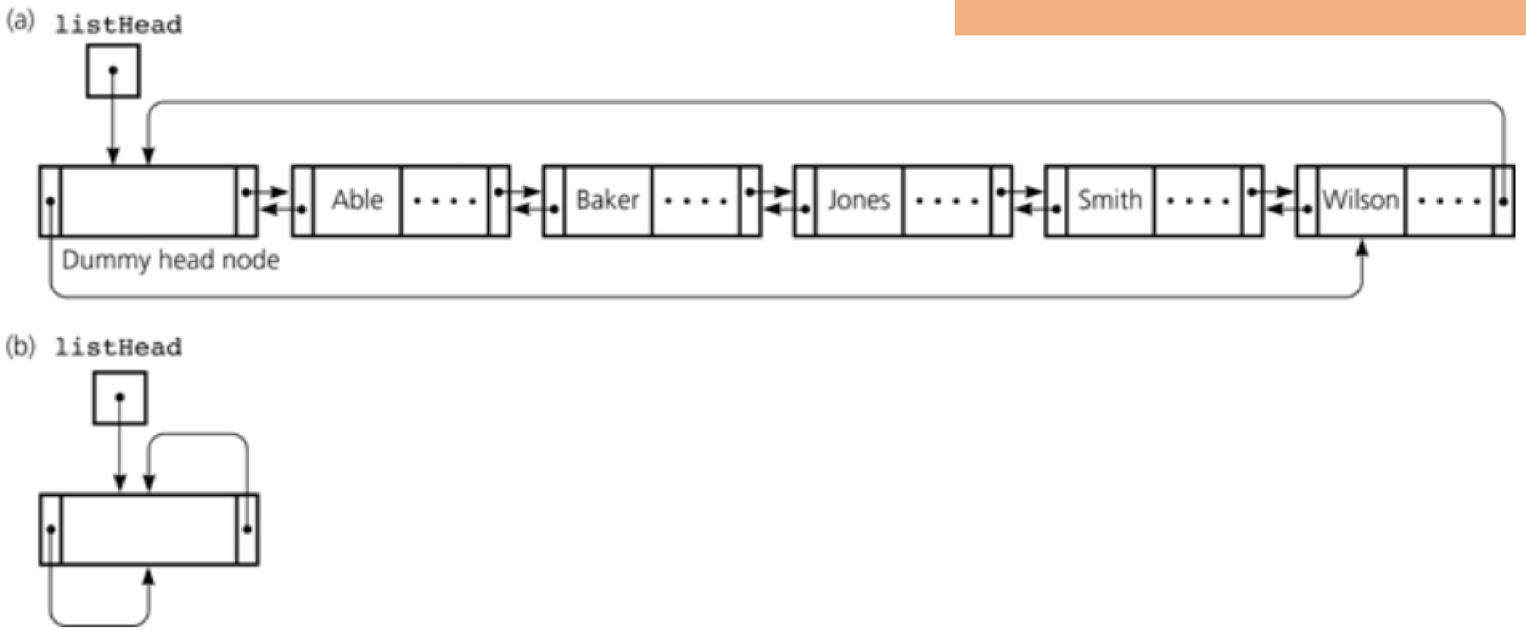


Figure 5-27

a) A circular doubly linked list with a dummy head node; b) an empty list with a dummy head node

Lecture 15 Chapter 5 -14

- Cleaner Implementation

https://www.cs.auckland.ac.nz/courses/compsci105ssc/archive/2007/lectures/lecture15.pdf

Recap

- Linked List is a data structure where we only keep address of the first data node and each data node keep address of the next one.

Methods	Best case	Worst case	Average case
Add into a linked list	$O(1)$	$O(1)$	$O(1)$
Insert into a linked list	$O(1)$	$O(1)$	$O(1)$
Find in a linked list	$O(1)$	$O(n)$	$O(n)$
Delete from a linked list	$O(1)$	$O(1)$	$O(1)$

- We can set/get at a specific index in O(n) times.
  - Other data structure could remedy this poor characteristic.
- We can add a new data using O(1) time if we insert at the front.
- Order or unordered linked list make no difference in its operations.
  - Exclude the cost of proper place for p
- We can use temporary head to help with delete (delete(int d))
  - Can be done similarly in insertion

Example challenge

- Given a head node referring to the head node of the list, return a deep copy of the second half.
  - Input: head -> 1 -> 2 -> 3 -> 4 -> 5 -> null
  - Output: copyHead -> 3 -> 4 -> 5 -> null

```
public class Solution {
  public static Node returnMid(Node head) {
    if (head == null) return null;

    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
       slow = slow.next;    // move every time
       fast = fast.next.next;
    }
```

```
      Node original = slow;
      Node copyHead = null, copyTail = null;

      while (original != null) {
         Node newNode = new Node(original.data);

         if (copyHead == null) {
             copyHead = newNode;
             copyTail = newNode;
         } else {
             copyTail.next = newNode;
             copyTail = newNode;
         }

         original = original.next;
      }
     // Head of the deep-copied second half
     return copyHead;
}
```