Data Structures and Algorithms (02-24-00108)

Course Presentation
by
Dr. Adel A. El-Zoghabi

Professor of Computer Science & Information Technology
Faculty of Computer and Data Science
Alexandria University

email: Adel.Elzoghby@alexu.edu.eg

Course Outline

Prerequisite: **Programming I (02-24-00105)**

Topics:

Basic Concepts of Data Structures	(Part 1: 1)
Dynamic Memory Allocations	(Part 2: 3)
Hashing Techniques	(Part 3: 1)
Stacks, Queues and Priority Queues	(Part 4: 1)
Binary Trees and Recursion	(Part 5: 2)
Complexity Analysis of Algorithms	(Part 6: 1)
Searching and Sorting Techniques	(Part 7: 2)
Heap Trees	(Part 8: 1)

Grading Policy:

Assignments (30%): Assigned Week 3, 6, 9 and Due Week 5, 8, 11
Midterm Exam (20%): Week 7,
Final Exam (50%): University Schedule

Part # 1

- **Basic Concepts of Data Structures**
  1. Introduction,
  2. Handling Problems,
  3. Abstraction of Real Problems,
  4. Abstract Data Types

Introduction

- The study of data structures, a fundamental component of IT education, serves as the foundation upon which many other fields are built,

- Some knowledge of data structures is a must for students who wish to do work in design, implementation, testing, or maintenance of software,

Introduction

- In this course, data structures are presented in the object-oriented setting in accordance with the current design and implementation paradigm,

- The Java language is widespread in industry and is also useful and natural in introducing data structures

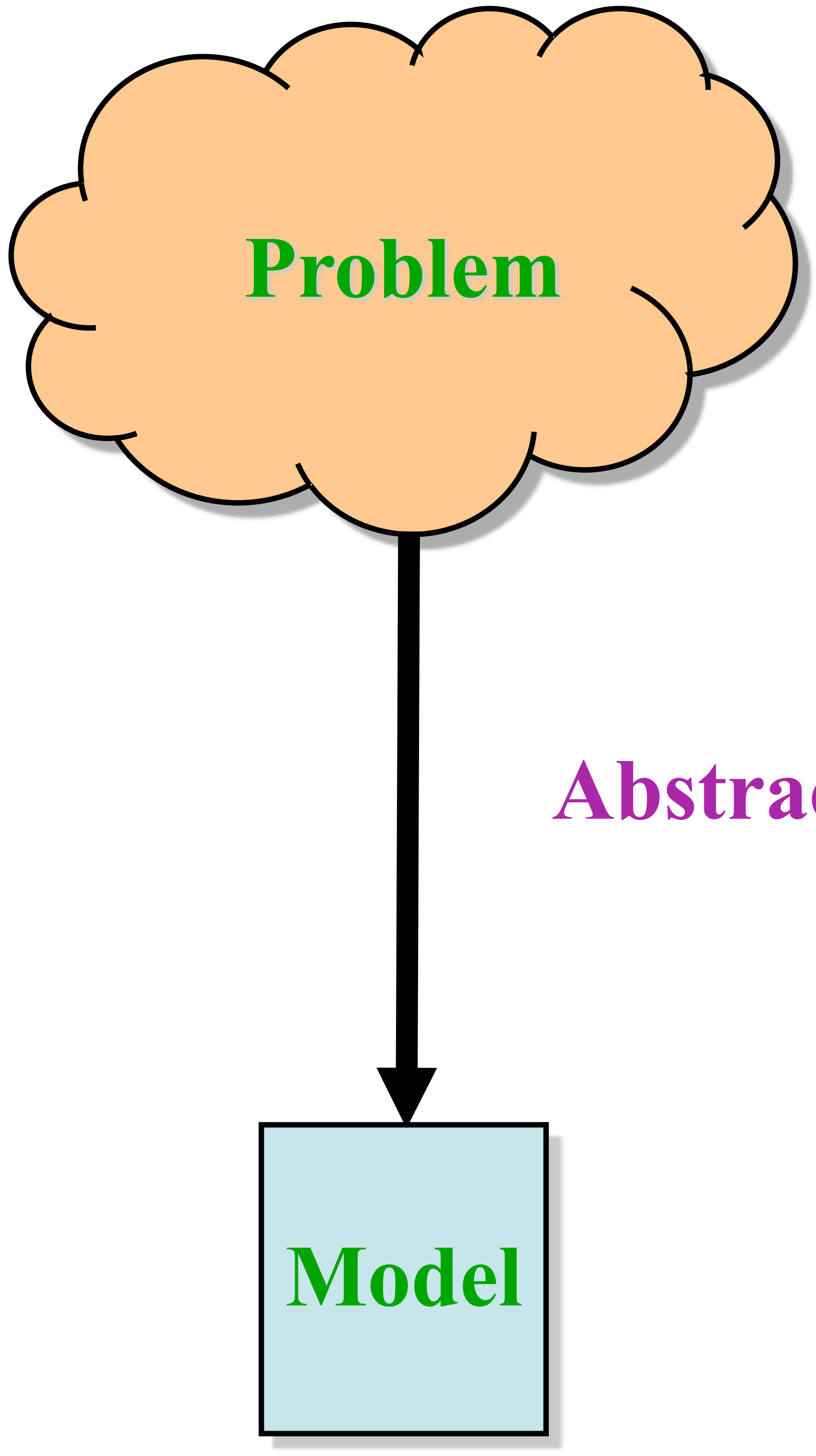
Handling Problems

- The first thing one is confronted in writing programs is the problem,
- You are confronted with real problems and your goal is to provide a program for the problem,
- Real problems are complicated by nature, and the first thing you have to do is to understand the problem, by separating necessary from unnecessary details,

Handling Problems

- In other words, you will try to obtain your own abstract view, or model, of the problem, a process called abstraction, see the next slide,
- The model focuses only on problem related stuff and you should try to define properties of the problem, including:
  - the data which are affected, and
  - the operations which are identified by the problem

Abstraction of Real Problems



Abstraction of Real Problems

- Consider the administration of employees in which it is required to create a program to administer the employees,
- We should know what information is needed by the administration? what tasks should be allowed?
- Employees can be characterized by:
  - name, date of birth, social number, room number, etc., these are called data for the employee model

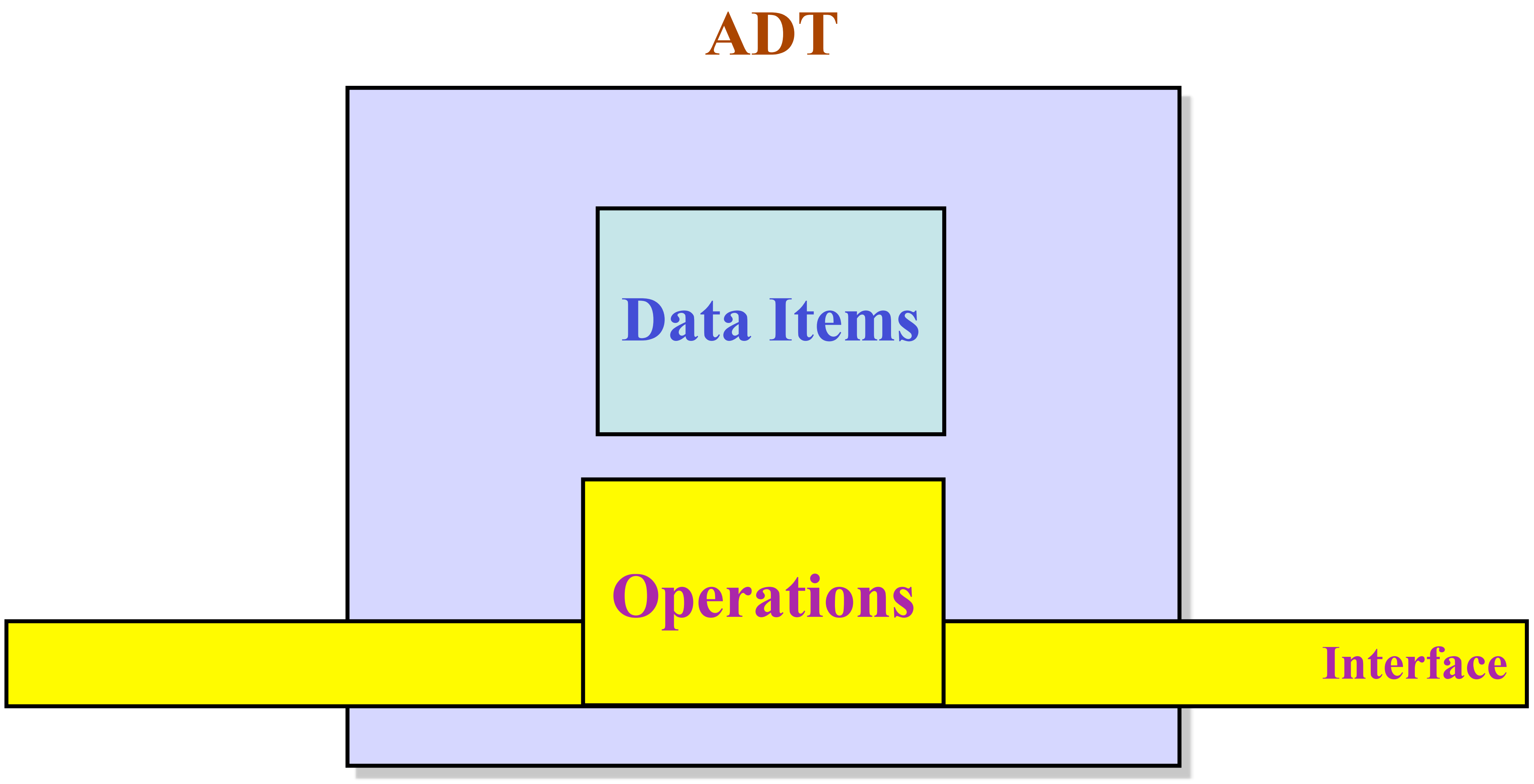
Abstraction of Real Problems

- There must be some operations, for example, an operation which allows you to create a new employee once a new person enters the institution, (Constructor)
- Consequently, you have to identify the operations that should be performed on an abstract employee and to decide how allowing access to the employees' data only with associated operations

Abstract Data Types

- Thus, with abstraction you create a well-defined entities that define the data structure of a set of items, each administered employee has a name, date of birth, and social number, etc.
- The data structure can only be accessed with defined operations called interface,
- An entity with the properties and the operations is called an abstract data type (ADT), see the next slide

Abstract Data Types



Abstract Data Types

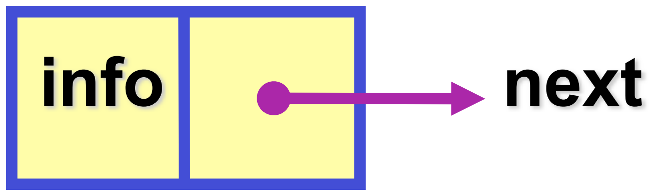
- **An ADT is usually characterized by the following properties:**
  - **it exports a type,**
  - **it exports a set of operations,**
  - **a subset of operations is called an interface, and**
  - **operations of the interface are the one and only access mechanism to the type's data structure**
- **An object-oriented language such as Java has a direct link to ADT by implementing them as a class**

Part # 2

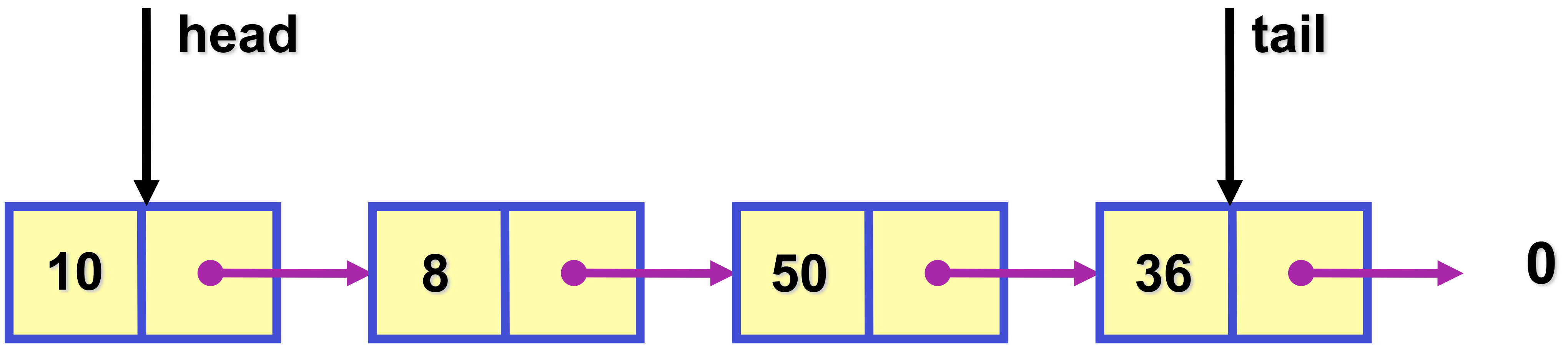
- Dynamic Memory Allocation
  1. Singly Linked Lists,
  2. Doubly Linked Lists,
  3. Circular Lists,
  4. Applications:
     A. Sparse Tables,
     B. Other Applications

Singly Linked Lists

- A singly linked list (SSL) is a concrete data structure consisting of a sequence of nodes,



- Each node stores:
  - A value called info of specific type, and
  - A pointer link to the next node, called next
- Two pointers of type node identify the list, these are: head and tail



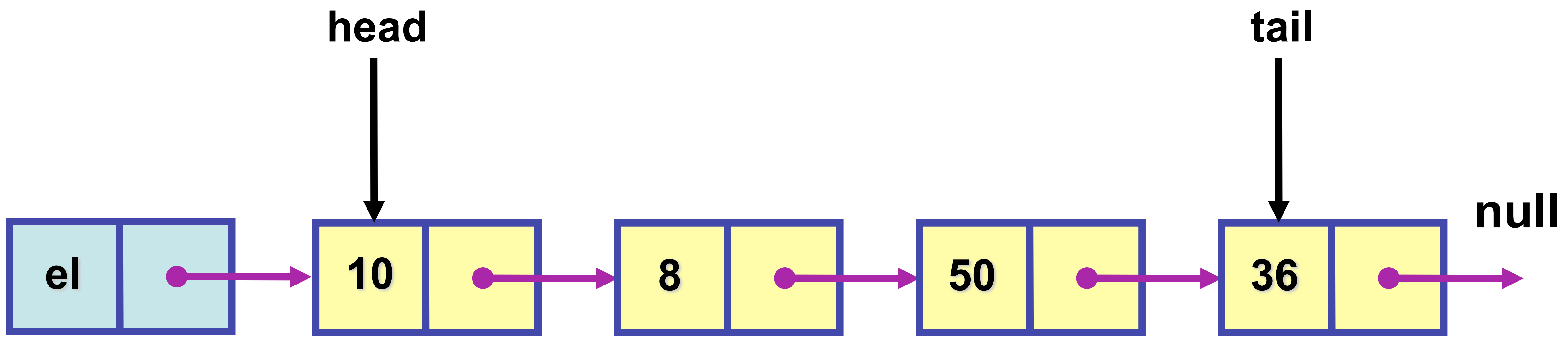
The SLL Node Class Definition

```
class Node {
   int info;
   Node next;
   Node() { info = 0; next = null; }
   Node(int el) {
         info = el;
         next = null;
   }
   Node(int el, Node n) {
        info = el;
        next = n;
   }
}
```

The SinglyLinkedList Class in Java

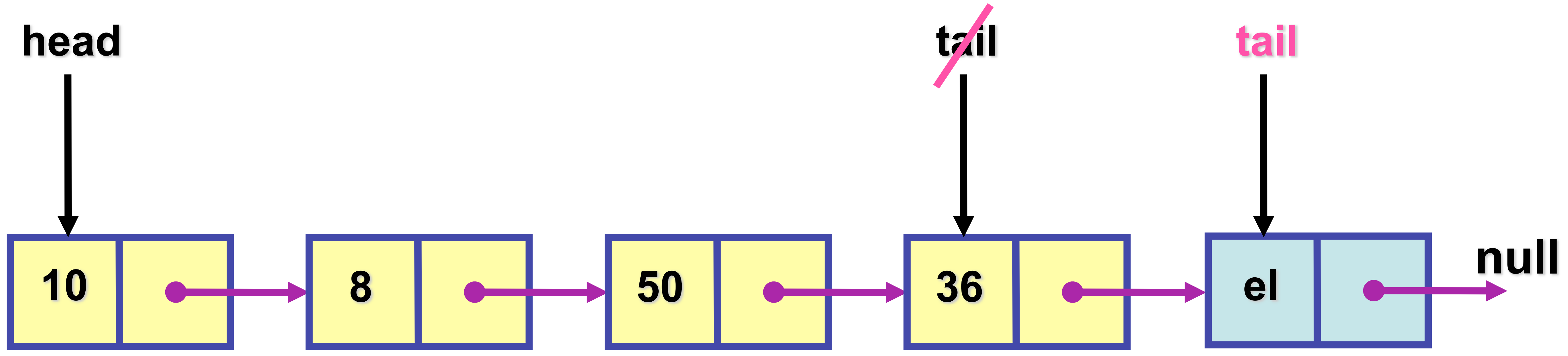
```
public class SinglyLinkedList {
    public static void main(String argv[]) {
        …
    }
    class Node {
        …
    }
    private Node head;
    private Node tail;
    SinglyLinkedList( ) {
        head = tail = null;
    }
    public void addToHead(int el)    {        …        }
    public void addToTail(int el)    {        …        }
    public int deleteFromHead()      {        …        }
    public int deleteFromTail()      {        …        }
    public int deleteNode()          {        …        }
    public boolean isInList(int el)  {        …        }
    public void printList()          {        …        }
}
```

Method addToHead



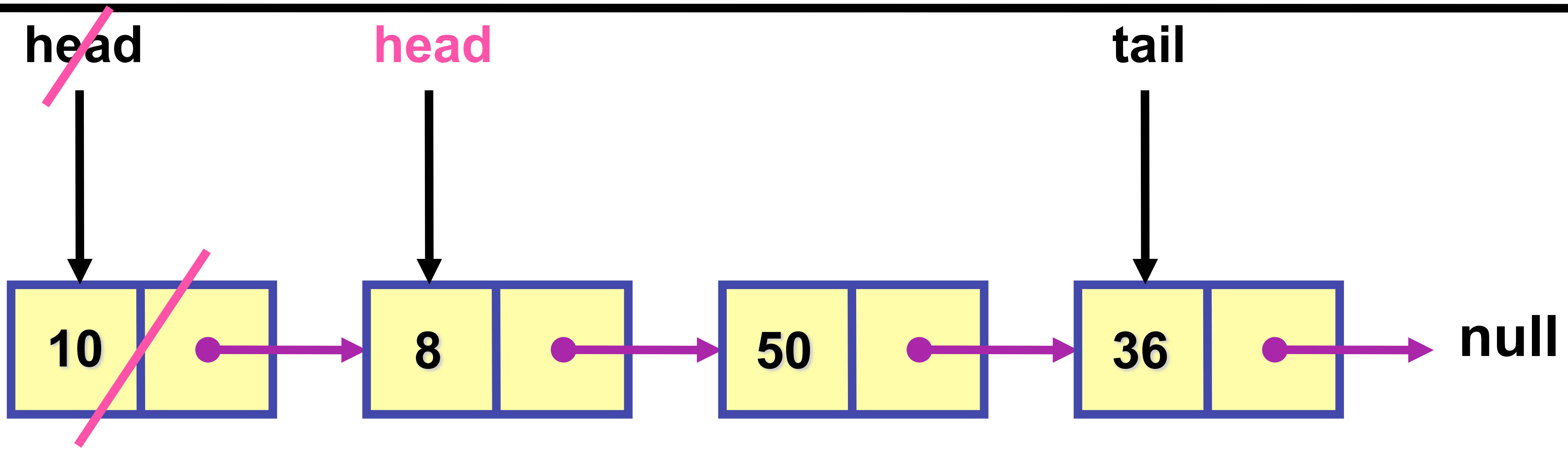
```
public void addToHead(int el) {
        if ((head == null) && (tail == null))
          { head = tail = new Node(el); }
        else
          { head = new Node(el, head); }
}
```

Method addToTail



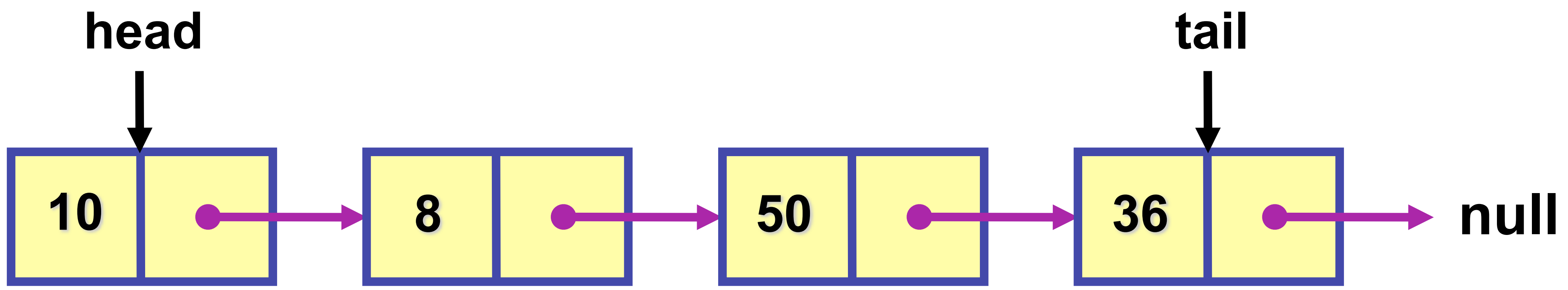
```
public void addToTail(int el) {
    if ((head == null) && (tail == null)) {
        head = tail = new Node(el);
    }
    else {
        tail.next = new Node(el);
        tail = tail.next;
    }
}
```

Method deleteFromHead



```
public int deleteFromHead() {
    if ( isEmpty() ) { return 0; } // Better to add a new method isEmpty
    else {
        int el = head.info;
        if (head == tail)    // if only one node in the list
            { head = tail = null; }
        else { head = head.next; }
        return el;
    }
}
```

Method deleteFromTail

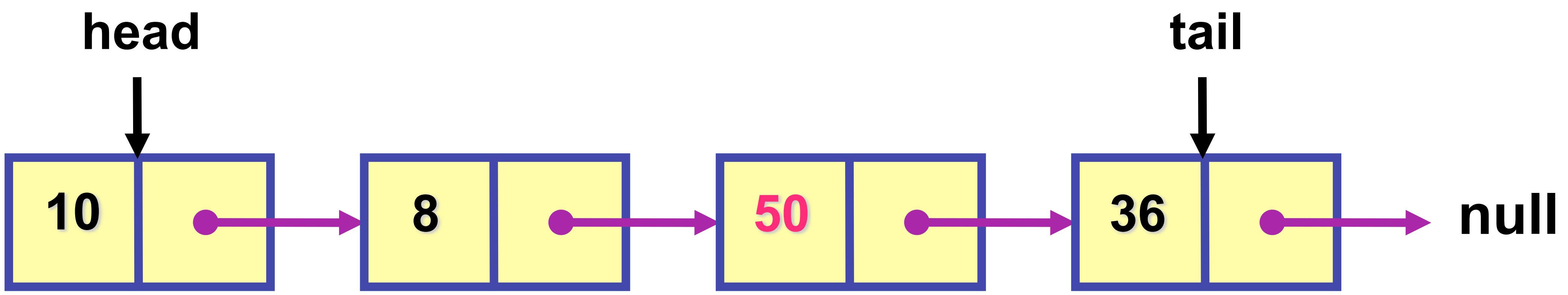


```
public int deleteFromTail() {
    if ( isEmpty() ) { return 0; }
    else {
        int el = tail.info;
        if (head == tail)  {  // if only one node in the list
            head = tail = null;
        }
        else { // if more than one node in the list, find the predecessor of tail
            Node tmp;
            for (tmp = head; tmp.next != tail; tmp = tmp.next);
            tail = tmp;     tail.next = null;
        }
        return el;
    }
}
```

Method deleteNode

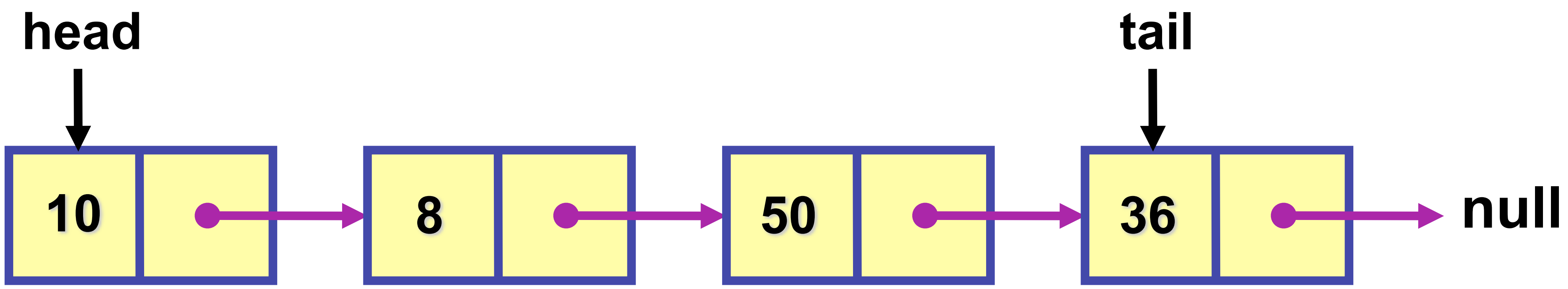
```
public void deleteNode(int el) {
    if ( !isEmpty() ) { // if not empty list
        if ((head == tail) && (el == head.info)) { // if only one node in the list
            head = tail = null;   // and el is the head
        }
        else if (el == head.info) { // if more than one in the list and el is head
            head = head.next;
        }
        else { // search for el starting at head, ending at tail
            Node pred, tmp;
            for (pred = head, tmp = head.next; (tmp != null) && !(tmp.info == el);
                                             pred = pred.next, tmp= tmp.next);
            if (tmp != null) {
                pred.next = tmp.next;
                if (tmp == tail) tail = pred;
            }
        }
    }
}
```

Method isInList



```
public boolean isInList(int el) {
          Node tmp;
          for (tmp = head; tmp != null && !(tmp.info == el);
                                             tmp = tmp.next);
          return (tmp != null);
}
```

Method printList



```
public void printList() {
   Node tmp = head;
   System.out.println("The Singly Linked List is:");
   while (tmp != null) {
      System.out.println(tmp.info);
      tmp = tmp.next;
   }
}
```

Testing the SinglyLinkedList Class

```
publix static void main() {
  SinglyLinkedList MyList = new SinglyLinkedList();
  MyList.addToHead(10);
  MyList.addToHead(20);
  MyList.printList();
  MyList.addToTail(100);
  MyList.addToTail(200);
  MyList.printList();
  System.out.println("Searching for 100: ", MyList.isInList(100));
  System.out.println("Deleted From Head: ", MyList.deleteFromHead());
  MyList.printList();
  System.out.println("Deleted From Tail: ", MyList.deleteFromTail());
  MyList.printList();
  System.out.println("Deleted Node 100: ");
  MyList.deleteNode(100);
  MyList.printList();
  System.out.println("Searching for 100: ", MyList.isInList(100));
}
```

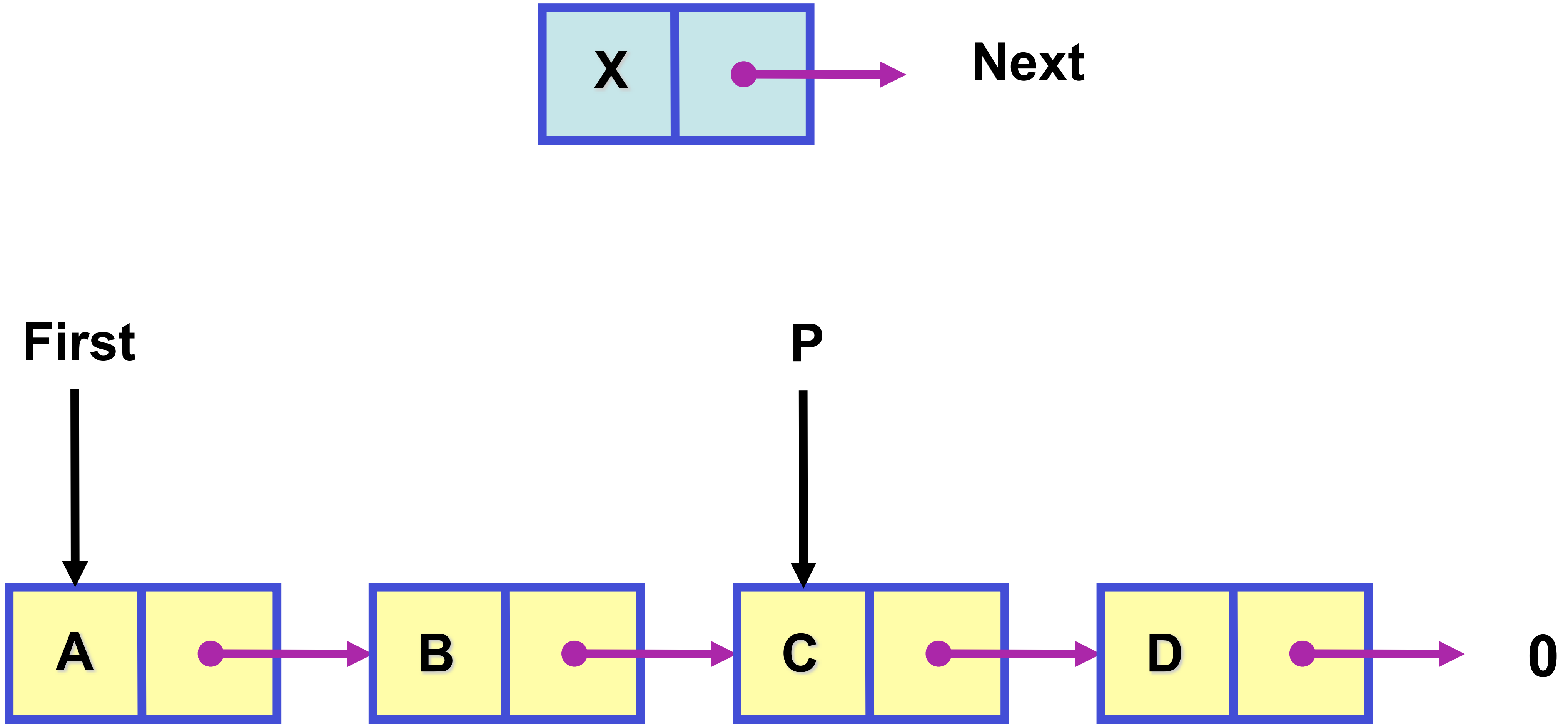
Testing the SinglyLinkedList Class

```
The Singly Linked List is:
20
10
The Singly Linked List is:
20
10
100
200
Searching for 100: true
Deleted From Head: 20
The Singly Linked List is:
10
100
200
```

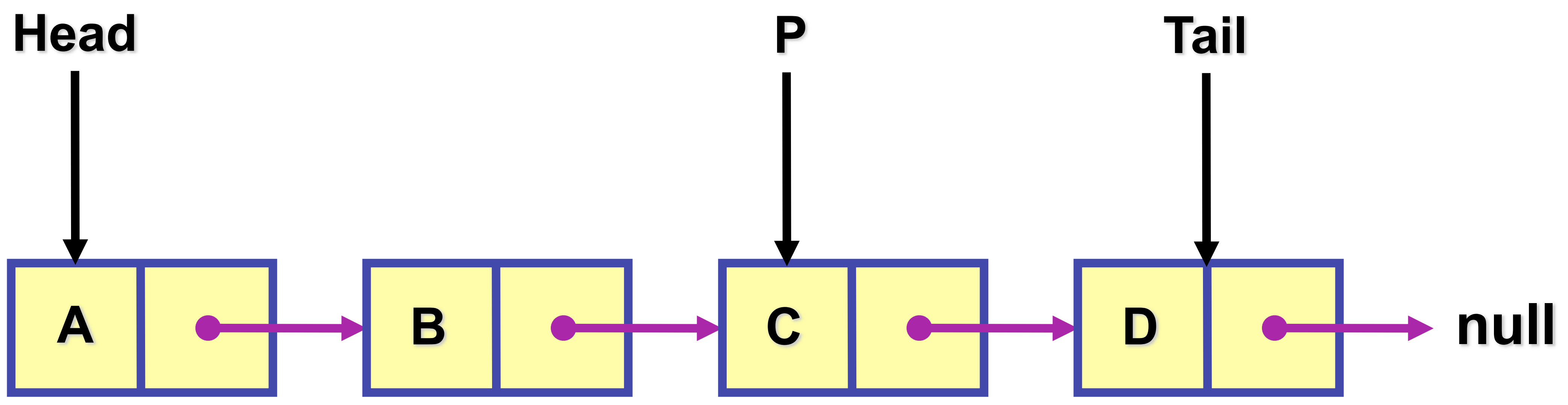
```
Deleted From Tail: 200
The Singly Linked List is:
10
100
Deleted Node 100:
The Singly Linked List is:
10
Searching for 100: false
```

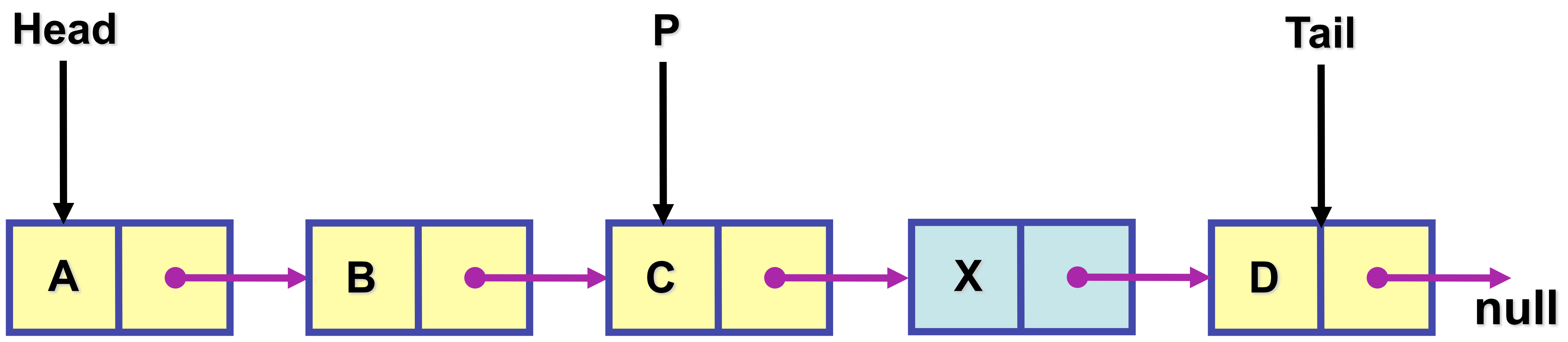
Operations on a Singly Linked Lists

- Inserting a node whose value is X after a node pointed to by position pointer P



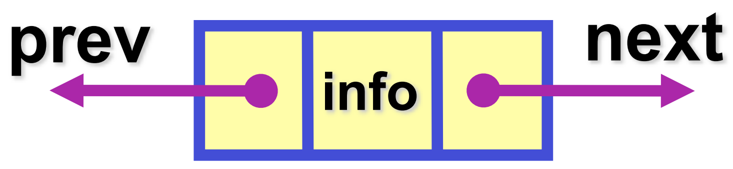
Singly Linked List: InsertAfter



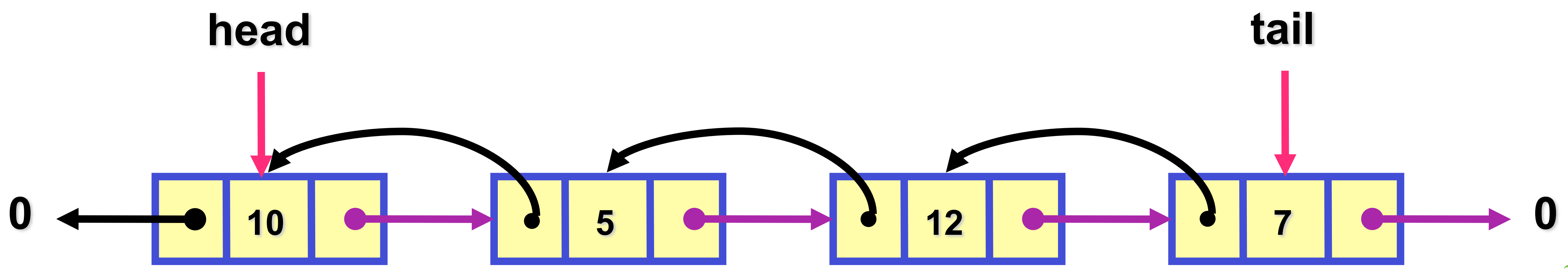


Doubly Linked Lists

- In a doubly linked, nodes store:
  - Element info,
  - Link to the previous node, and
  - Link to the next node



- Two pointers tail and head,
- In order to be able to design nodes with general type (int, real, char, …, etc.), we will use templates



The Node Class for Doubly Linked List

```
class Node {
        Object info;
        Node next, prev;
        Node() { info = null; next = null; prev = null; }
        Node(Object el) {
                info = el;
                next = null; prev = null;
        }
        Node(Object el, Node n, Node p) {
                info = el;
                next = n; prev = p;
        }
}
```

The DoublyLinkedList Class

```
public class DoublyLinkedList {
        public static void main(String argv[]) { … }
        class Node {
           …
        }
        private Node head;
        private Node tail;
        DoublyLinkedList( ) {
                head = tail = null;
        }
        public void addToTail(Object el) { … }
        public Object deleteFromTail() { … }
        public void printList() { … }
}
```

Method addToTail

```
public void addToTail(Object el) {
    if (tail != null) {
        tail = new Node(el, null, tail);
        tail.prev.next=tail;
    }
    else head = tail = new Node(el);
}
```

Method deleteFromTail

```
public Object deleteFromTail() {
   if (head  == null) return null;
   else {
      Object el = tail.info;
      if (head ==tail) { // if only one node in the list
         head = tail = null;
      }
      else {
         tail = tail.prev;
        tail.next = null;
      }
      return el;
   }
}
```

Method printList

```
public void printList() {
    Node tmp = head;
    System.out.println("The Doubly Linked List is:");
    while (tmp != null) {
        System.out.println(tmp.info);
        tmp = tmp.next;
    }
}
```

Testing the DoublyLinkedList Class

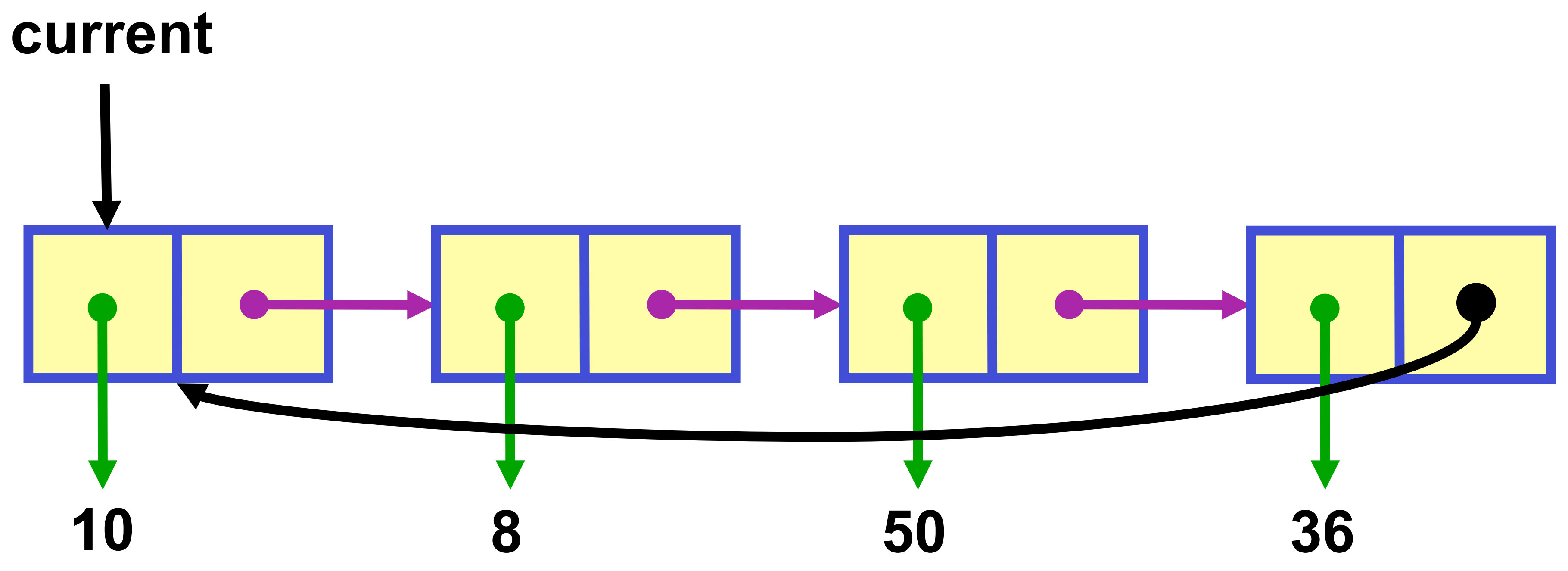
```
public static void main(String argv[]) {
   DoublyLinkedList MyList = new DoublyLinkedList();
   MyList.addToTail(new Integer(10));
   MyList.printList();
   MyList.addToTail(new Integer(20));
   MyList.printList();
   MyList.addToTail(new Integer(30));
   MyList.printList();
   System.out.println
            ("Deleted From Tail: "+MyList.deleteFromTail());
   MyList.printList();
}
```

Testing the DoublyLinkedList Class

```
The Doubly Linked List is:
10
The Doubly Linked List is:
10
20
The Doubly Linked List is:
10
20
30
Deleted From Tail: 30
The Doubly Linked List is:
10
20
```

Circular Lists

- In some situations, a circular list is needed in which nodes form a ring, or in other words, the list is finite and each node has a successor,
- Below is an example of circular singly linked list

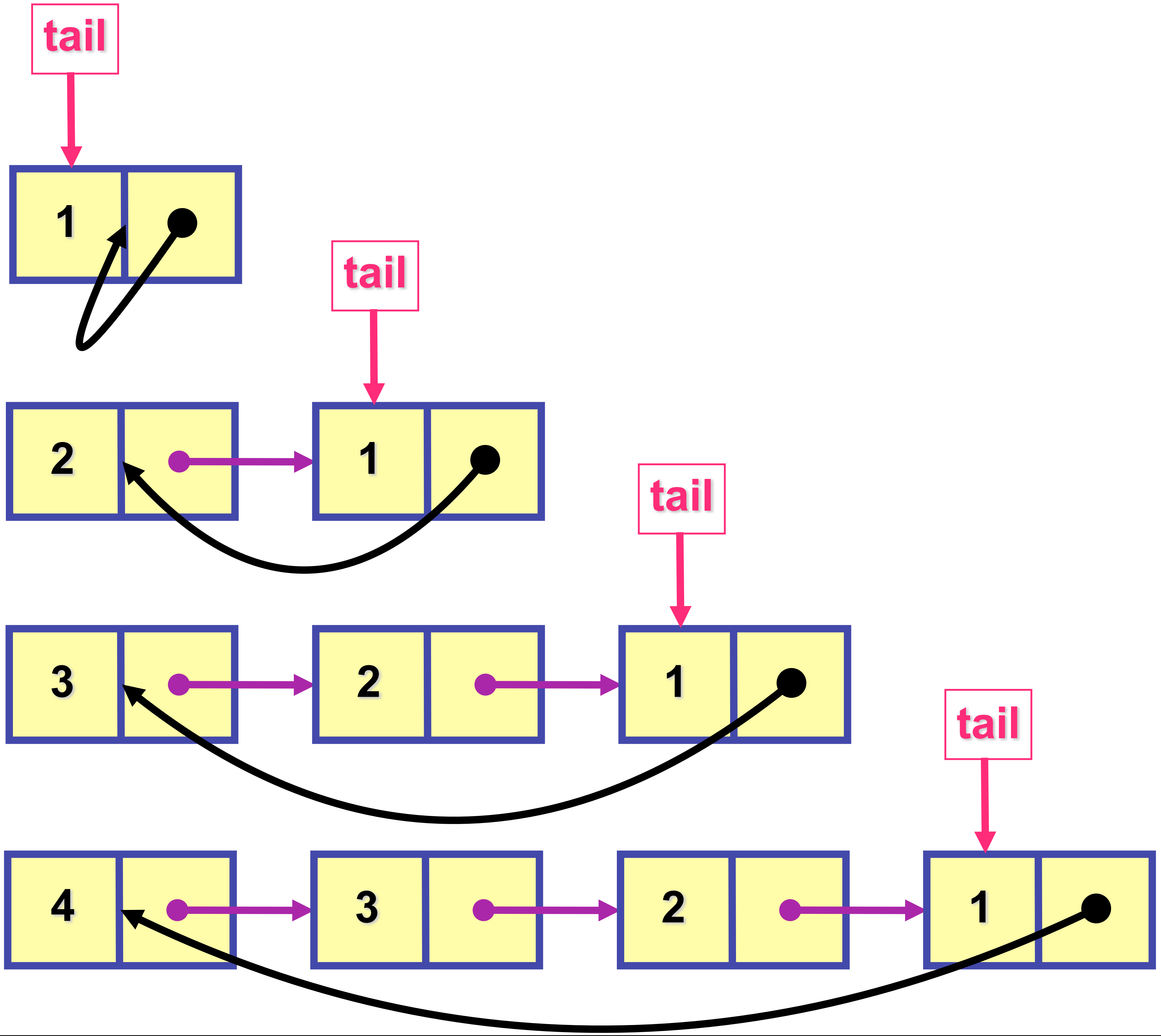


Circular Singly Linked Lists

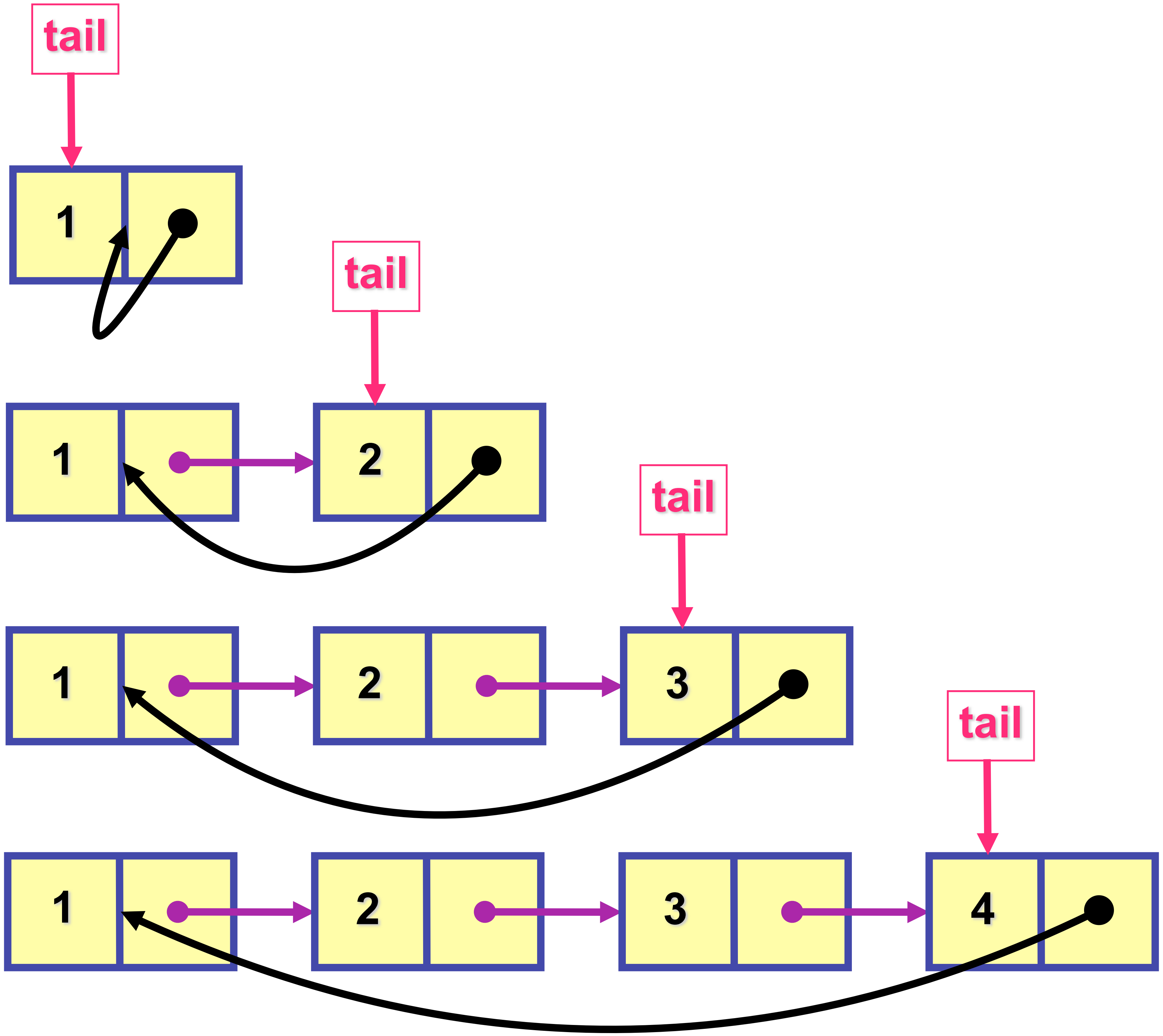
- In an implementation of a circular singly linked list, we can use only one permanent pointer, tail, to the list even so operations on the list require access to the tail and its predecessor, the head,

- The next two slides show a sequence of insertions at the front and at the end of the circular list

Inserting at the Front of a Circular SLL



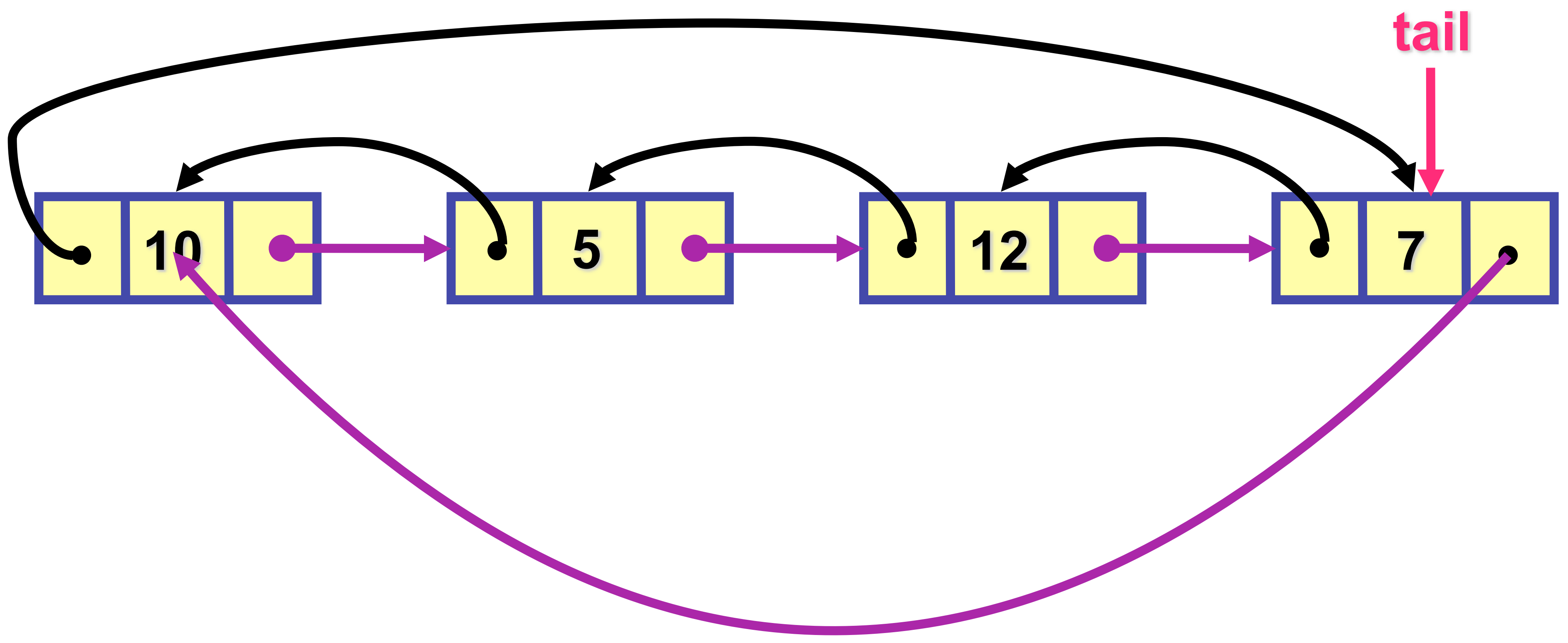
Inserting at the End of a Circular SLL



Circular Doubly Linked Lists

- **The implementation of Circular SLL is not without problems,**
- **A member function for deleting of the tail node requires a loop so that tail can be set after delete to its predecessor,**
- **Moreover, processing data in the reverse order is not effective,**
- **To avoid these problems, A circular doubly linked list can be used**

Circular Doubly Linked List



Applications: Sparse Tables

- In many real-world applications, the choice of a table seems to be the most natural one, but space consideration may preclude this choice,
- This is particularly true if only a small fraction of the table is actually used,
- A table of this type is called a sparse table since the table is populated sparsely by data and most of its cells are empty, so linked lists are better

Sparse Table Example

Students

1.	Sheaver Geo
2.	Weaver Henry
3.	Shelton Mary
	…
404.	Crawford William
405.	Lawson Earl
	…
5206.	Fulton Jenny
5207.	Craft Donald
5208.	Oates Key
	…

Classes

1.	Anatomy/Physiology
2.	Microbiology
	…
20	Advanced Writing
31	Chaucer
	…
115	Data Structures
116	Cryptology
117	Computer Ethics
	…

Grade Code

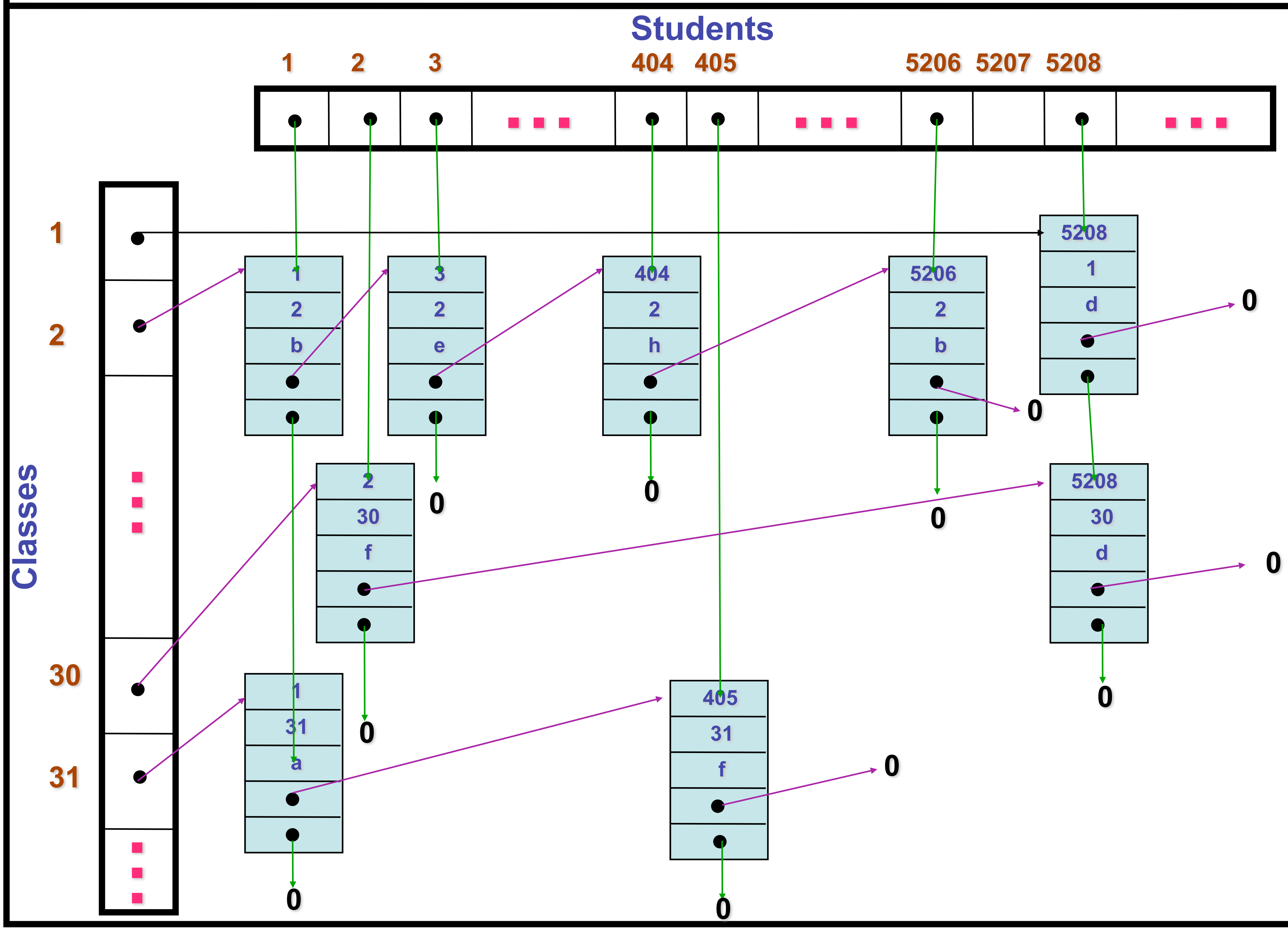
a	A
b	A-
c	B+
d	B
e	B-
f	C+
g	C
h	C-
i	D
j	F

Student

Class

	1	2	3	…	404	405	…	5206	5207	5208	…	8000
1										d		
2.	b		e		h			b				
…												
30.	f								d			
31.	a					f						
..												
200												

Implementation using Linked Lists



Other Applications

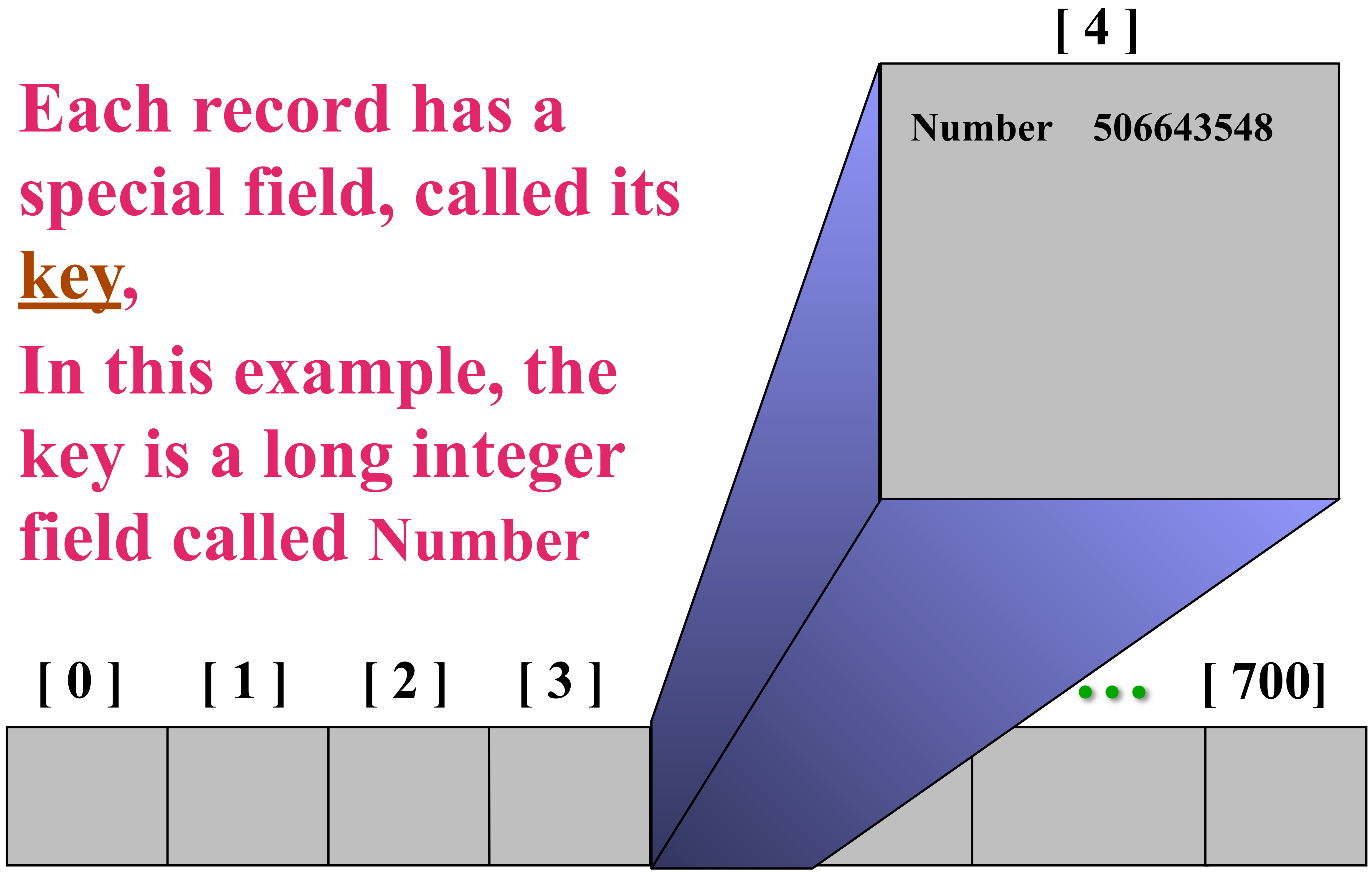
- Various other applications presented and discussed in class including:
  - Dynamic Allocation Problems,
  - The File System in an Operating System,
  - Implementing the base class for other dynamic data structures, such as stacks, trees and graphs

Part # 3

- **Hashing Techniques**
  1. **Concepts of Hashing,**
  2. **Hash Tables and Hash Functions,**
  3. **Basic Operations in Hash Tables,**
  4. **Resolving Collisions**

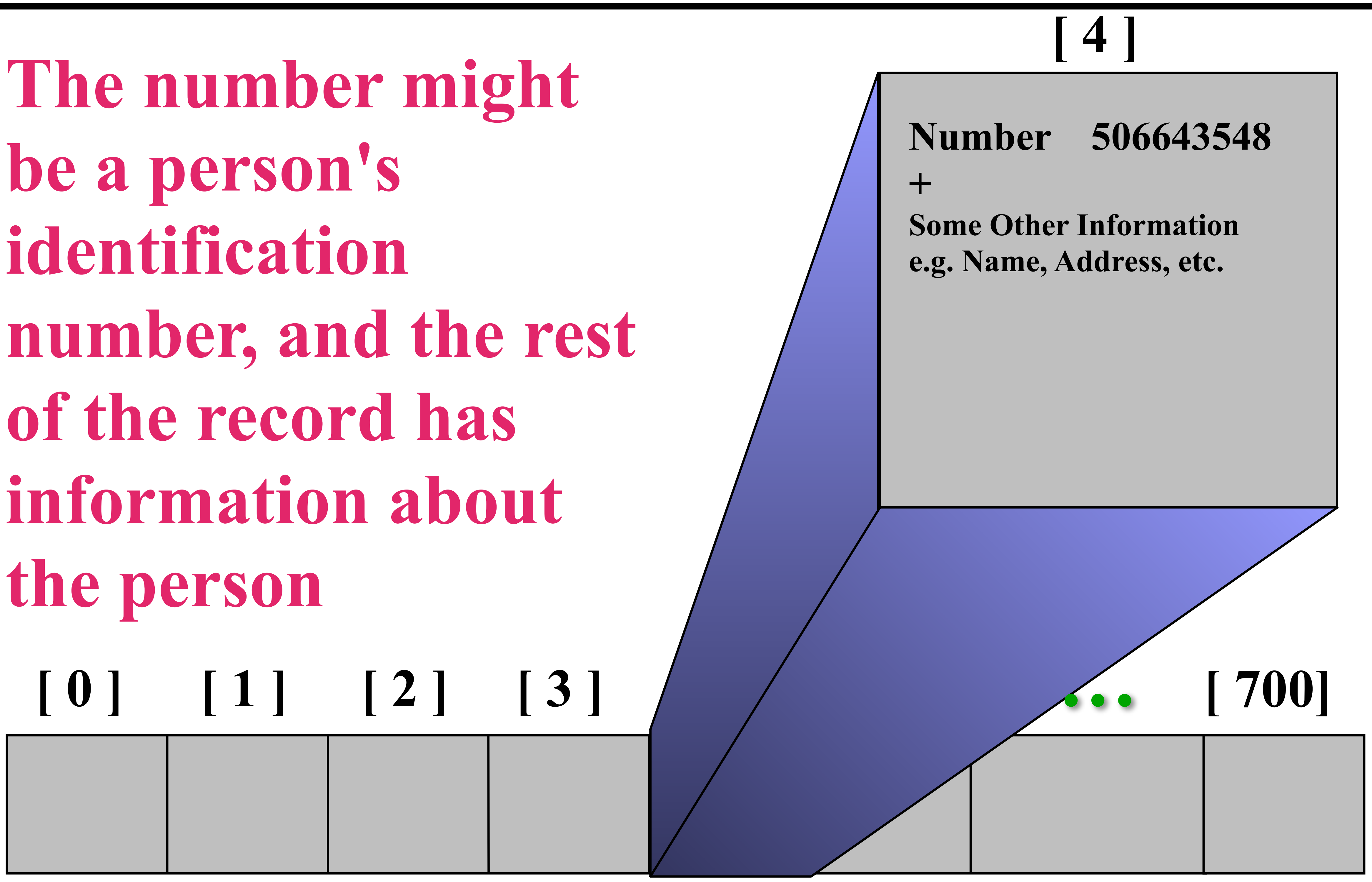
What is a Hash Table?

- Each record has a special field, called its key,
- In this example, the key is a long integer field called Number



What is a Hash Table?

- The number might be a person's identification number, and the rest of the record has information about the person



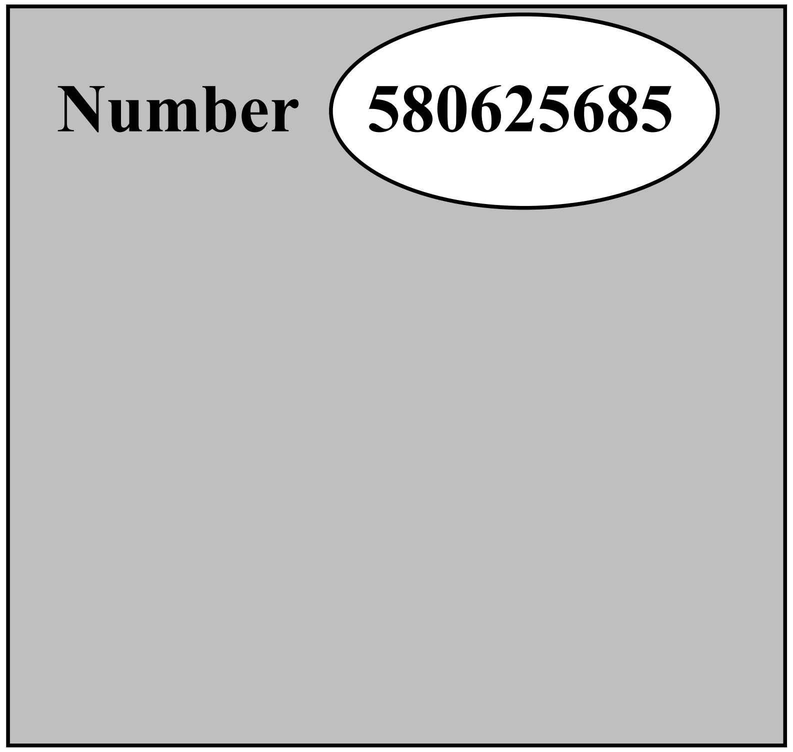
What is a Hash Table?

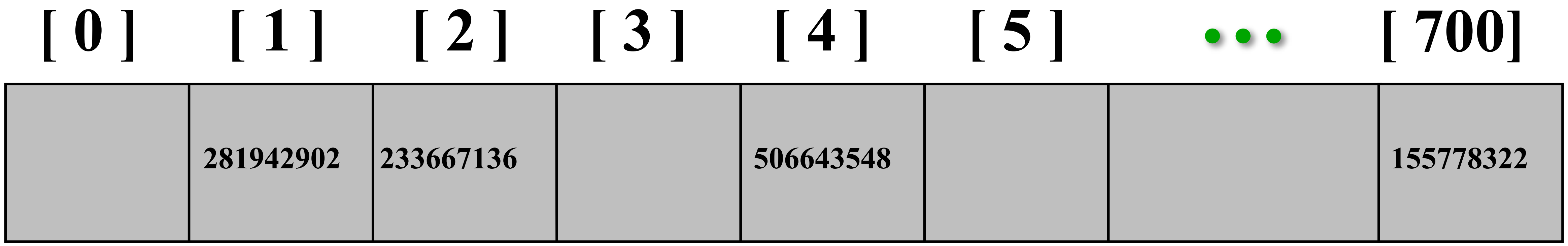
- When a hash table is in use, some spots contain valid records, and other spots are "empty"

[0]	[1]	[2]	[3]	[4]	[5]	• • •	[700]
	281942902	233667136		506643548			155778322

Inserting a New Record

- To insert a new record, the key must be converted to an array index,
- The index is called the hash value of the key



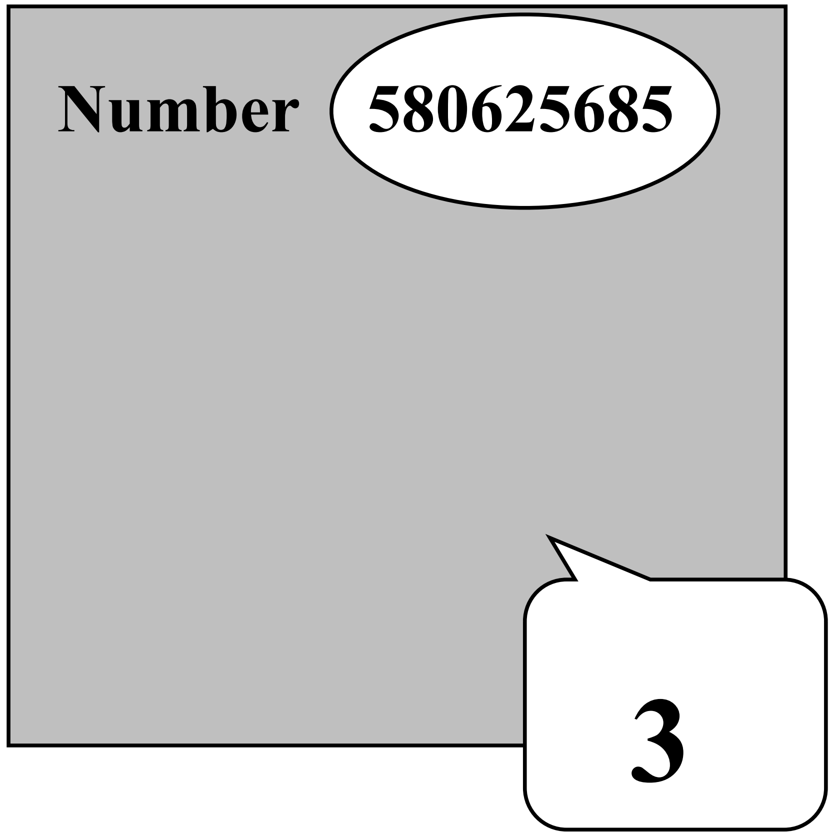


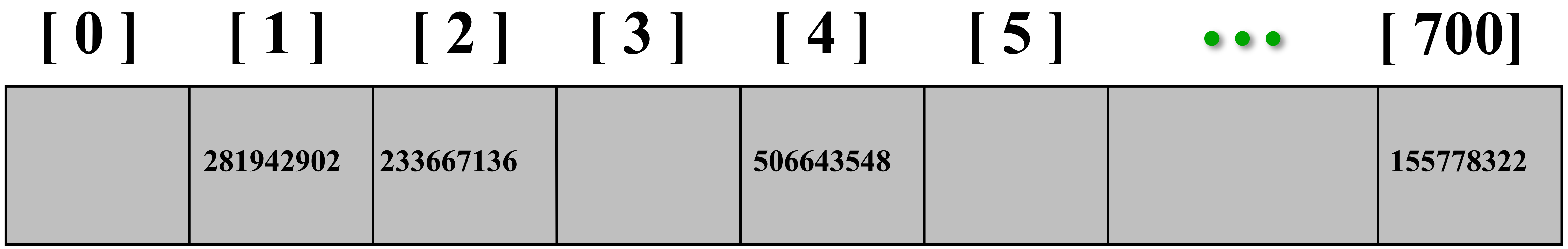
Inserting a New Record

- **Typical way to create a hash value:**

(Number mod 701)

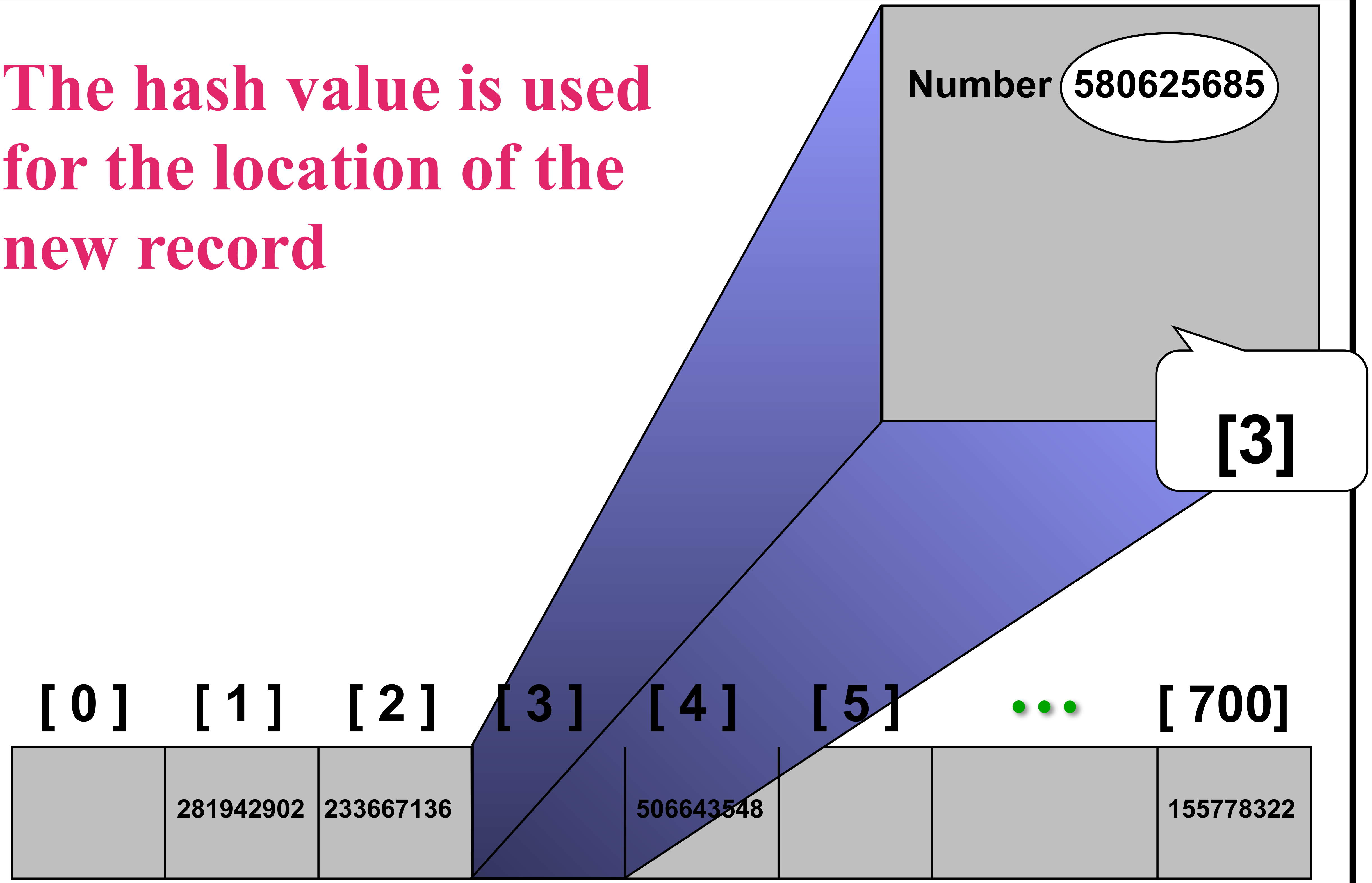
What is (580625685 mod 701) ?





Inserting a New Record

- The hash value is used for the location of the new record



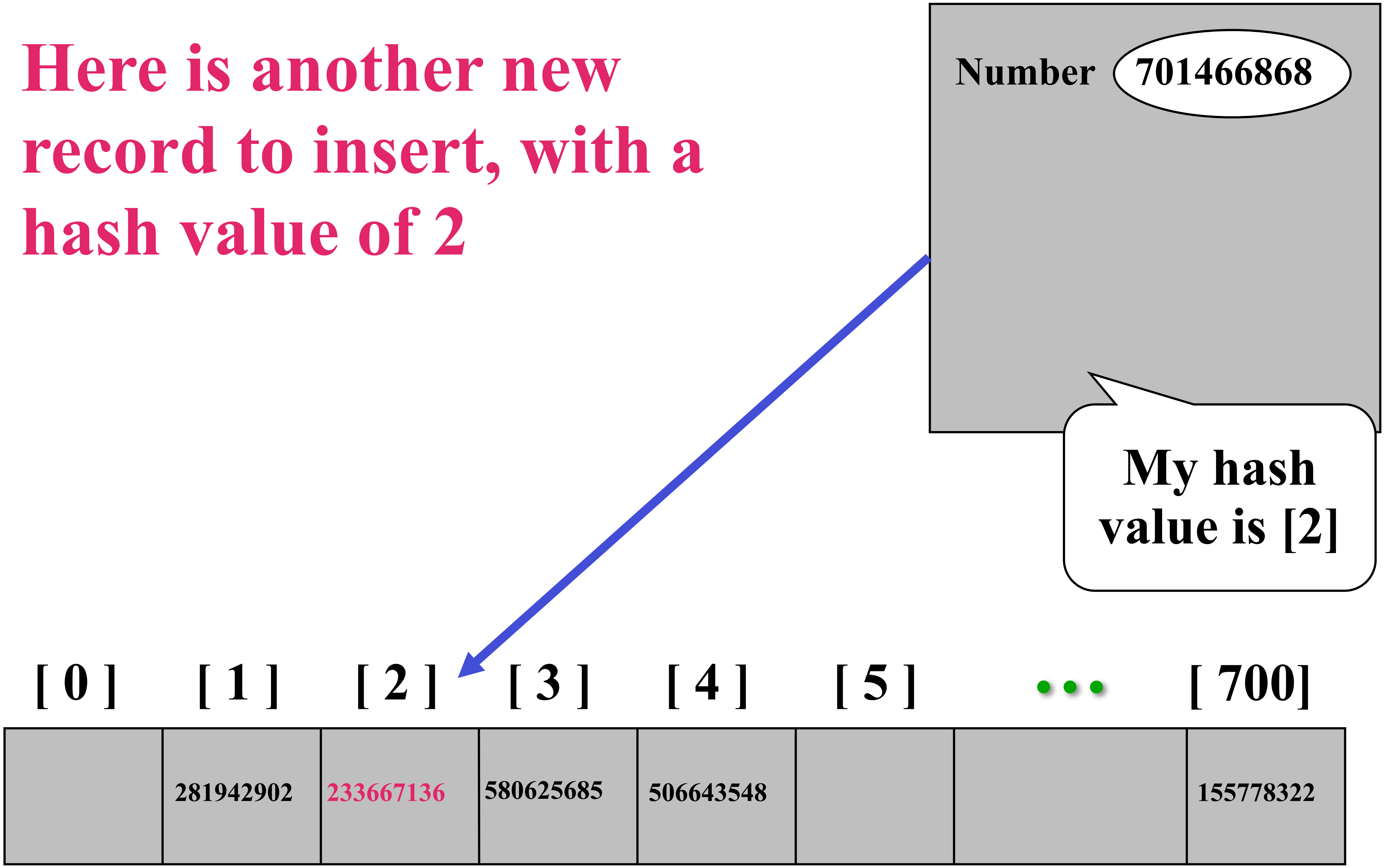
Inserting a New Record

- The hash value is used for the location of the new record

[0]	[1]	[2]	[3]	[4]	[5]	•••	[700]
	281942902	233667136	580625685	506643548			155778322

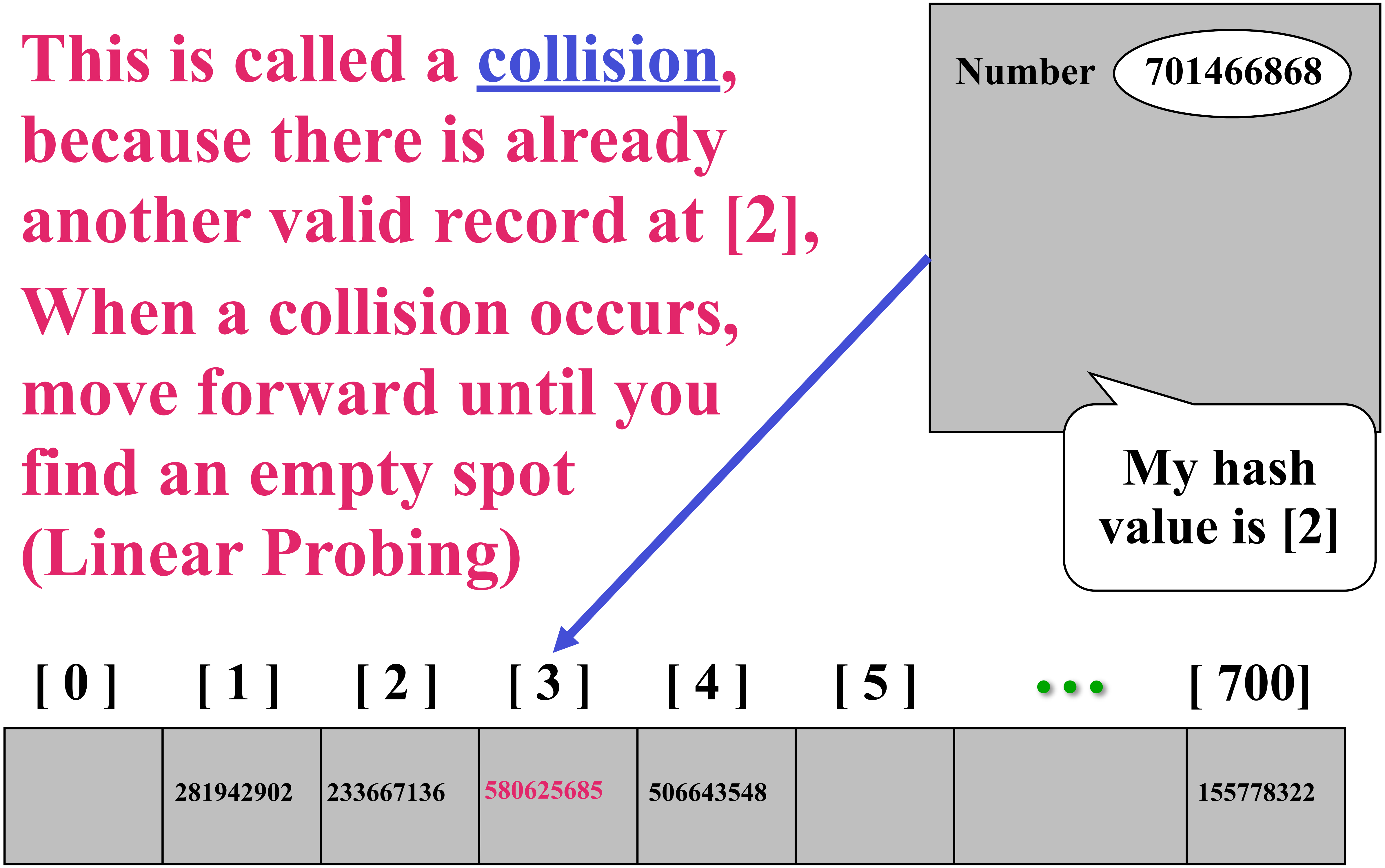
Collisions

- Here is another new record to insert, with a hash value of 2



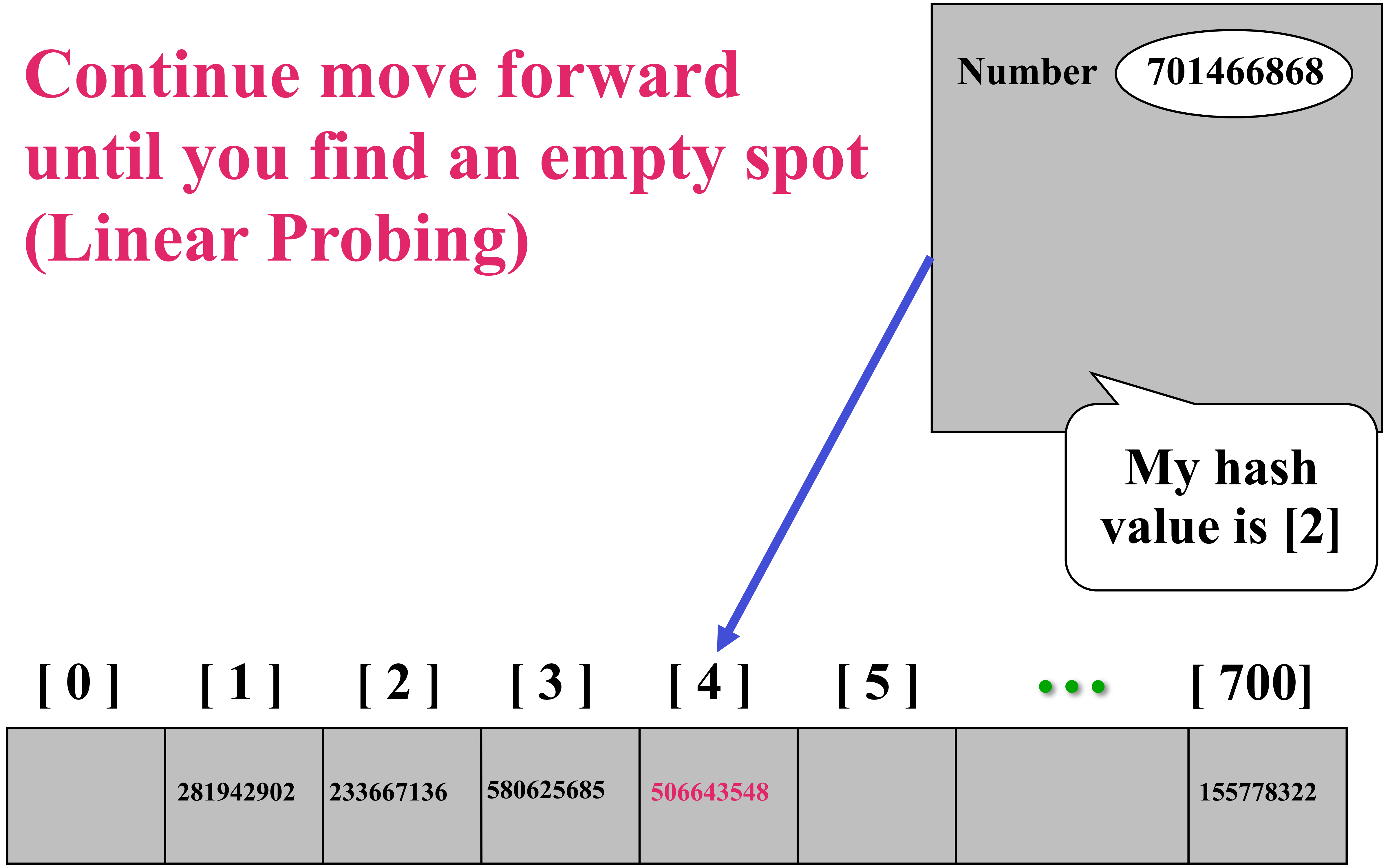
Collisions

- This is called a collision, because there is already another valid record at [2],
- When a collision occurs, move forward until you find an empty spot (Linear Probing)



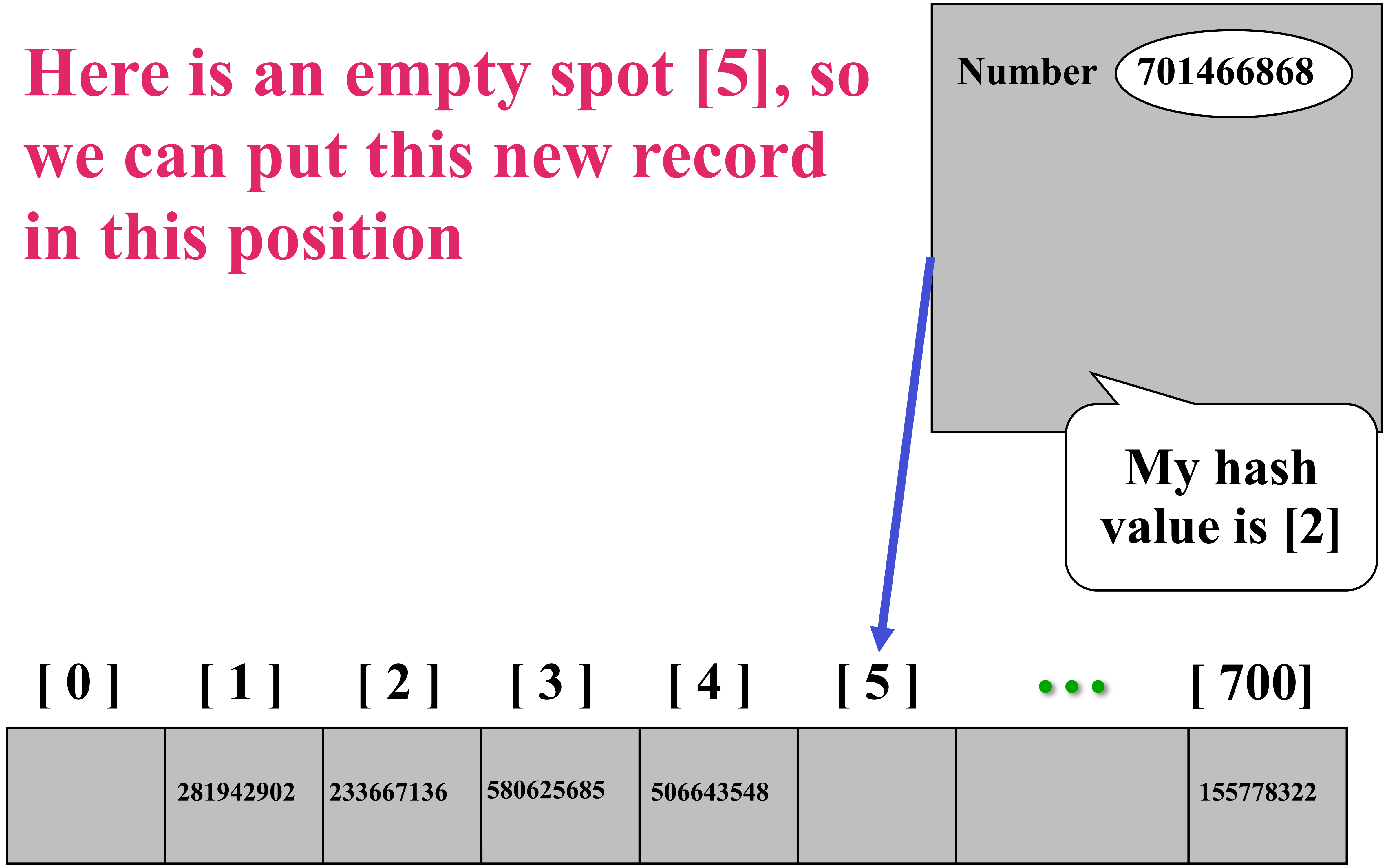
Collisions

- Continue move forward until you find an empty spot (Linear Probing)



Collisions

- Here is an empty spot [5], so we can put this new record in this position



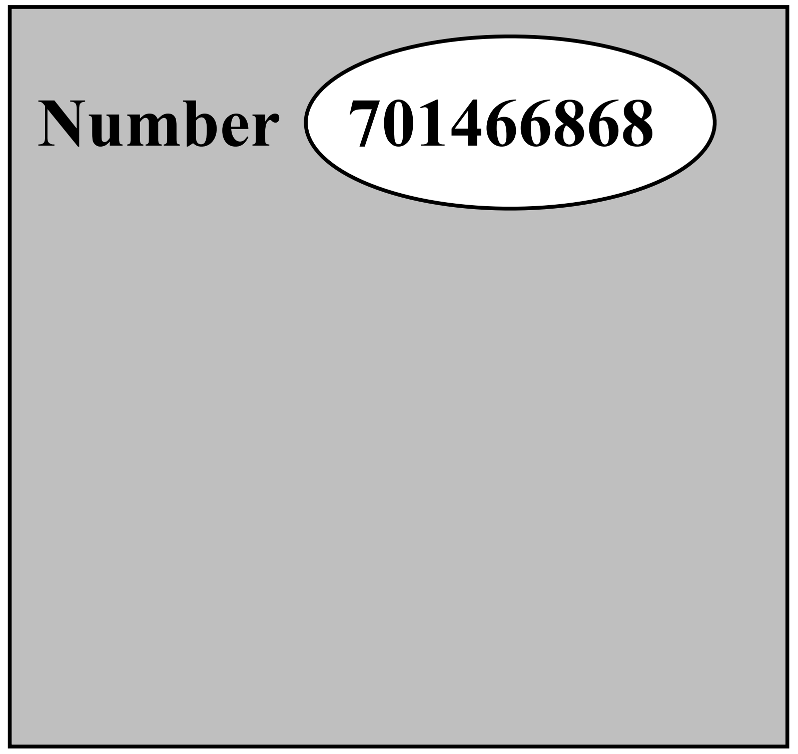
Collisions

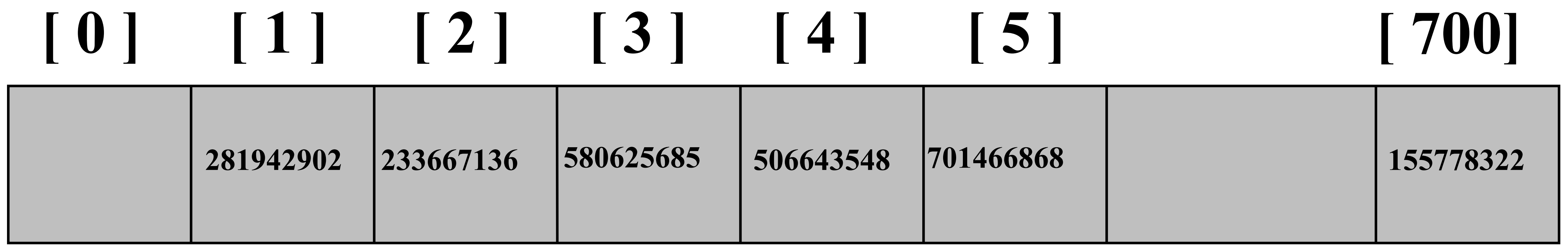
- The new record goes in the empty slot

[0]	[1]	[2]	[3]	[4]	[5]	• • •	[700]
	281942902	233667136	580625685	506643548	701466868		155778322

Searching for a Key

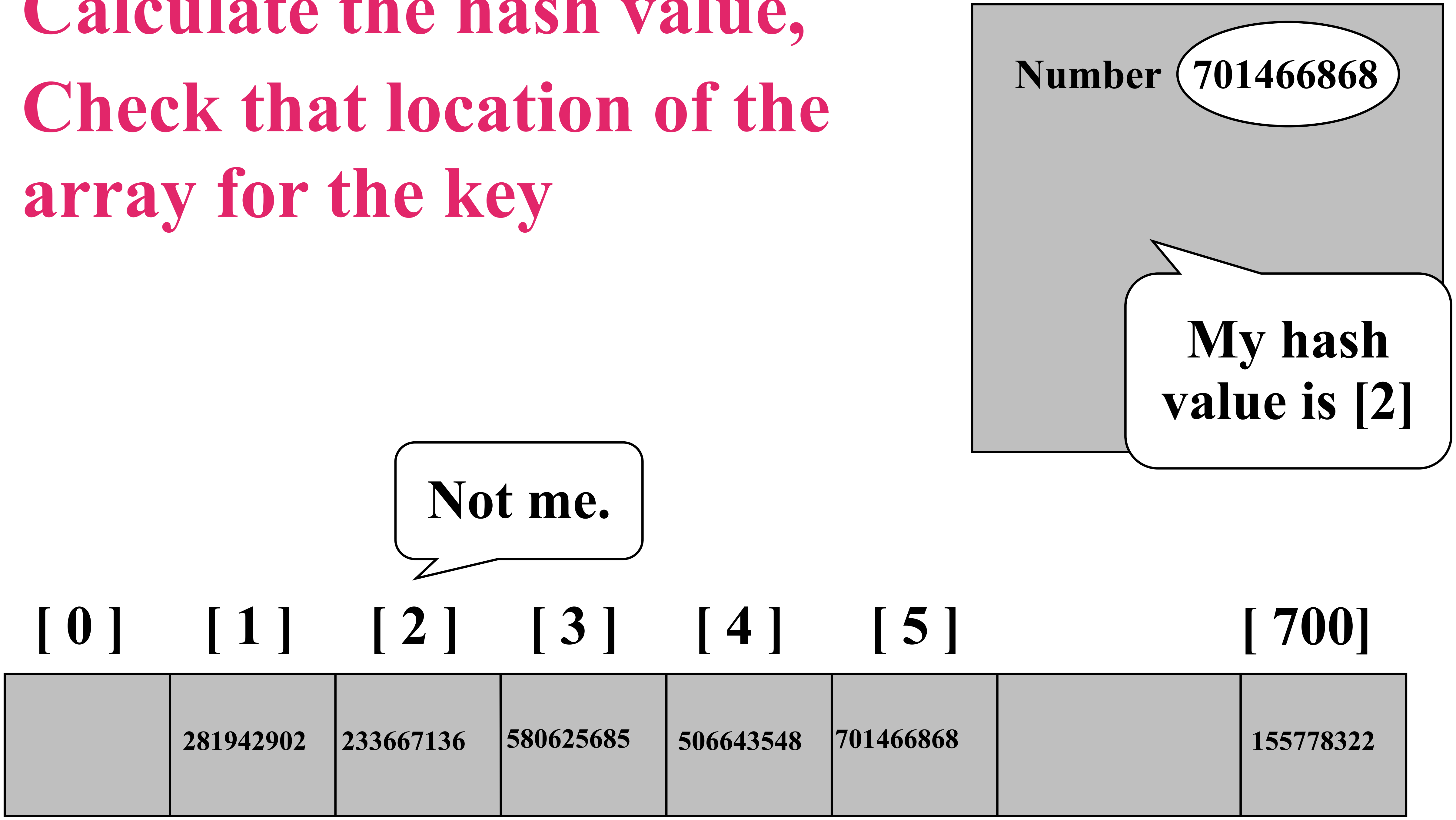
- The data that's attached to a key can be found fairly quickly





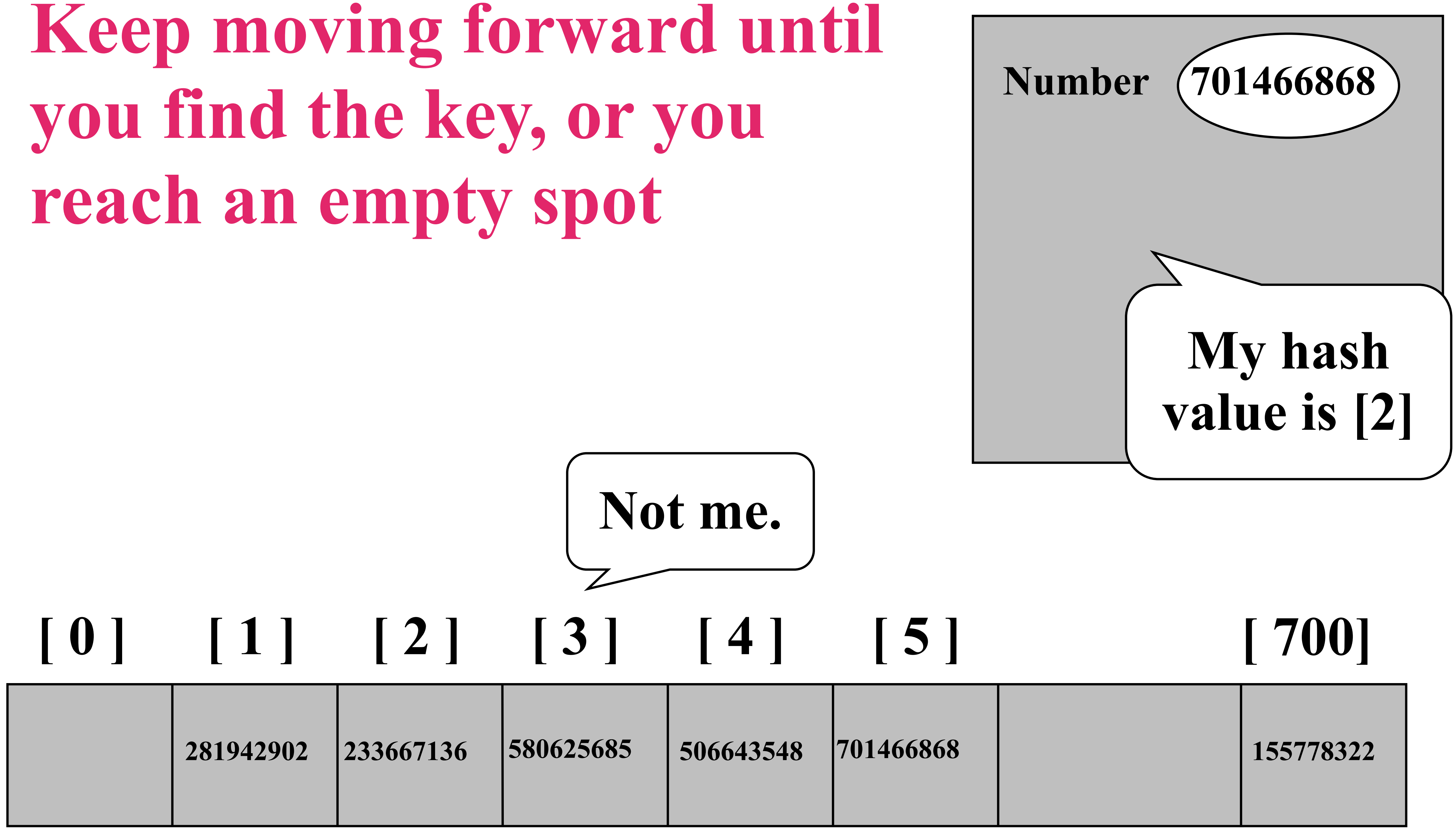
Searching for a Key

- Calculate the hash value,
- Check that location of the array for the key



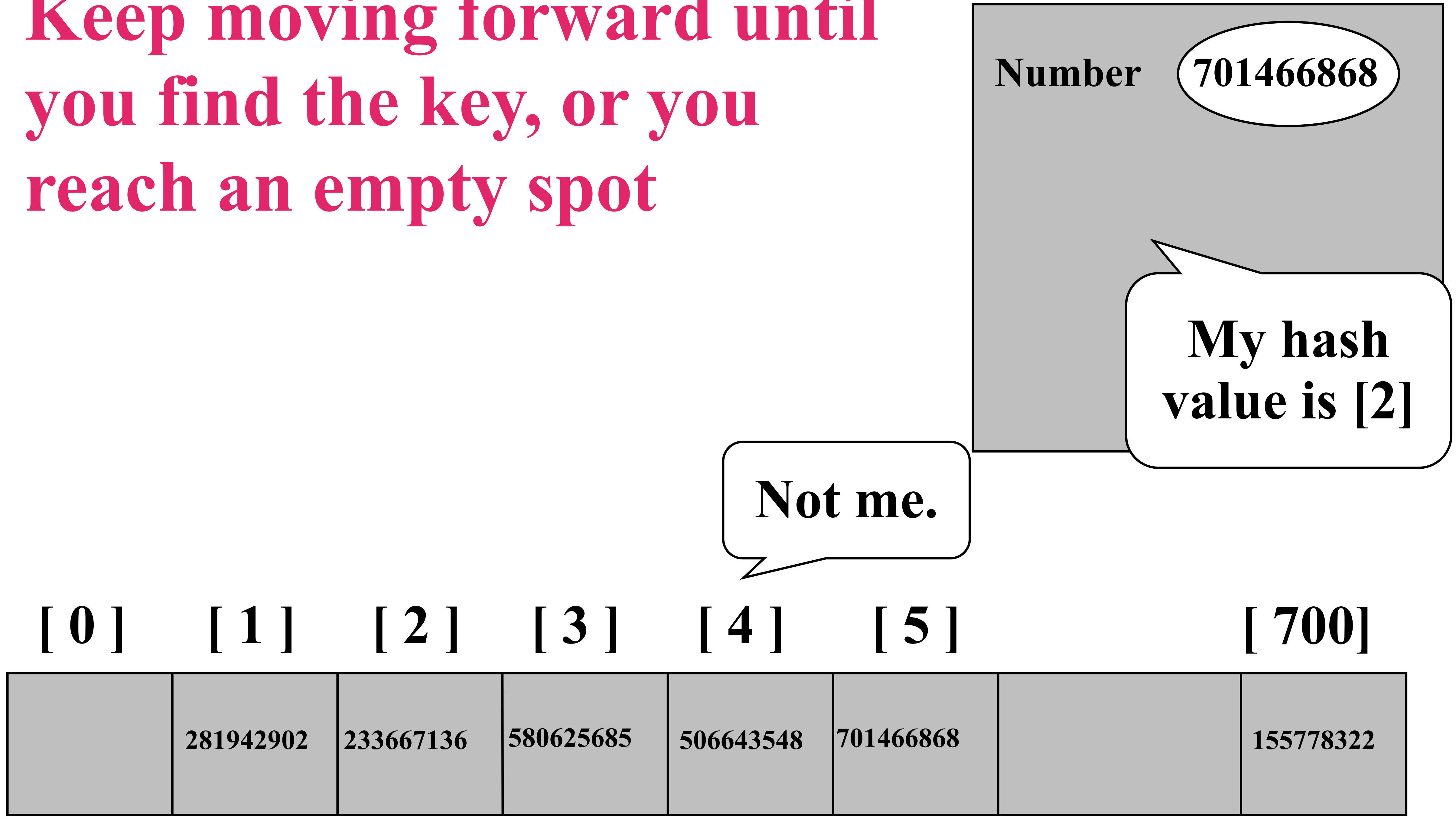
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot



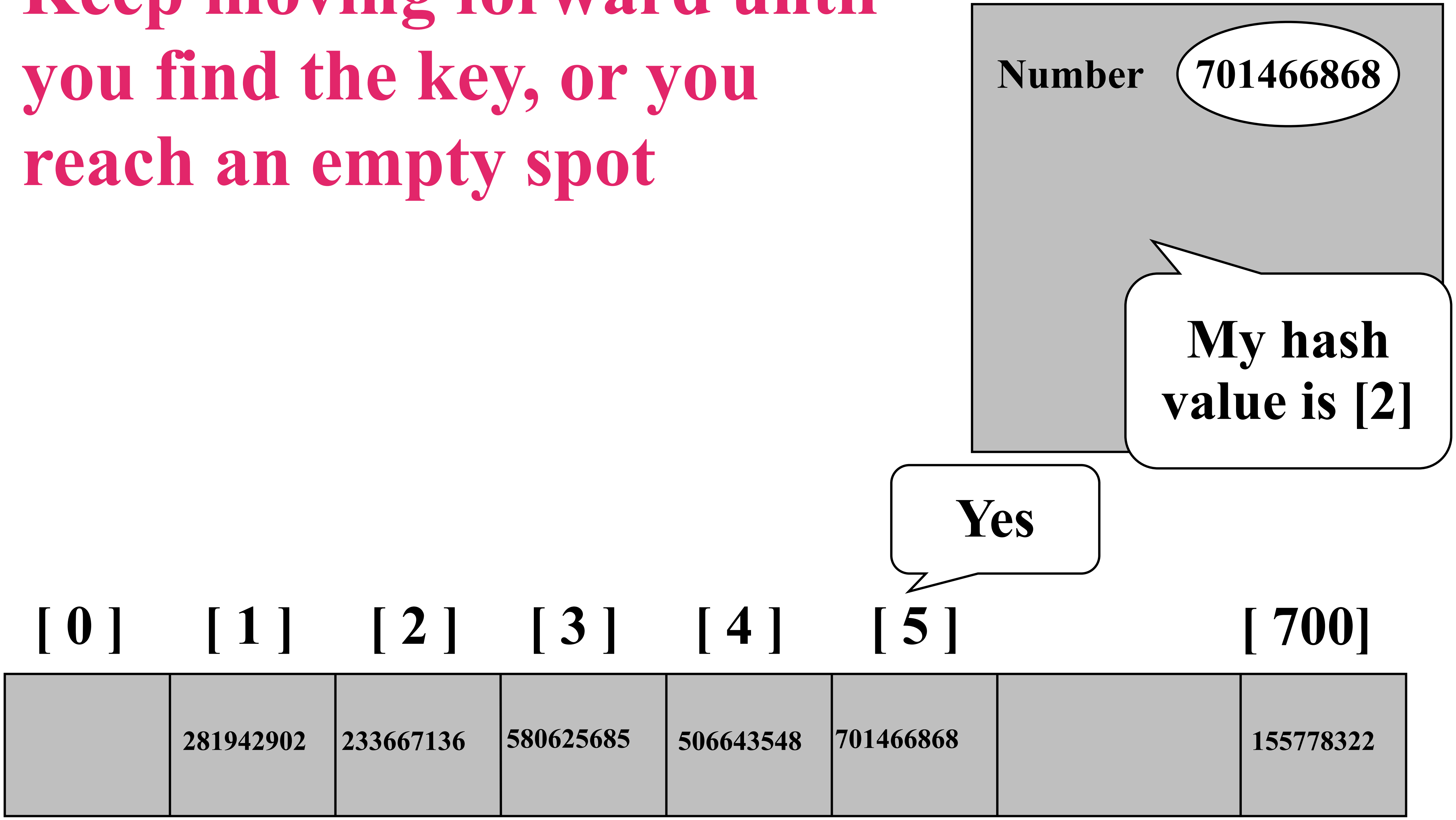
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot



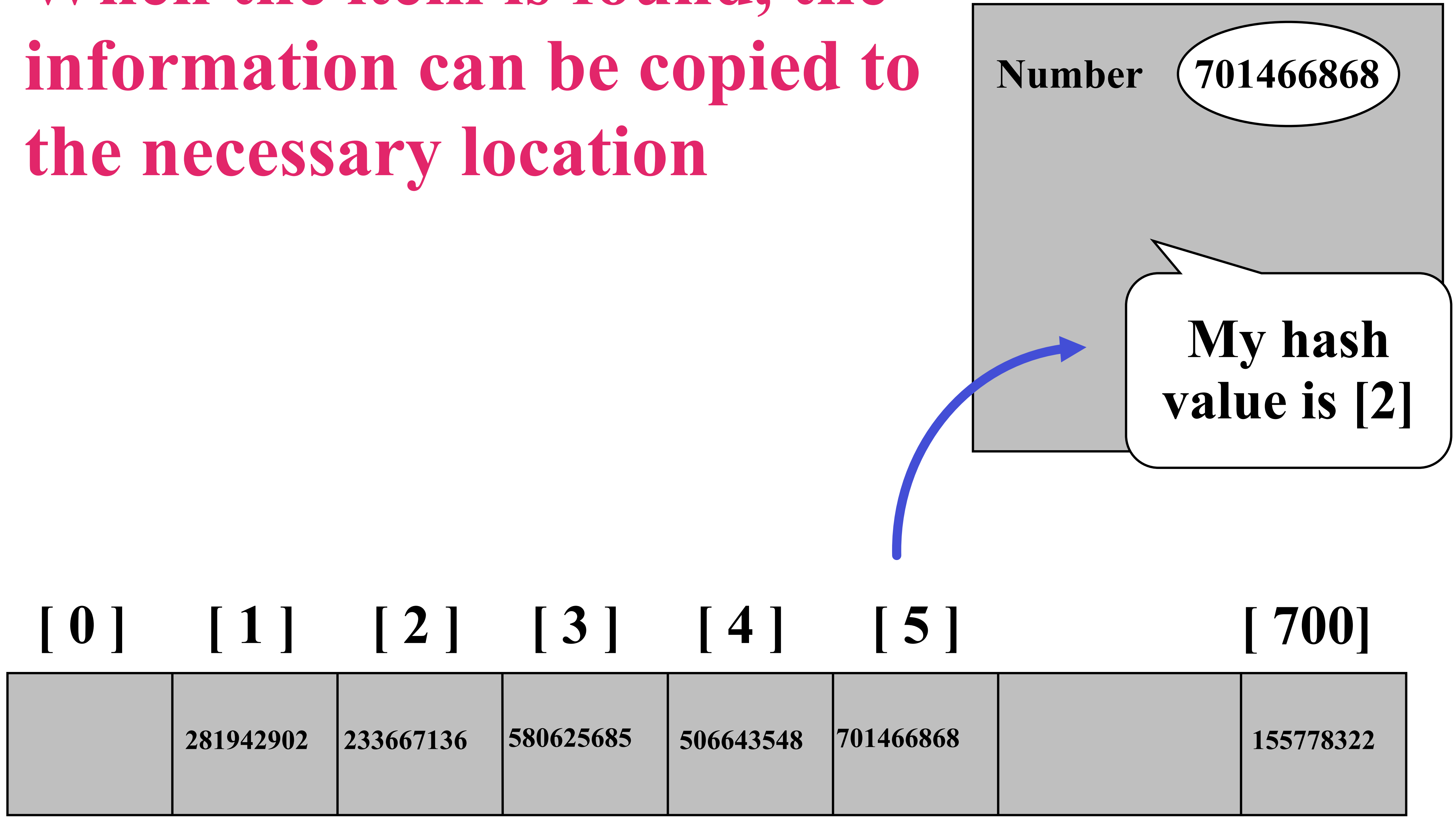
Searching for a Key

- Keep moving forward until you find the key, or you reach an empty spot



Searching for a Key

- When the item is found, the information can be copied to the necessary location



Deleting a Record

- Records may be deleted from a hash table, but the location must not be left as an ordinary "empty spot" since that could interfere with searches,
- The location must be marked so that a search can tell that the spot used to have something in it (i.e. the item was deleted)

[0]	[1]	[2]	[3]	[4]	[5]		[700]
	281942902	233667136	580625685	▨	701466868		155778322

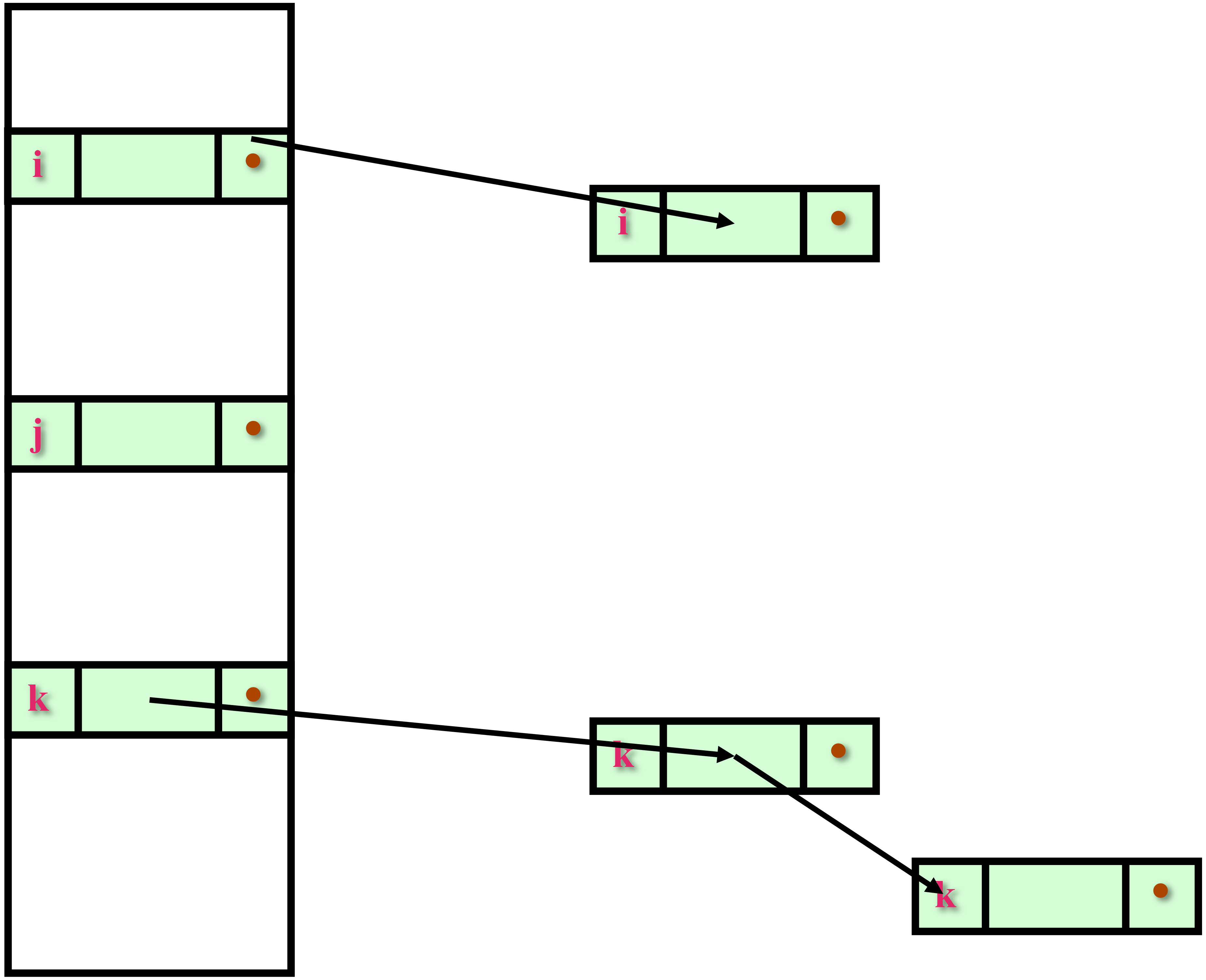
Non-Linear Probing

- Other probing methods can be used to reduce the clustering and to speed up the average search performance, so instead of:
  - inc(i) = i + l mod MaxItems,
  - i = 1, 2, 3, …, etc.
- We get the non-linear hashing:
  - inc(i, p) = (i + p) mod MaxItems,
  - Where p is the variable step to move each time
- Or quadratic hashing:
  - inc (i, p) = (i + ap + bp^2) mod MaxItems,
  - where a, b are constants

Resolving Collisions by Chaining

- One simple scheme is to chain all collisions in lists attached to the appropriate slot,
- This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection,
- The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent

Resolving Collisions by Chaining



Resolving Collisions using Overflow Area

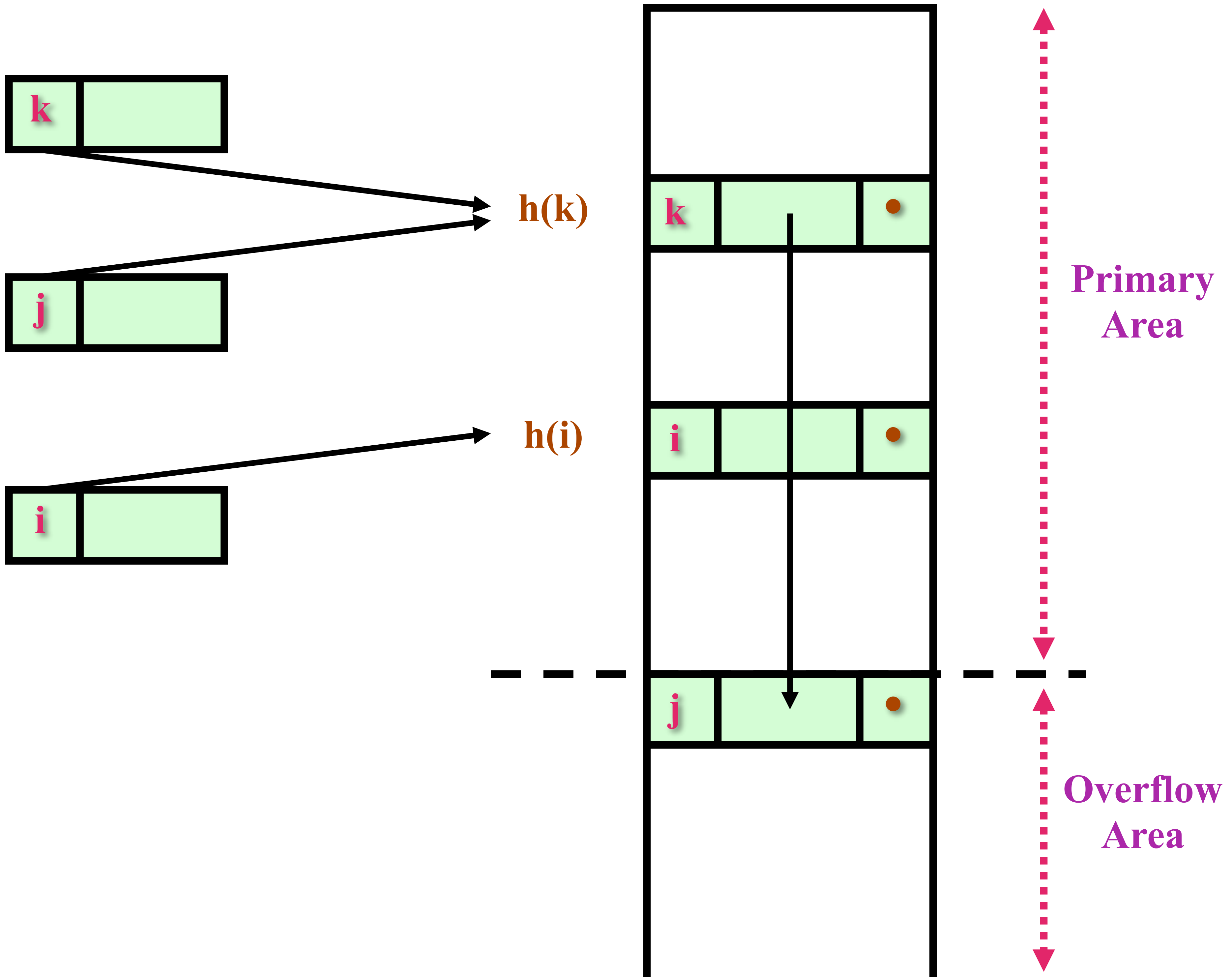
- Another scheme will divide the pre-allocated table into two sections:
  - the primary area to which keys are mapped, and
  - an area for collisions, normally termed the overflow area
- When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system,

Resolving Collisions using Overflow Area

- This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access,

- As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

Resolving Collisions using Overflow Area



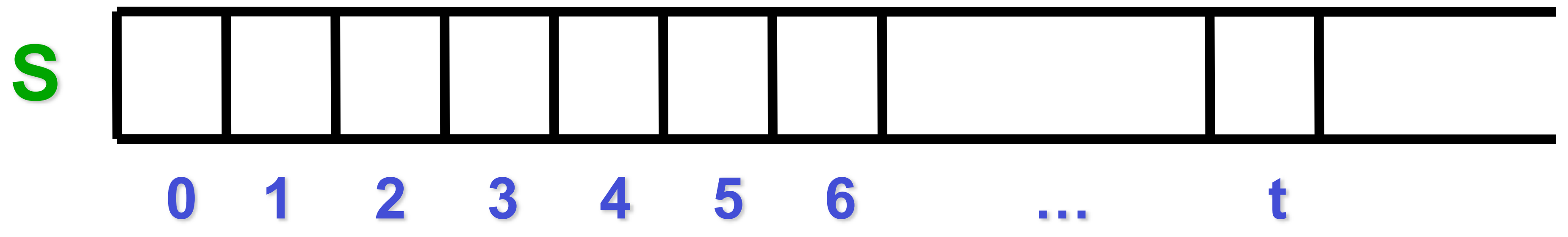
Part # 4

- **Stacks, Queues, and Priority Queues**

1. Basic Concepts of Stacks,
2. Implementation of Stacks ADT,
3. Applications,
4. Basic Concepts of Queues,
5. Implementation of Queues ADT,
6. Priority Queues,
7. Applications

Stacks

- A stack is a linear data structure which can be accessed only at one of its end for storing and retrieving data,



- For that reason, a stack is called Last In First Out structure (LIFO),
- A simple way of implementing the stack ADT is by using an array S and elements are added from left to right (Static Stack)

Static Stacks

- A variable *t* keeps track of the index of the top element,
- The array storing the stack elements may thus become full,
- The main stack operations are:
  - push(object): inserts an element,
  - object pop(): removes and returns the last inserted element,
  - object top(): returns the last inserted element without removing it,
  - integer size(): returns the number of elements stored, and
  - boolean isEmpty(): indicates whether no elements are stored

The Stack ADT

```
public class Stack {
   public static void main( String argv[]) {   …   }
   private int  t, MAX_STACK_SIZE=100;
   private int [] S;
   public Stack() { t = -1;  S = new int [MAX_STACK_SIZE];}
   public boolean isEmpty() { return (t == -1); }
   public int size() { return (t + 1); }
   public void push(int el) {
            if (t == S.length - 1) { System.out.println("Stack is Full"); }
            else { t = t + 1;    S[t] = el; }
   }
   public int pop() {
            if (isEmpty()) { System.out.println("Stack is Empty"); return -1; }
            else { t = t - 1; return S[t + 1]; }
   }
   public int top() {
            if (isEmpty()) { System.out.println("Stack is Empty"); return -1; }
            else { return S[t]; }
   }
}
```

Testing the Static Stack ADT

```
public static void main( String argv[]) {
      Stack S1 = new Stack();
            System.out.println("Initial stack status: "+S1.isEmpty());
            S1.push(10);
            S1.push(20);
            S1.push(30);
            S1.push(40);
            S1.push(50);
            System.out.println("Stack status: "+S1.isEmpty());
            System.out.println("Stack size is: "+S1.size());
            System.out.println("Top element of stack is: "+S1.top());
            System.out.println("Pop from stack: "+S1.pop());
            System.out.println("Stack status: "+S1.isEmpty());
            System.out.println("Stack size is: "+S1.size());
  }
```

```
Initial stack status: 1
Stack status: 0
Stack size is: 5
Top element of stack is: 50
Pop from stack: 50
Stack status: 0
Stack size is: 4
```

Dynamic Stacks

- A Stack:
  - Is considered as a constrained version of a singly linked list,
  - Add and remove nodes only to and from the top of the stack,
  - Push method adds node to top of stack,
  - Pop method removes node from top of stack

The Dynamic Stack ADT

```
public class DynamicStack extends SinglyLinkedList {
    public static void main( String argv[]) {   …   }
    private int count;
    public DynamicStack() { count = -1; }
    public boolean isEmpty() { return (count == -1); }
    public int size() { return (count + 1); }
    public void push(int el) { addToHead(el);   ++count; }
    public int pop() {
        if (isEmpty()) { System.out.println("Stack is Empty"); return -1; }
        else {  --count;  return deleteFromHead(); }
    }
    public int top() {
        if (isEmpty()) { System.out.println("Stack is Empty"); return -1; }
        else {
            int tmp = deleteFromHead();
            addToHead(tmp);
            return tmp;
        }
    }
}
```

Testing the Dynamic Stack ADT

```
public static void main( String argv[]) {
    DynamicStack S1 = new DynamicStack();
        System.out.println("Initial stack status: "+S1.isEmpty());
        S1.push(10);
        S1.push(20);
        S1.push(30);
        S1.push(40);
        S1.push(50);
        System.out.println("Stack status: "+S1.isEmpty());
        System.out.println("Stack size is: "+S1.size());
        System.out.println("Top element of stack is: "+S1.top());
        System.out.println("Pop from stack: "+S1.pop());
        System.out.println("Stack status: "+S1.isEmpty());
        System.out.println("Stack size is: "+S1.size());
 }
```

```
Initial stack status: 1
Stack status: 0
Stack size is: 5
Top element of stack is: 50
Pop from stack: 50
Stack status: 0
Stack size is: 4
```

Applications of Stacks

- Here are some example applications of the use of stacks:
  - Converting Infix expressions to Postfix,
  - Evaluation of Postfix expressions, and
  - Function calls and recursion
- The next few slides show a mapping of infix expression to postfix, an example of converting infix expression to postfix, and finally the evaluation of a postfix expression

Mapping Infix to Postfix Expressions

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 –	(4 * 7) – 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Conversion of: w - 5.1 / sum * 2

Next Token	Action	Effect on operatorStack	Effect on postfix
`w`	Append `w` to `postfix`		`w`
`-`	The stack is empty Push – onto the stack	`-`	`w`
`5.1`	Append 5.1 to `postfix`	`-`	`w 5.1`
`/`	`precedence(/) > precedence(-),` Push / onto the stack	`/` `-`	`w 5.1`
`sum`	Append `sum` to `postfix`	`/` `-`	`w 5.1 sum`
`*`	`precedence(*)` equals `precedence(/)` Pop / off of stack and append to postfix	`-`	`w 5.1 sum /`
`*`	`precedence(*) > precedence(-),` Push * onto the stack	`*` `-`	`w 5.1 sum /`
`2`	Append 2 to `postfix`	`*` `-`	`w 5.1 sum / 2`
End of input	Stack is not empty, Pop * off the stack and append to `postfix`	`-`	`w 5.1 sum / 2 *`
End of input	Stack is not empty, Pop – off the stack and append to `postfix`		`w 5.1 sum / 2 * -`

Evaluating a Postfix Expression

Expression	Action	Stack
$\underset{\uparrow}{4}$ 7 * 20 –	Push 4	$\begin{array}{\|c\|}4\\\hline\end{array}$
4 $\underset{\uparrow}{7}$ * 20 –	Push 7	$\begin{array}{\|c\|}7\\4\end{array}$
4 7 $\underset{\uparrow}{*}$ 20 –	Pop 7 and 4 Evaluate 4 * 7 Push 28	$\begin{array}{\|c\|}28\\\hline\end{array}$
4 7 * $\underset{\uparrow}{20}$ –	Push 20	$\begin{array}{\|c\|}20\\28\end{array}$
4 7 * 20 $\underset{\uparrow}{–}$	Pop 20 and 28 Evaluate 28 – 20 Push 8	$\begin{array}{\|c\|}8\\\hline\end{array}$
4 7 * 20 – $\uparrow$	Pop 8 Stack is empty Result is 8	$\begin{array}{\|c\|}\ \\\hline\end{array}$

Queues

- A Queue is a linear data structure similar to stack, except that the first item to be inserted is the first one to be removed that is First-In-First-Out (FIFO),
- Placing an item in a queue is called "enqueue", which is done at the end of the queue called "rear",
- Removing an item from a queue is called "dequeue", which is done at the other end of the queue called "front",

Queues

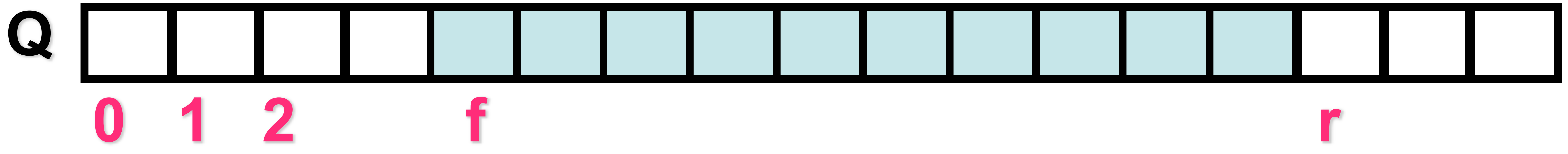
- Main queue operations:
  - enqueue(object): inserts an object at the end of the queue,
  - object dequeue(): removes and returns the object at the front of the queue,
  - object front(): returns the element at the front without removing it,
  - integer size(): return the number of elements stored, and
  - boolean isEmpty(): indicates whether no elements are stored

Static Circular Queues

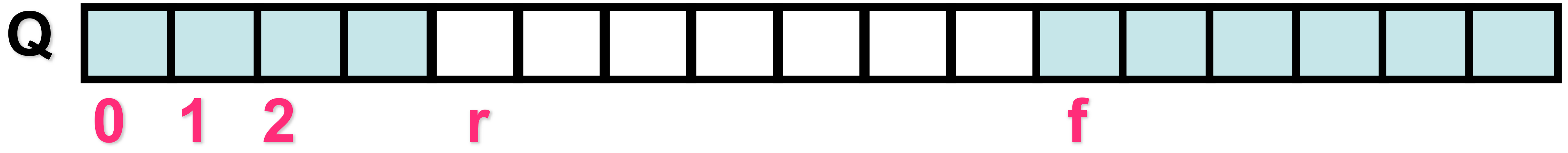
- Use an array of size N in a circular fashion,
- Two variables keep track of the front and rear:
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty,
- The modulo operator (remainder of division) is used

Static Circular Queues

Normal Configuration



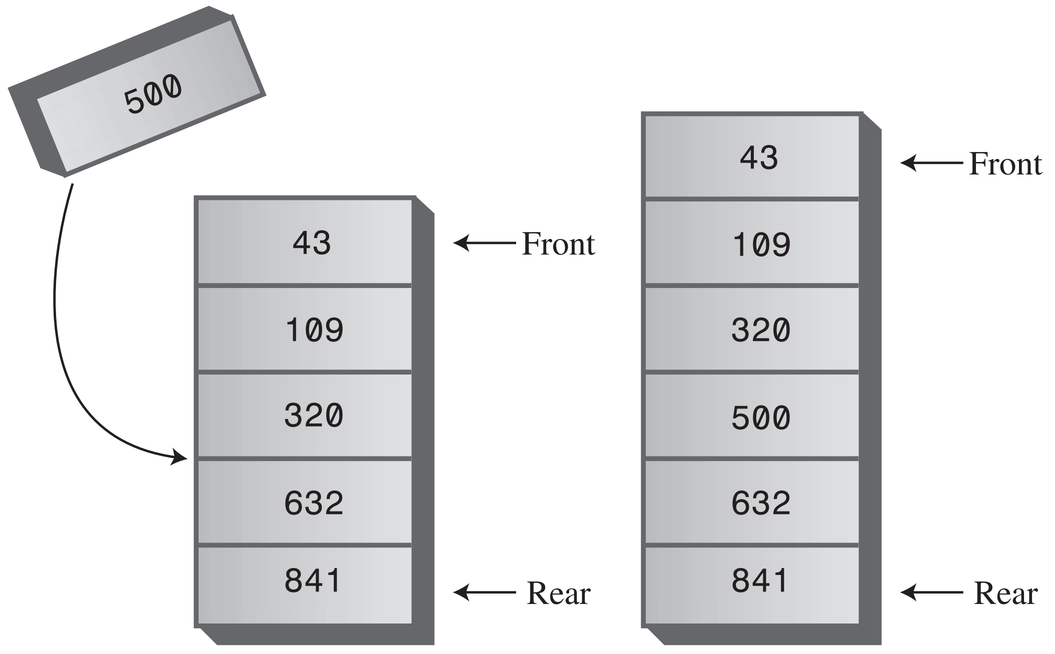
Wrapped-Around Configuration



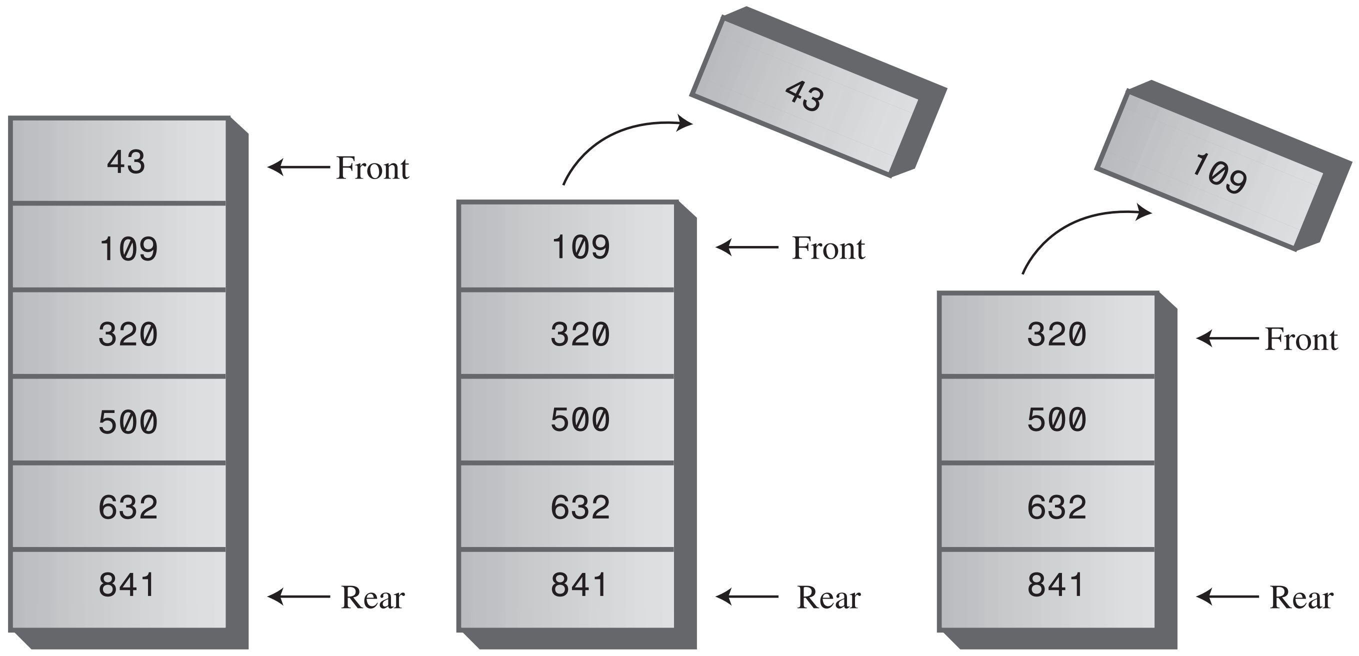
Priority Queues

- A priority queue is a more specialised data structure than a stack or a queue,
- Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front,
- However, in a priority queue, items are ordered by key value so that the item with the lowest key (or in some implementations the highest key) is always at the front

Operations of the Priority Queues



New item inserted in priority queue



Two items removed from front of priority queue

Operations of the Priority Queues

- **Main priority queue operations:**
  - **insert(object): inserts an object and shifting up all items with higher priority,**
  - **object delete(): removes and returns the object at the front of the queue,**
  - **boolean isFull(): indicates whether the queue is full or not,**
  - **integer size(): return the number of elements stored, and**
  - **boolean isEmpty(): indicates whether no elements are stored**

Applications

- Here are some example applications of the use of queues:
  - printer queue (operating systems),
  - keystroke queues (system software), and
  - waiting lists (service applications)

Applications

- Like stacks and queues, priority queues are often used as programmer’s tools,
- Examples in finding something called a minimum spanning tree for a graph,
- Also, like ordinary queues, priority queues are used in various ways in certain computer systems such as a preemptive multitasking operating system, for example, where programs may be placed in a priority queue so the highest-priority program is the next one to receive a time-slice that allows it to execute

Part # 7

- **Binary Trees and Recursion**

1. Introduction,
2. Binary Tree Traversals,
3. Expressions Transformation,
4. Binary Tree Abstract Data Type,
5. Binary Search Trees (BST),
6. Inserting/Deleting Elements in a BST,

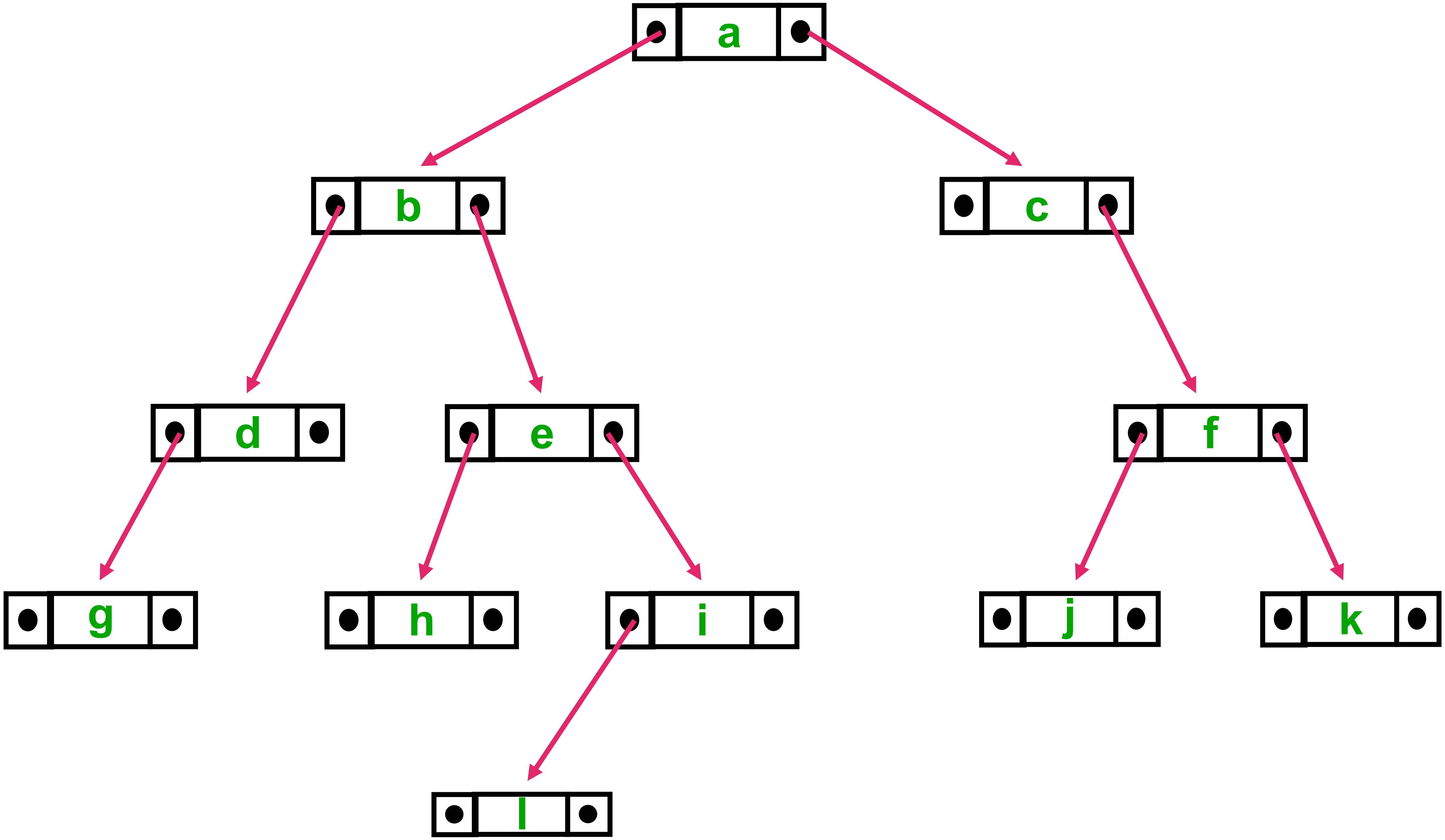
Introduction

- Here are some of the data structures we have studied so far:
  - Arrays,
  - Singly Linked Lists,
  - Doubly Linked Lists,
  - Circular Lists,
  - Stacks, Queues, and Priority Queues
- These all have the property that their elements can be displayed in a straight line,
- Binary trees are one of the simplest nonlinear data structures

Introduction

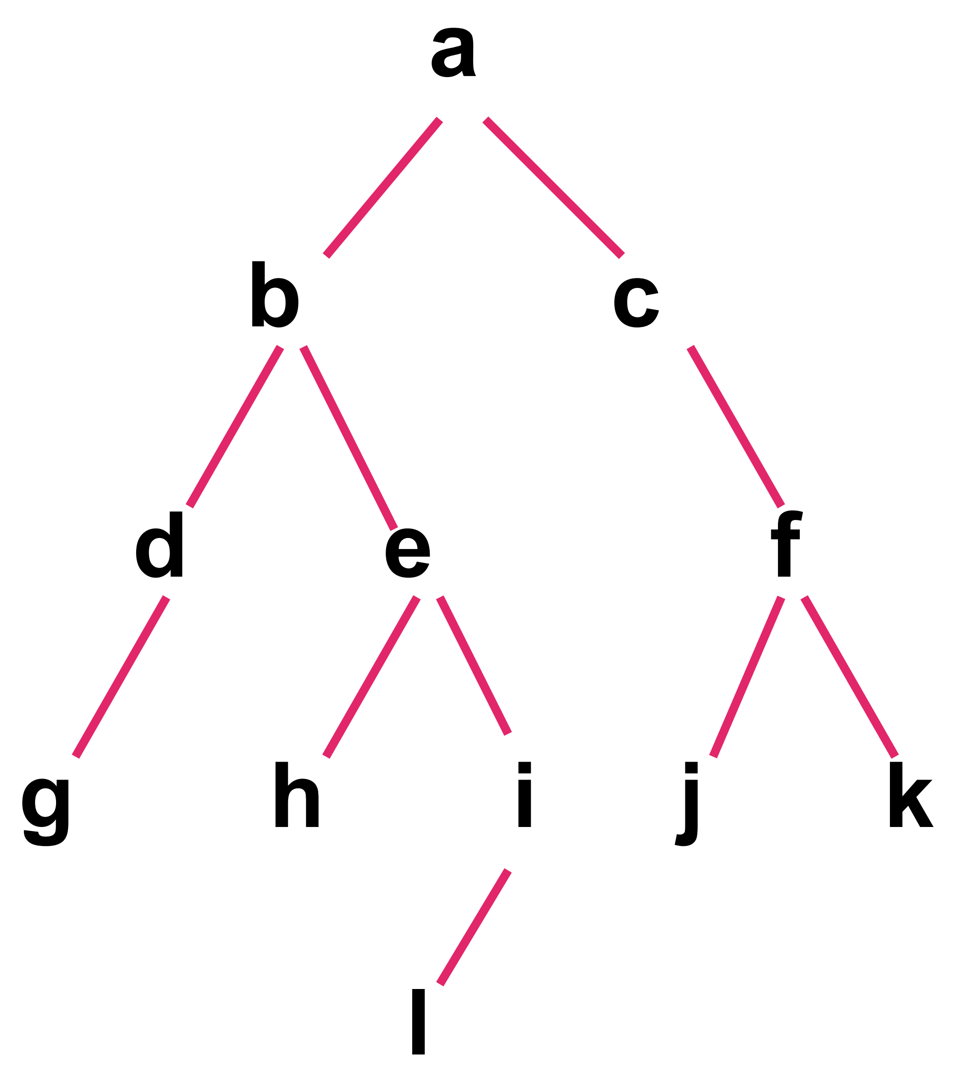
- A binary tree is composed of zero or more nodes, each contains:
  - A value (some sort of data item),
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be empty,
- If not empty, a binary tree has a root node
  - Every node in the binary tree is reachable from the root node by a unique path
- A node with neither a left child nor a right child is called a leaf,
- The next slide is an example of a binary tree for storing characters

Introduction



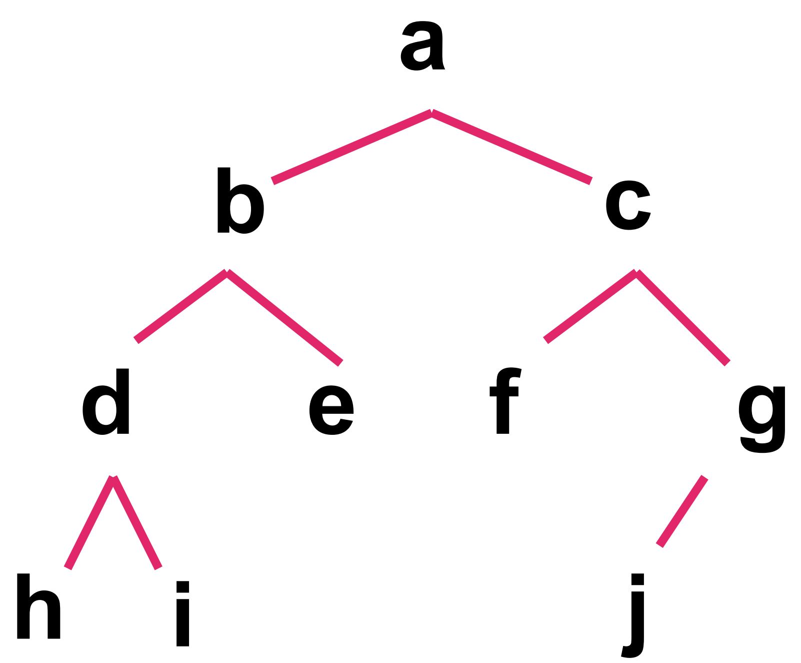
Size, Depth, and Height

- The size of a binary tree is the number of nodes in it
  - This tree has size 12
- The depth of a node is its distance from the root
  - a is at depth zero,
  - e is at depth 2
- The height of a binary tree is the depth of its deepest node
  - This tree has height 4

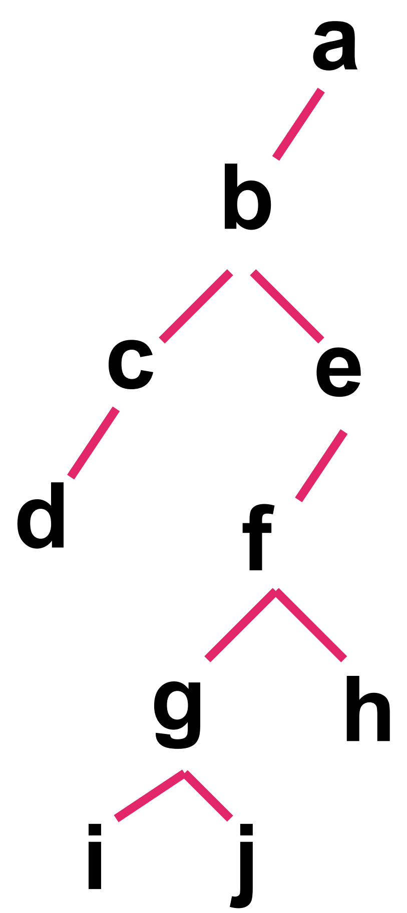


Balance

- A binary tree is balanced if every level above the lowest is "full" (contains $2^n - 1$ nodes)
- In most applications, a reasonably balanced binary tree is desirable



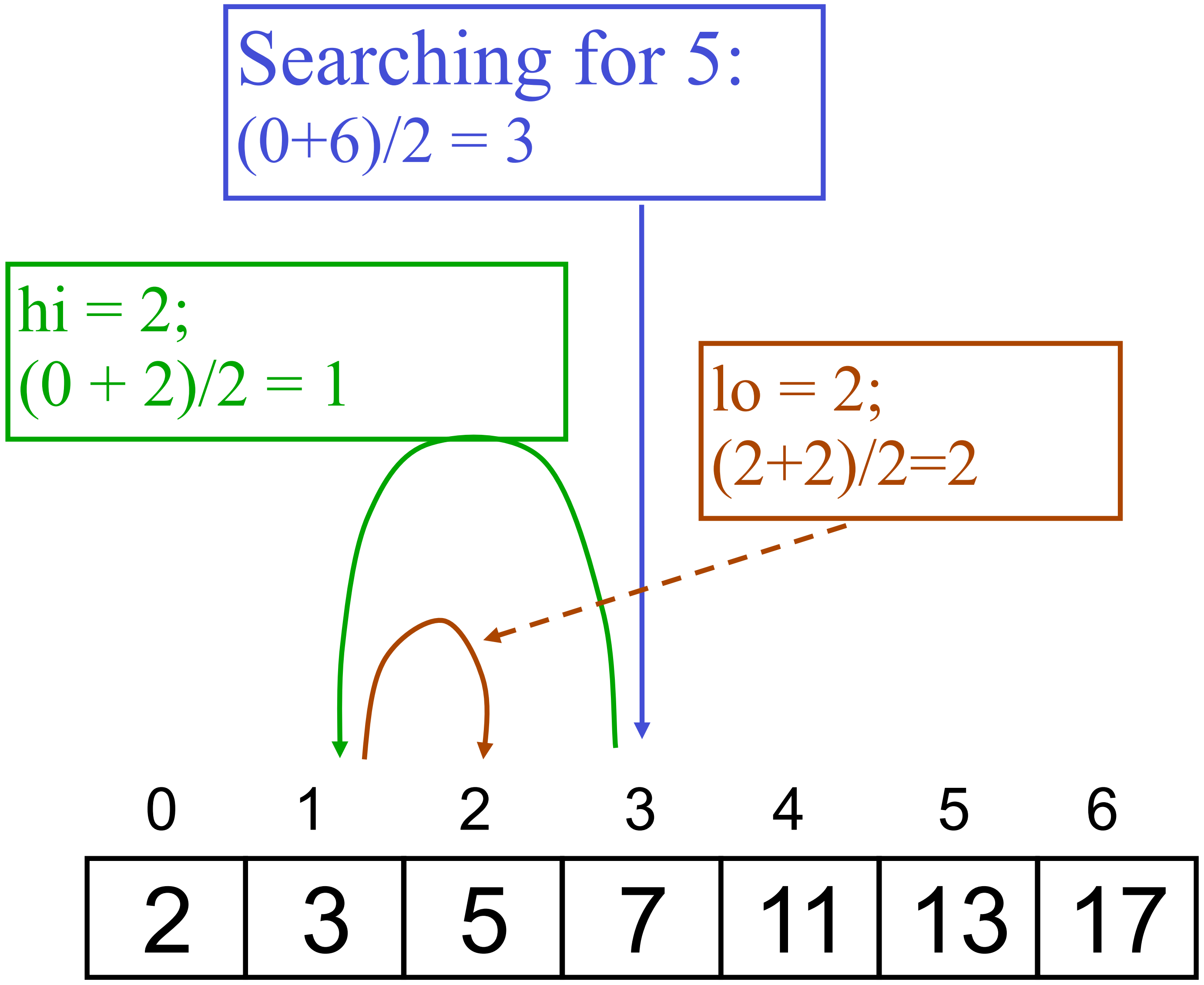
A balanced binary tree



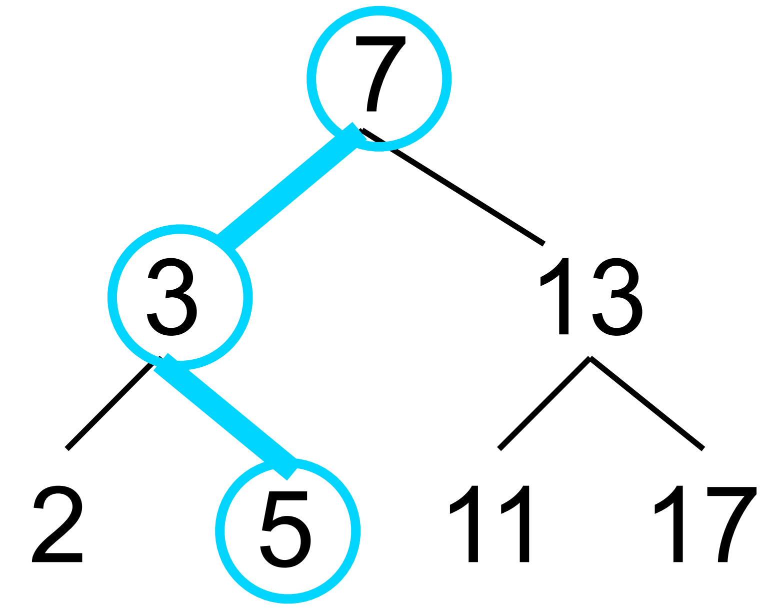
An unbalanced binary tree

Binary Search in an Array (Revisited)

- Look at array location (lo + hi)/2



Using a binary search tree



Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree,
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once,
- Tree traversals are naturally recursive,
- Since a binary tree has three "parts," there are six possible ways to traverse the tree:
  - root, left, right & root, right, left
  - left, root, right & right, root, left
  - left, right, root & right, left, root

Traversal Example

```
          (8)
         /   \
        /     \
     (2)      (21)
    /   \     /
 (1)   (5) (13)
        /
     (3)
```

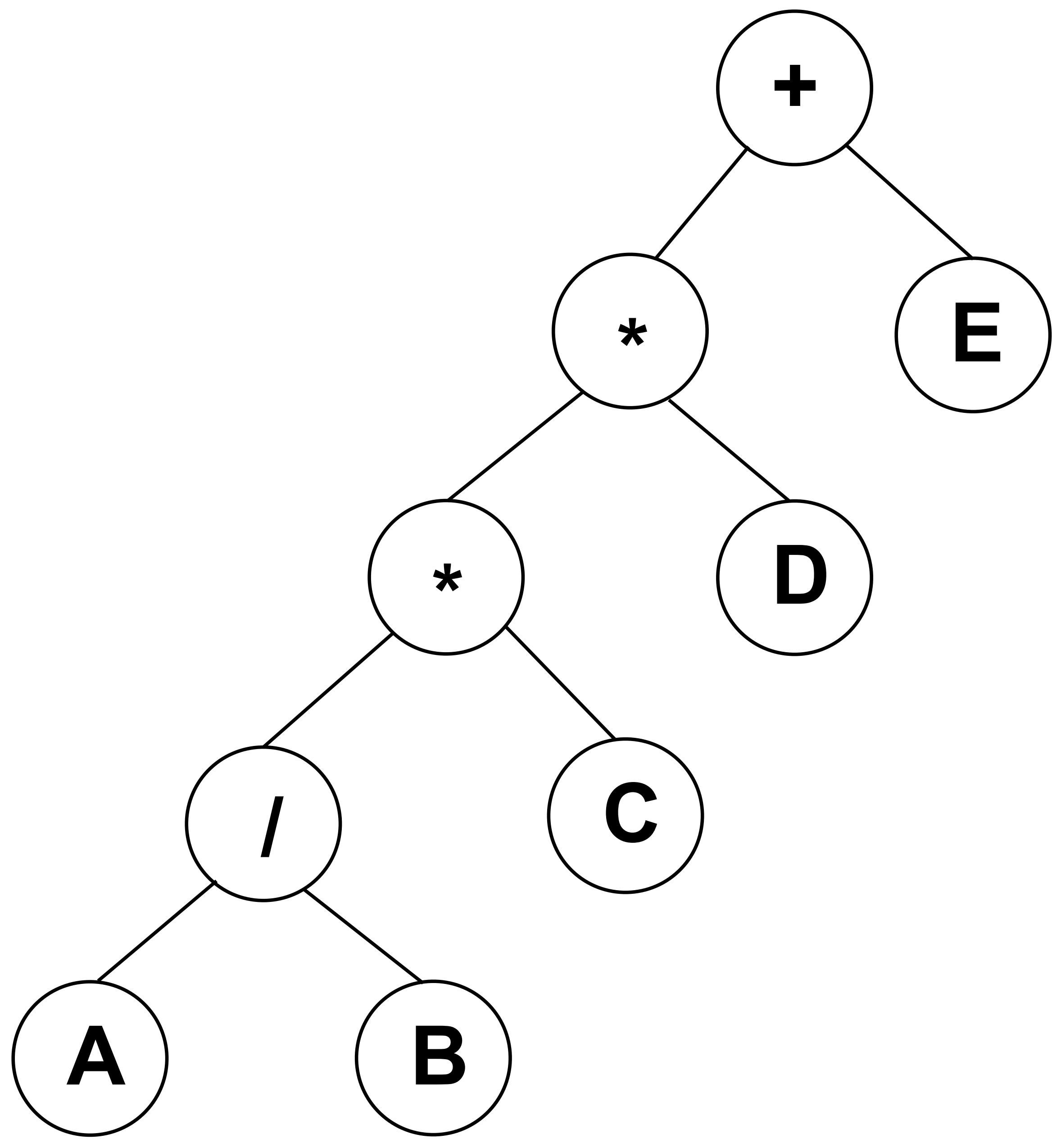
pre-order: 8, 2, 1, 5, 3, 21, 13
post-order: 1, 3, 5, 2, 13, 21, 8
in-order: 1, 2, 3, 5, 8, 13, 21

Traversal Example

inorder traversal
A / B * C * D + E
infix expression

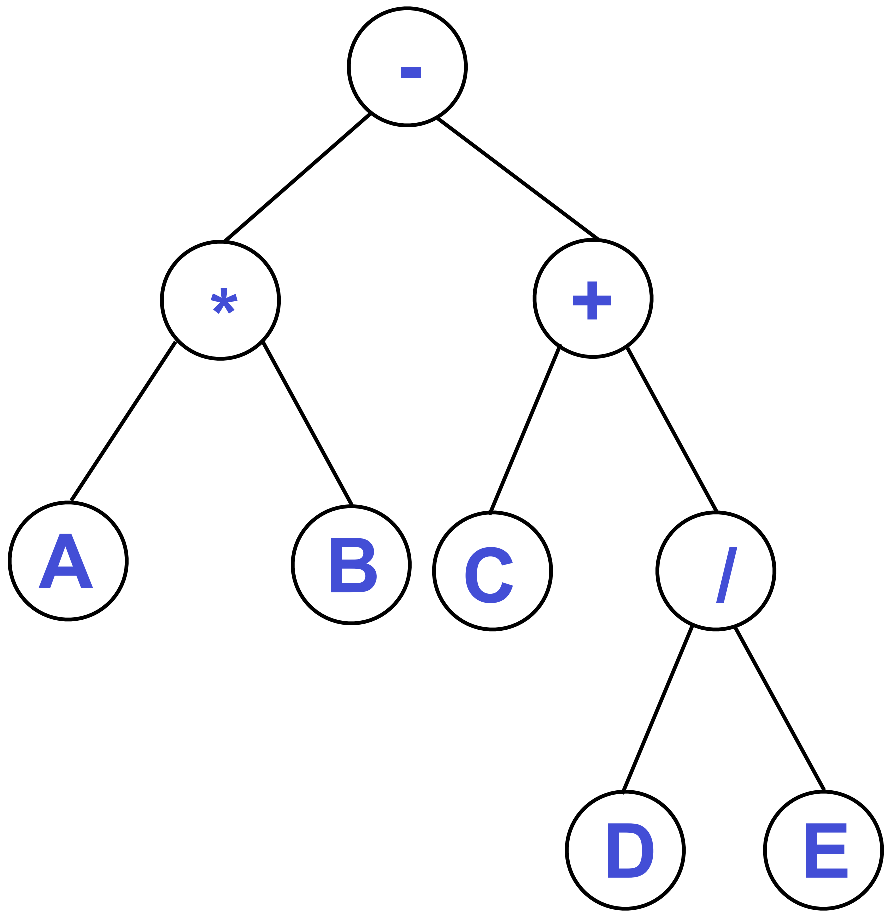
preorder traversal
+ * * / A B C D E
prefix expression

postorder traversal
A B / C * D * E +
postfix expression



The Transformation of Expressions

Given and Infix Expression: (A * B) – (C + (D / E))



Step 1:

Represent the expression as a Binary Tree with Leafs as variables and intermediate nodes as operators

Step 1:

Traverse the Tree

Pre-Order Traverse (prefix expression): -*AB+C/DE

Post-Order Traverse (postfix expression): AB*CDE/+-

The Binary Trees ADT

```
public class BinaryTree{
    public static void main(String argv[]) {  …  }

    class Node{
        Object info;  Node left;  Node right;
        Node() { info = null; left = null; right = null; }
        Node(Object el) { info = el; left = null; right = null; }
        Node(Object el, Node l, Node r) { info = el; left = l; right = r; }
    }

    private Node root;

    BinaryTree( ) { root = null; }

    public boolean isLeaf(Node node) {return ((node.left==null) && (node.right==null));}
    public void visit(Node node) { System.out.println(node.info); }
    public void inOrder( Node node ) {  …  }
    public void preOrder( Node node ) {  …  }
    public void postOrder( Node node) {  …  }
    public int max( int x, int y) { if ( x > y ) { return x; } else { return y; } }
    public int size( Node node ) {  …  }
    public int height( Node node ) {  …  }
    public Node search( Object key ) {  …  }
    public void insertNode( Object value ) {  …  }
}
```

Inorder Traversal

- In inorder, the root is visited in the middle,
- Here's an inorder traversal to visit the elements in the binary tree node:

```
public void inOrder( Node node ) {
  if (node.left != null) { inOrder(node.left); }
  visit(node);
  if (node.right != null) { inOrder(node.right); }
}
```

Preorder Traversal

- In preorder, the root is visited **first**,
- Here’s a preorder traversal to visit the elements in the binary tree node:

```
public void preOrder( Node node ) {
  visit(node);
  if (node.left != null) { preOrder(node.left); }
  if (node.right != null) { preOrder(node.right); }
}
```

Postorder Traversal

- In postorder, the root is visited **last**,
- Here’s a postorder traversal to visit all the elements in the binary tree pointed to by node:

```
public void postOrder( Node node) {
  if (node.left != null) { postOrder(node.left); }
  if (node.right != null) { postOrder(node.right); }
  visit(node);
}
```

Other Traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
    - Reverse preorder: root, right subtree, left subtree
    - Reverse inorder: right subtree, root, left subtree
    - Reverse postorder: right subtree, left subtree, root

The Binary Tree Size

- The size of a binary tree is the number of nodes in it,
- Here’s the method that calculates the size of the binary tree pointed to by node:

```
public int size( Node node ) {
  if ( node == null ) { return 0; }
  else if ( isLeaf(node) ) { return 1; }
  else if ( node.left == null ) { return (1+size(node.right)); }
  else if ( node.right == null) { return (1+size(node.left)); }
  else return (1 + size(node.left) + size(node.right));
}
```

The Binary Tree Height

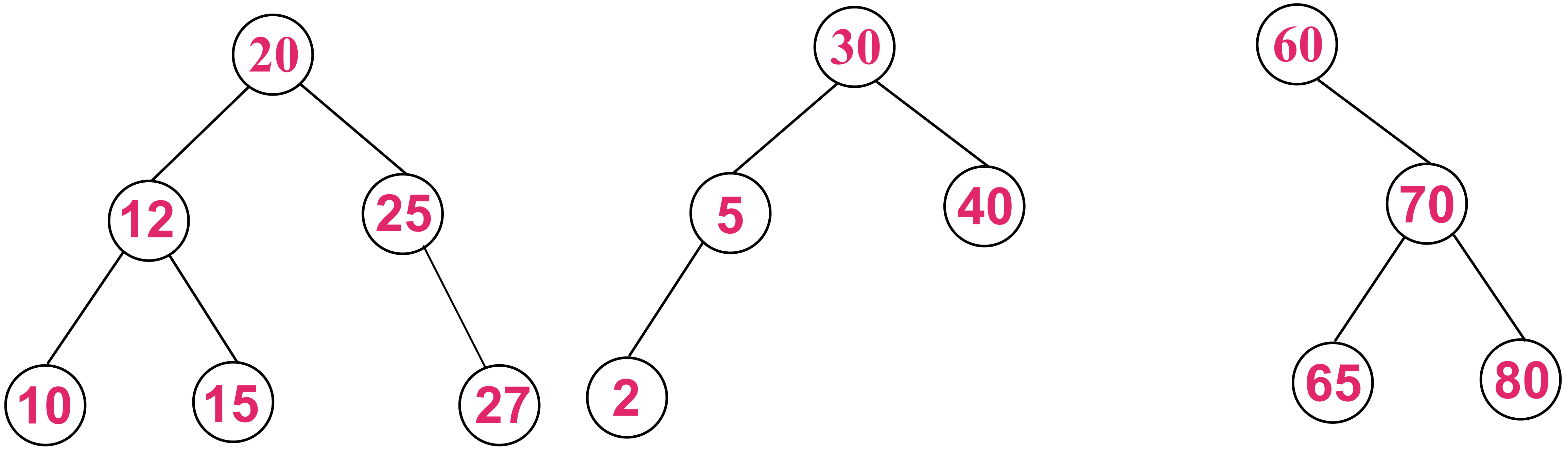
- The height of a binary tree is the depth of its deepest node,
- Here’s the method that calculates the height of the binary tree pointed to by node:

```
public int height( Node node ) {
    if ( node == null ) { return 0; }
    else if ( isLeaf(node) ) { return 0; }
    else if ( node.left == null ) { return (1+height(node.right)); }
    else if ( node.right == null) { return (1+height(node.left)); }
    else return (1+max(height(node.left), height(node.right)));
}
```

Binary Search Trees (BST)

- Every element in a binary search tree has a unique key,
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree,
- The left and right subtrees are also binary search trees,
- The next slide shows two different examples of binary search trees

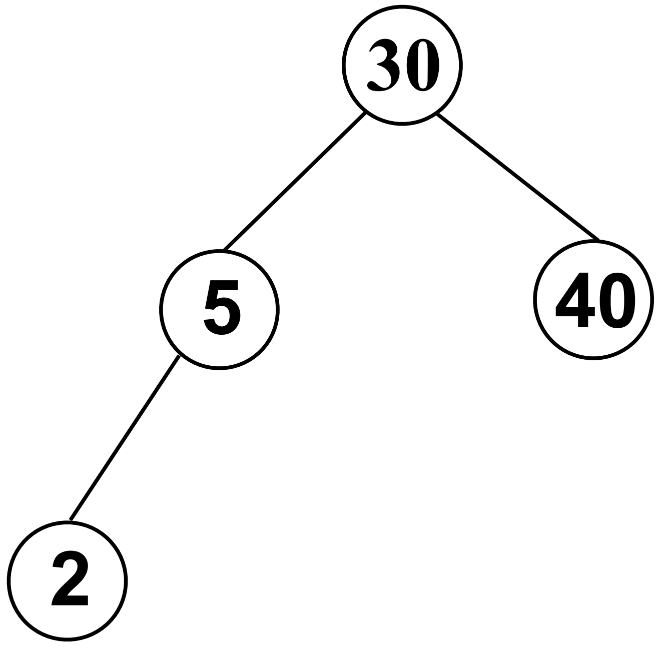
Examples of BST



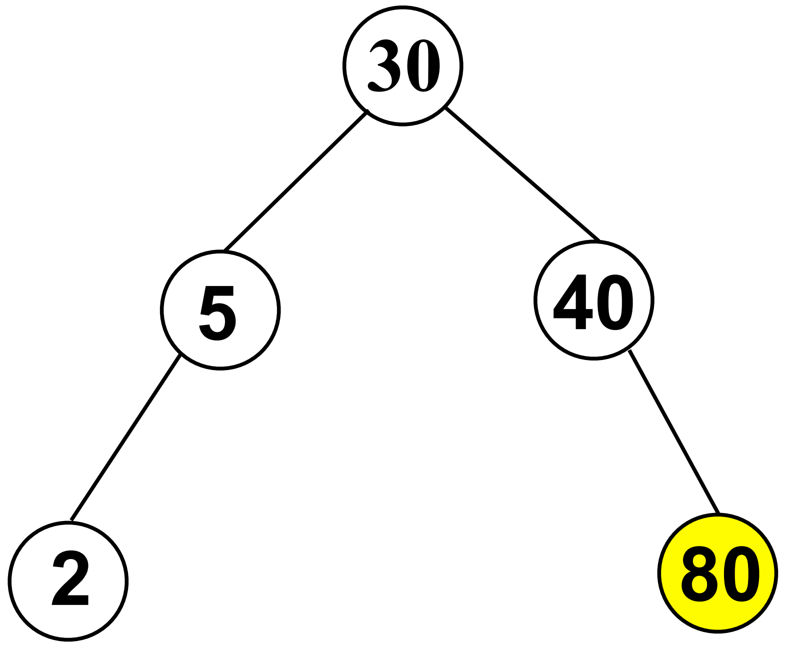
Searching in a BST

```
public Node search( Object key ) {
      Node temp = root;
      while ( temp != null ) {
          int comp = ((Comparable) key).compareTo(temp.info);
          if ( comp < 0 ) { temp = temp.left; }
          else if ( comp > 0 ) { temp = temp.right; }
          else { return temp; }
      }
      return null;
   }
```

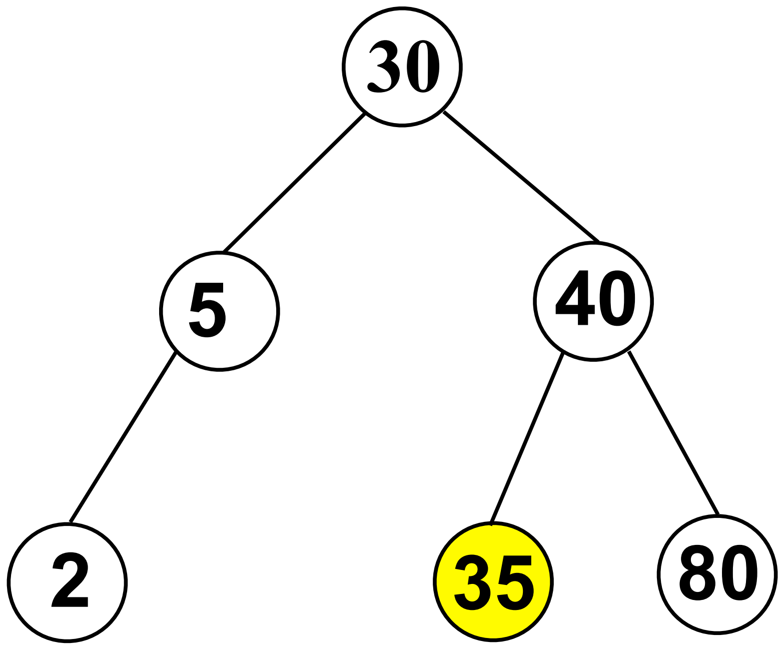
Insert Node in a BST



Original



Insert 80



Insert 35

Insert Node in a BST

```
public void insertNode( Object value ) {
   Node BT = new Node(value);
   Node temp = search(value);
   if ( temp == null ) {// not found in tree (assume unique keys)
            if ( root == null )  { root = BT; } // was empty tree
      else {
         temp = insertPoint(root, value);
         int comp = ((Comparable) value).compareTo(temp.info);
         if ( comp < 0 ) { temp.left = BT; }
         else { temp.right = BT; }
     }
   }
}
```

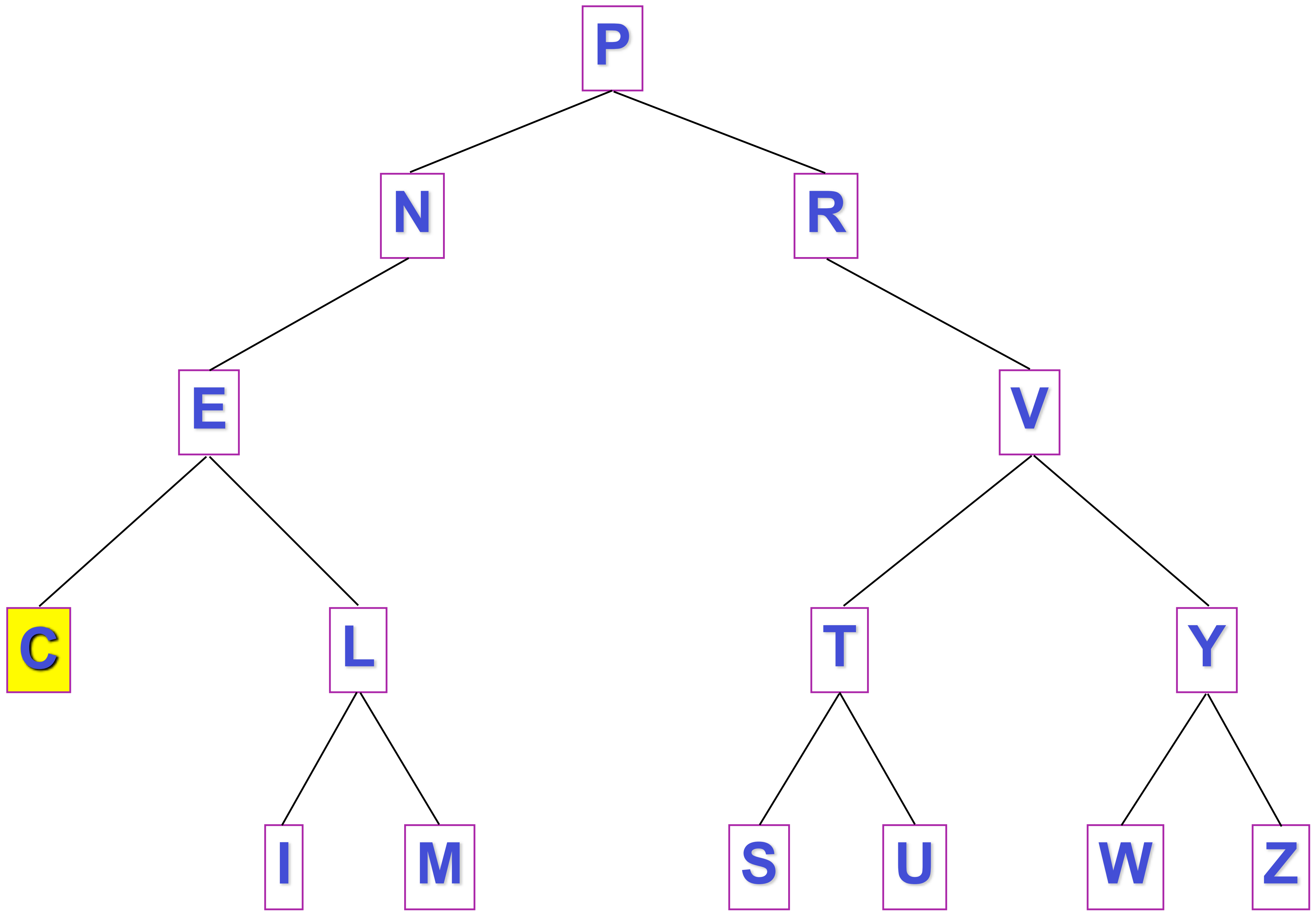
Insert Node in a BST

```
public Node insertPoint(Node node, Object key) {
    if ( isLeaf(node) ) return node;
    else {
        int comp = ((Comparable) key).compareTo(node.info);
        if ( ( comp < 0 ) && (node.left != null) ) {
            return insertPoint(node.left, key);
        }
        else if ( ( comp > 0 ) && (node.right != null)) {
            return insertPoint(node.right, key);
        }
        else { return node; }
    }
}
```

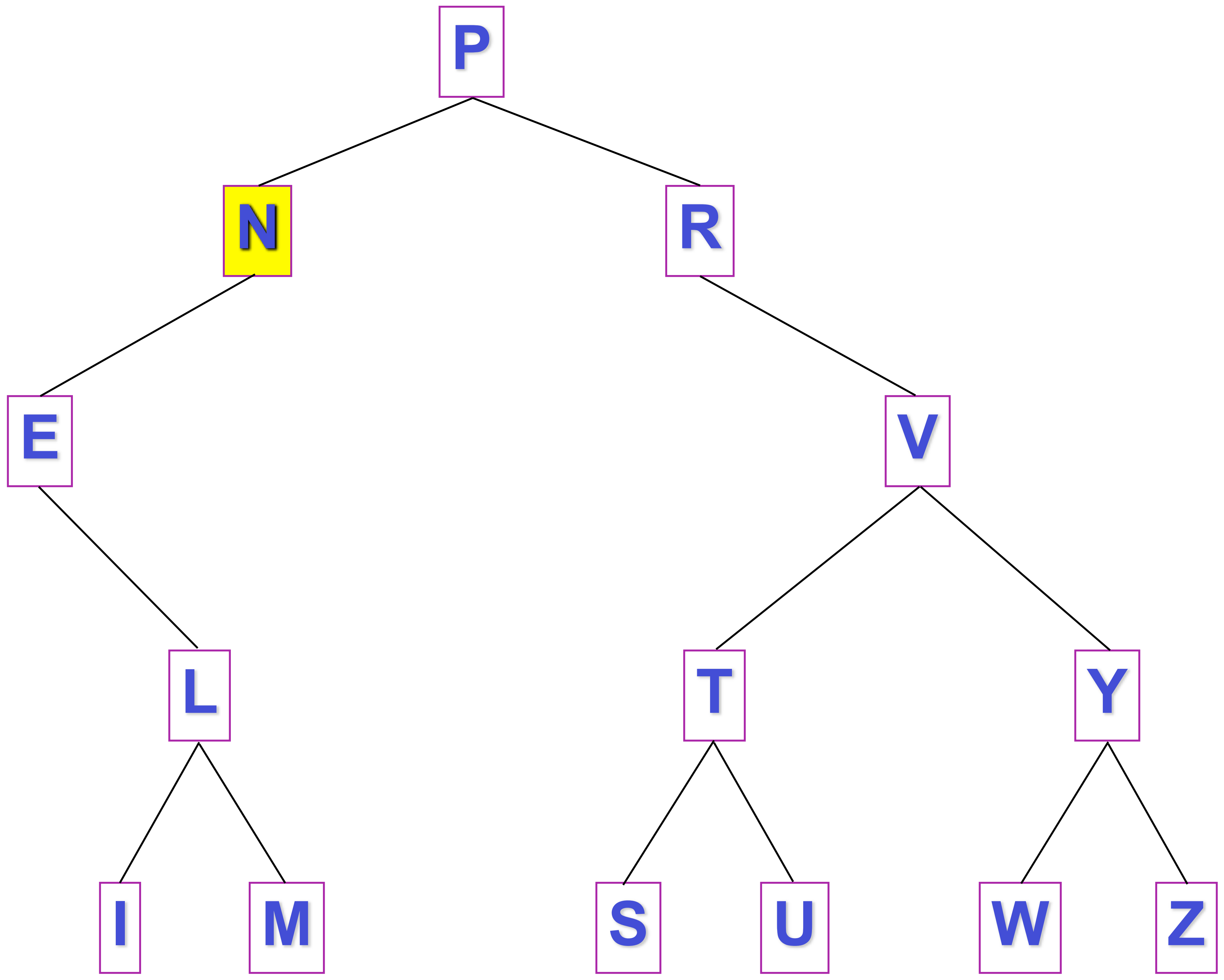
Deleting a Node from a BST

- To delete a node from a BST:
  1. Locate the desired node by a search; call it t,
  2. If t is a leaf, disconnect it from its parent,
  3. If t has a left child but no right child, remove t from the tree by making t's parent point to t's left child,
  4. If t has a right child but no left child, remove t from the tree by making t's parent point to t's right child,
  5. Otherwise, find the node in t's right subtree with the smallest key, copy this node's information into t; delete the node

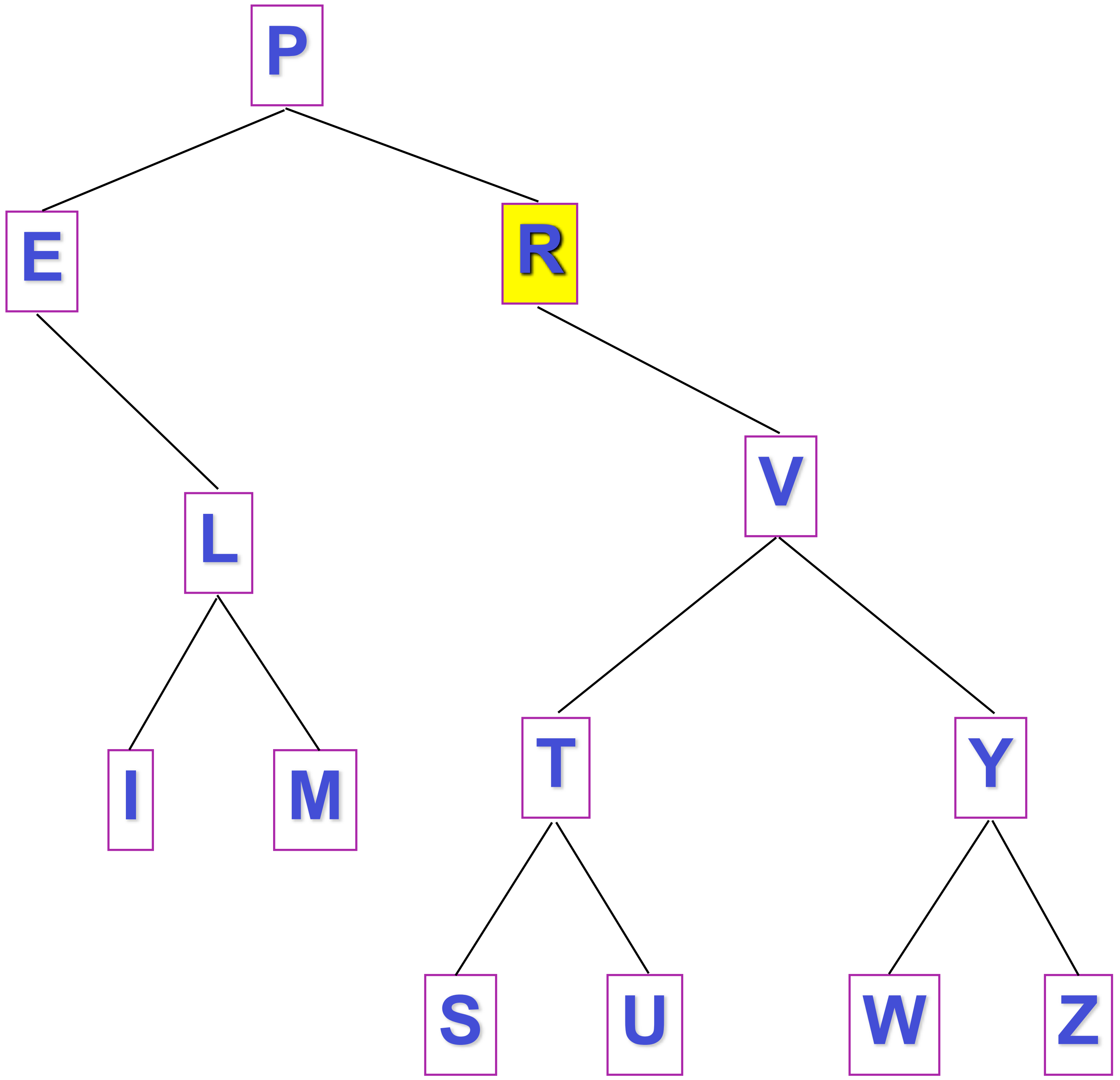
Example: The Original Tree



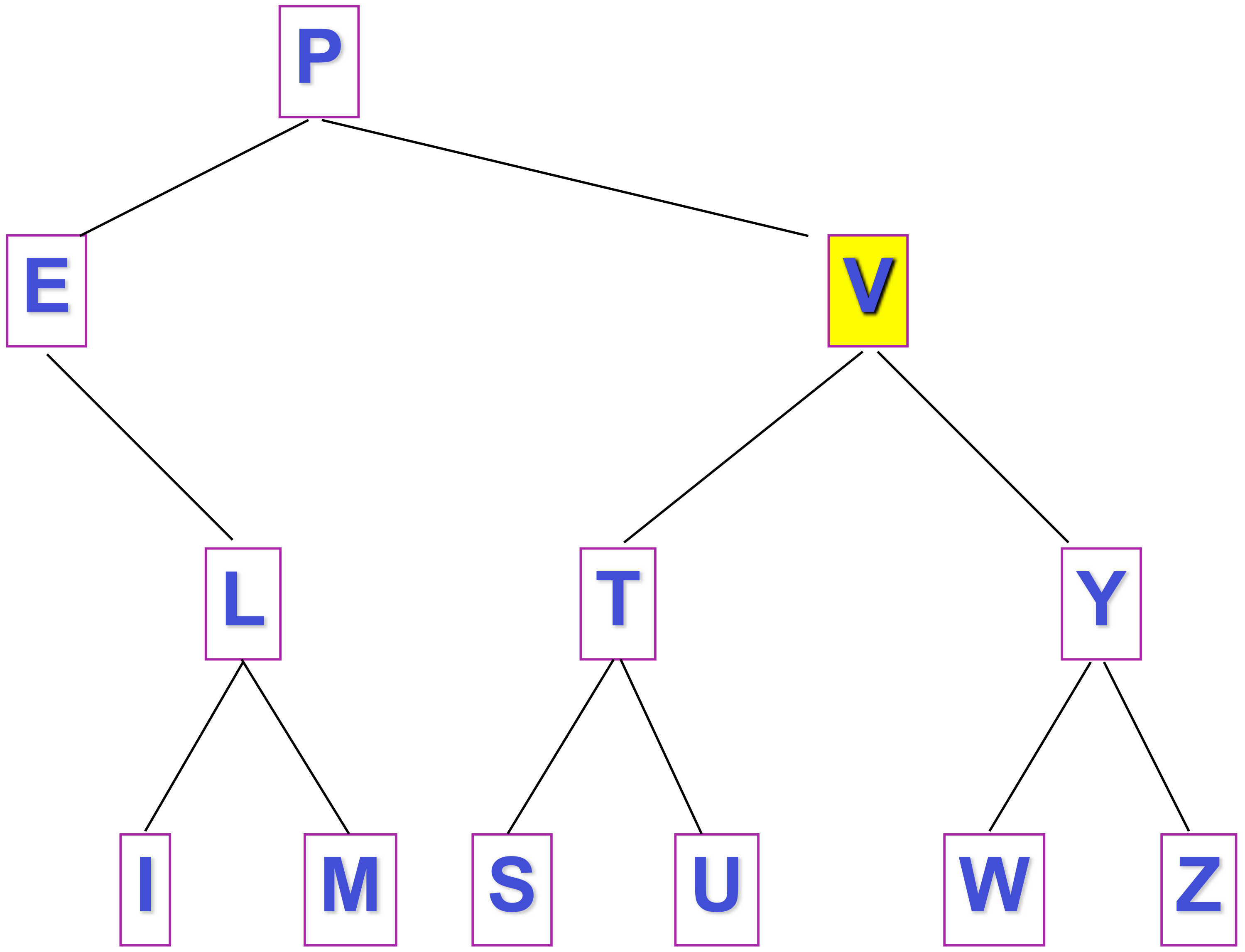
Deleting a Leaf ‘C’



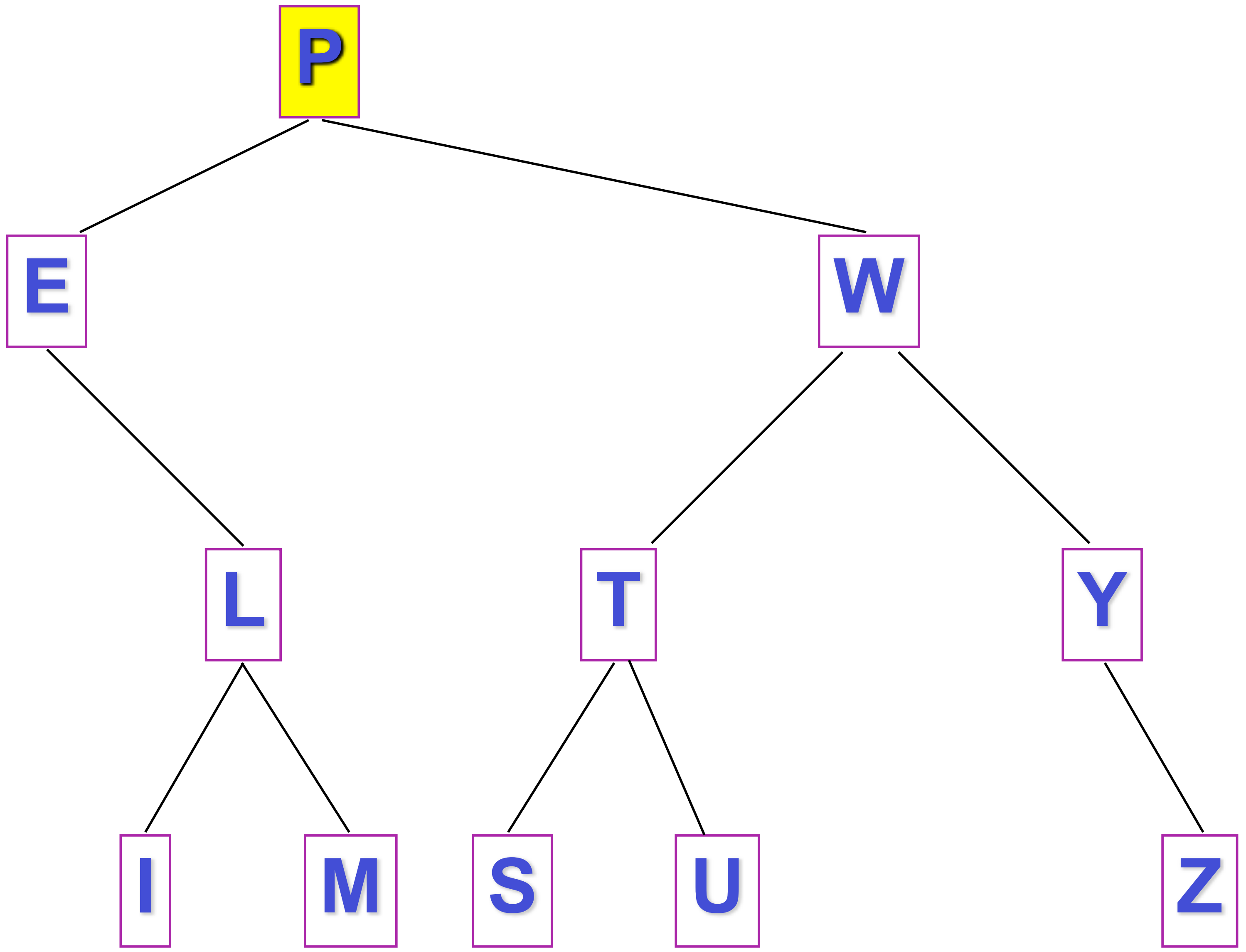
Deleting a Left Child Only ‘N’



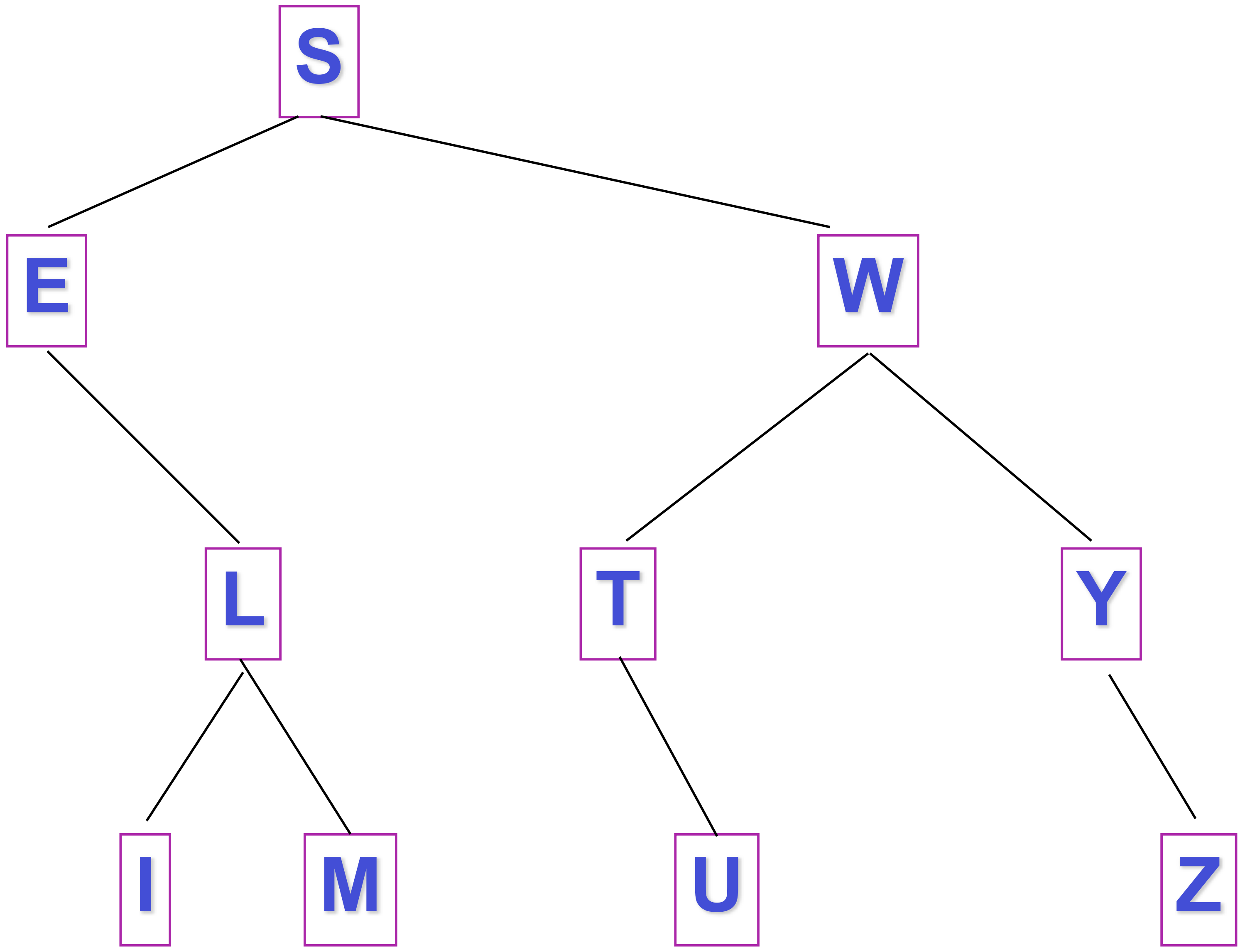
Deleting a Right Child Only ‘R’



Deleting a Both Children ‘V’



Deleting a Both Children ‘P’



Testing the Binary Tree ADT

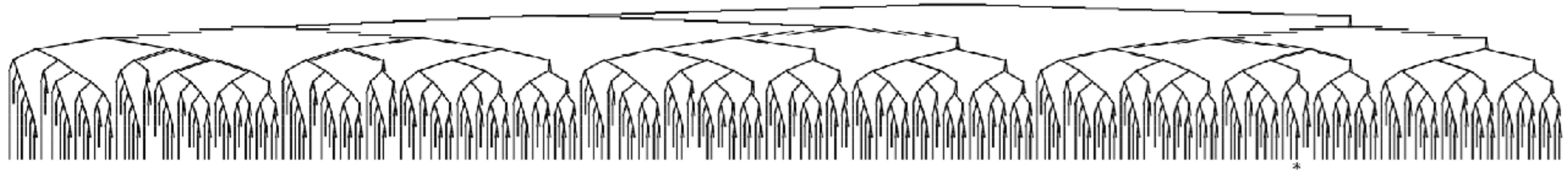
```
public static void main(String argv[]) {
    BinaryTree MyTree = new BinaryTree();
    System.out.println("Initially, the Binary Tree size is: "+MyTree.size(MyTree.root));
    MyTree.insertNode( new Integer(30) );
    MyTree.insertNode( new Integer(5) );
    MyTree.insertNode( new Integer(40) );
    MyTree.insertNode( new Integer(2) );
    MyTree.insertNode( new Integer(80) );
    MyTree.insertNode( new Integer(35) );
    System.out.println("The tree size is: "+MyTree.size(MyTree.root));
    System.out.println("The tree height is: "+MyTree.height(MyTree.root));
    System.out.println("Searching the Binary Tree for 40: ");
    if (MyTree.search(new Integer(40))==null) {System.out.println("Value 40 Not Found");}
    else { System.out.println("Value 40 Found"); }
    System.out.println("Searching the Binary Tree for 60: ");
    if (MyTree.search(new Integer(60))==null) {System.out.println("Value 60 Not Found");}
    else { System.out.println("Value 60 Found"); }
    System.out.println("In Order Traverse of the Binary Tree:");
    MyTree.inOrder(MyTree.root);
    System.out.println("Pre Order Traverse of the Binary Tree:");
    MyTree.preOrder(MyTree.root);
    System.out.println("Post Order Traverse of the Binary Tree:");
    MyTree.postOrder(MyTree.root);
}
```

Testing the Binary Tree ADT

```
Initially, the Binary Tree size is: 0
The tree size is: 6
The tree height is: 2
Searching the Binary Tree for 40:
Value 40 Found
Searching the Binary Tree for 60:
Value 60 Not Found
In Order Traverse of the Binary Tree:
2
5
30
35
40
80
Pre Order Traverse of the Binary Tree:
30
5
2
40
35
80
Post Order Traverse of the Binary Tree:
2
5
35
80
40
30
```

Advanced Searching Techniques

- The search for a solution in a large search tree presents a problem for nearly all inference systems,
- From the starting state there are many possibilities for the first inference step,
- For each of these possibilities there are again many possibilities in the next step, and so on,



- The tree above was cut off at a depth of 14 and has a solution in the leaf node marked by ∗.

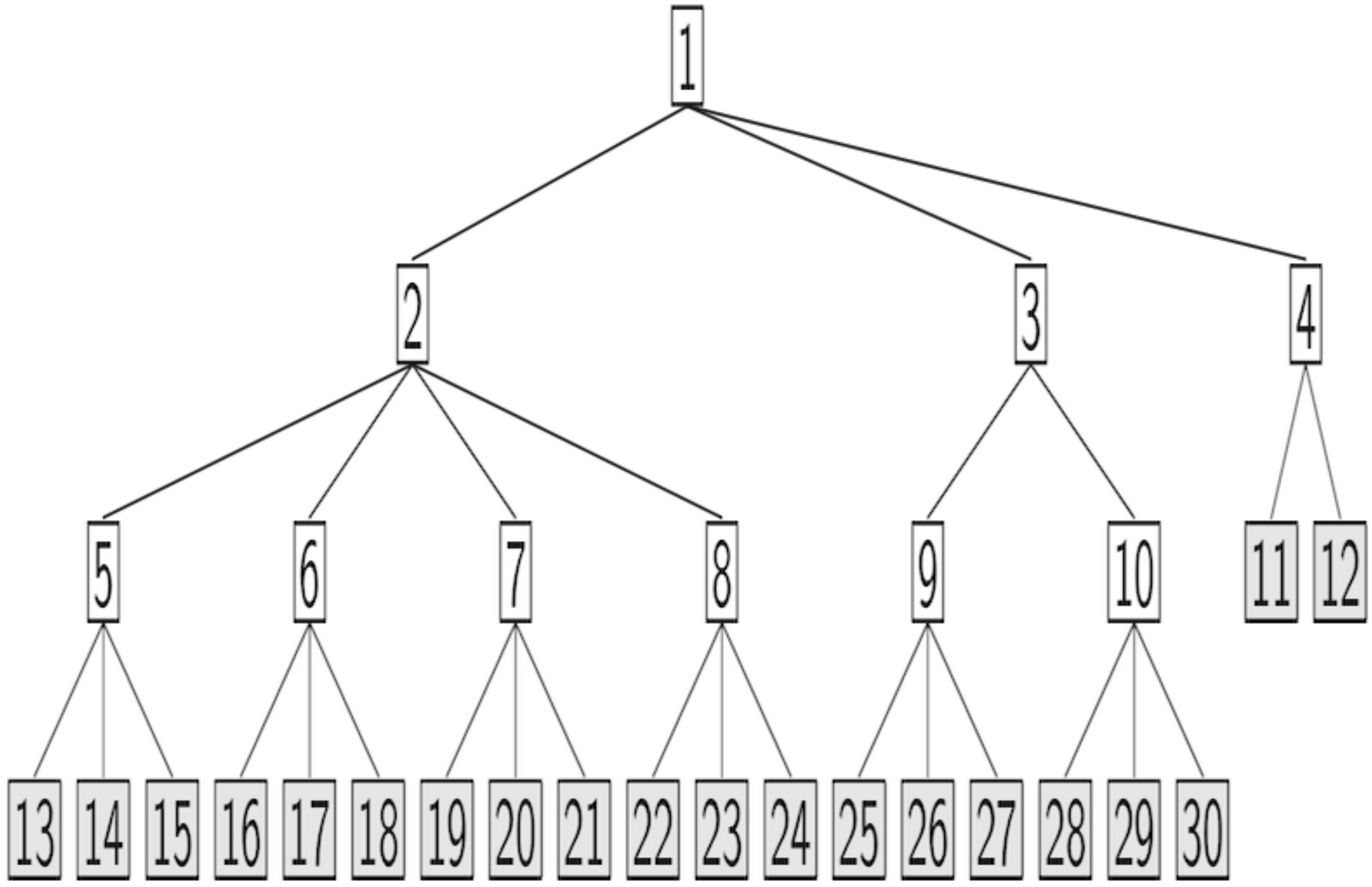
Breadth First Search (BFS)

- In breadth-first search, the search tree is explored from top to bottom according to the algorithm given in the next slide until a solution is found,
- First, every node in the node list is tested for whether it is a goal node, and in the case of success, the program is stopped,
- Otherwise all successors of the node are generated and the search is then continued recursively on the list of all newly generated nodes,
- The whole thing repeats until no more successors are generated

The BFS Algorithm

```
BREADTH-FIRST-SEARCH(NodeList, Goal)

NewNodes = $\emptyset$
**For all** Node $\in$ NodeList
    **If** GoalReached(Node, Goal)
        **Return**("Solution found", Node)
    NewNodes = **Append**(NewNodes, Successors(Node))
**If** NewNodes $\neq \emptyset$
    **Return**(BREADTH-FIRST-SEARCH(NewNodes, Goal))
**Else**
    **Return**("No solution")
```

Example of BFS



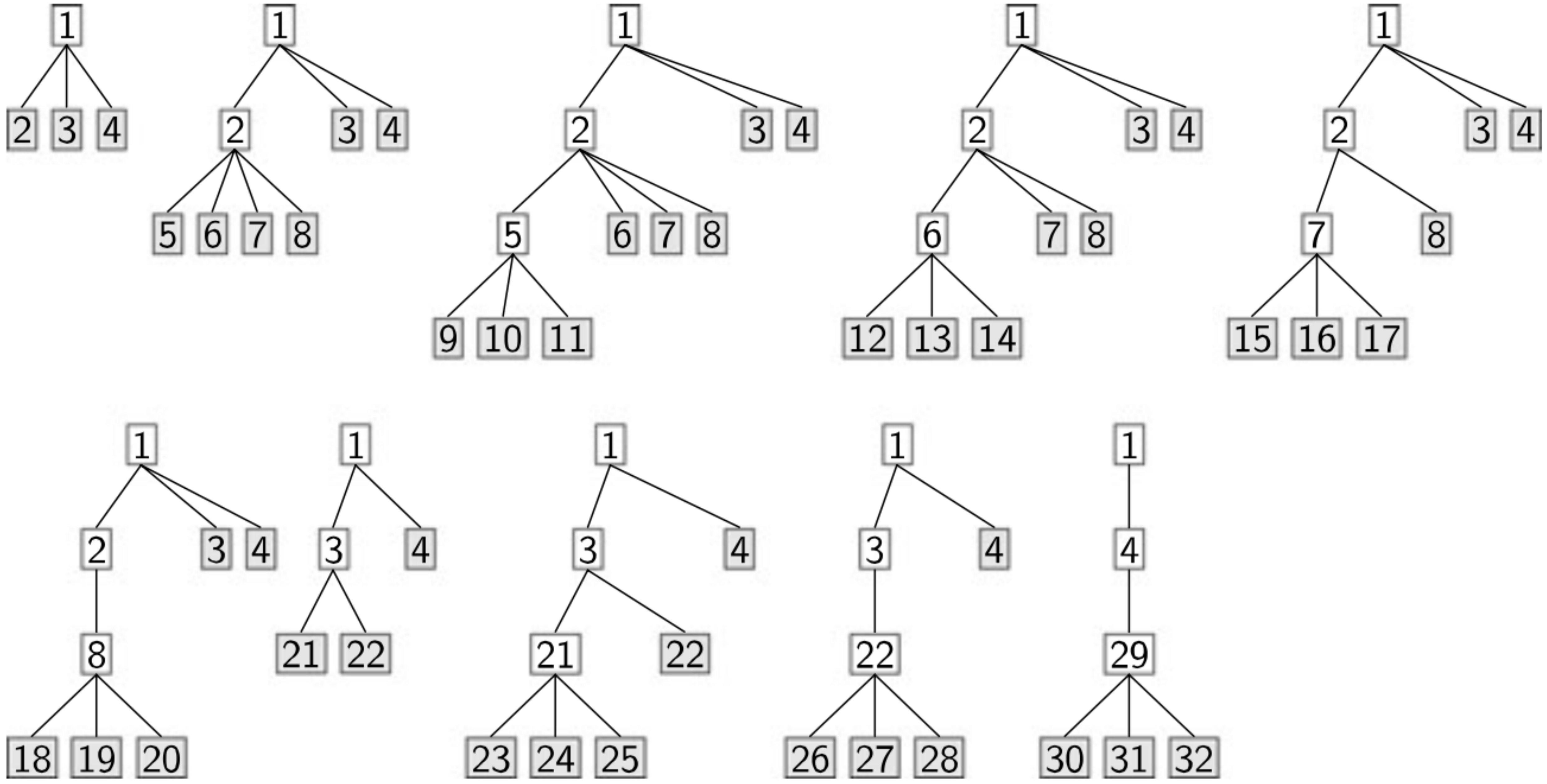
Depth First Search (DFS)

- In depth-first search only a few nodes are stored in memory at one time,
- After the expansion of a node only its successors are saved, and the first successor node is immediately expanded, thus the search quickly becomes very deep,
- Only when a node has no successors and the search fails at that depth is the next open node expanded via backtracking to the last branch, and so on,
- We can best perceive this in the elegant recursive algorithm the next slide and in the search tree in the slide that follow

The DFS Algorithm

```
DEPTH-FIRST-SEARCH(Node, Goal)

If GoalReached(Node, Goal) Return("Solution found")
NewNodes = Successors(Node)
While NewNodes $\neq \emptyset$
    Result = DEPTH-FIRST-SEARCH(First(NewNodes), Goal)
    If Result = "Solution found" Return("Solution found")
    NewNodes = Rest(NewNodes)
Return("No solution")
```

Example of DFS



Part # 2

- **Complexity Analysis of Algorithms**
  1. **Introduction to Algorithms,**
  2. **The Complexity Analysis of Algorithms,**
  3. **The Big O Notation,**

Introduction to Algorithms

- What is an algorithm?
  - an algorithm is “a step-by-step procedure for accomplishing some tasks”
- How an algorithm can be written?
  - an algorithm, in general, can be written down in English (pseudo code), however, we are interested in algorithms which have been precisely specified using an object-oriented programming language (Java)

Introduction to Algorithms

- Given such an algorithm, what can we do with it?
  - obviously, we can run the program and observes its behavior,
  - however, this is not likely to be informative in the general case,
  - running a program that represents an algorithm on a particular computer with a particular set of inputs will give us its behavior in a single instance

Introduction to Algorithms

- We should analyze algorithms, or programs representing them, to draw conclusions about how the implementation will perform in general, for example:
  - the running time as a function of the inputs,
  - the total or maximum memory space needed,
  - the total size of the program code,
  - whether the algorithm correctly computes the desired result,
  - how easy it is to read, understand, and modify,
  - the robustness of the program

Why to Analyze Algorithms?

- **Practical Reasons:**
  - to estimate computer resources for processing a given amount of data,
  - to save valuable resources
- **Theoretical Reasons:**
  - to choose the most efficient algorithm for solving a given problem,
  - to develop new efficient algorithms

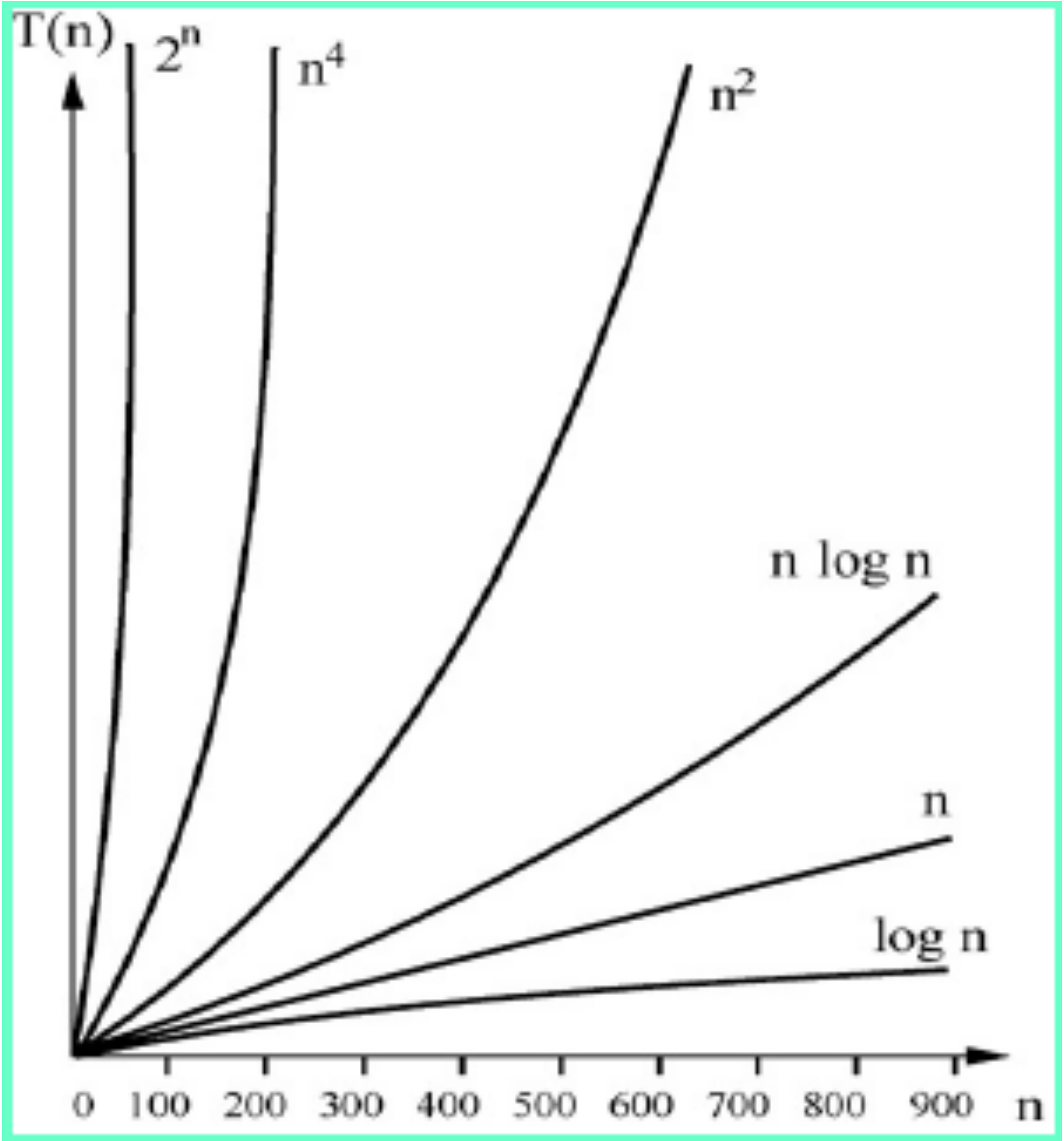
Complexity Analysis of Algorithms

- The big oh notation or simply O is used to denote the general behavior of an algorithm as a function of the problem size n,
  - For example, an algorithm is O(log n) if its running time T(n) to solve a problem of size n is proportional to log(n),
    - O(1): Pronounced Order 1, denoting a function that run in constant time,
    - O(N): Pronounced Order N, denoting a function that run in linear time,
    - O(N^2): Pronounced Order N^2, denoting a function that run in quadratic time,
    - O(logN): Pronounced Order log N, denoting a function that run in logarithmic time,
    - O(N log N): Pronounced Order N log N, denoting a function that run in time proportional to the size of the problem multiplied by the logarithmic time, and
    - O(N!): Pronounced Order N factorial, denoting a function that run in factorial time,

Complexity Analysis of Algorithms

- **Some properties of the big O:**
  - Constant factors may be ignored
    - an^2 and bn^2 are both $O(n^2)$, 2^{n+k} is $O(2^n)$
  - The growth rate of a sum of terms is the growth rate of its fastest growing term
    - $an^2 + bn^3$ is $O(n^3)$,
  - The product of upper bounds of functions gives an upper bounds for the product of the functions
    - if f is $O(n^2)$ and g is $O(\log n)$, then fg is $O(n^2 \log n)$,
  - All logarithms grow at the same rate
    - $\log_b n = \log_b 2 * \log_2 n$ which is $O(\log_2 n)$ or $O(\log n)$
  - Exponential functions growth faster than powers
    - $2^n + n^4$ is $O(2^n)$ and $e^n + n^4$ is $O(e^n)$

Typical Curves for Time Complexity



Part # 3

- **Searching and Sorting Techniques**
  1. **The Searching Problem**
     - Linear Search,
     - Binary Search, and
     - Advanced Searching techniques (at the end of Part 7)
  2. **The Sorting Problem**
     - Selection Sort,
     - Bubble Sort,
     - Quick Sort, and
     - Heap Sort (at the end of Part 8)

Searching

- The Searching problem:
  - given an array of values (of size n),
  - determine whether a given value v is in the array

- There are 2 approaches for searching:
  - Sequential (Linear) Search,
  - Binary Search

Searching

- **Sequential (Linear) search**
  - Compare each element of the array with a value v,
  - Useful for small and unsorted arrays,
  - Simple (one loop), complexity is O(n)
- **Binary search**
  - For sorted arrays only,
  - Compares middle element with a value v
    - If equal, match found,
    - If value v < middle, looks in first half of array,
    - If value v > middle, looks in last half
  - Very fast, at most r steps, where $2^r > $ number of elements, thus complexity is $O(\log_2 n)$
    - 30 element array takes at most 5 steps

      $2^5 > 30$

The Two Search Algorithms in Java

```
public static int linearSearch(int target, int[] a) {
        for (int i = 0; i < a.length; i++) {
                if (target == a[i]) return i;
        }
        return -1; // not a legal index
}
```

```
public static int binarySearch(int target, int[] a, int left, int right){

   while (left <= right) {
      int middle = (left  + right) / 2;
      if (a[middle] == target) return middle; // found
      else if (a[middle] > target)  right = middle  – 1;
      else left  = middle  + 1;
   }
   return -1;  // -1 means "not found"
}
```

Sorting Data

- The Sorting Problem:
  - given an array A of size n of values,
  - arrange the values in sorted order
- Most sorting algorithms involve comparing values with two loops,
- The obvious algorithms are $O(n^2)$, while the clever ones are $O(n \log n)$,
- It is not possible (in the general case) to have a comparison sort with a worst-case time better than $O(n \log n)$

The Selection Sort Algorithm

- **Idea:**
  - find the smallest value in A; put it in A[0],
  - find the 2nd smallest value in A; put it in A[1],
  - etc.
- **Approach:**
  - use one loop from 0 to n-1 (position in A to fill),
  - loop invariant: after k iterations A[0] through A[k-1] contain their final values so after n iterations, A[0] through A[n-1] contain their final values,
  - each time around, use a nested loop to find the smallest value (and its index) in the unsorted part of the array and swap that value with A[k],

The Selection Sort Algorithm in Java

```
public static void selectionSort(int[] a) {
    int outer, inner, min;
    for (outer = 0; outer < a.length - 1; outer++) { // outer counts down
        min = outer;
        for (inner = outer + 1; inner < a.length; inner++) {
            if (a[inner] < a[min]) {
                min = inner;
            }
            // Invariant: for all i, if outer <= i <= inner, then a[min] <= a[i]
        }
        // a[min] is least among a[outer]..a[a.length - 1]
        int temp = a[outer];
        a[outer] = a[min];
        a[min] = temp;
        // Invariant: for all i <= outer, if i < j then a[i] <= a[j]
    }
}
```

Complexity of the Selection Sort

- **The Complexity:**
  - The inner loop executes a different # of times each time around outer loop, so:
    - 1st iteration of outer : inner executes n - 1
    - 2nd " : " " n - 2
    - nth - 1 " : " " 1
    - $1 + 2 + \ + (n - 1)=((n - 1)/2)*((n-1) + 1)*1=n(n-1)/2=O(n^2)$
  - Thus, the selection sort is always $O(n^2)$,
  - It won't do any swaps if array is already sorted, but still looks at values $O(n^2)$ times

The Bubble Sort Algorithm

- **Idea:**
  - Several passes (n at most) through the array,
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change if sorting ascending,
    - If decreasing order, elements exchanged if sorting ascending,
    - The above two rules are switched if sorting descending
  - Repeat,
  - One can stop if no change is made ($< O(n^2)$),
  - The Bubble sort worst case is $O(n^2)$

The Bubble Sort Algorithm in Java

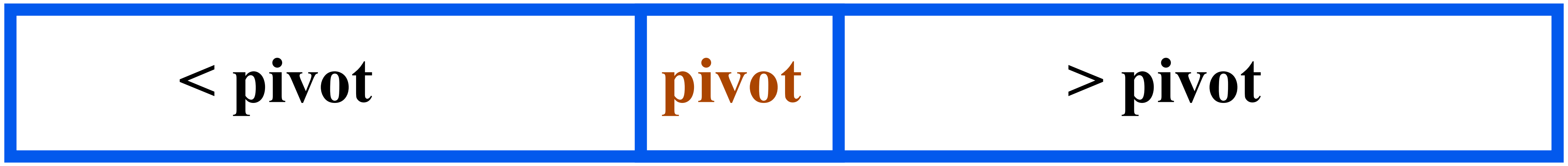
```
public static void bubbleSort(int a[]) {

    for ( int pass = 0; pass < a.length; pass++ )
        for ( i = 0; i < a.length - 1; i++ )
                if ( a[ i ] > a[ i + 1 ] ) {
                    int temp = a[i];
                        a[i] = a[i + 1];
                        a[i + 1] = temp;
                 }
}
```

The Quick Sort Algorithm

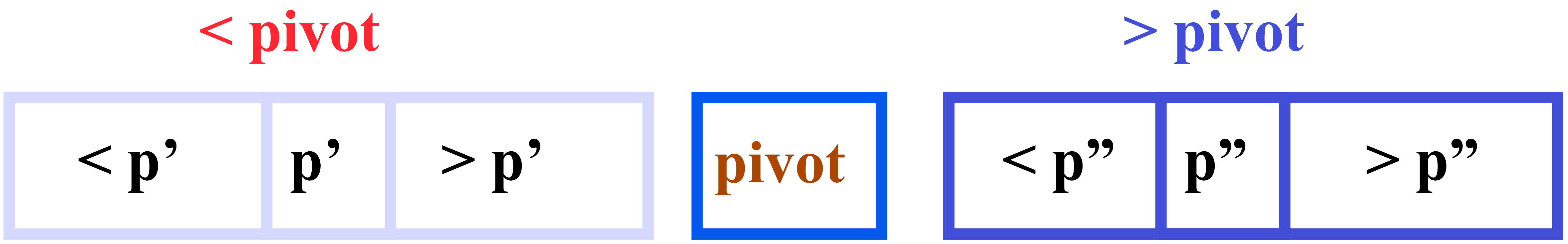
- The quick sort is a sorting algorithm based on the divide-and-conquer paradigm, Discovered by C.A.R. Hoare:
  - Divide: pick a random element x (called pivot) and partition S into two sets:
    - L elements less than x, and
    - G elements greater than x
  - Recur: quick sort L and G
  - Conquer: join L, x, and G
  - Thus the complexity of the Quick Sort is $O(n \log_2 n)$ most of the time, but worst case is $O(n^2)$

The Quick Sort Algorithm

- **Partition**
  - Choose a pivot,
  - Find the position for the pivot so that:
    - all elements to the left are less, and
    - all elements to the right are greater



- **Conquer**
  - Apply the same algorithm to each half



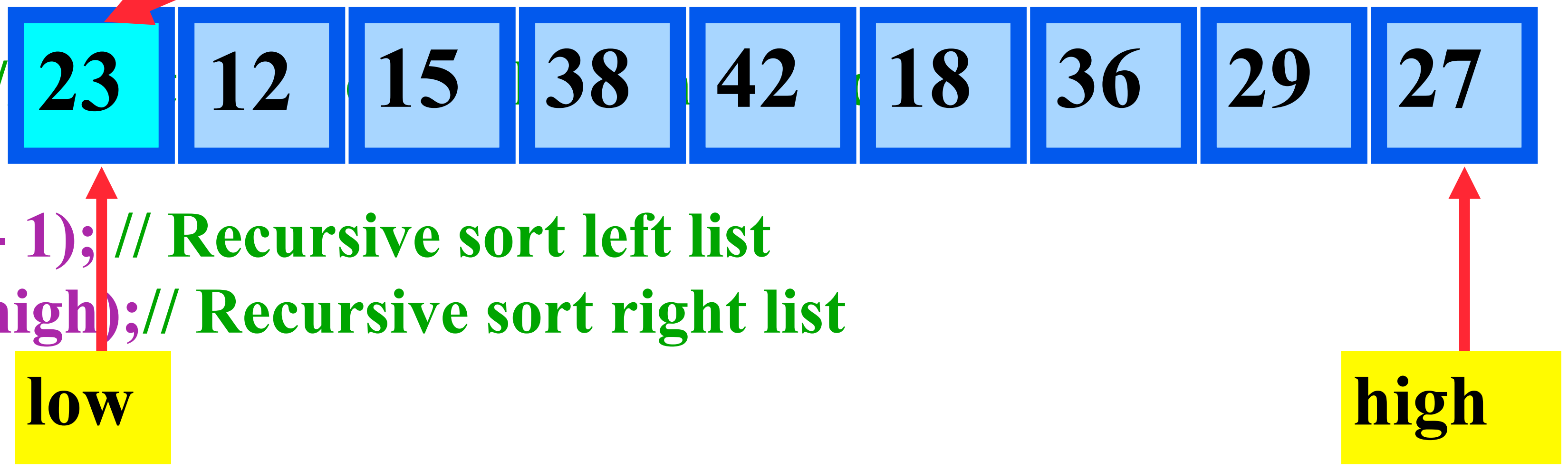
The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
        if (low < high) {
                    int pivotIndex = low; // Assume first element is the pivot
                    int pivot = A[low]; // The pivot value
                    A[pivotIndex] = A[high]; // Swap pivot with last item
                    A[high] = pivot;
                    int i = low - 1;
                    int j = high;
                    do {
                        do { i++; } while (A[i] < pivot);
                        do { j--; } while (A[j] > pivot);
                        if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
                    } while ( i < j );
                    A[high] = A[i]; // Put the pivot back in the middle
                    A[i] = pivot;
                    qSort(A, low, i - 1); // Recursive sort left list
                    qSort(A, i + 1, high);// Recursive sort right list
        }
}
```

The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
        if (low < high) {
                    int pivotIndex = low; // Assume first element is the pivot
                    int pivot = A[low]; // The pivot value
                    A[pivotIndex] = A[high]; // Swap pivot with last item
                    A[high] = pivot;
                    int i = low - 1;
                    int j = high;
                    do {
                        do { i++; } while (A[i] < pivot);
                        do { j--; } while (A[j] > pivot);
                        if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
                    } while ( i < j );
                    A[high] = A[i]; /
                    A[i] = pivot;
                    qSort(A, low, i - 1); // Recursive sort left list
                    qSort(A, i + 1, high);// Recursive sort right list
        }
}
```

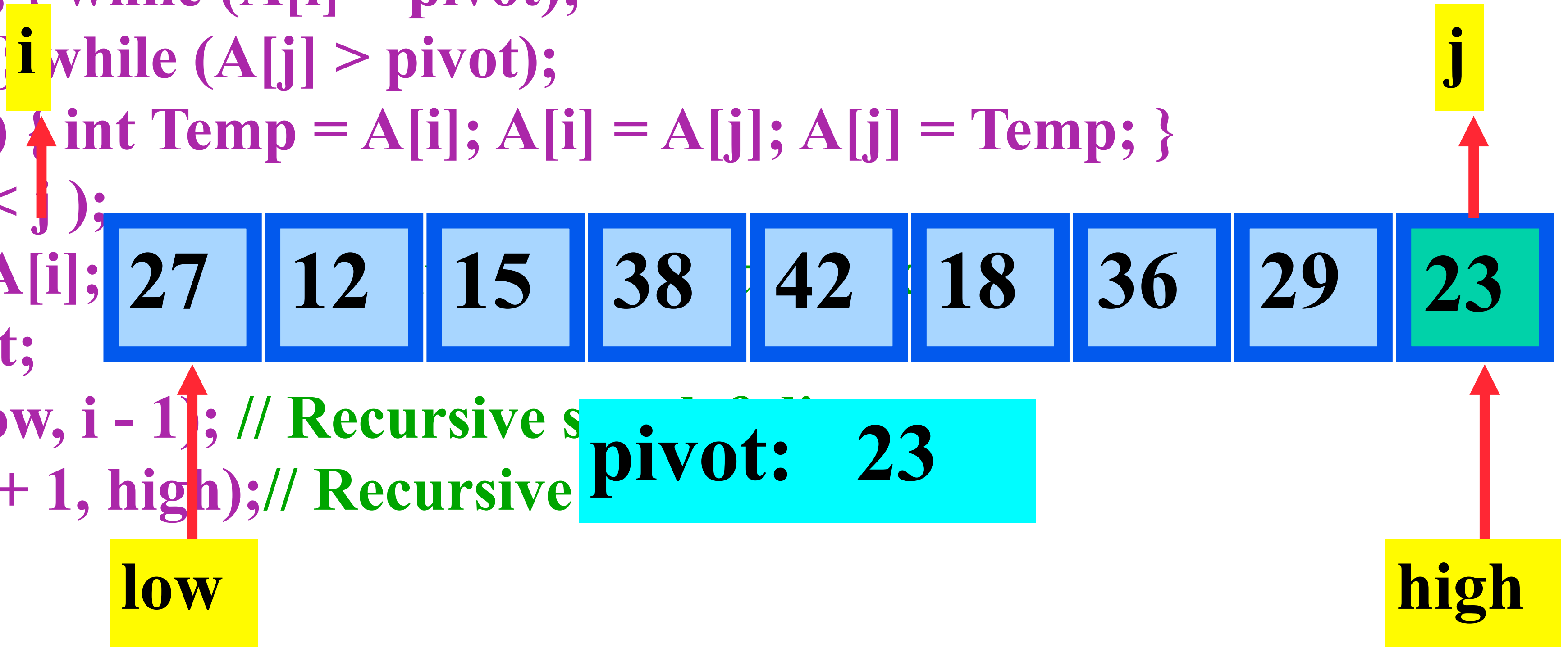
Any item will do as the pivot, choose the leftmost one!



The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
    if (low < high) {
        int pivotIndex = low; // Assume first element is the pivot
        int pivot = A[low]; // The pivot value
        A[pivotIndex] = A[high]; // Swap pivot with last item
        A[high] = pivot;
        int i = low - 1;
        int j = high;
        do {
          do { i++; } while (A[i] < pivot);
          do { j--; } while (A[j] > pivot);
          if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
        } while ( i < j );
        A[high] = A[i];
        A[i] = pivot;
        qSort(A, low, i - 1); // Recursive s
        qSort(A, i + 1, high);// Recursive
    }
}
```

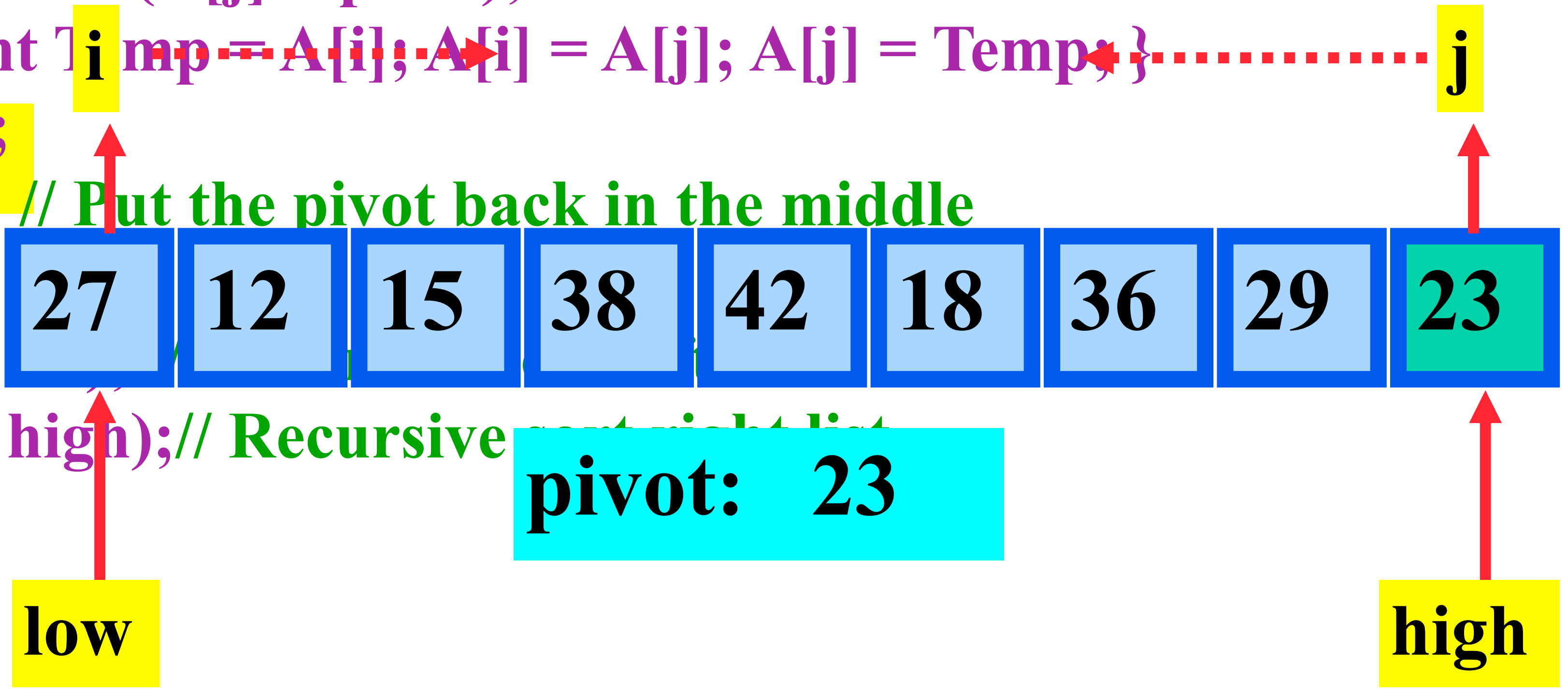
Set left (i) and right (j)



The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
        if (low < high) {
                int pivotIndex = low; // Assume first element is the pivot
                int pivot = A[low]; // The pivot value
                A[pivotIndex] = A[high]; // Swap pivot with last item
                A[high] = pivot;
                int i = low - 1;
                int j = high;
                do {
                    do { i++; } while (A[i] < pivot);
                    do { j--; } while (A[j] > pivot);
                    if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
                } while ( i < j );
                A[high] = A[i]; // Put the pivot back in the middle
                A[i] = pivot;
                qSort(A, low, i
                qSort(A, i + 1, high);// Recursive
        }
}
```

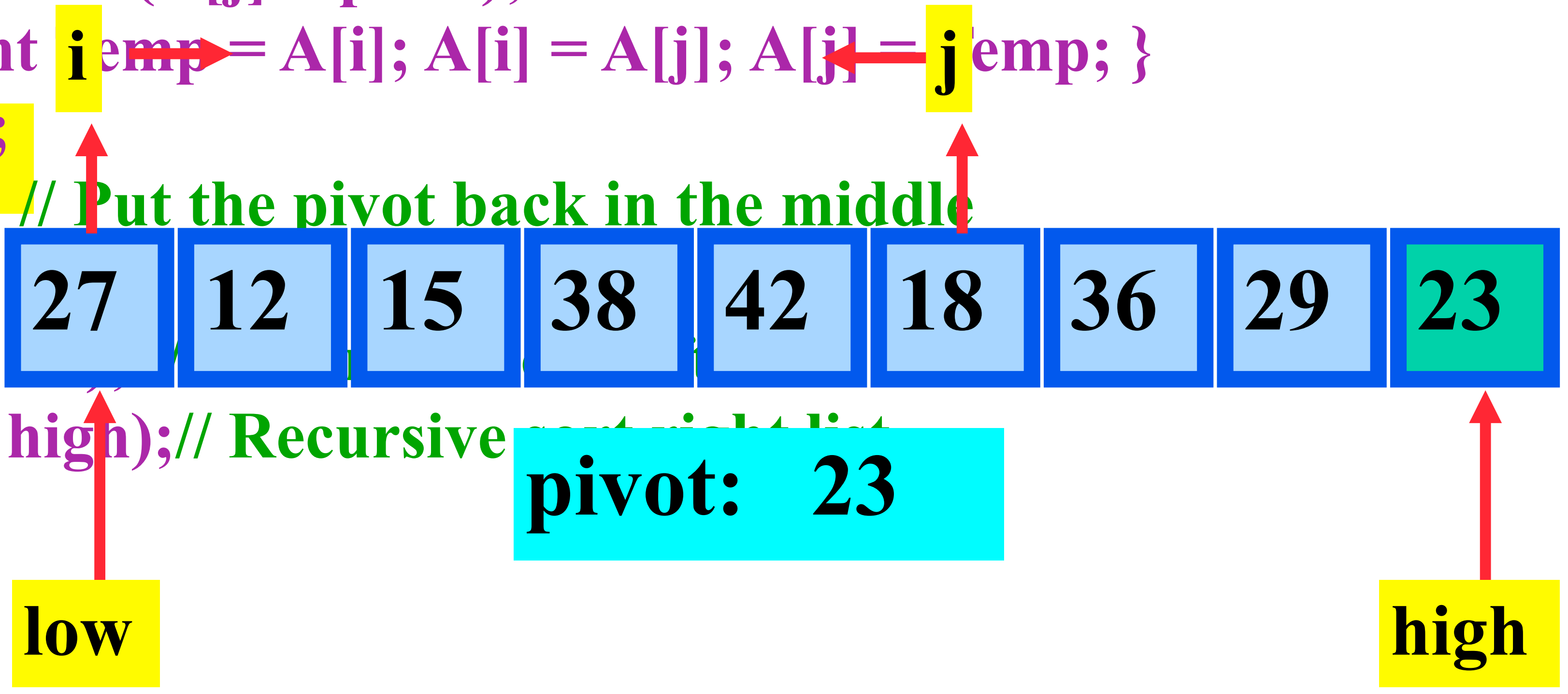
Move the Markers until they Cross



The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
        if (low < high) {
                int pivotIndex = low; // Assume first element is the pivot
                int pivot = A[low]; // The pivot value
                A[pivotIndex] = A[high]; // Swap pivot with last item
                A[high] = pivot;
                int i = low - 1;
                int j = high;
                do {
                    do { i++; } while (A[i] < pivot);
                    do { j--; } while (A[j] > pivot);
                    if ( i < j ) { int temp = A[i]; A[i] = A[j]; A[j] = temp; }
                } while ( i < j );
                A[high] = A[i]; // Put the pivot back in the middle
                A[i] = pivot;
                qSort(A, low, i
                qSort(A, i + 1, high);// Recursive
        }
}
```

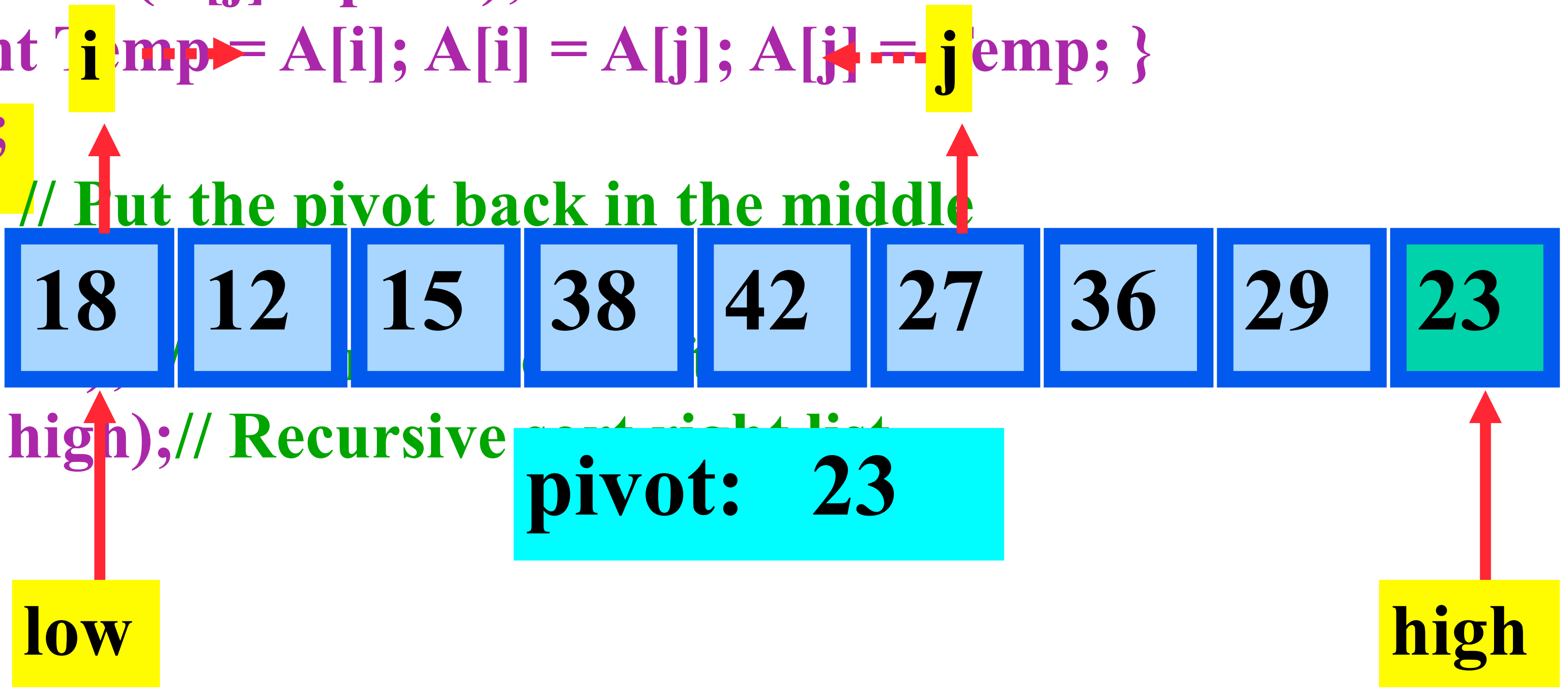
A[i] is GT pivot, A[j] is LT pivot, i < j
Swap A[i] with A[j]



The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
        if (low < high) {
                int pivotIndex = low; // Assume first element is the pivot
                int pivot = A[low]; // The pivot value
                A[pivotIndex] = A[high]; // Swap pivot with last item
                A[high] = pivot;
                int i = low - 1;
                int j = high;
                do {
                    do { i++; } while (A[i] < pivot);
                    do { j--;  } while (A[j] > pivot);
                    if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
                } while ( i < j );
                A[high] = A[i]; // Put the pivot back in the middle
                A[i] = pivot;
                qSort(A, low, i
                qSort(A, i + 1, high);// Recursive
        }
}
```

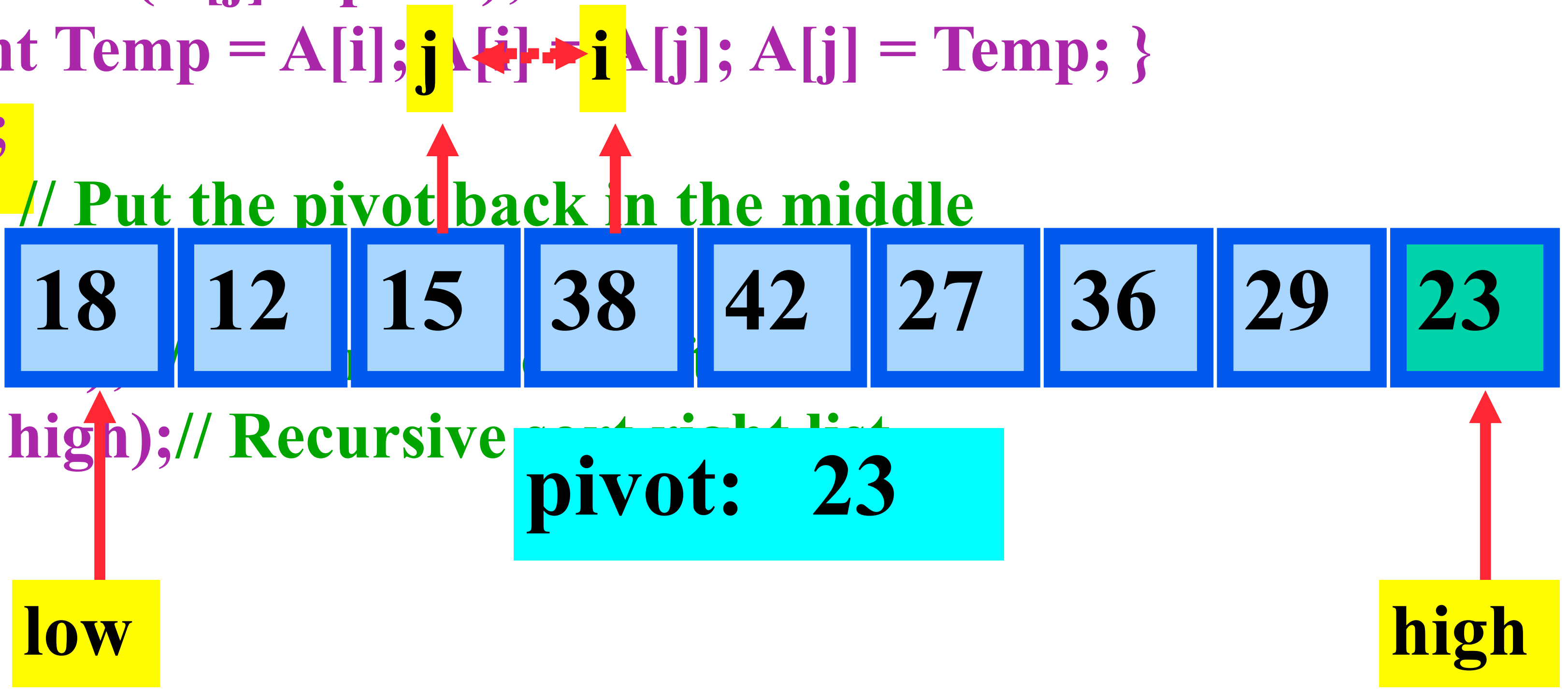
Continue until i, j Cross



The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
    if (low < high) {
            int pivotIndex = low; // Assume first element is the pivot
            int pivot = A[low]; // The pivot value
            A[pivotIndex] = A[high]; // Swap pivot with last item
            A[high] = pivot;
            int i = low - 1;
            int j = high;
            do {
                do { i++; } while (A[i] < pivot);
                do { j--; } while (A[j] > pivot);
                if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
            } while ( i < j );
            A[high] = A[i]; // Put the pivot back in the middle
            A[i] = pivot;
            qSort(A, low, i
            qSort(A, i + 1, high);// Recursive sort right list
    }
}
```

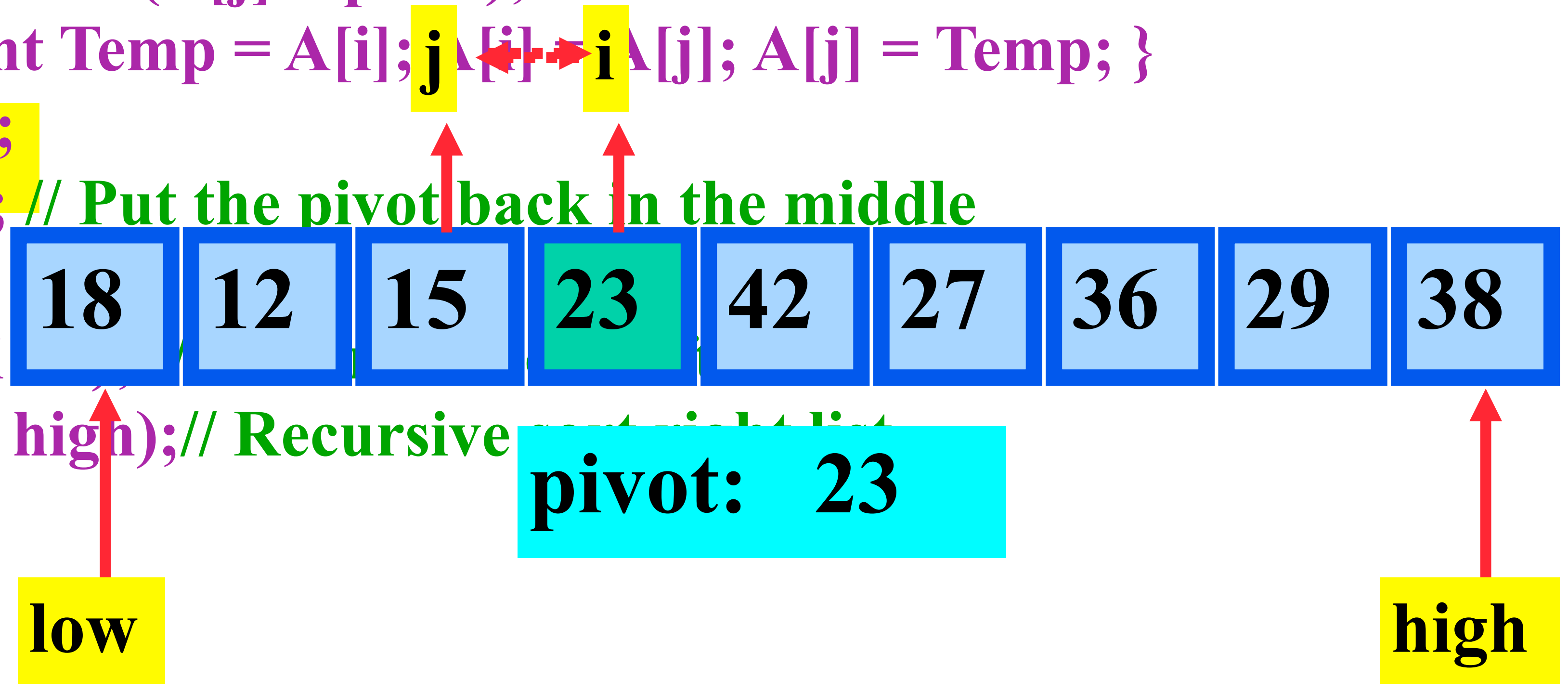
i, j Cross Over Stop



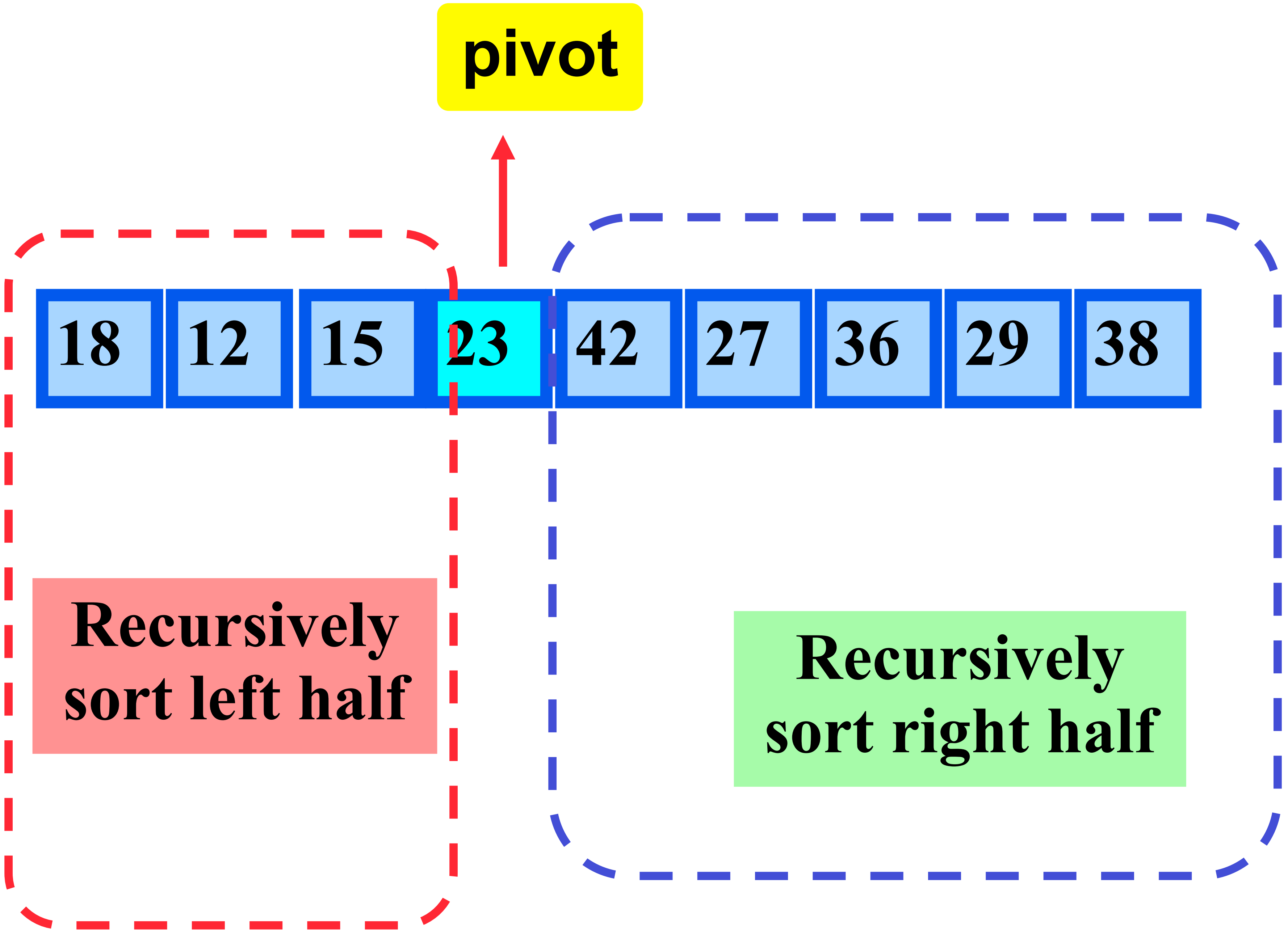
The Recursive Quick Sort Implementation

```
public static void qSort(int A[], int low, int high) {
    if (low < high) {
        int pivotIndex = low; // Assume first element is the pivot
        int pivot = A[low]; // The pivot value
        A[pivotIndex] = A[high]; // Swap pivot with last item
        A[high] = pivot;
        int i = low - 1;
        int j = high;
        do {
            do { i++; } while (A[i] < pivot);
            do { j--; } while (A[j] > pivot);
            if ( i < j ) { int Temp = A[i]; A[i] = A[j]; A[j] = Temp; }
        } while ( i < j );
        A[high] = A[i]; // Put the pivot back in the middle
        A[i] = pivot;
        qSort(A, low, i
        qSort(A, i + 1, high);// Recursive sort right list
    }
}
```

Put the pivot in its Normal Position
By Swapping [high] with A[i]



The Partitioned List



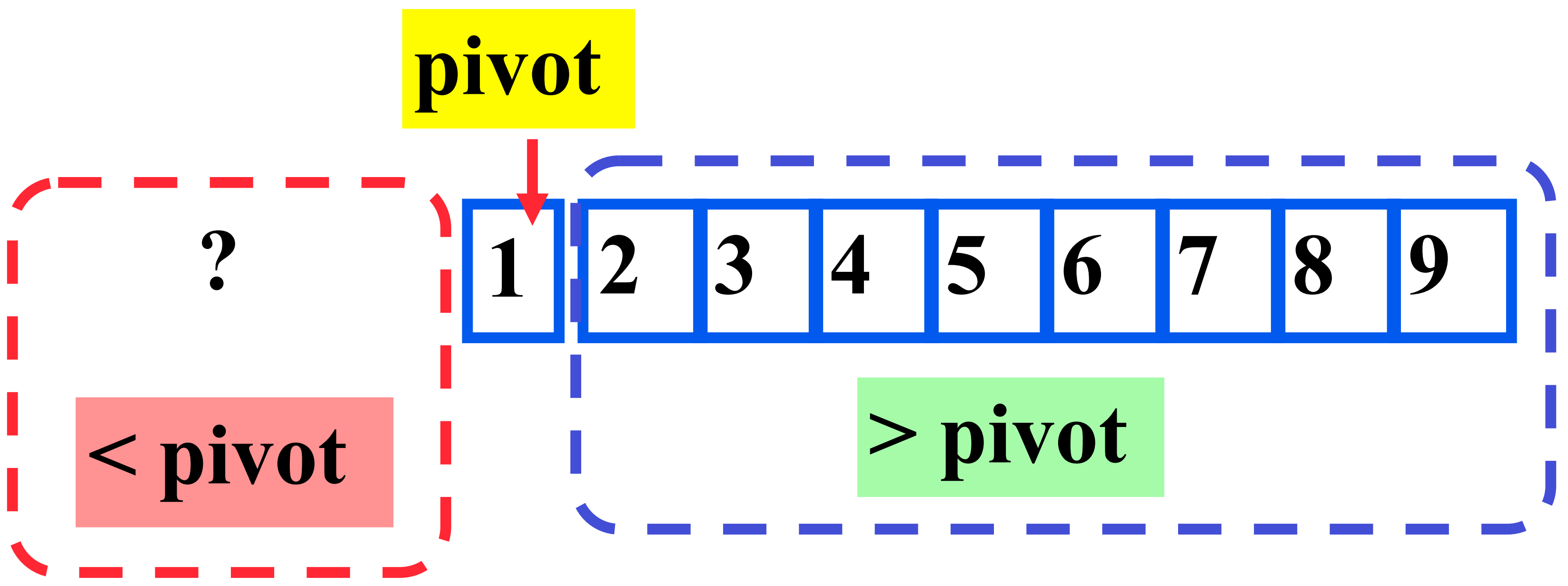
pivot: 23

Complexity Analysis of Quick Sort

- Partition
  - Check every item once: O(n)
- Conquer
  - Divide data in half: $O(\log_2 n)$
- Total
  - Product: $O(n \log_2 n)$
- Thus, the Quick Sort complexity in the best case is $O(n \log_2 n)$

Complexity Analysis of Quick Sort

- What happens if we use the Quick Sort algorithm on data that's already sorted (or nearly sorted)?
- What is the expected performance of the algorithm?
- Consider the Sorted Data given below:

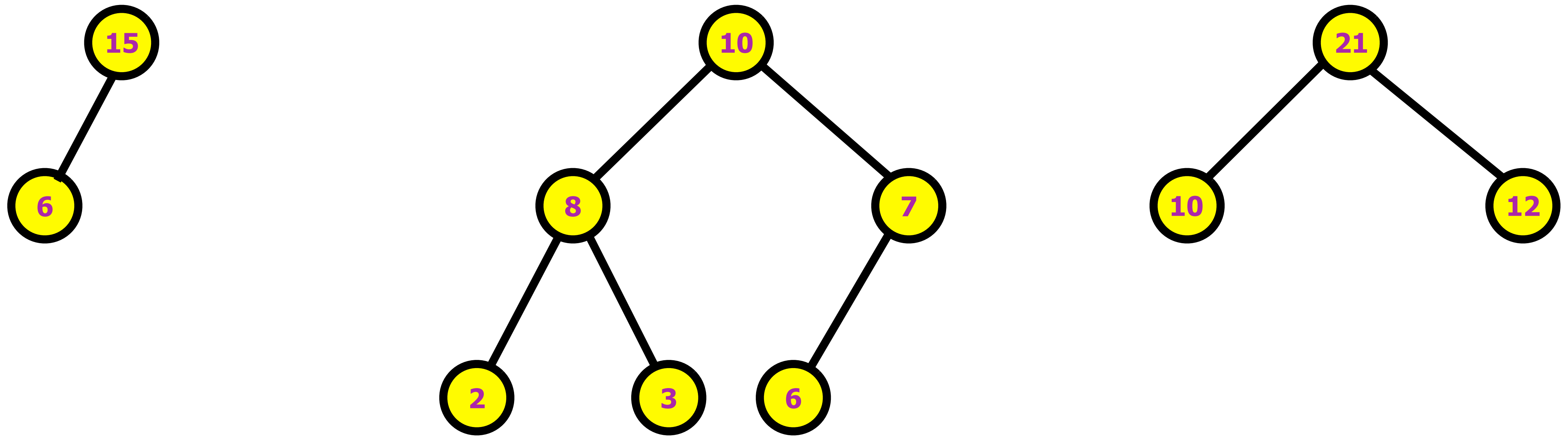


Part # 8

- **Heap Trees**
  1. The Concept of Heap Trees,
  2. Organizing Arrays as Heaps,
  3. Inserting/Deleting Elements in Heaps,
  4. The Heap Sort Algorithm

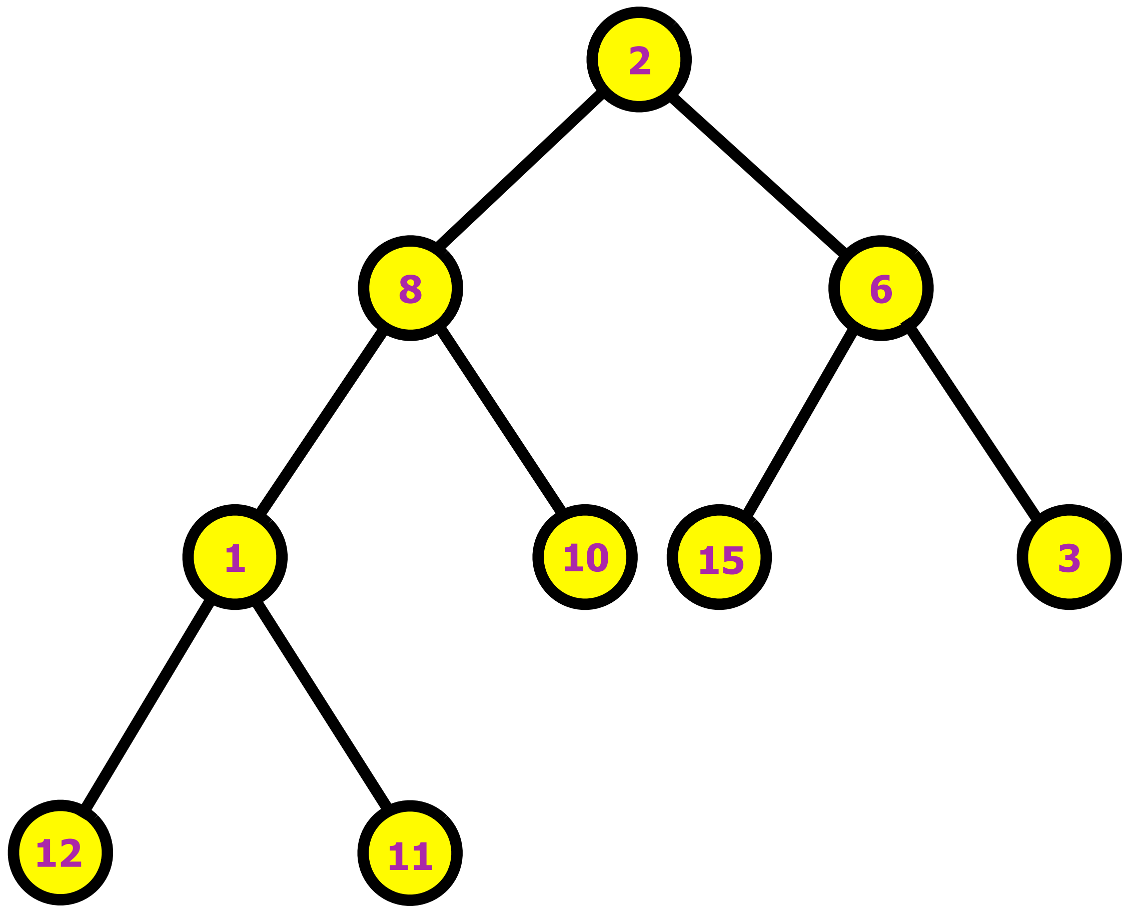
The Concept of Heap Trees

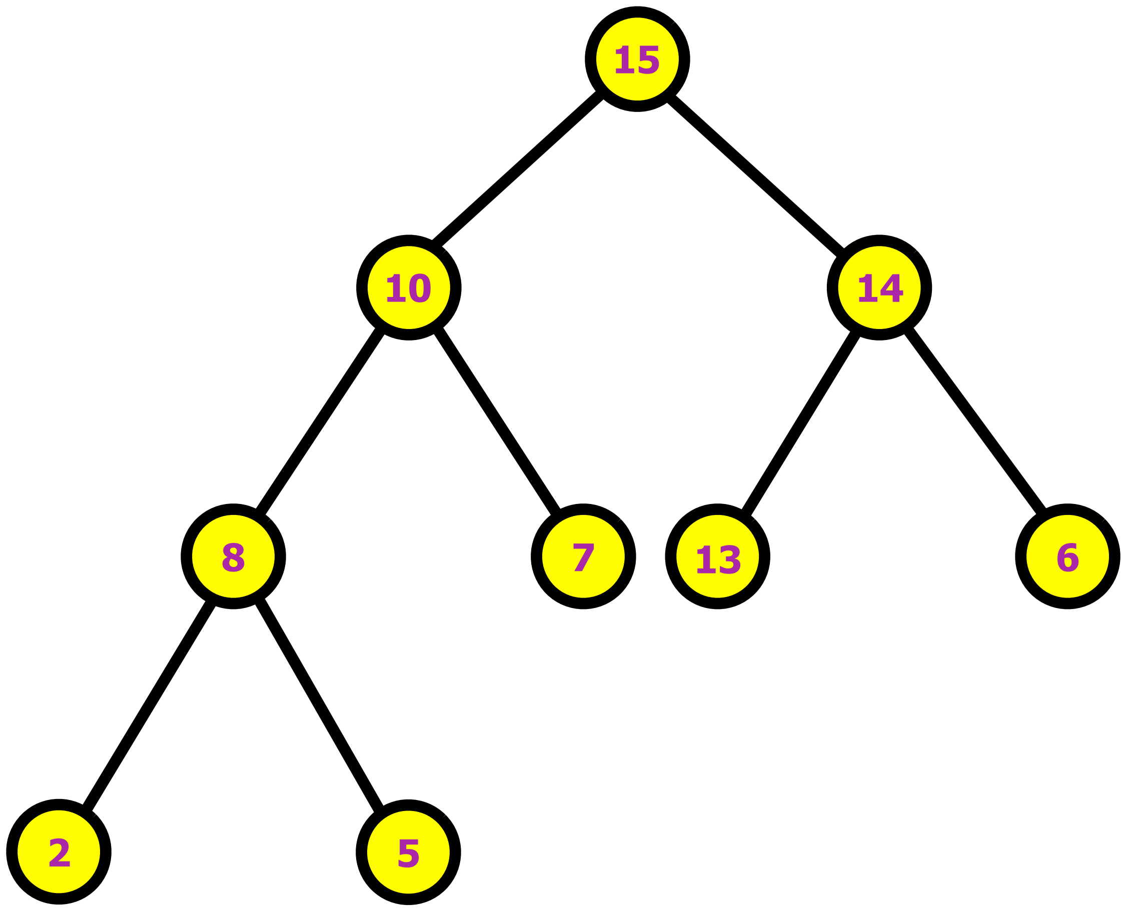
- **A particular kind of binary tree, called a heap, has the following properties:**
  - **The value of each node is not less than the values stored in each of its children,**
  - **The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions**



The Concept of Heap Trees

- **Interestingly, heaps can be implemented by arrays, for example**
  - the array data = [2 8 6 1 10 15 3 12 11] can be represented by the following non heap and heap trees

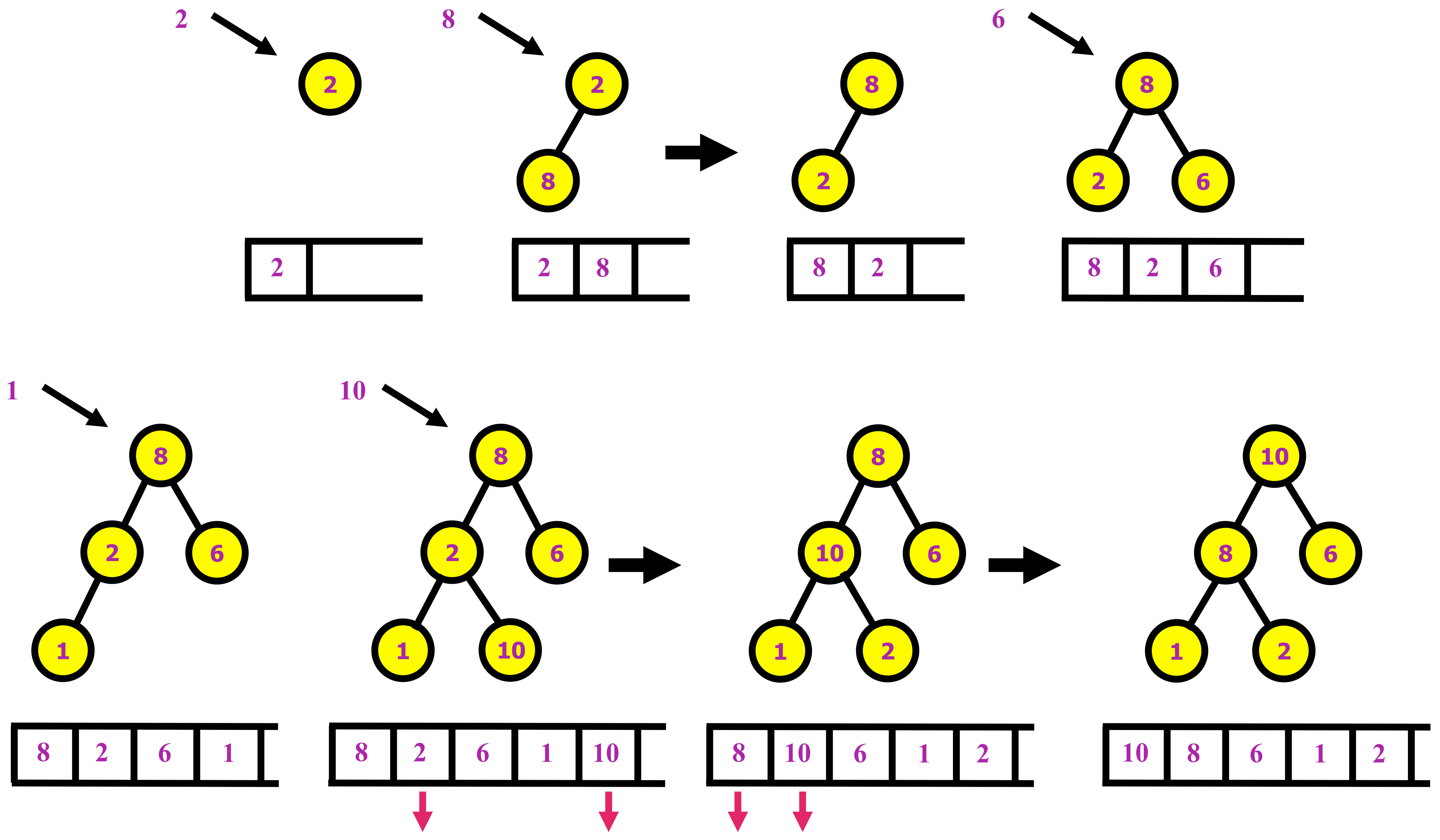




Organizing Arrays as Heaps

- Heaps can be implemented as arrays, and in that sense, each heap is an array, but all arrays are not heaps,
- In some situations, most notably in heap sort, we need to convert an array into a heap,
- This means that the data in the original array are reorganized to represent a heap, as shown by the example of the next slide

Organizing Arrays as Heaps



Organizing Arrays as Heaps

- Now, a heap can be defined as an array heap of length n in which:

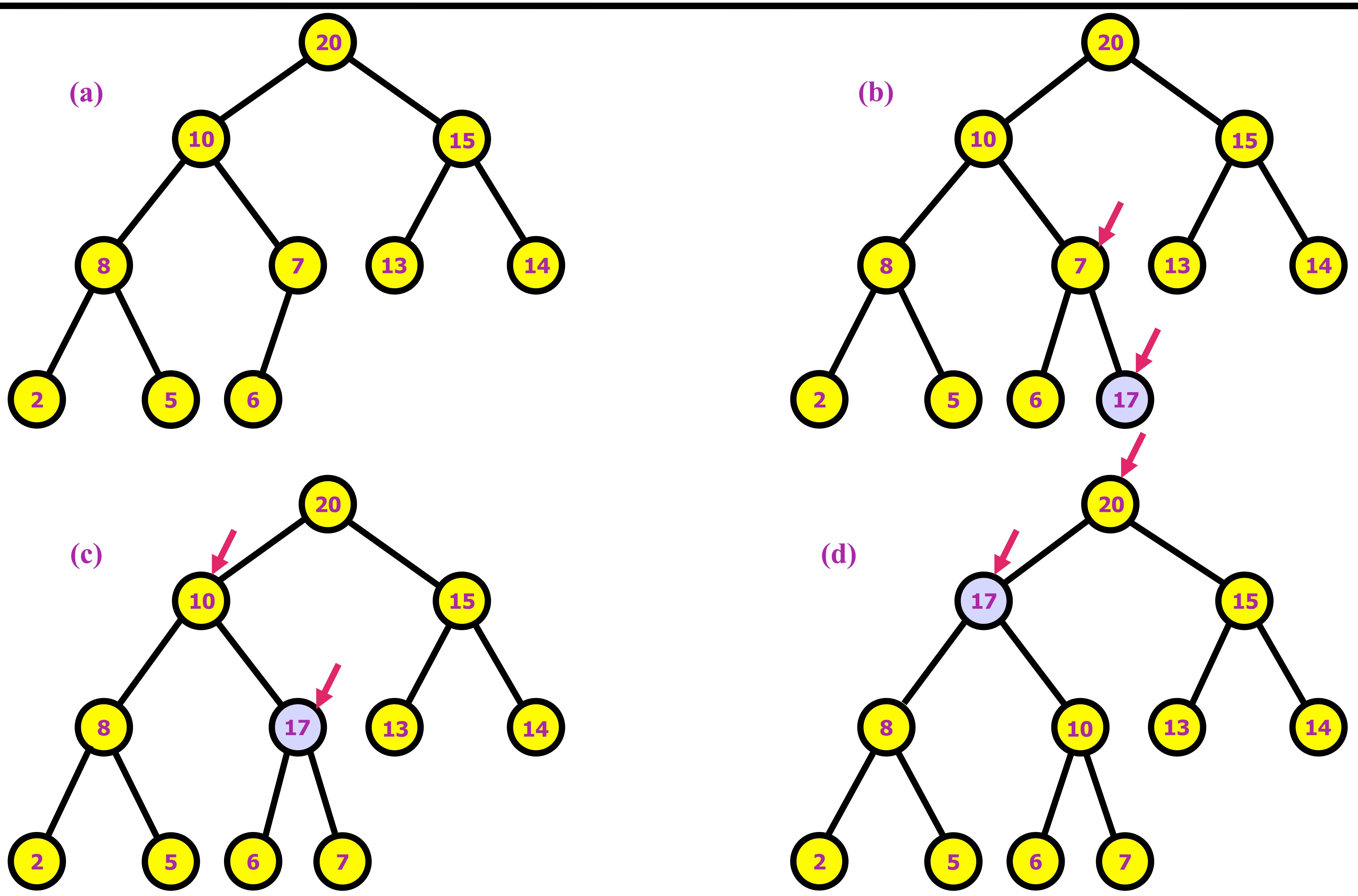
  heap[i] $\geq$ heap[2 * i + 1], for $0 \leq i \leq (n - 1)/2$, and
  heap[i] $\geq$ heap[2 * i + 2], for $0 \leq i \leq (n - 2)/2$
- Elements in a heap are not perfectly ordered, but we know only that the largest element is in the root node and that, for each node, all its descendant are less than or equal to that node

Adding Elements to the Heap

- The element is added at the end of the heap as the last leaf and restoring the heap property is achieved by moving from the last leaf toward the root, the algorithm is as follows:

```
heapAdd( el ) {
  put el at the end of heap;
  while el is not in the root and el > parent( el )
        swap el with its parent;
}
```

Adding Elements to the Heap



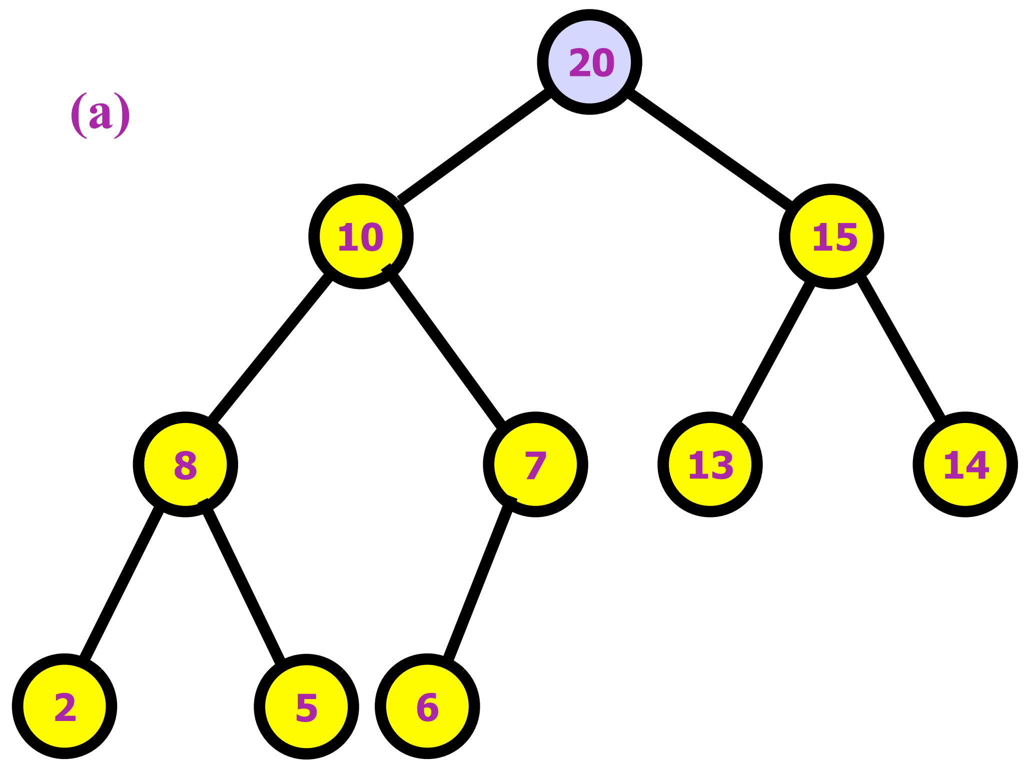
Deleting Elements from the Heap

- Deleting an element from the heap consists of removing the root element from the heap, then the last leaf is put in its place, and the heap property is restored

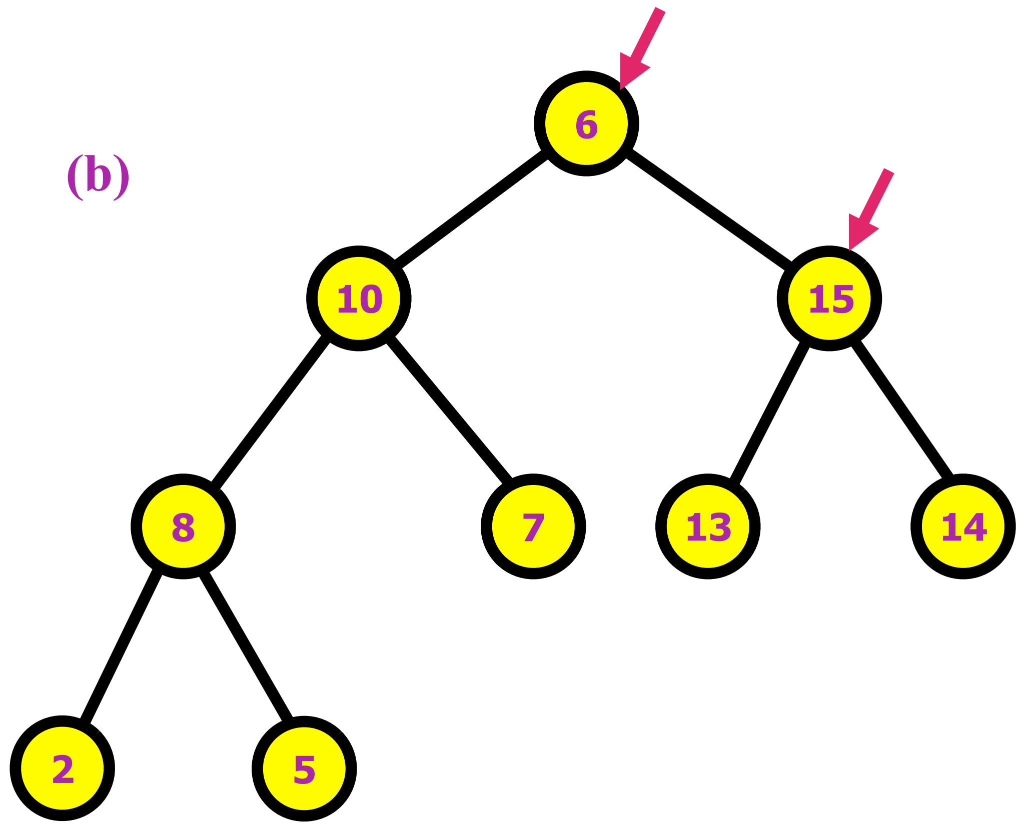
```
heapDelete( el ) {
  extract the element from the root;
  put the element from the last leaf in its place;
  p = the root;
  while p is not a leaf and p < any of its childrens
        swap p with the larger child;
}
```

Deleting Elements from the Heap

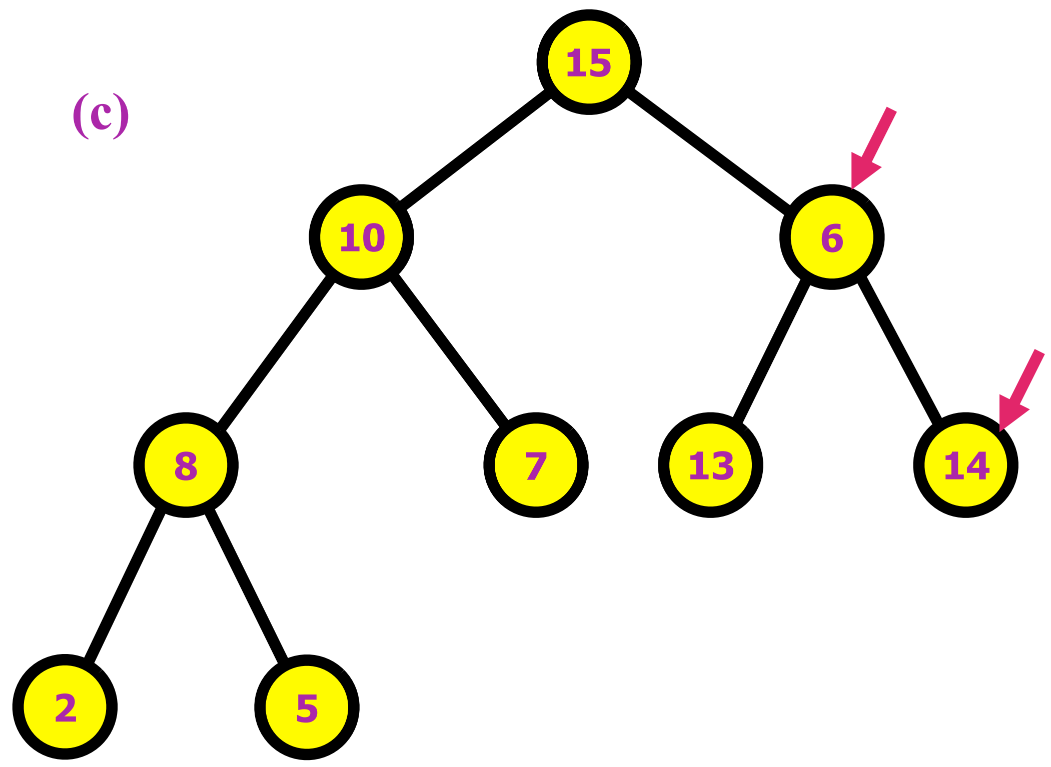
(a)



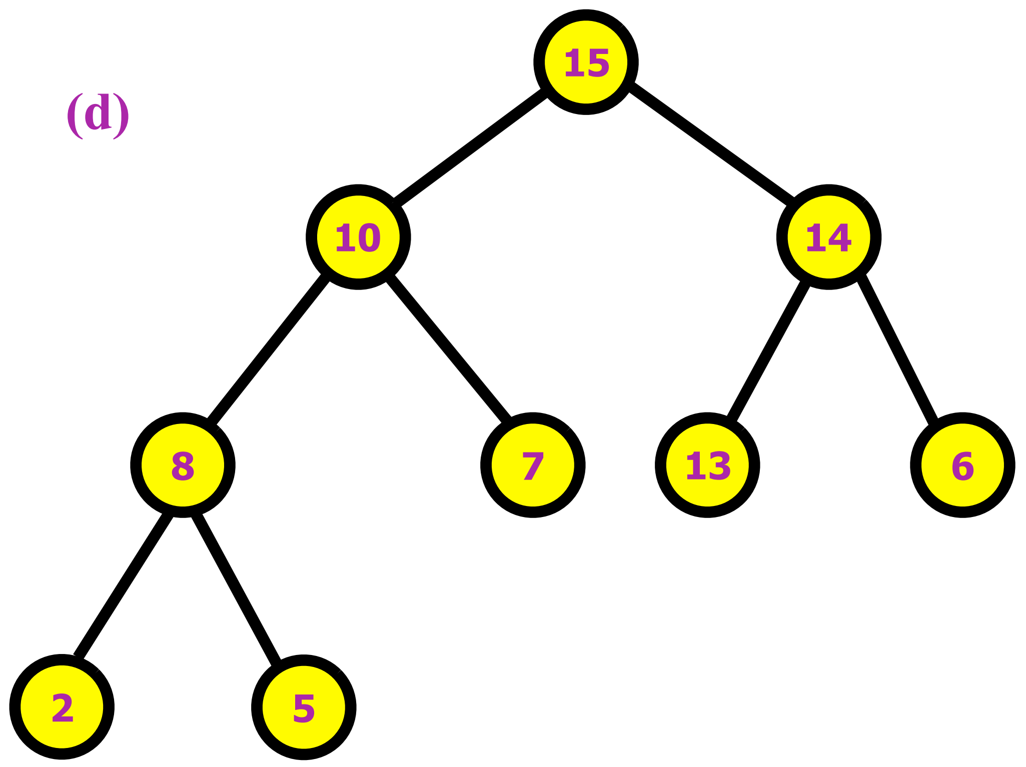
(b)



(c)



(d)



The Heap Sort

- As discussed, selection (linear) sort makes O(n^2) comparisons and is very inefficient, especially for large n,
- Heap sort uses the approach inherited from the selection sort by putting the largest element at the end of the array, then the second largest in front of it, and so on,
- Thus heap sort starts from the end of the array by finding the largest element,

The Heap Sort

- The heap sort algorithm can be easily described in Java as follow:

```
void heapsort( int data[], int size ) {
    for ( int i = size / 2 – 1; i >= 0; i-- )   // create the heap
            moveDown( data, i, size – 1 );
    for ( i  = size – 1; i >= 1; i-- ) {
            swap( data[0], data[i] );    // move the largest item to data[i]
            moveDown( data, 0, i – 1 );   // restore the heap property
    }
}
```

- For example, consider the array:
  - 2, 8, 6, 1, 10, 15, 3, 12, 11
  - The index of the first and last elements are 0 and 1
  - The value of size is 9, i starts from 3 down to 0

Complexity Analysis of the Heap Sort

- In the first phase, to create the heap, moveDown is called O(n) steps, each log(n),
- In the second phase, the root element is exchanged with element i n-1 times which in the worst case causes moveDown to iterate log(i) times to bring the root down to the level of the leaves,
- Thus the number of moves in all executions of moveDown in the second phase is
  - log(1) + log(2) + … + log(n-1) which is O(n log n)

Complexity Analysis of the Heap Sort

- **To conclude:**
  - in the worst case, heap sort requires O(n) steps in the first phase, each with log(n),
  - in the second phase , n-1 swaps and O(n log n) operations to restore the heap property,

- **Which gives:**
  - O(n) * log(n) + O(n log n) + n-1 = O(n log n) exchanges for the whole process in the worst case

Thank You