Intro to Algorithms and Data Structure

CSE 203 – Week 1, Class 1-3

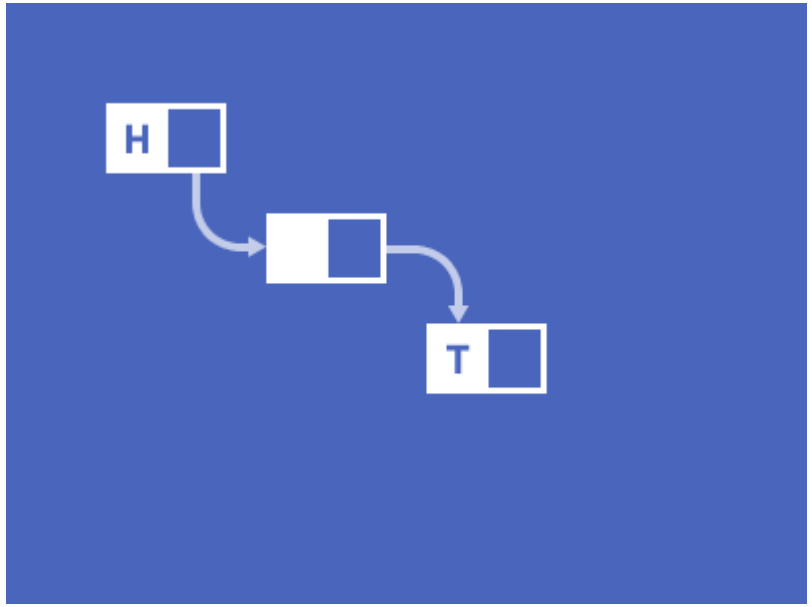
Lec Raiyan Rahman
Dept of CSE, MIST
raiyan@cse.mist.ac.bd

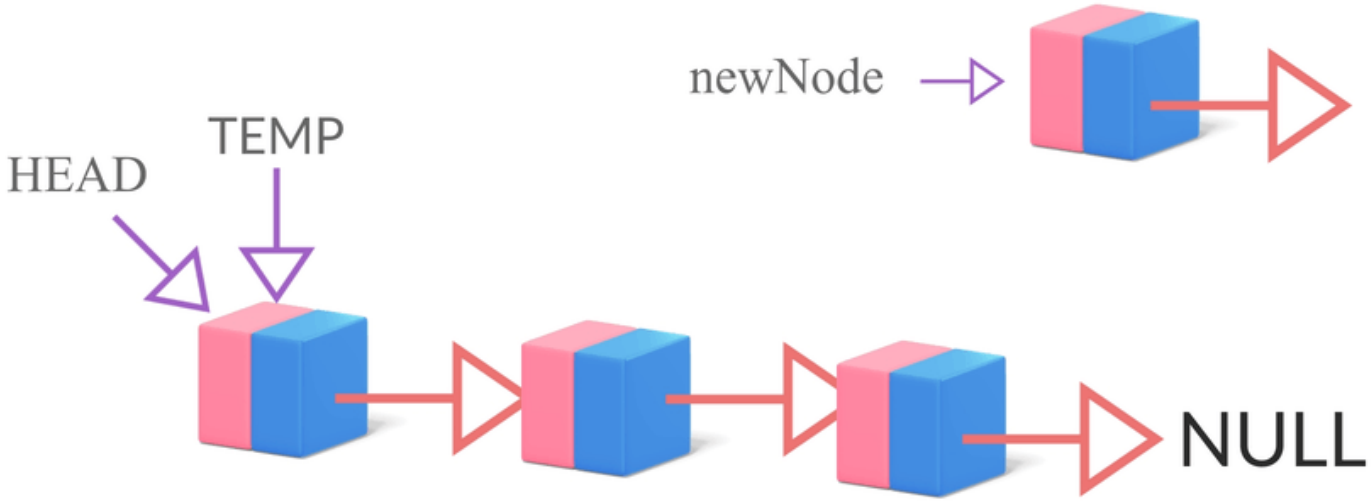
Hello and Good Morning!

Hello and Good Morning!

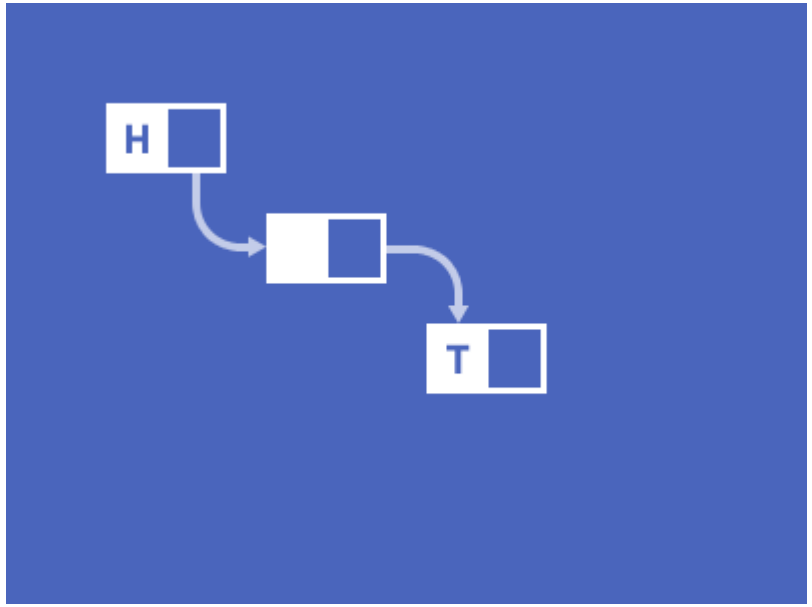
A Little Introduction ☺

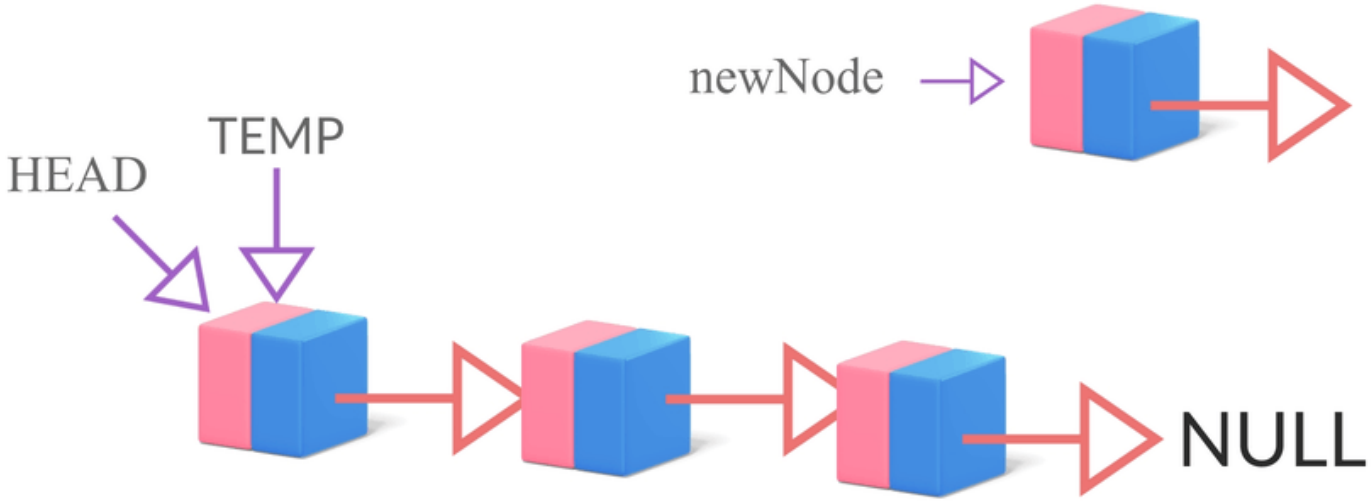
Welcome to Data Structures and Algorithm – I



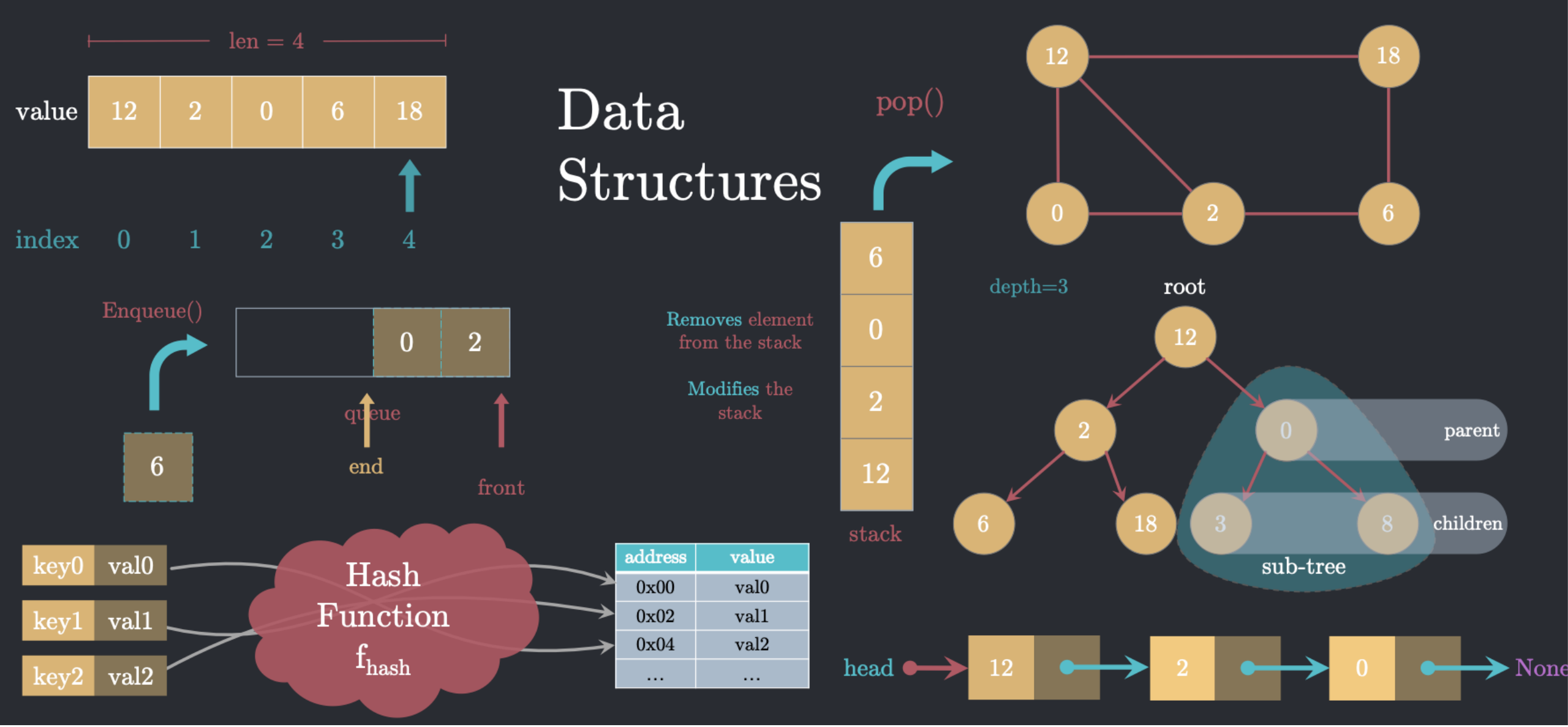


Do you know of Data Structures?





What is this Particular DS?



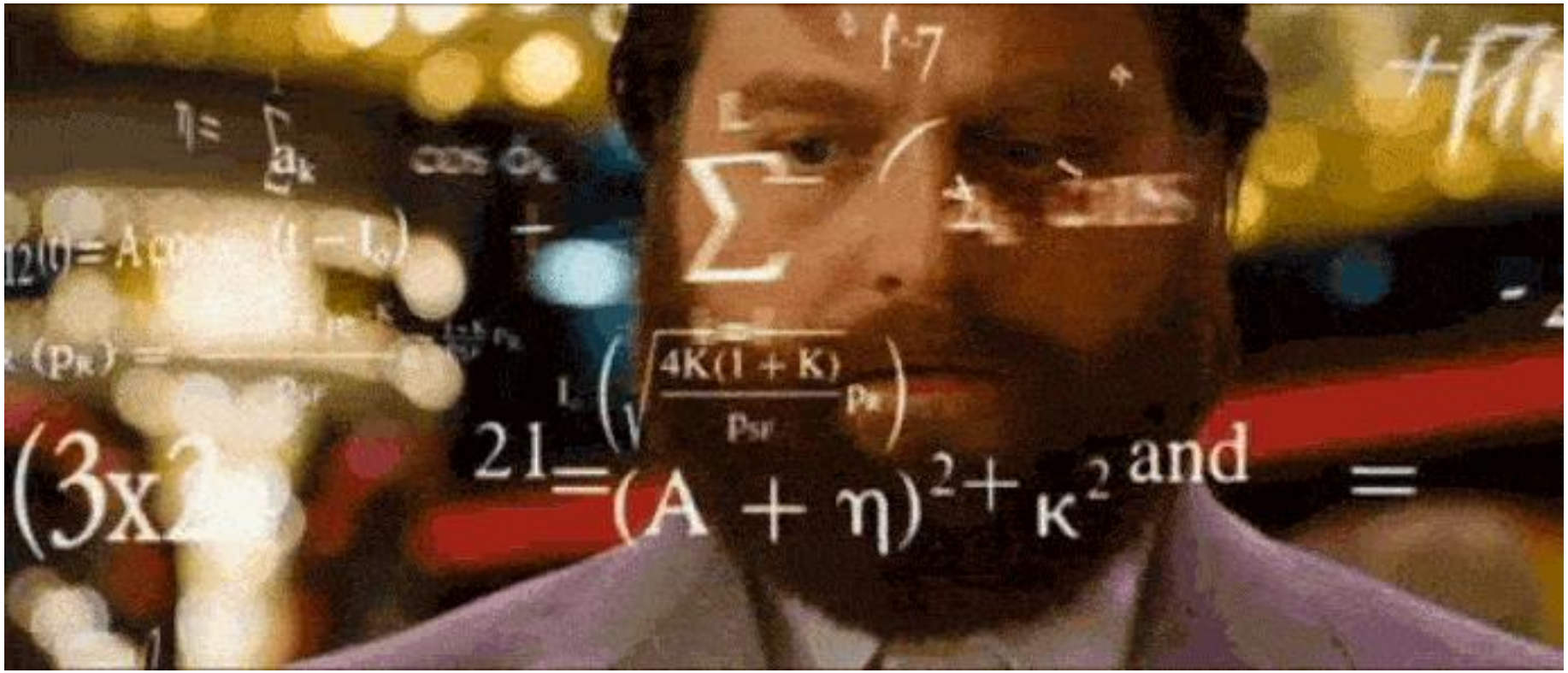
So many data structures, so little time!

But What are Data Structures?

Let’s **Understand** What DS are!

But first, we need to know **why** we need DS.

(Ans: Algorithms!)



So now, what are Algorithms?


CSE Annual Picnic 2023
Stand-Up Comedy
LIVE

Problem Scenario 1

You only remember the face somewhat.
Now, think of a way to find him.

PS:

1. You have to do it **covertly** (can't ask everyone at once and directly)
2. But try to find him by asking the **minimum** number of students

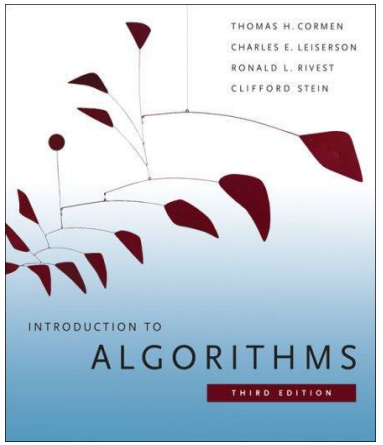
Algorithms

Introduction

The Course

- Purpose: a rigorous introduction to the design and analysis of algorithms
  - Not a programming course
  - Not a math course, either

- Textbook: *Introduction to Algorithms* (3rd edition)
  Cormen, Leiserson, Rivest, and Stein
  - An excellent reference you should own



What is an Algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a tool for solving a well-specified computational problem.
  - An algorithm is said to be **correct** if, for every input instance, it halts with the correct output.
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired output.

What is a Program?

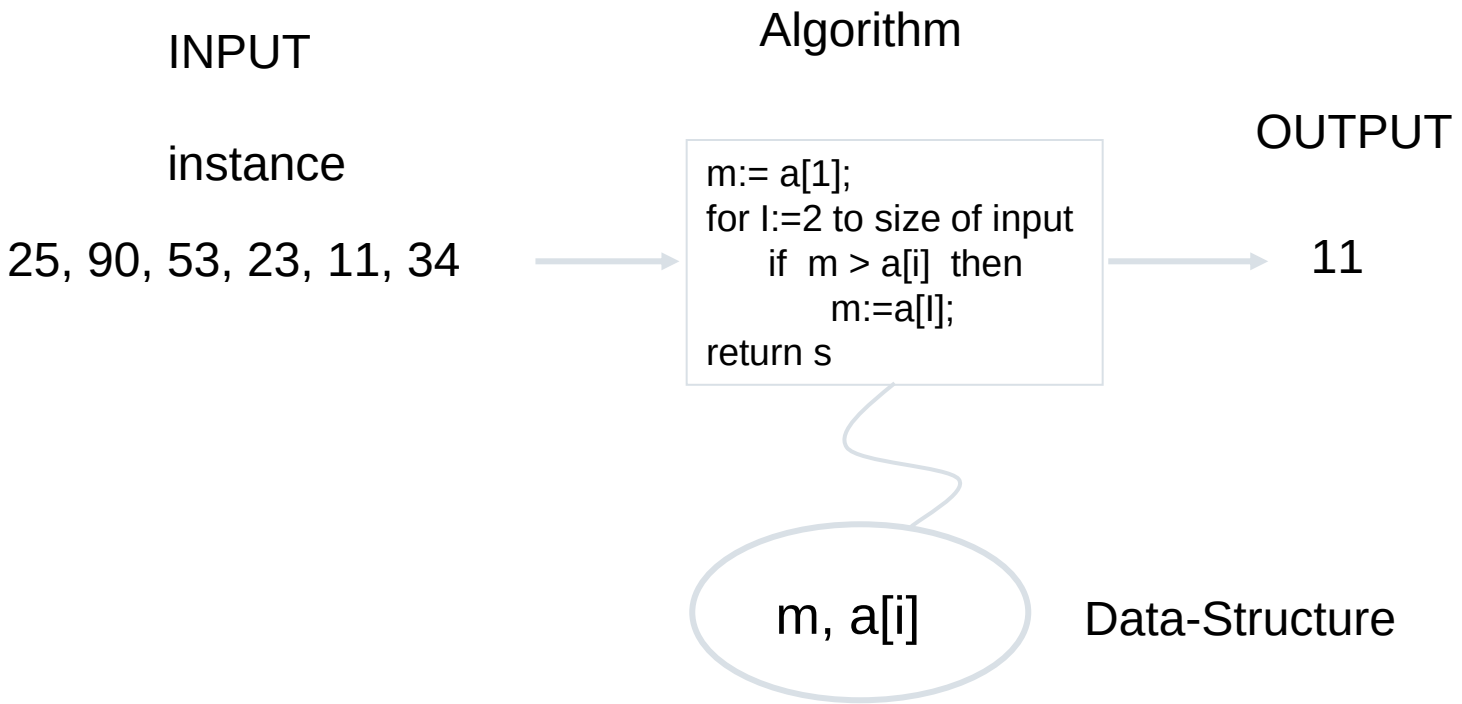
- A program is the expression of an algorithm in a programming language
- A set of instructions which the computer will follow to solve a problem

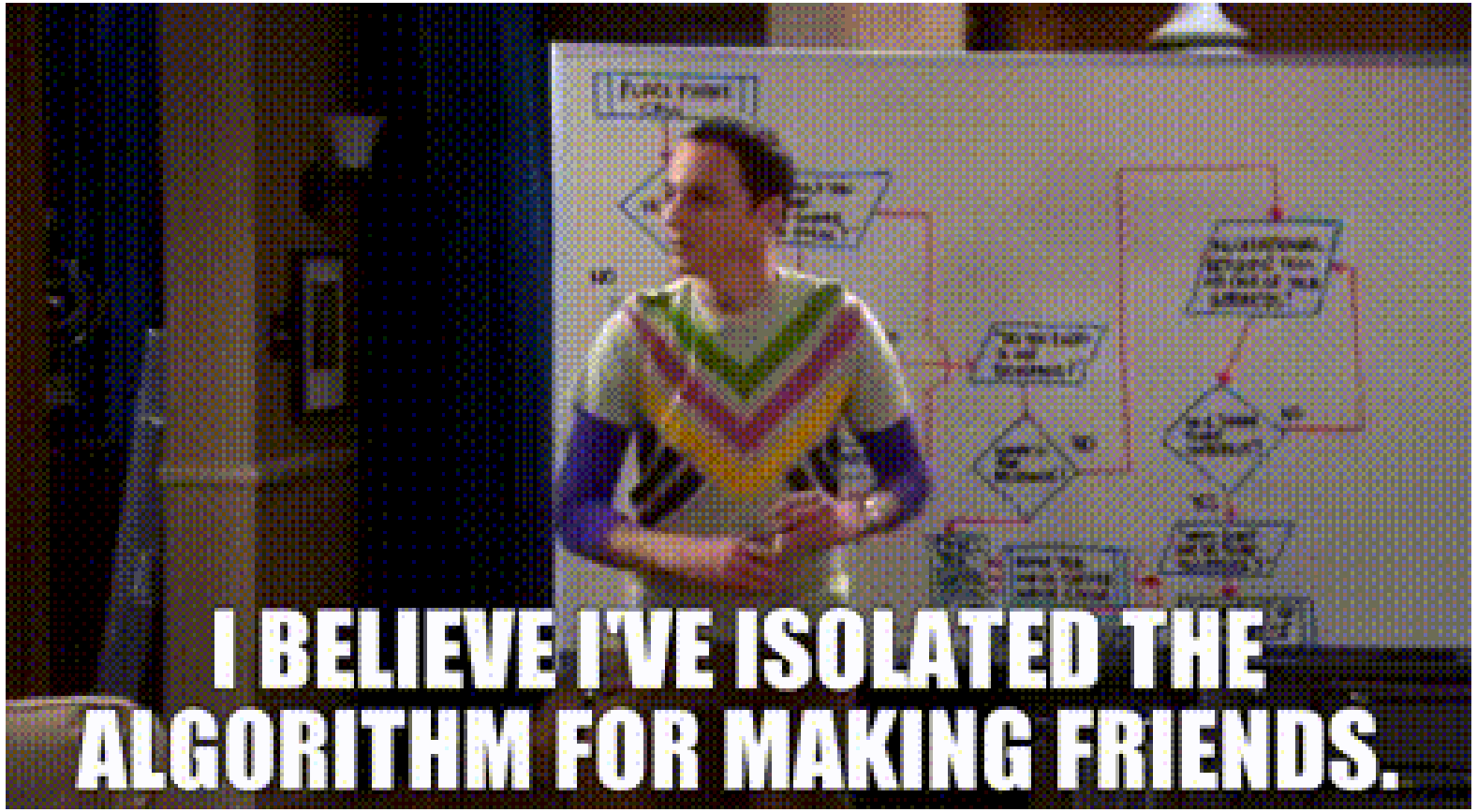
Define a Problem, and Solve It

- **Problem**:
  - Description of Input-Output relationship
- **Algorithm**:
  - A sequence of computational steps that transform the input into the output.
- **Data Structure:**
  - An organized method of storing and retrieving data.
- **Our Task:**
  - Given a problem, design a ***correct*** and ***good*** algorithm that solves it.

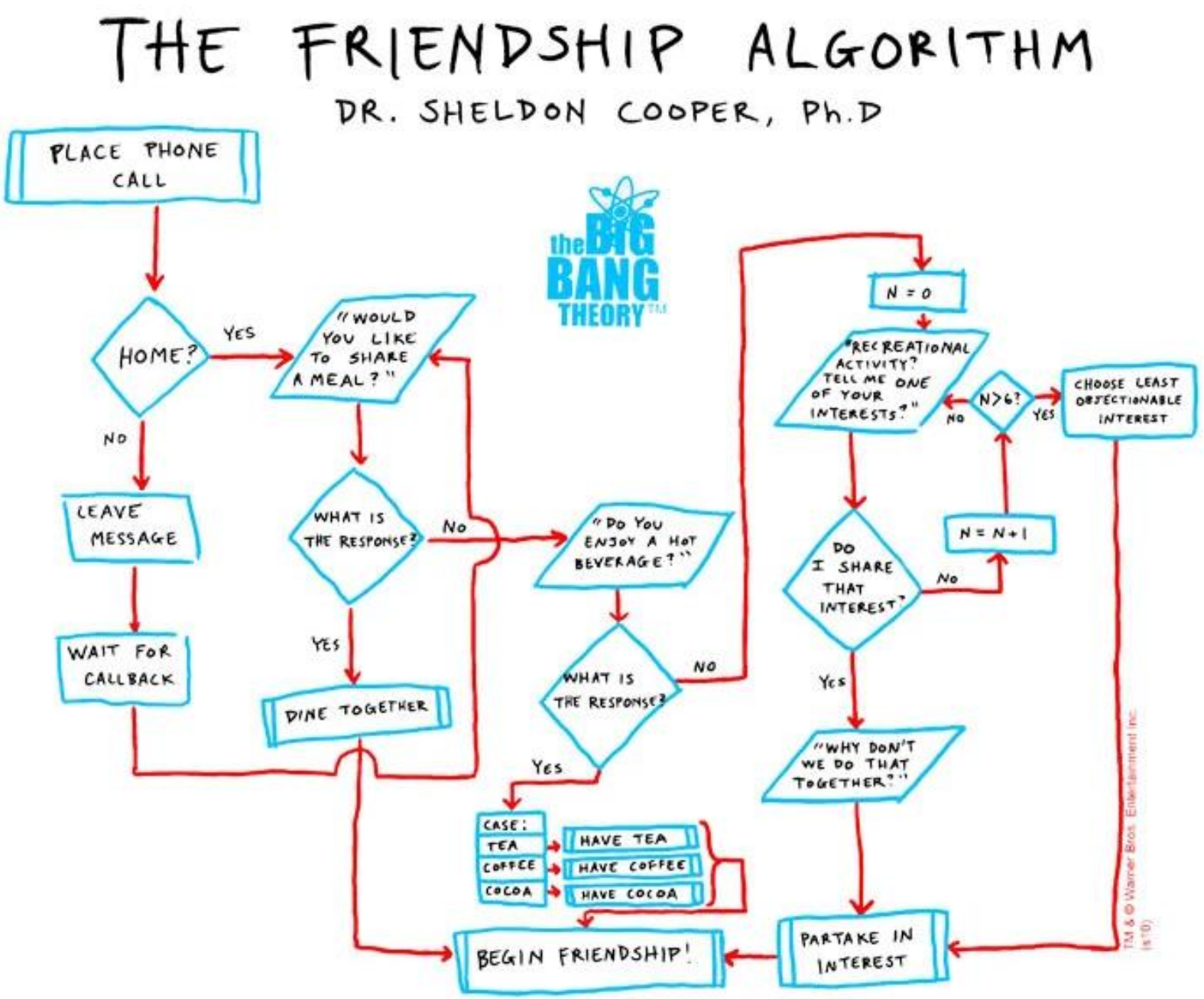
Define a Problem, and Solve It

Problem: Input is a sequence of integers stored in an array.
Output the minimum.





Let’s try another problem, making friends! ☹


THE FRIENDSHIP ALGORITHM
DR. SHELDON COOPER, Ph.D
PLACE PHONE CALL
HOME?
YES
NO
"WOULD YOU LIKE TO SHARE A MEAL?"
LEAVE MESSAGE
WAIT FOR CALLBACK
WHAT IS THE RESPONSE?
No
YES
DINE TOGETHER
"DO YOU ENJOY A HOT BEVERAGE?"
WHAT IS THE RESPONSE?
NO
Yes
CASE:
TEA
COFFEE
COCOA
HAVE TEA
HAVE COFFEE
HAVE COCOA
BEGIN FRIENDSHIP!
N = 0
"RECREATIONAL ACTIVITY? TELL ME ONE OF YOUR INTERESTS?"
N>6?
NO
YES
CHOOSE LEAST OBJECTIONABLE INTEREST
N = N+1
DO I SHARE THAT INTEREST?
No
Yes
"WHY DON'T WE DO THAT TOGETHER?"
PARTAKE IN INTEREST
the BIG BANG THEORY
TM & © Warner Bros. Entertainment Inc.

Problem Scenario 2

Say I want to find the name of the student with the highest CGPA.

PS:

1. Again, I have to do it **covertly**
   (can't ask the question directly and all at once)
2. But I'll try to find the name with the **minimum** number of asks

Problem Scenario 3

Now, what if I needed to find the top 5 students as per CGPA?

Data Structures

Introduction

The Course

- Text Books
  - *Introduction to Algorithms* (Third edition)
    Cormen, Leiserson, Rivest, and Stein
    - An excellent reference you should own

  - *Data Structures and Algorithms in C++*
    Goodrich, Tamassia, and Mount

What is a Data Structure?

- Data is a collection of facts, such as values, numbers, words, measurements, or observations.
- Structure means a set of rules that holds the data together.

- A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
  - Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.
  - Data structures provide a means to manage huge amount of data efficiently.
  - Usually, efficient data structures are a key to designing efficient algorithms.
  - Data structures can be nested.

Types of Data Structures

- Data structures are classified as either
  - Linear (*e.g*, arrays, linked lists), or
  - Nonlinear (*e.g*, trees, graphs, etc.)

- A data structure is said to be linear if it satisfies the following four conditions
  - There is a unique element called the first
  - There is a unique element called the last
  - Every element, except the last, has a unique successor
  - Every element, except the first, has a unique predecessor

- There are two ways of representing a linear data structure in memory
  - By means of sequential memory locations (arrays)
  - By means of pointers or links (linked lists)

Arrays

- Data is often available in tabular form
- Tabular data is often represented in arrays
- Matrix is an example of tabular data and is often represented as a 2-dimensional array
  - Matrices are normally indexed beginning at 1 rather than 0

Properties of Arrays

- The components of an array are all of the same type.
- Array is a random access data structure.
- Array is a static data structure.
- Access time for an array element is constant, that is, $O(1)$.
- An array is a suitable structure **when a small number of insertions and deletions are required**.
- An array is a suitable structure **when a lot of searching and retrieval are required**.

Why? We'll see soon!

Parameters of Arrays

- **Base Address (b):** The memory address of the first byte of the first array component.
- **Component Length (L):** The memory required to store one component of an array.
- **Upper and Lower Bounds (l_i, u_i):** Each index type has a smallest value and a largest value.
- **Dimension**

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts index value to component address

- **Linear (1 D) Arrays:**

  a : array $[l_1 .. u_1]$ of element_type

  Then addr($a[i]$) $= b + (i - l_1) \times L$

  $= c_0 + c_1 \times i$

  Therefore, the time for calculating the address of an element is same for any value of i.

  Hence, lookup time for any elements in the array is constant and the same.

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts index value to component address

- **2 D Arrays:**

  a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

  In which order are the elements stored?
  - Row major order (C, C++, Java support it)
  - Column major order (Fortran supports it)

Representation of Arrays in Memory

- **2 D Arrays:**

The elements of a 2-dimensional array a are declared as:

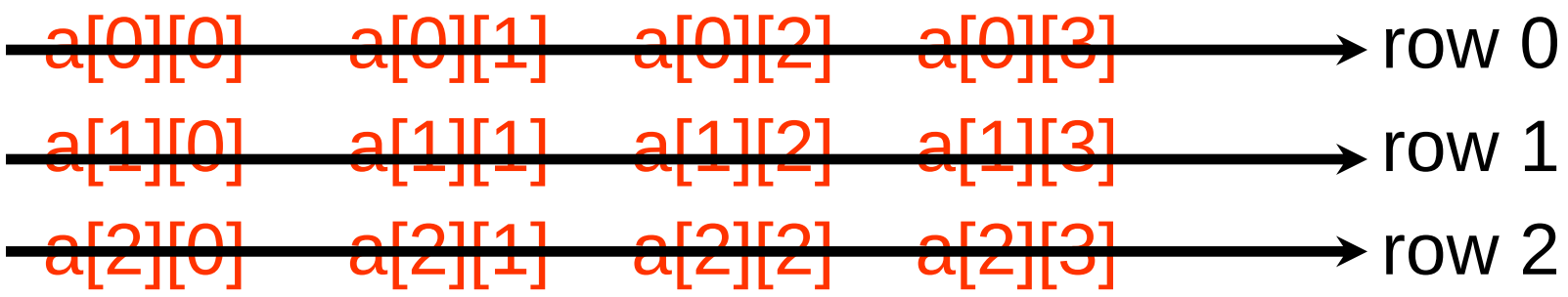
int a[3][4];

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

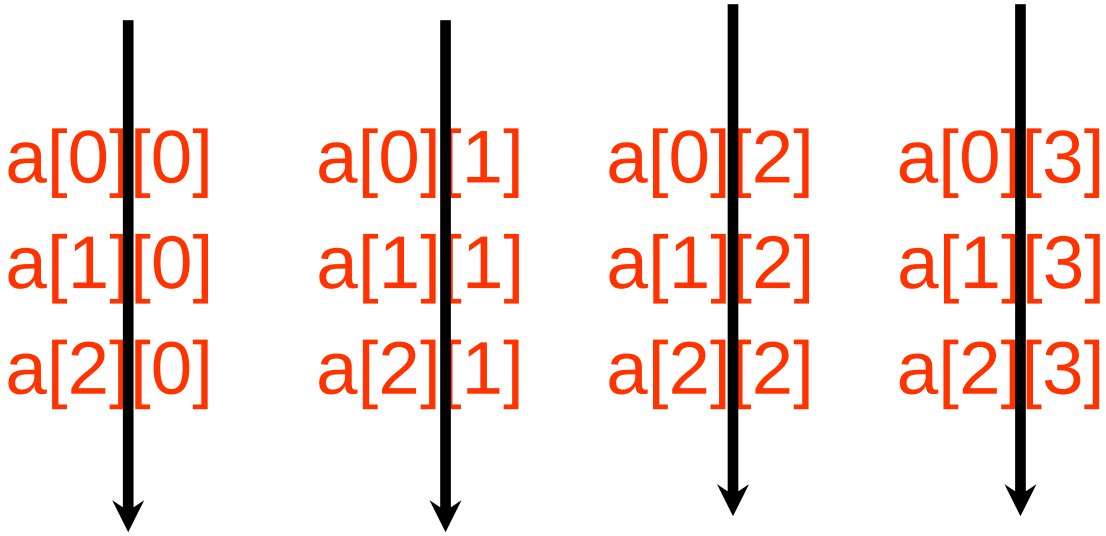
Representation of Arrays in Memory

Row Major Order:



Representation of Arrays in Memory

Column Major Order:



Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts Index value to component address

- **2 D Arrays (Row Major Order):**

a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

Then addr$(a[i, j]) = b + (i - l_1)\times(u_2 - l_2 + 1)\times L + (j - l_2)\times L$

$= c_0 + c_1\times i + c_2\times j$

Therefore, the time for calculating the address of an element is same for any value of (i, j).

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts Index value to component address

- **2 D Arrays (Row Major Order):**

  a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

int a[3][4];

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Then addr($a[i, j]$) = $b + (i - l_1)\times(u_2 - l_2 + 1)\times L + (j - l_2)\times L$

(move past many full rows till I reach target row)

(move to a particular position of the target row)

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts Index value to component address

- **2 D Arrays (Column Major Order):**

  a : array $[l_1 .. u_1, l_2 .. u_2]$ of element_type

  Then addr$(a[i, j]) = b + (j - l_2)\times(u_1 - l_1 + 1)\times L + (i - l_1)\times L$

  $= c_0 + c_1\times i + c_2\times j$

  Therefore, the time for calculating the address of an element is same for any value of (i, j).

  Hence, lookup time for any elements in a 2D array is also constant and the same.

Representation of Arrays in Memory

- **Array Mapping Function (AMF)**
  - AMF converts Index value to component address

- **3 D Arrays :**

a : array $[l_1 .. u_1, l_2 .. u_2, l_3 .. u_3]$ of element_type

$$\text{Then addr}(a[i, j, k]) = b + (i - l_1)\times(u_2 - l_2 + 1)\times(u_3 - l_3 + 1)\times L + (j - l_2)\times(u_3 - l_3 + 1)\times L + (k - l_3)\times L$$

$$= c_0 + c_1\times i + c_2\times j + c_3\times k$$

Therefore, the time for calculating the address of an element is same for any value of (i, j, k).

Arrays in C/C++

- Arrays are defined to be a sequence/set of data elements of the same type. Having an array, each array element can be accessed by its position in the sequence of the array.
- **Declaration of the Arrays**: Any array declaration contains:
  - the *array name*,
  - the *element type*
  - the *array size* and
  - the *dimension*
- **Examples:**
  - int a[20], b[3], c[7];
  - float f[5][2], d[2];
  - char m[4], n[20];

Arrays in C/C++

- **Initialization** of an array is the process of assigning initial values. Typically declaration and initialization are combined.

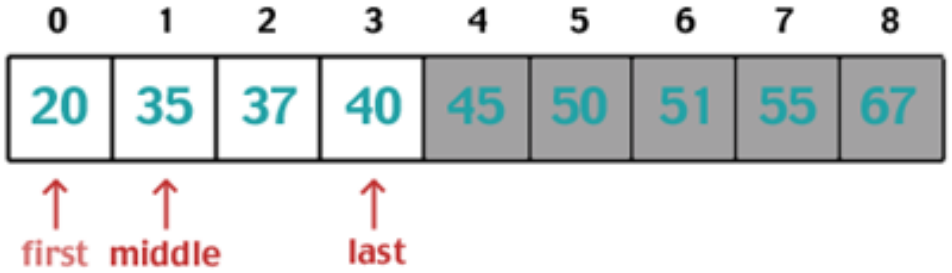
- **Examples:**
  - int a[4]={1, 3, 5, 2};
  - float b[3]={2.0, 5.5, 3.14};
  - char name[5]= {‘N’, ‘a’, ‘o’, ‘m’ , ‘i’};

Example

- Write a program to calculate and print the average of the following array of integers.
  (4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

```
#include<stdio.h>
#define SIZE 10
int main() {
      int x[10] = {4, 3, 7, -1, 7, 2, 0, 4, 2, 13};
      int i, sum=0;
      float av;
      for(i=0; i<SIZE; i++)
            sum = sum + x[i];
      av = (float)sum / SIZE;
      printf("The average of the numbers = %.2f\n", av);
      return 0;
}
```

Arrays,
Searching & Sorting



Searching

- The process of finding a particular element in an array is called searching. There two popular searching techniques:
  - *Linear search*, and
  - *Binary search*.
- The ***linear search*** compares each array element with the ***search key***.
- If the ***search key*** is a member of the array, typically the location of the search key is reported to indicate the presence of the search key in the array. Otherwise, a *sentinel* value is reported to indicate the absence of the search key in the array.

Linear Search

- Each member of the array is visited until the search key is found.

- **Example:**

  Write a program to search for the search key entered by the user in the following array:

  (9, 4, 5, 1, 7, 78, 22, 15, 96, 45)

  You can use the linear search in this example.

Linear Search

```
/* This program is an example of the Linear Search*/
#include <stdio.h>
#define SIZE 10
int LinearSearch(int [], int);
int main() {
      int a[SIZE]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45};
      int key, pos;
      printf("Enter the Search Key\n");
      scanf("%d", &key);
      pos = LinearSearch(a, key);
      if(pos == −1)
            printf("The search key is not in the array\n");
      else
            printf("The search key %d is at location %d\n", key, pos);
      return 0;
}
```

Linear Search

```
int LinearSearch (int b[ ], int skey) {
        int i;
        for (i=0; i < SIZE; i++)
                if(b[i] == skey)
                        return i;
        return −1;
}
```

Sorting

- **Sorting** an array is the ordering the array elements in
  - ***ascending*** (increasing: from min to max), or
  - ***descending*** (decreasing: from max to min) order.

- **Example:**
  - $\{2, 1, 5, 3, 2\} \rightarrow \{1, 2, 2, 3, 5\}$ ***ascending*** order
  - $\{2, 1, 5, 3, 2\} \rightarrow \{5, 3, 2, 2, 1\}$ ***descending*** order

Bubble Sort

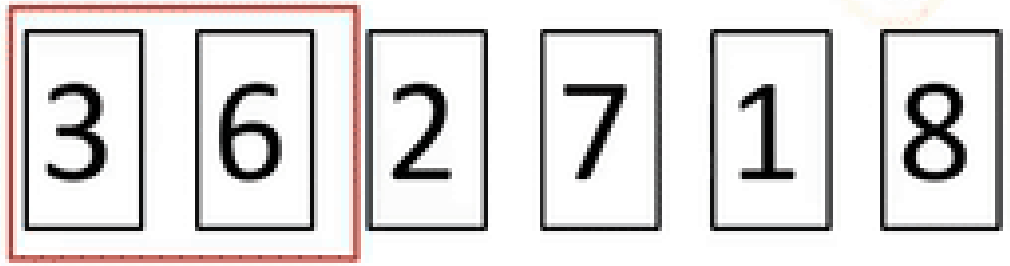
- Smaller values in the list gradually “bubble” their way upward to the top of the array.

- The technique makes several passes through the array. On each pass successive pairs of elements are compared. If the pair is in increasing order (or equal) the pair is unchanged. If a pair is in descending order, their values are swapped in the array.

Consider how air bubbles underwater find their way up, because they’re lighter.

Similarly, in bubble sort, smaller values bubble their way to the start of the array!

Bubble Sort



But how many times does this need to happen to confidently say the array will be sorted?

Bubble Sort

Pass = 1	Pass = 2	Pass = 3	Pass = 4
2 1 5 3 2	**1 2** 3 2 5	**1 2** 2 3 5	**1 2** 2 3 5
1 **2 5** 3 2	1 **2 3** 2 5	1 **2 2** 3 5	1 **2 2** 3 5
1 2 **5 3** 2	1 2 **3 2** 5	1 2 **2 3** 5	1 2 **2 3** 5
1 2 3 **5 2**	1 2 2 **3 5**	1 2 2 **3 5**	1 2 2 **3 5**
1 2 3 2 5	1 2 2 3 5	1 2 2 3 5	1 2 2 3 5

Solution reached. But remember, the computer doesn't know that for sure!

- Underlined pairs show the comparisons. For each pass there are **size-1** comparisons.
- Total number of comparisons = **(size-1)^2**

Bubble Sort

Pass = 1	Pass = 2	Pass = 3	Pass = 4
5 4 3 2 1	**4 3** 2 1 5	**3 2** 1 4 5	**2 1** 3 4 5
4 **5 3** 2 1	3 **4 2** 1 5	2 **3 1** 4 5	1 **2 3** 4 5
4 3 **5 2** 1	3 2 **4 1** 5	2 1 **3 4** 5	1 2 **3 4** 5
4 3 2 **5 1**	3 2 1 **4 5**	2 1 3 **4 5**	1 2 3 **4 5**
4 3 2 1 5	3 2 1 4 5	2 1 3 4 5	1 2 3 4 5

- Underlined pairs show the comparisons. For each pass there are **size-1** comparisons.
- Total number of comparisons = **(size-1)^2**

Bubble Sort : C Code

```
/* This program sorts the array elements in the ascending order*/

#include <stdio.h>
#define SIZE 5
void BubbleSort(int [ ]);

int main() {
      int a[SIZE]= {2, 1, 5, 3, 2};
      int i;
      printf("The elements of the array before sorting\n");
      for (i=0; i < SIZE; i++)
            printf("%4d", a[i]);
      BubbleSort(a);
      printf("\n\nThe elements of the array after sorting\n");
      for (i=0; i< SIZE; i++)
            printf("%4d", a[i]);
      return 0;
}
```

Bubble Sort : C Code

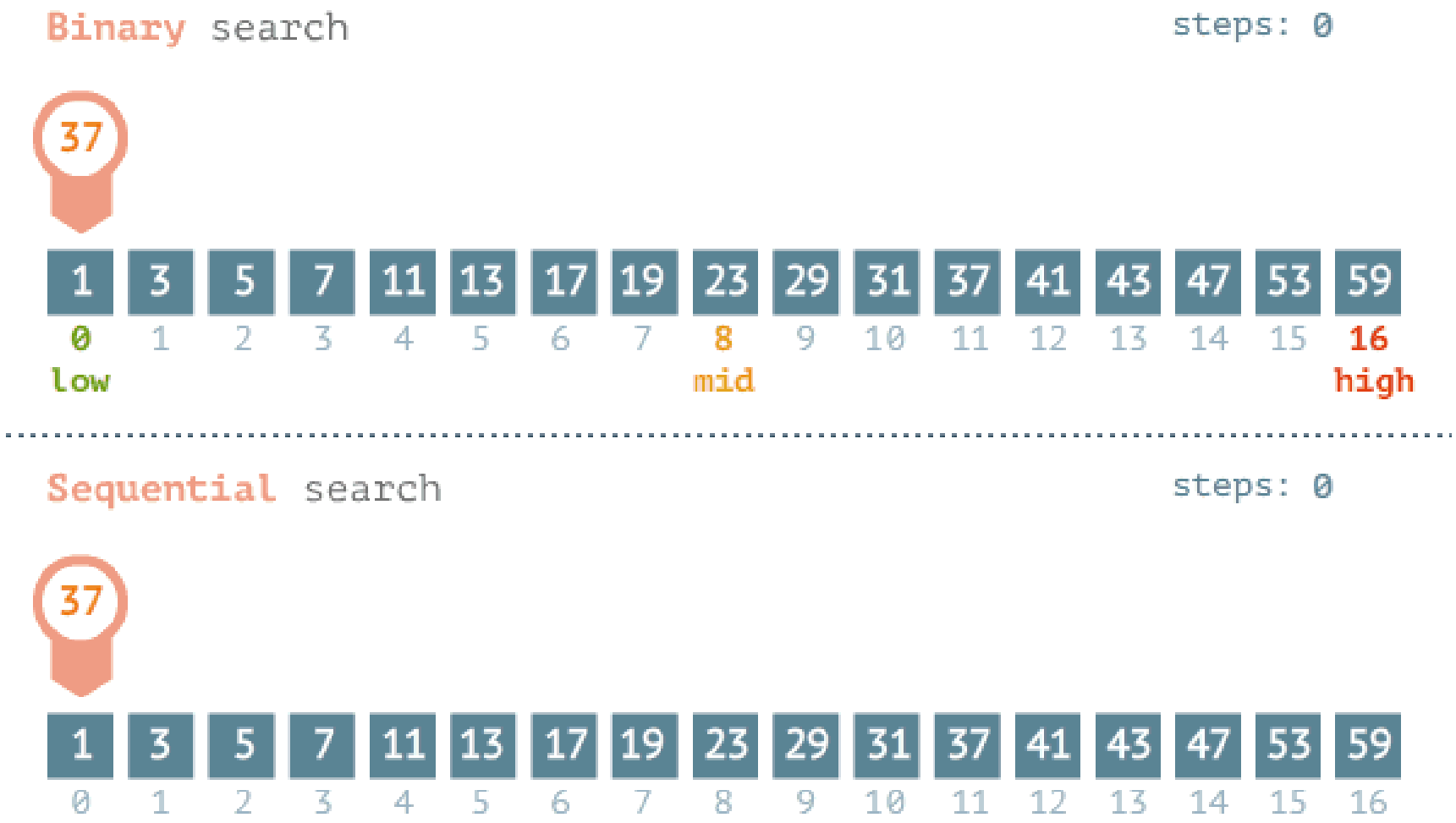
```
void BubbleSort(int A[ ]) {
int i, pass, temp;
for (pass=1; pass < SIZE; pass++)
for (i=0; i < SIZE-1; i++)
if(A[i] > A[i+1]){
temp = A[i];
A[i] = A[i+1];
A[i+1] = temp;
}
}
```

Binary Search

- Given a sorted array, ***Binary Search*** algorithm can be used to perform fast searching of a search key on the **sorted** array.
- The following program uses pointer notation to implement the binary search algorithm for the search key entered by the user in the following array:

(3, 5, 9, 11, 15, 17, 22, 25, 37, 68)

Binary Search



Binary Search

Binary Search Algorithm: The basic steps to perform Binary Search are:

1. Sort the array in ascending order.
2. Set the low index to the first element of the array and the high index to the last element.
3. Set the middle index to the average of the low and high indices.
4. If the element at the middle index is the target element, return the middle index.
5. If the target element is less than the element at the middle index, set the high index to the middle index – 1. Calculate new mid.
6. If the target element is greater than the element at the middle index, set the low index to the middle index + 1. Calculate new mid.
7. Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Binary Search

```
#include <stdio.h>
#define SIZE 10
int BinarySearch(int [ ], int);
int main(){
      int a[SIZE]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68};
      int key, pos;
      printf("Enter the Search Key\n");
      scanf("%d",&key);
      pos = BinarySearch(a, key);
      if(pos == −1)
          printf("The search key is not in the array\n");
      else
          printf("The search key %d is at location %d\n", key, pos);
      return 0;
}
```

Binary Search

```
int BinarySearch (int A[], int skey){
    int low=0, high=SIZE−1, middle;
    while(low <= high){
        middle = (low+high)/2;
        if (skey == A[middle])
            return middle;
        else if(skey <A[middle])
            high = middle − 1;
        else
            low = middle + 1;
    }
    return −1;
}
```

Computational Complexity

- The *Computational Complexity* of the *Binary Search* algorithm is measured by the maximum (worst case) number of comparisons it performs for searching operations.

  The searched array is divided by 2 for each comparison/iteration.

  Therefore, the maximum number of comparisons is measured by:

  $\log_2(n)$, where n is the size of the array.

- **Example**:

  If a given sorted array 1024 elements, then the maximum number of comparisons required is:

  $\log_2(1024) = 10$ (only 10 comparisons is enough)

Computational Complexity

- Note that the *Computational Complexity* of the *Linear Search* is the maximum number of comparisons you need to search the array. As you are visiting all the array elements in the worst case, then, the number of comparisons required is:

  n (n is the size of the array)

- **Example**:

  If a given an array of 1024 elements, then the maximum number of comparisons required is:

  $n = 1024$ (As many as 1024 comparisons may be required)

Problem Scenario 3

Now, what if I needed to find the top 5 students as per CGPA?

Steps:

1. Take input of all CGPAs and store in an array - DS.
2. Bubble sort the arrays - algo. (expand the steps!)
3. Return the last 5 arrays (if sorted in ascending order).

Problem Scenario 4

What of you were asked to find if a particular ID has registered for CSE 203?

Steps:

1. Take input of all IDs who've registered for CSE 203 and store in an array - DS.
2. Do Linear Search on the array. - algo

Steps:

1. Take input of all IDs who've registered for CSE 203 and store in an array - DS.
2. Bubble sort the arrays - algo
3. Do Binary Search on the sorted array. - algo

Thank you

Working with Dynamic Arrays - Data Structures and Algorithm Sessional-I

CSE 204 – Week 2

LEC RAIYAN RAHMAN
Dept of CSE, MIST
raiyan@cse.mist.ac.bd

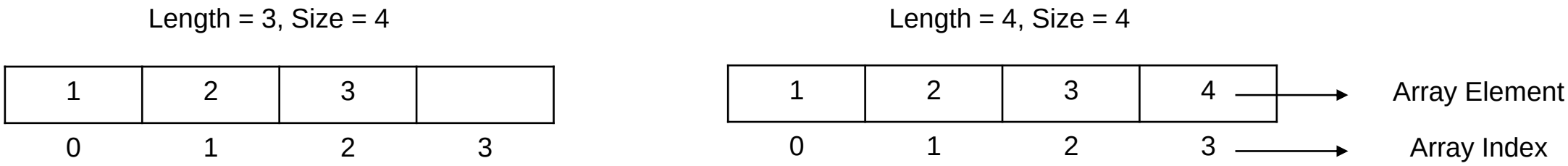
Arrays

- An array is a collection of homogeneous (same type) data items stored in contiguous memory locations.
  - For example if an array is of type “int”, it can only store integer elements and cannot allow the elements of other types such as double, float, char etc.

- Array is called data structure because it also helps in organizing and efficiently using a large number of data of a particular type.
  - Suppose, in case we need to store and manage the roll numbers of fixed number of students in a class then we consider to use an array of integer type.

Array

- The items of an array are allocated at adjacent memory locations. These memory locations are called **elements** of that array.
- The total number of elements in an array is called **length**.
- The details of an array are accessed about its position. This reference is called **index**.
- The maximum length of an array is called the **size** of the array.



Static Array List – Sample Operations

- Searching:
  - Index of an element
- Inserting:
  - Element at the last
  - Element at any position
- Deleting:
  - Last element
  - Element at any position

Static Array

- An array created at compile time by specifying size in the source code.
- **It has a fixed size and cannot be modified at run time.**
- The process of allocating memory at compile time is known as static memory allocation.
- The arrays that receives static memory allocation are called static arrays.
  - *Advantage:* It has efficient execution time. (no processing needed at runtime)
  - *Disadvantage:* Space cannot be expanded or reduced when required. (limitation, either wasted space or not enough space)

Problems with Static Array

- We are usually wasting a lot of space.
- Or we are not being able to accommodate space in case the client/user needs more.
- Either way, we're guessing the required space (usually a upper limit) and not being exact in accommodating space.

Solution - Dynamic Arrays

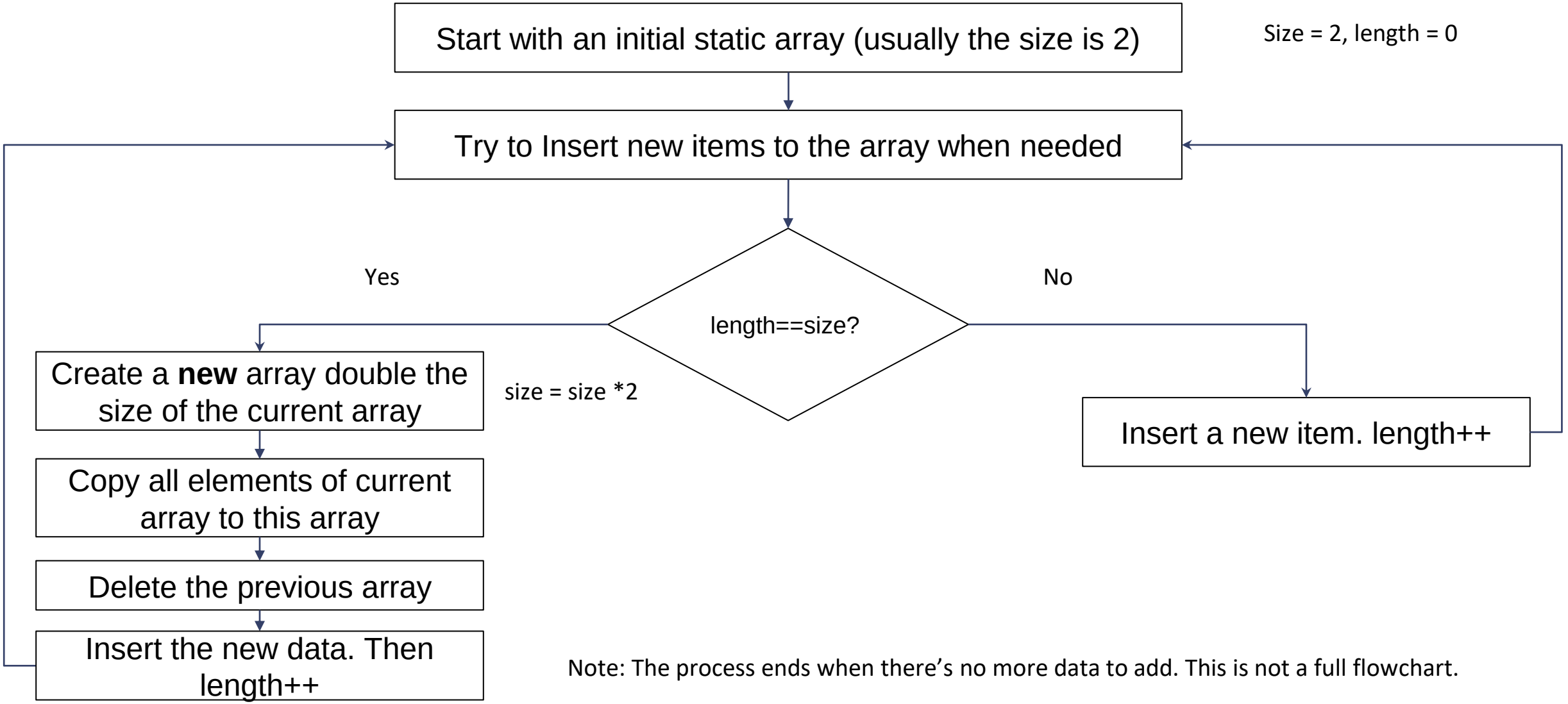
- Arrays that ‘grow’ as we store more data into it.
- It’s a space efficient way of storing information.
- This ‘growing’ of the array as per user’s need, happens at runtime and needs a little bit of processing. This processing might slow the program down a bit, but there are ways of optimizing that as well, so that execution time isn’t affected too much.
- So, formally, a dynamic array is a contiguous area of memory (or a block of memory) the size of which grows dynamically as we insert new data.

How do Dynamic Arrays Work?

It’s actually an static array at a given time. Then we just **increase the size** when the max capacity of the current array is exceeded.

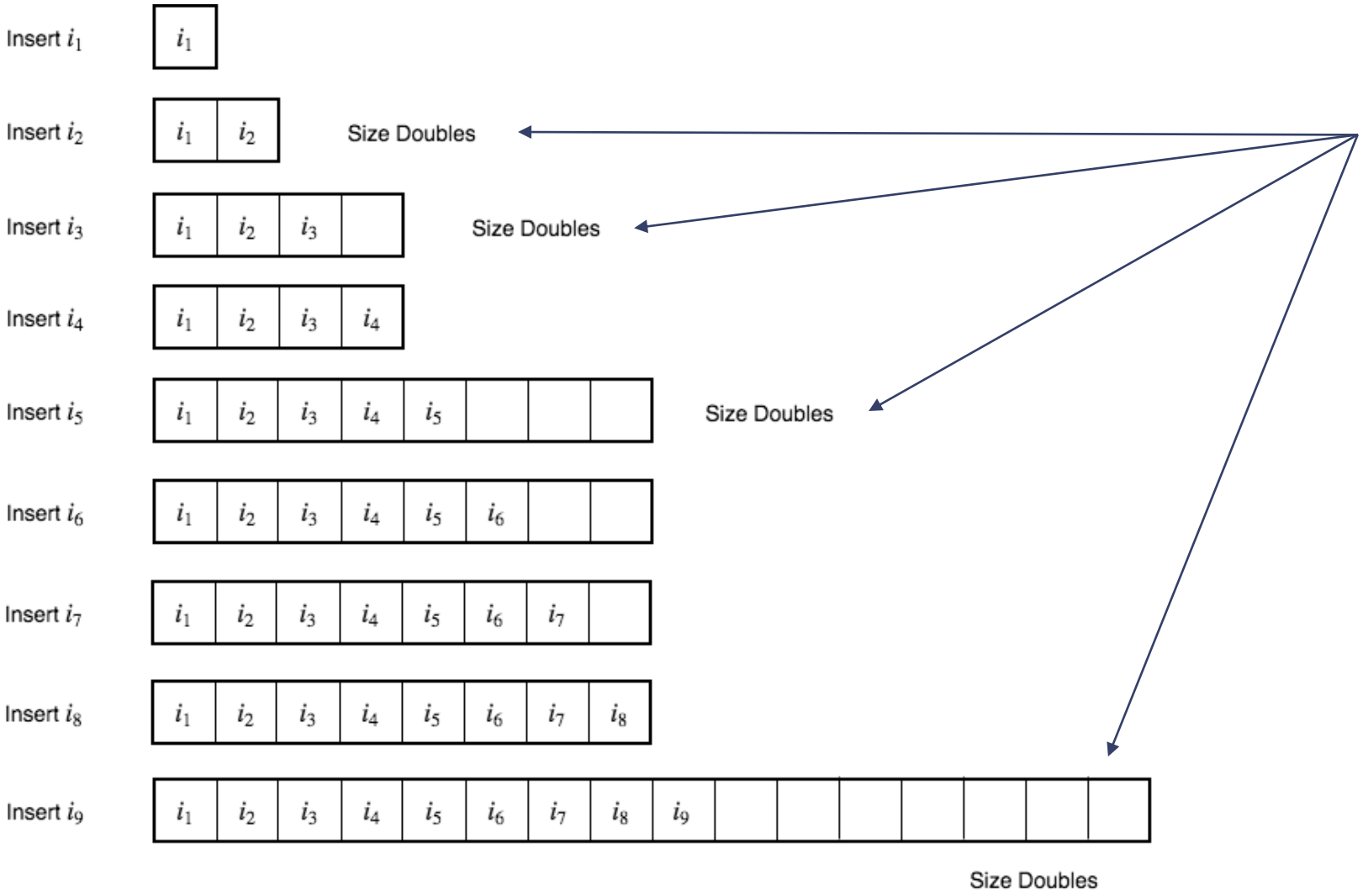
Well, there’s a bit more to it though, let’s see!

How do Dynamic Arrays Work?



Note: The process ends when there's no more data to add. This is not a full flowchart.

How do Dynamic Arrays Work?



New array with double the size is created and previous array's existing elements are copied to it. This will slow down the runtime.

But good thing is, this does not happen often and as per our optimization we can limit it even more.

Implementing Dynamic Arrays

- We can make a structure using C or C++ that stores data the way we described in the previous slide.
- Or, we could also use a data structure defined already in C++ Standard Template Library (STL), which is called **Vector**.
- There are advantages and disadvantages to both ways. So, let's learn both!

Some Concepts We'll Need: Constructors

- We'll learn more about it in CSE 205 (OOP Theory).
- For now, understand that, they are functions that are automatically called whenever a new object of a particular class is created. The name of this function is the **same** as the class's name.
- The job of this function is usually to allocate the memory for each (or some) or the attributes (variables) of a class and also to initialize them with a value.
- However, we can rewrite these functions and write whatever we need there. So, if there are some additional commands i want the program to do, we can place them in the constructor function.

Some Concepts We'll Need: Constructors

```
 6  template <class T> // this is a template class. It's he
 7  class DynamicArray {
 8  private:
 9      T * list; //a pointer that we'll use to store whate
10      int max_size; //to store the current array's max pc
11      int length; //to store array's current length
12  public:
13      DynamicArray() //a constructor function, it's calle
14
15      {
16          // initially allocate a single memory block
17          max_size = 2;
18          list = new T[max_size];   // like int * a = new
19          length = 0;
20      }
```

The Constructor for our program (DAL from scratch). Notice that the function name is same as the class and we're initializing each of the attributes of our class as we need.

Some Concepts We'll Need: Destructors

- We'll learn more about it in CSE 205 (OOP Theory).
- Again, for now, understand that, they are functions that are automatically called when we terminate the program and/or the objects created are no longer needed.
- It's job is to deallocate the memory occupied by the objects of the class.
- Again, we can rewrite it and do anything else to do (if we need it) when the objects are deallocated.
- In this program, we just deallocate the memory occupied by our DAL.

Some Concepts We'll Need: Destructors

```
61
62      ~DynamicArray() //a destructor
63      {
64          delete [] list;
65      }
66
```

The Destructor for our program (DAL from scratch). Notice that the function name is same as the class with a ~ sign before it and just deallocating the memory.

Let’s see it’s functionality directly in the sample code

Code 1: Implementing Dynamic Arrays from scratch in C++

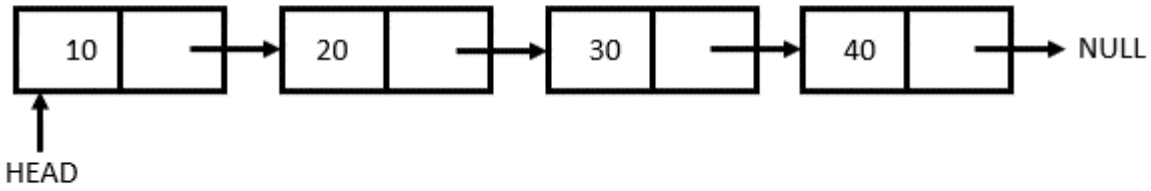
Code 2: Implementing Dynamic Arrays with std::Vector (C++)

But what is STL and what is std::vector?

- The C++ STL (Standard Template Library) is a powerful set of built-in C++ classes to provide functionality that can be used to implement many popular and commonly used algorithms and data structures like lists, **dynamic arrays**, queues, and stacks.
- One such template class is Vector, which essentially acts just like a dynamic array list and works exactly as we described in slide 10.

Thank you

Linked Lists



OUTPUT:

CSE 203 – Week 3+4

LEC RAIYAN RAHMAN

Dept of CSE, MIST

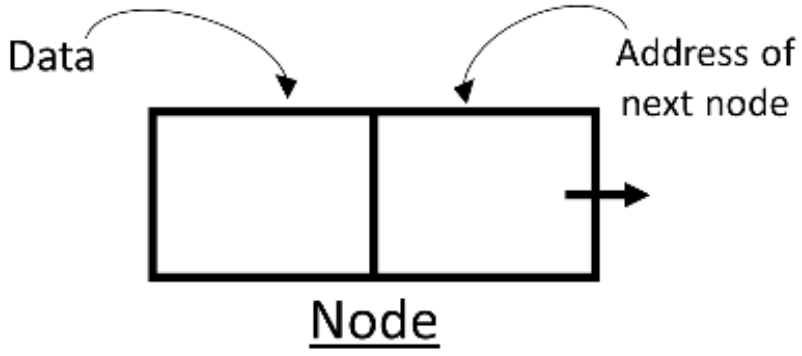
raiyan@cse.mist.ac.bd

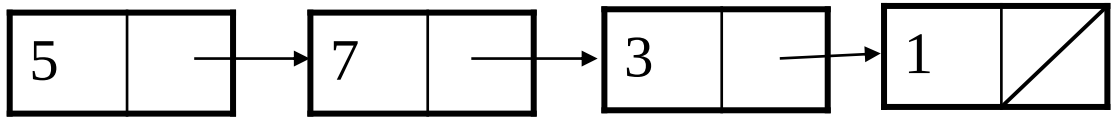
Courtesy: Prof. Dr. Abul Kashem Mia

Linked List

- A linked list is a linear collection of data elements (called nodes), where the linear order is given by means of pointers.
- Each node is divided into 2 parts:
  - 1st part contains the information of the element.
  - 2nd part is called the link field or next pointer field which contains the address of the next node in the list.

```
struct node {
    int value;
    struct node *next;
}
```



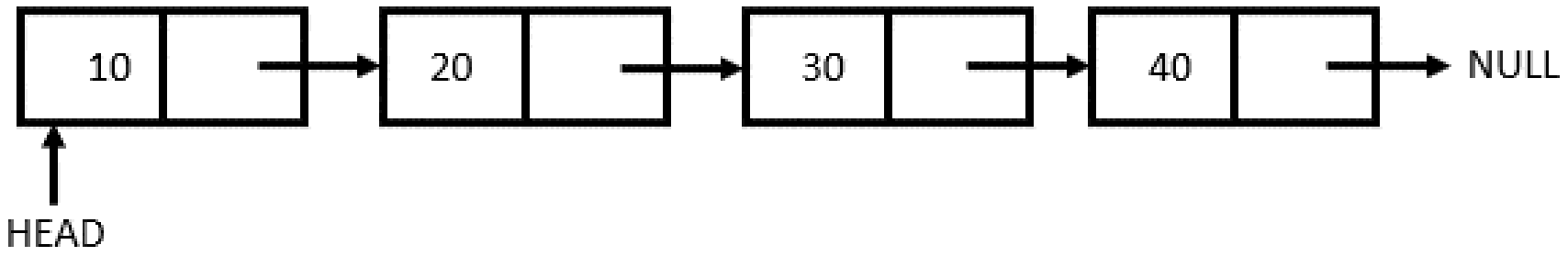


Basic Operations

- Traverse: Go through the list starting from first and print.
- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

Traversal of a Linear Linked list

- We start from the head. Point a node ‘temp’ to head.
- Need to continue and advance ‘temp’ to next node till node->next is not null.
- Print the values of each node as we go.



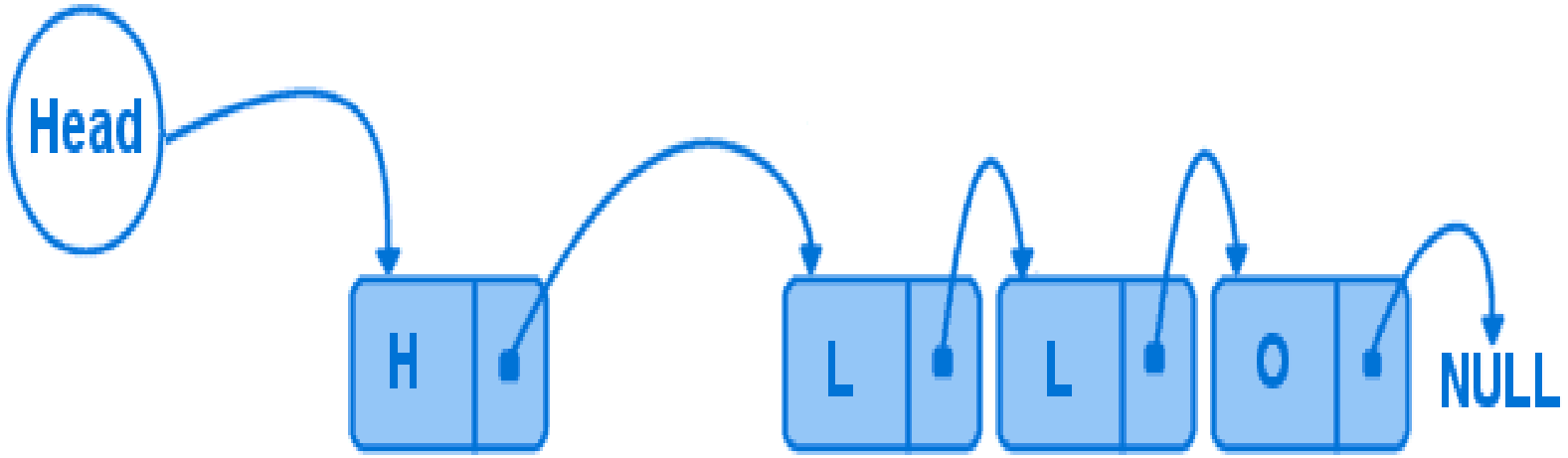
OUTPUT:

Printing or Traversing List

```
void printList()
{
  if (head = = NULL)   // no list at all
       return;
  struct node *temp =  head;
  while (temp != NULL)
  {
       printf("%d \t", temp->value);
     temp = temp->next;
  }
}
```

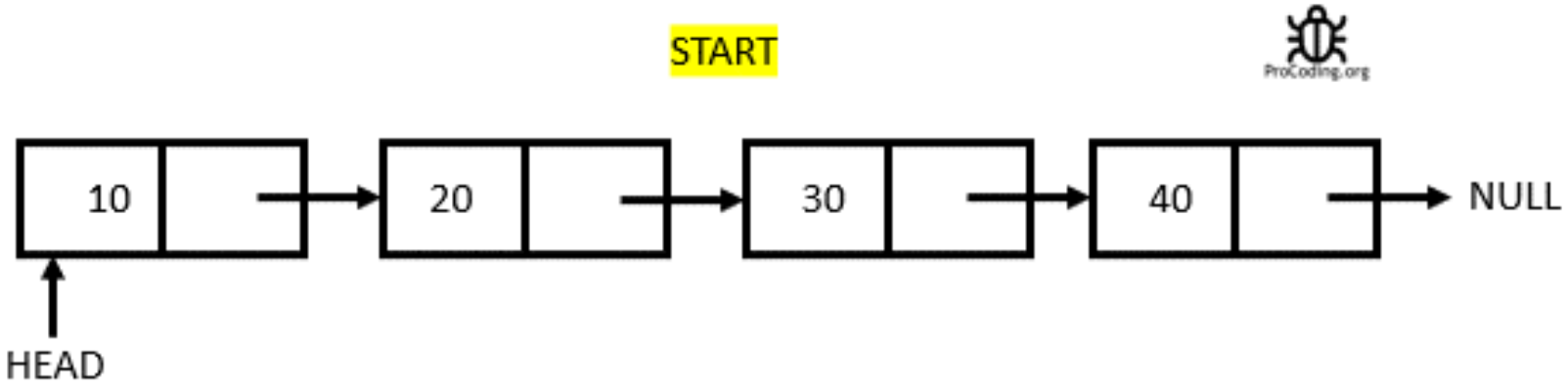
Insertion to a Linear Linked list

- Add a new node at the first, last or interior of a linked list.



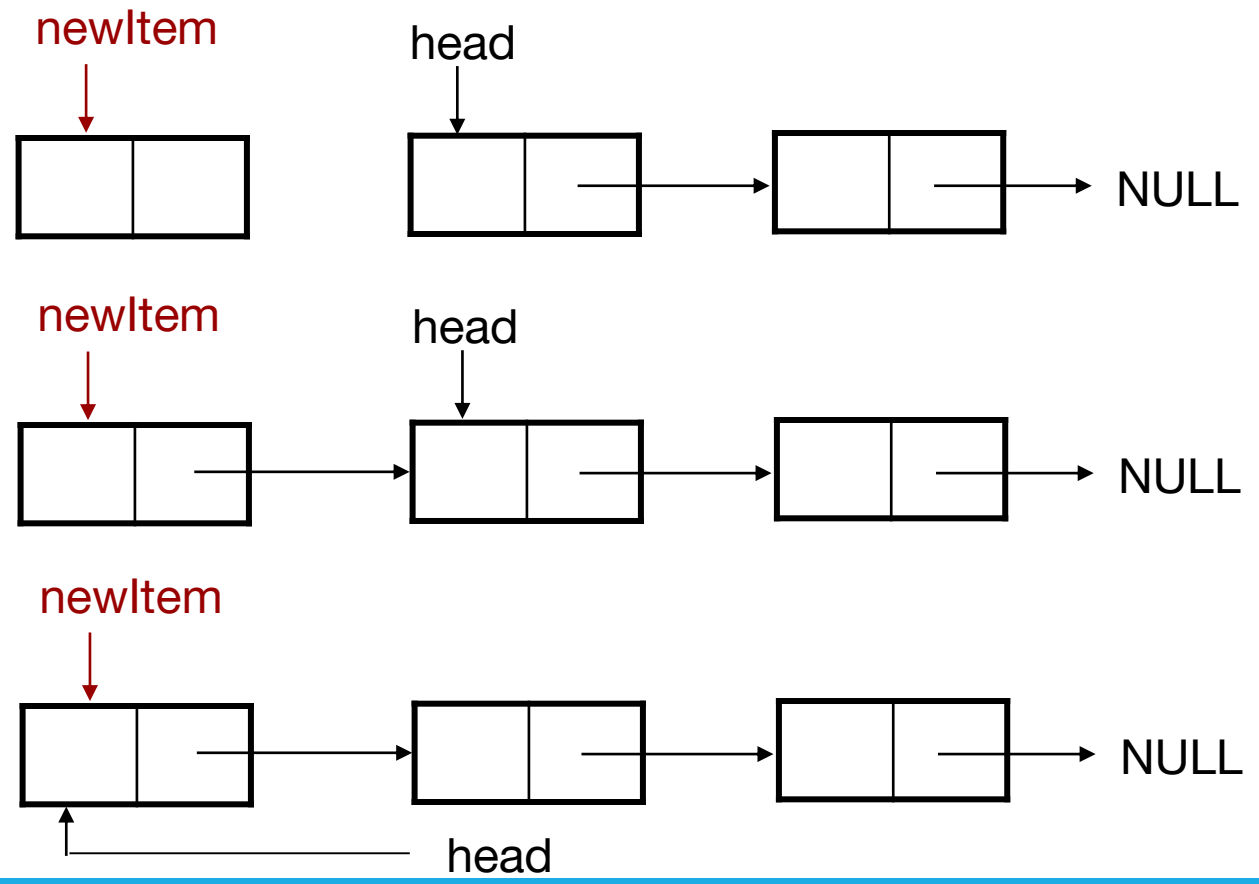
Insert First

- To add a new node to the head of the linear linked list, we need to construct a new node that is pointed by pointer *newitem*.
- Assume there is a global variable *head* which points to the first node in the list.
- The new node points to the first node in the list. The *head* is then set to point to the new node.



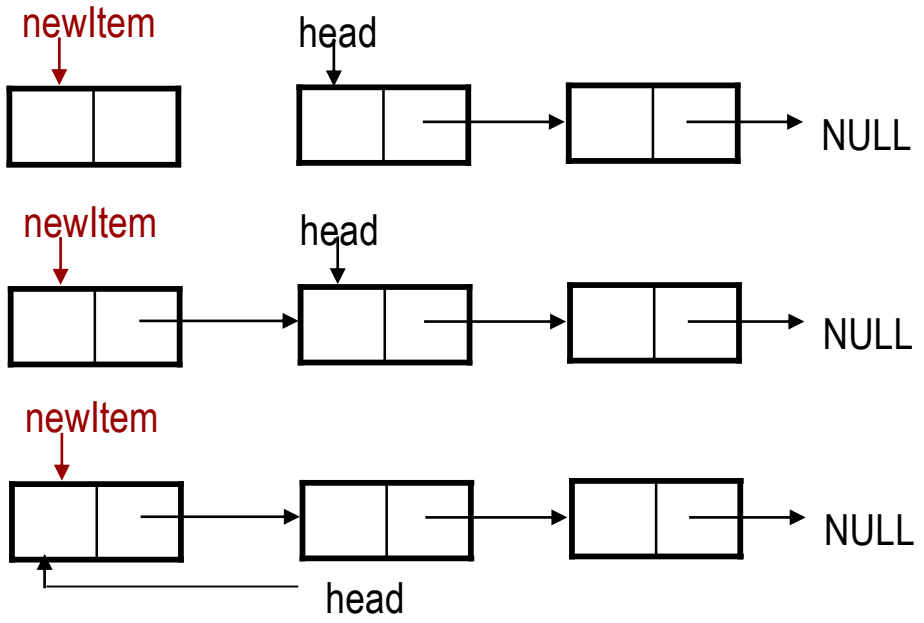
Insert First (Cont.)

- Step 1. Create a new node that is pointed by pointer *newItem*.
- Step 2. Link the new node to the first node of the linked list.
- Step 3. Set the pointer *head* to the new node.



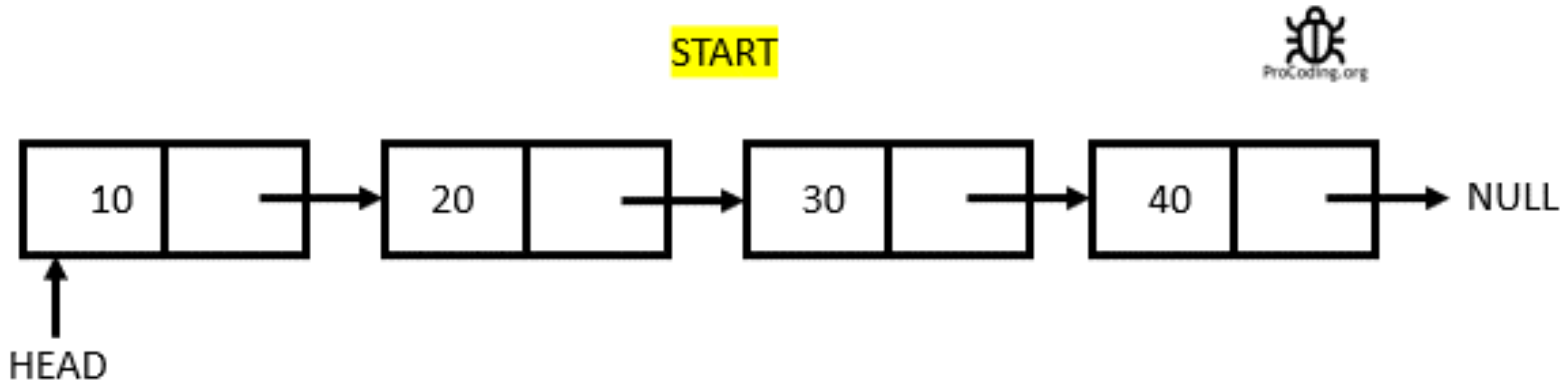
Insert First (Cont.)

```
struct node{
    int value;
    struct node *next;
};
struct node *head;

void insertHead(int num){
  //create a new node
  struct node *newItem;
  newItem=(struct node *)malloc(sizeof(struct node));
  newItem->value = num;
  newItem->next = NULL;
  //insert the new node at the head
  newItem->next = head;
  head = newItem;
}
```



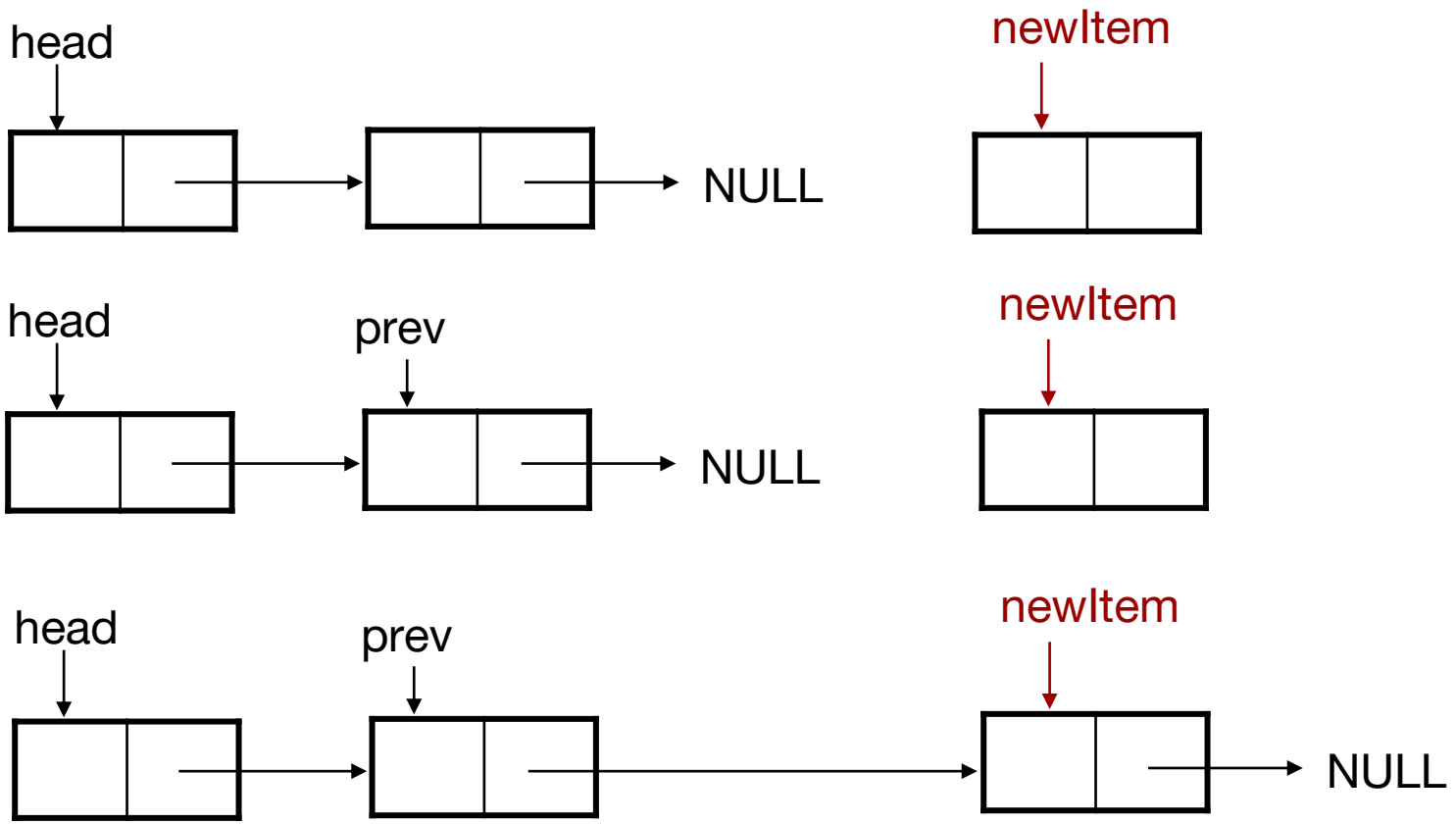
Insert Last

- To add a new node to the tail of the linear linked list, we need to construct a new node and set it's link field to “NULL".
- Assume the list is not empty, locate the last node and change it's link field to point to the new node.



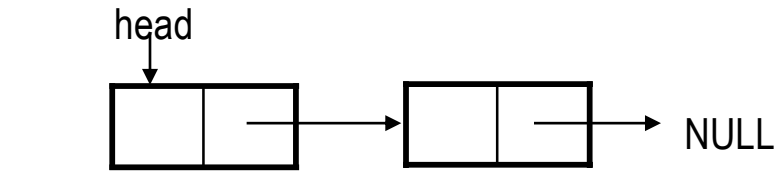
Insert Last (Cont.)

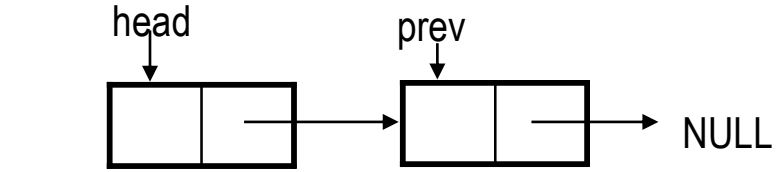
- Step1. Create the new node.
- Step2. Set a temporary pointer prev to point to the last node.
- Step3. Set prev to point to the new node and new node as last node.

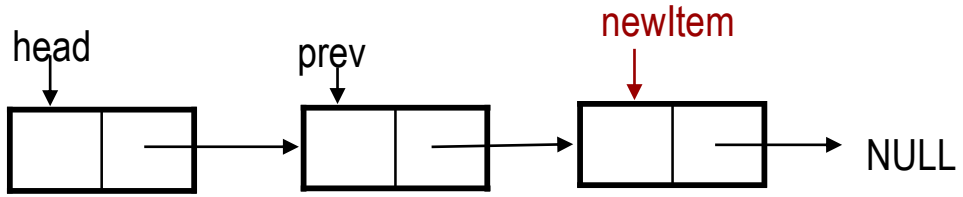


Insert Last (Cont.)

```
struct node{
     int value;
     struct node *next;
};
struct node *head;

void insertTail(int num){
   //create a new node to be inserted
   struct node *newItem;
   newItem=(struct node *)malloc(sizeof(struct node));
   newItem->value = num;
   newItem->next = NULL;
   // set prev to point to the last node of the list
   struct node *prev = head;
   while (prev->next != NULL)
       prev = prev->next;
   prev->next = newItem;
}
```

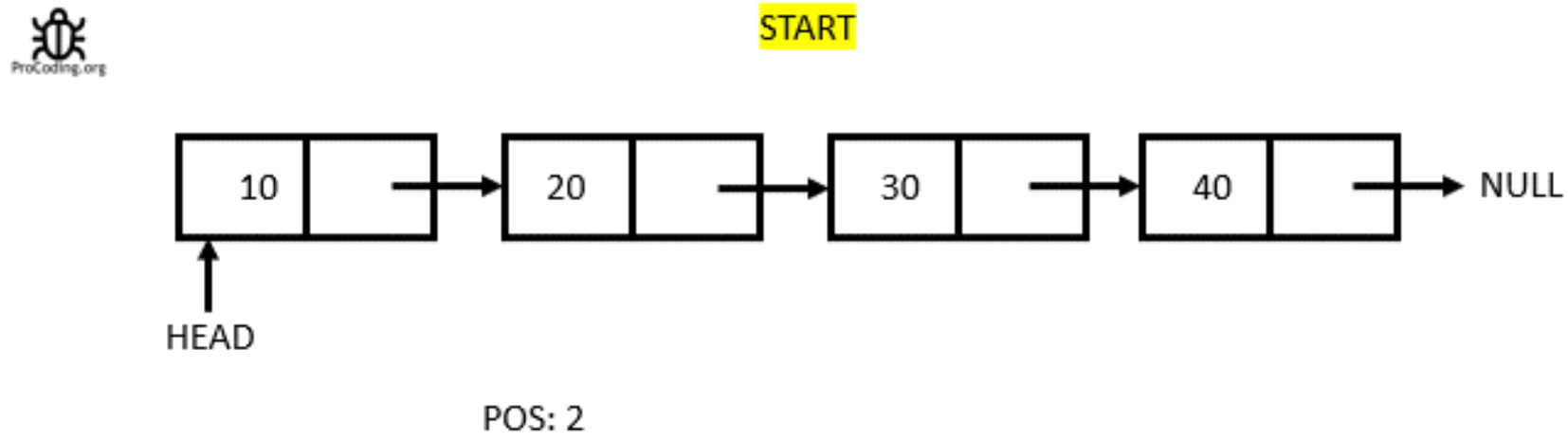






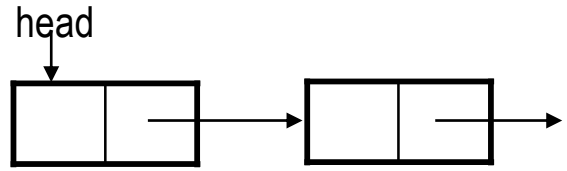
Insertion After a Target Value or to a Specific Index Position

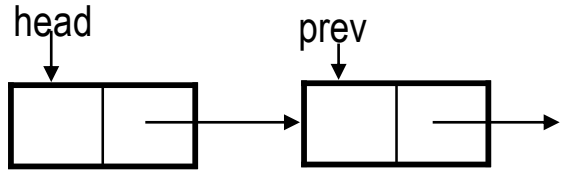
- To insert to a desired location/position or after a target value, we need to traverse to that specific position first.
- Then, need to link the new node (node to be inserted) in such a way that the chain remains unbroken. That is, link the new node to the previous and next one.

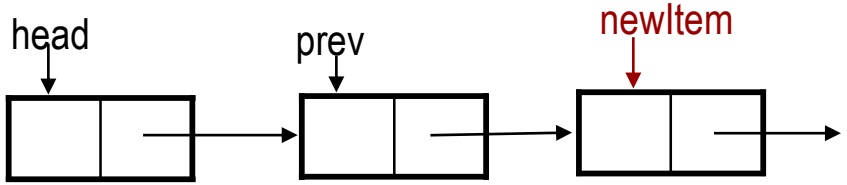


Insert Middle (after a desired node value)

```
struct node{
     int value;
     struct node *next;
};
struct node *head;

void insertMiddle(int num, int val){
   //create a new node to be inserted
   struct node *newItem;
   newItem=(struct node *)malloc(sizeof(struct node));
   newItem->value = num;
   newItem->next = NULL;
   // set prev to point to the desired node of the list
   struct node * prev = head;
   while (prev->value != val){
       prev = prev->next;
   }
   newItem->next = prev->next;
   prev->next = newItem;
}
```

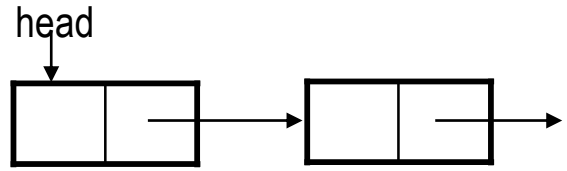


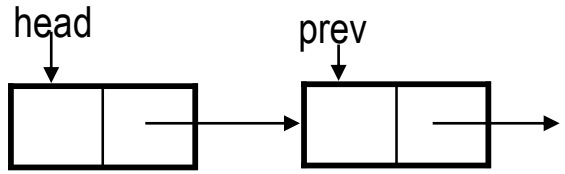


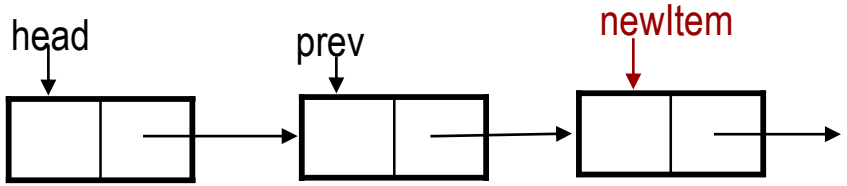


Insert Middle (to a specific index position)

```
struct node{
     int value;
     struct node *next;
};
struct node *head;

void insertatPosition(int num, int pos){
   //create a new node to be inserted
   struct node *newItem;
   newItem=(struct node *)malloc(sizeof(struct node));
   newItem->value = num;
   newItem->next = NULL;
   // set prev to point to the desired node of the list
   struct node *prev = head;
   for (int i=0;i<pos-1;i++)
       prev = prev->next;

   newItem->next = prev->next;
   prev->next = newItem;
}
```

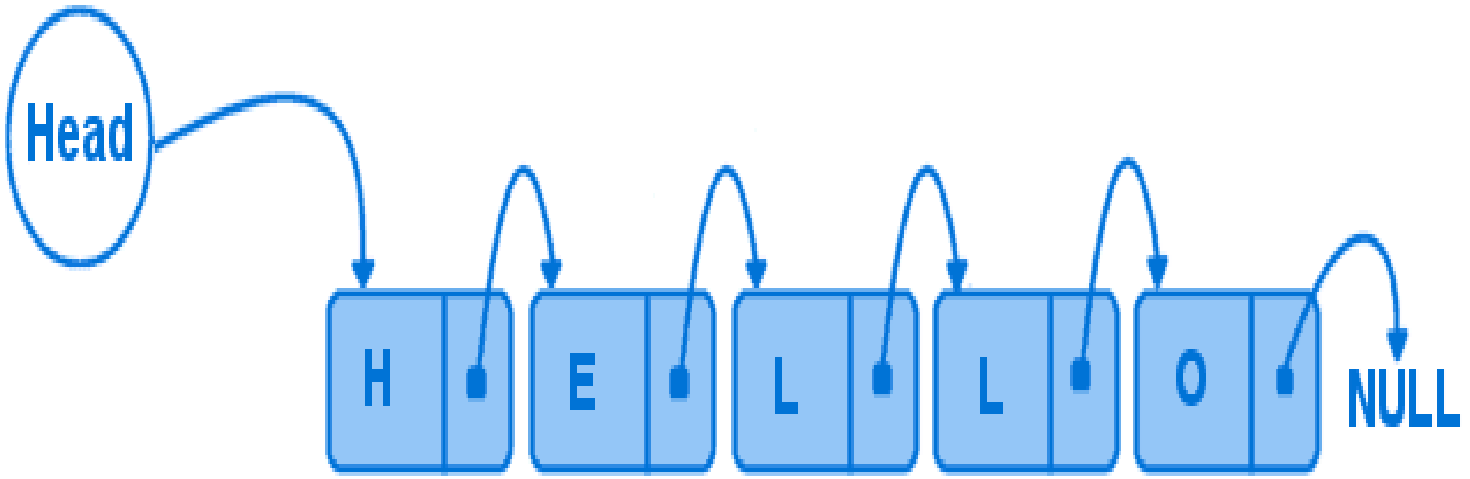






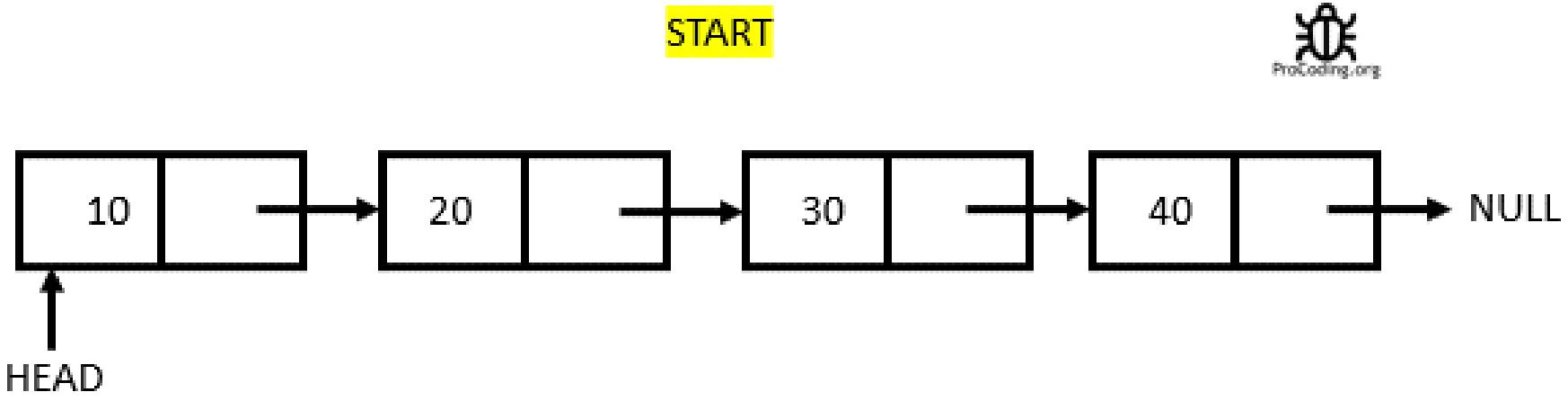
Deletion from a Linear Linked list

- Deletion can be done
  - At the first node of a linked list.
  - At the end of a linked list.
  - Within the linked list.



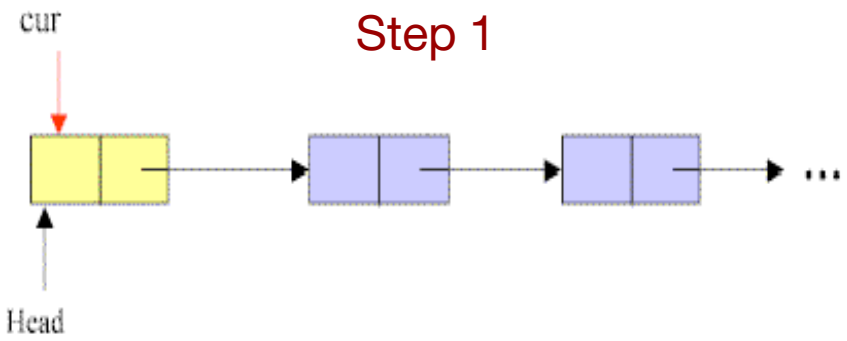
Delete First

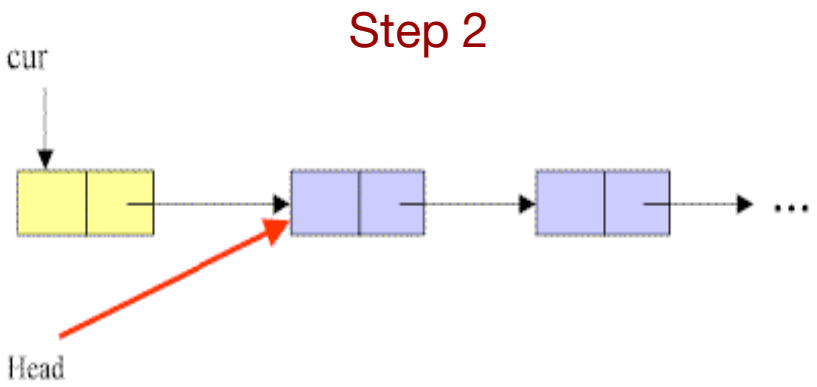
- To delete the first node of the linked list, we not only want to advance the pointer *head* to the second node, but we also want to release the memory occupied by the abandoned node.

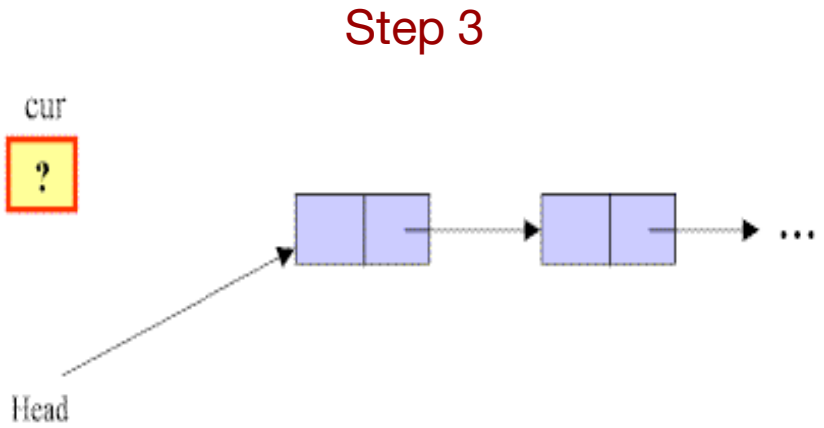


Delete First (Cont.)

- Step1. Initialize the pointer *cur* point to the first node of the list.
- Step2. Move the pointer *head* to the second node of the list.
- Step3. Remove the node that is pointed by the pointer *cur*.





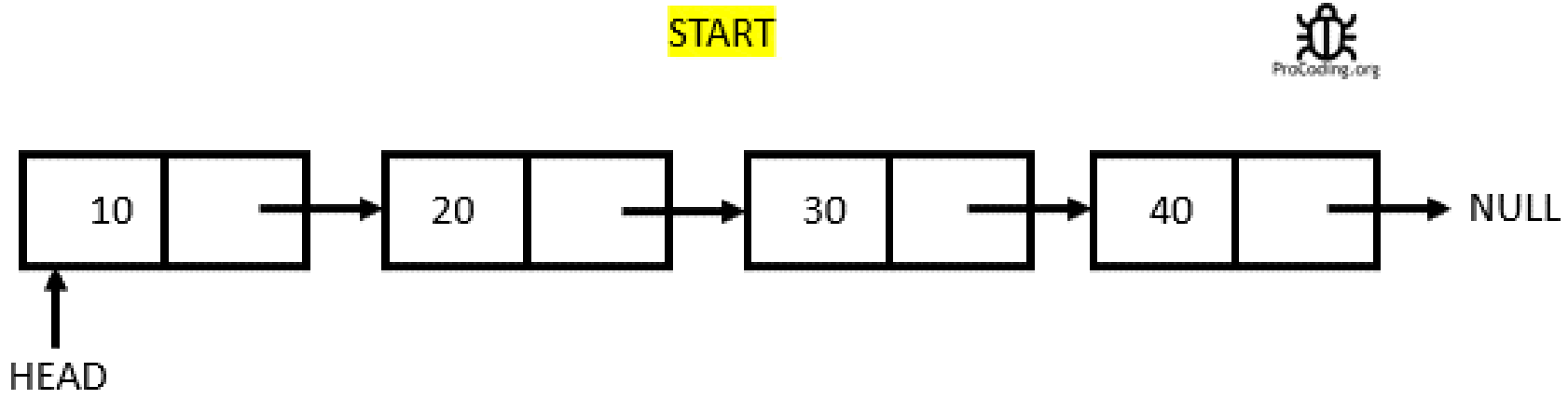


Delete First (Cont.)

```
void deleteHead()
{
   struct node *cur;
   if (head = = NULL)  //list empty
        return;
   cur = head;  // save head pointer
   head = head->next; //advance head
   free(cur);
}
```

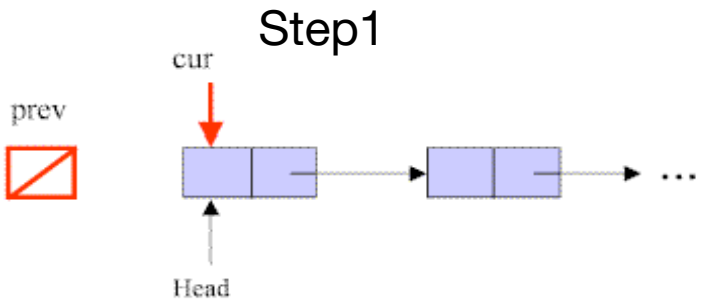
Delete Last

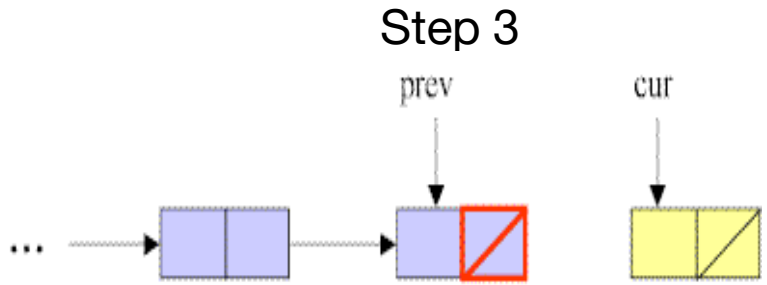
- To **delete** the last node in a linked list, we use a local variable, *cur*, to point to the last node. We also use another variable, *prev*, to point to the second last node in the linked list.

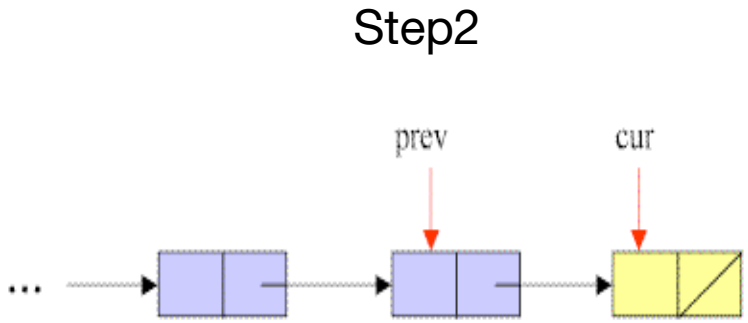


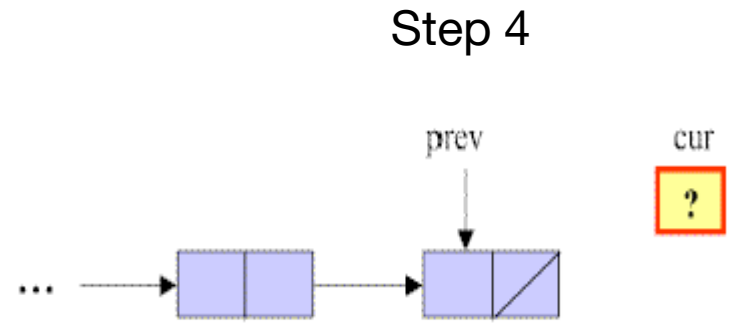
Delete Last (Cont.)

- Step1. Initialize pointer *cur* to point to the first node of the list, while the pointer *prev* has a value of NULL.
- Step2. Traverse the entire list until the pointer *cur* points to the last node of the list.
- Step3. Set NULL to *next* field of the node pointed by the pointer *prev*.
- Step4. Remove the last node that is pointed by the pointer *cur*.



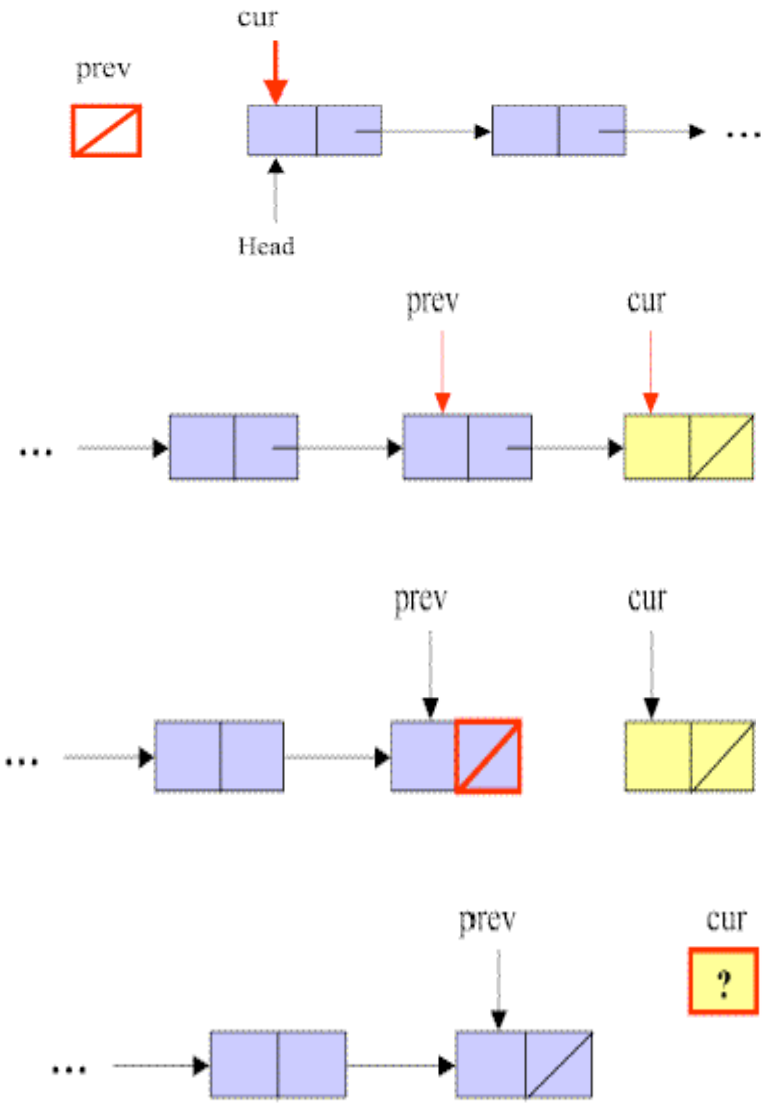






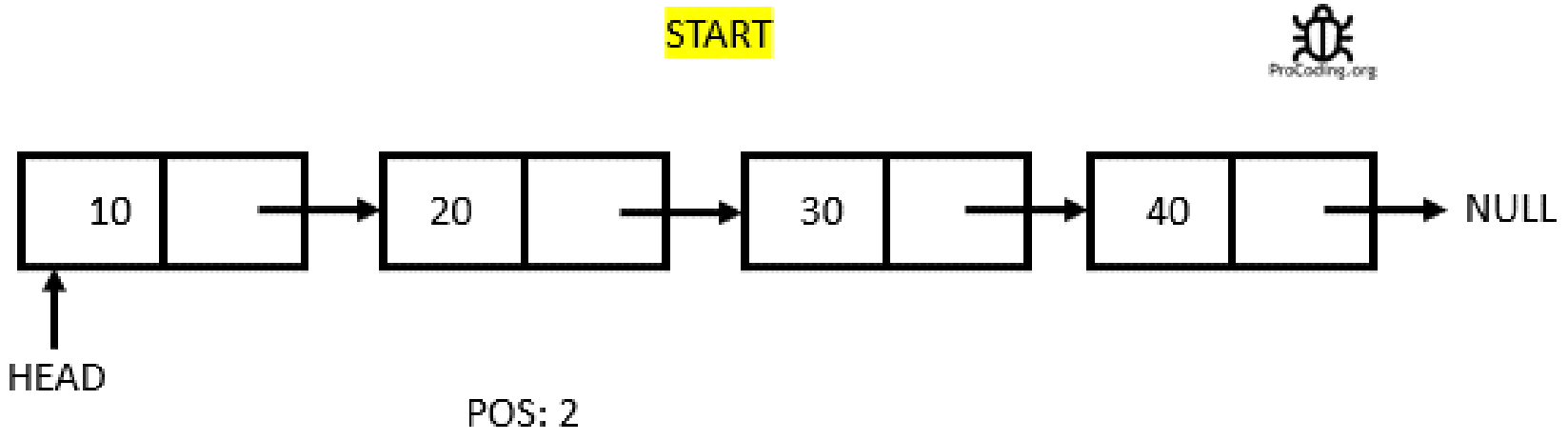
Delete Last (Cont.)

```
void deleteTail(){
   if (head = = NULL)       //list empty
        return;
   struct node *cur = head;
   struct node *prev = NULL;
   while (cur->next != NULL){
        prev = cur;
        cur=cur->next;
   }
   if (prev != NULL)
        prev->next = NULL;
   free(cur);
}
```



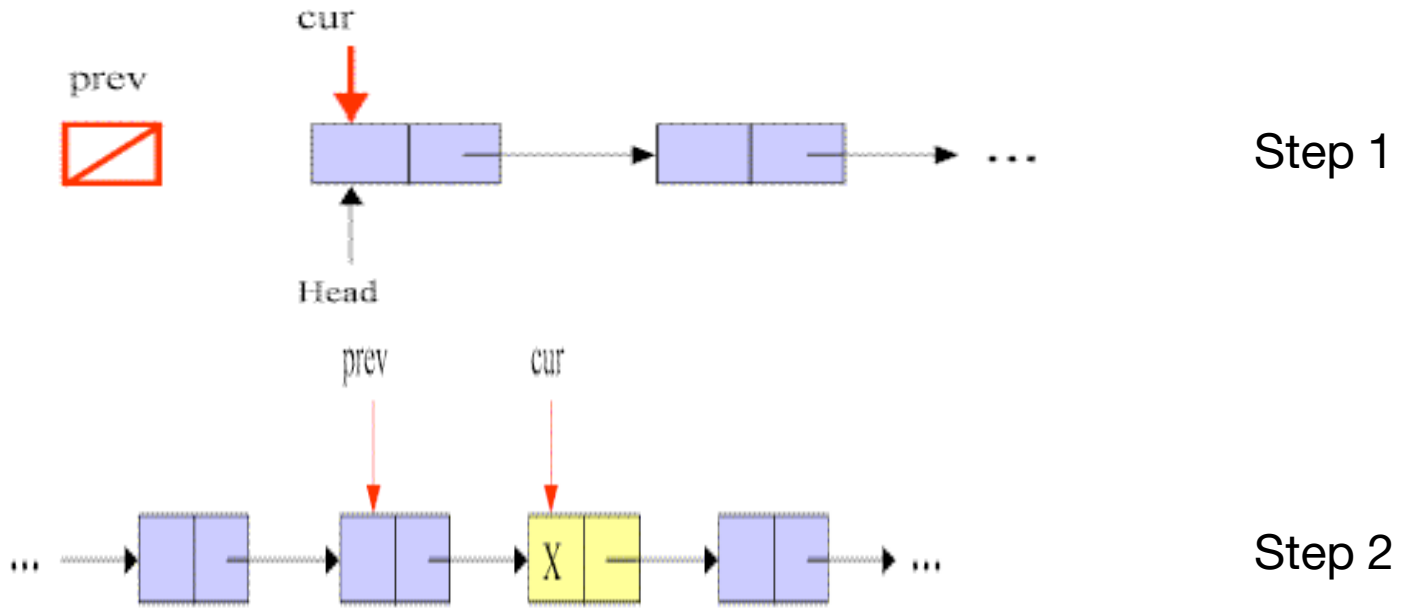
Delete from Matching Target Value

- To **delete** a node that contains a particular value *x* in a linked list or is in a particular position, we use a local variable, *cur*, to point to this node, and another variable, *prev*, to hold the previous node.



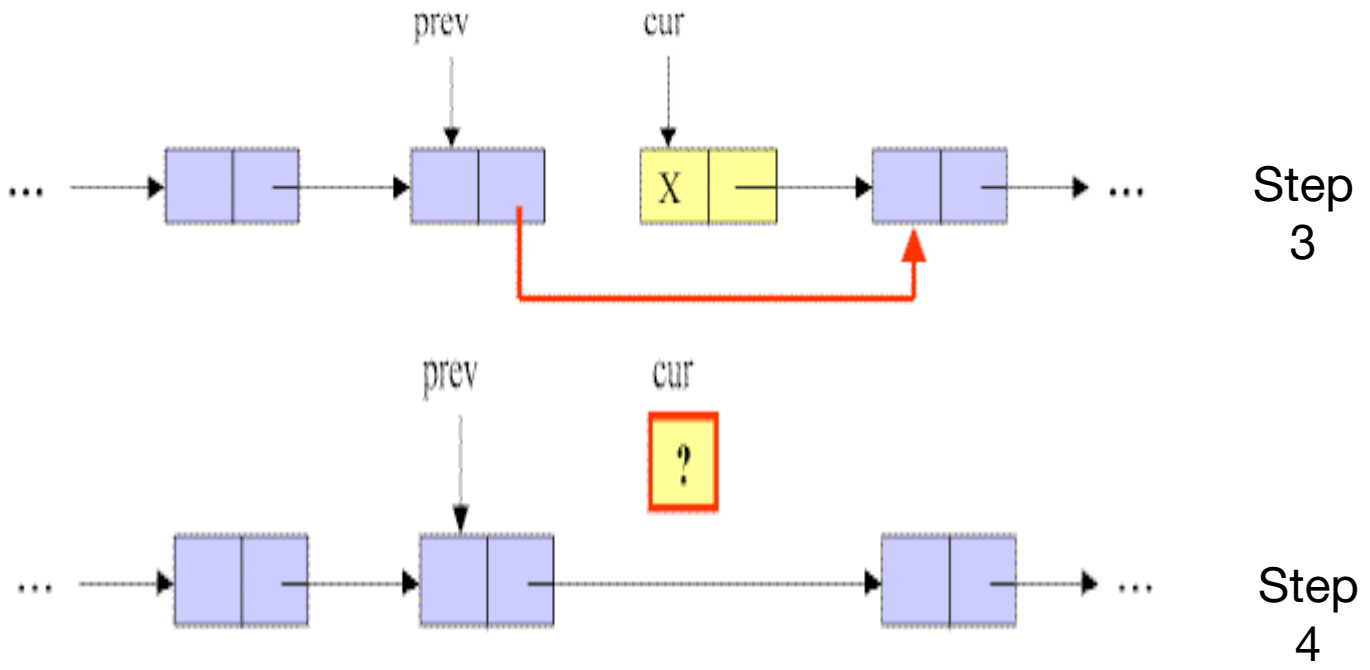
Delete Any (Cont.)

- Step1. Initialize pointer *cur* to point to the first node of the list, while the pointer *prev* has a value of null.
- Step2. Traverse the entire list until the pointer *cur* points to the node that contains value of *x*, and *prev* points to the previous node.
- ……..



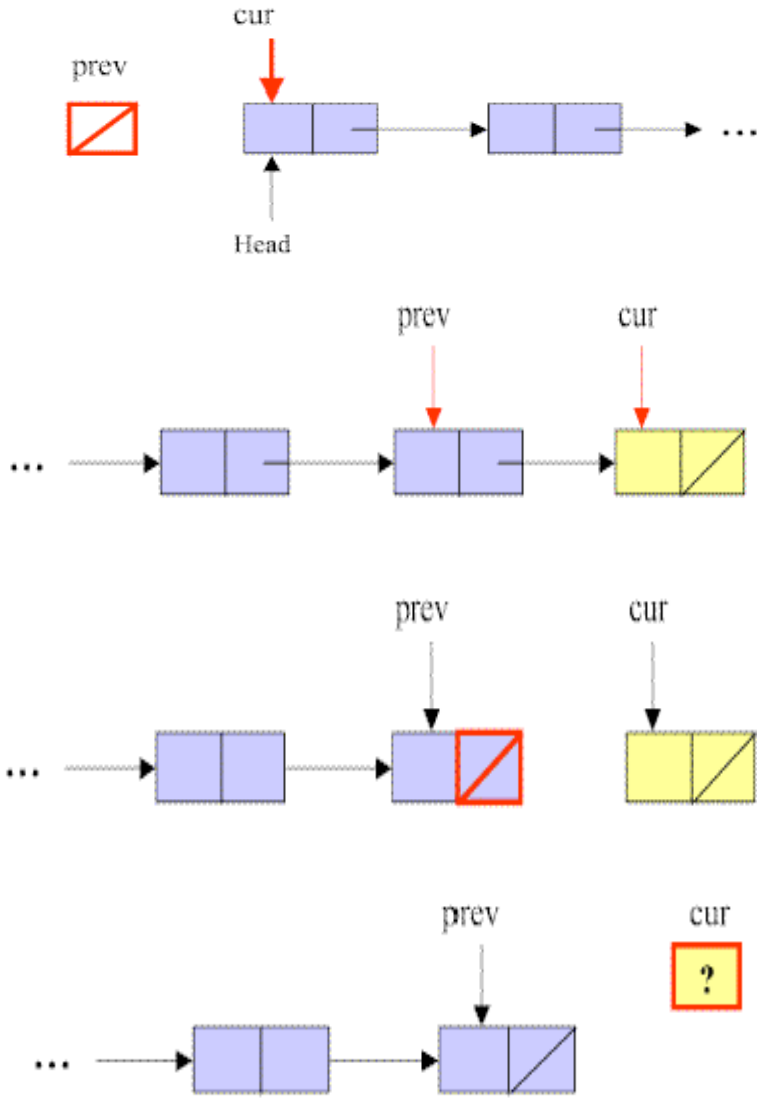
Delete Any (Cont.)

- …….
- Step3. Link the node pointed by pointer *prev* to the node after the *cur*'s node.
- Step4. Remove the node pointed by *cur*.



Delete Any (Cont.)

```
void deleteAny( int x ){
   if (head = = NULL)       //list empty
        return;
   struct node *cur = head;
   struct node *prev = NULL;
   while (cur->value != x){
        prev = cur;
        cur=cur->next;
   }
 if (prev != NULL)
        prev->next = cur->next;
 free(cur);
}
```



Check **SLL.c** for implementation of Single Linked List. Add more functions on your own for better understanding and practice!

An **evaluation** will be held on SLL (insertion or equivalent only) on Week 4 sessional (11 April 2023)

Single Link Lists Vs Arrays

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Single Link Lists & Arrays – A Tough Choice

	Array	Linked List
Strength	• Random Access (Fast Search Time) • Less memory needed per element • Better cache locality	• Fast Insertion/Deletion Time • Dynamic Size • Efficient memory allocation/utilization
Weakness	• Slow Insertion/Deletion Time • Fixed Size • Inefficient memory allocation/utilization	• Slow Search Time • More memory needed per node as additional storage required for pointers

Single Link Lists Vs Arrays

- **Time Complexity:** The biggest improvement in linked list over arrays is in insertion and deletion time. Both operations themselves can be done here in **O(1)**, although keep in mind that, you may need to look up (search) a value before inserting or deleting. Time complexity of search and traversal is **O(n)**.

- **Searching:** Then, can we speed up the searching process? Unfortunately, as for searching, it's also essential to state that there is no method of optimizing search in linked lists. In the array, at least we could keep the array sorted. Also, as we don't have any idea how long the linked list is, there is no way of accomplishing a binary search.

Single Link Lists Vs Arrays

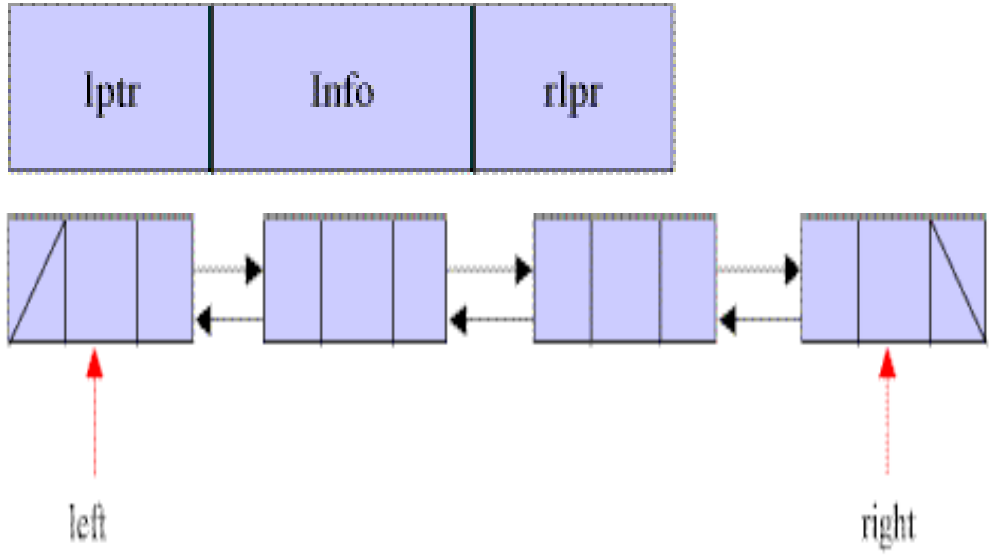
- **Space Complexity:** Linked lists carry two major chunks of data, namely, the value and the pointer per node. This indicates that the amount of data stowed grows linearly with the number of nodes in the list. Thus, the space complexity of the linked list is linear, i.e. **O(n).**

- **Application:** Linked lists are used in a variety of cases, most prominently being the building block for other complex data structures. They may be used to implement stacks, queues, graphs and hash tables. They also have particular use cases in applications, such as:

  - Next and Previous page in a web browser – We can access the previous and next "url" searched in a web browser by pressing the "back" and "next" buttons as they are connected as a linked list.
  - Music Player – Songs in the music player are connected to the previous and next song. You can listen to the songs either from the beginning or end of the list.

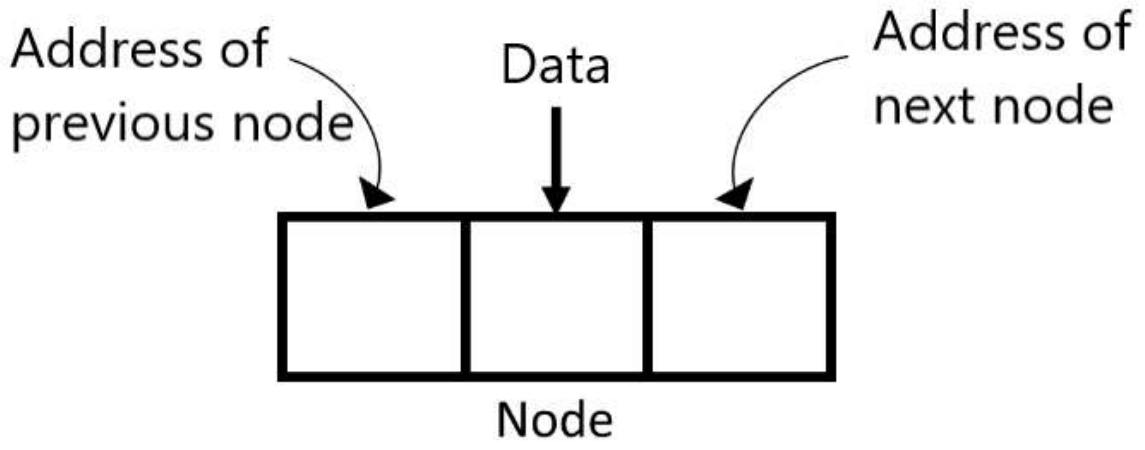
Introduction to Doubly Linked List

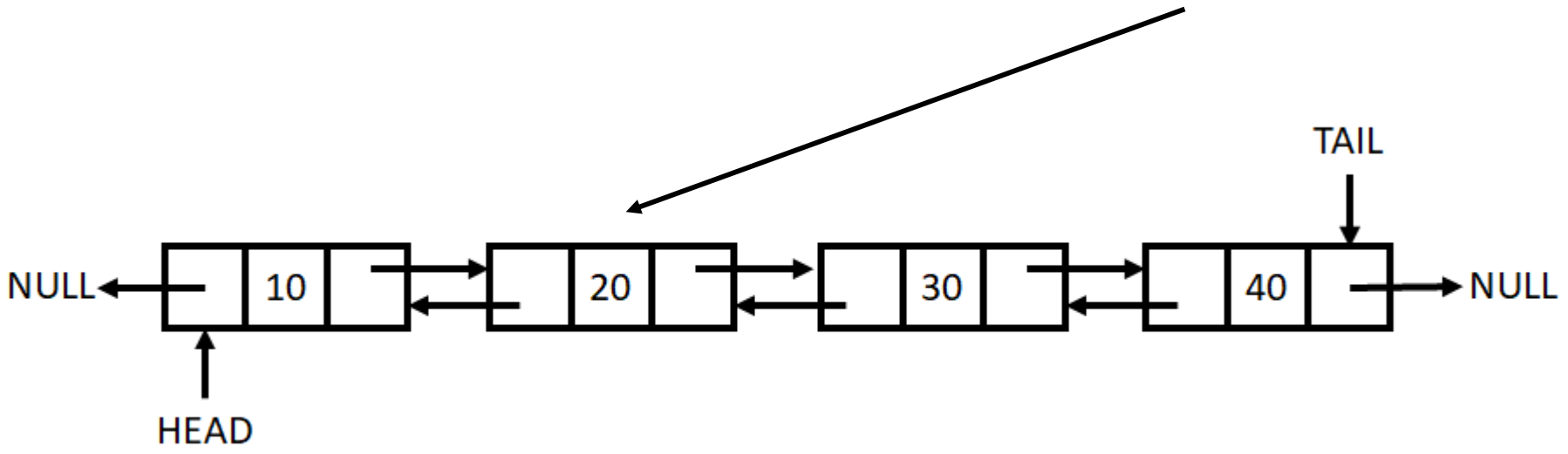
- We have discussed the details of linear linked list. In the linear linked list, we can only traverse the linked list in one direction.
- But sometimes, it is very desirable to traverse a linked list in either a forward or reverse manner.
- This property of a linked list implies that each node must contain two link fields instead of one. The links are used to denote the predecessor and successor of a node. The link denoting the predecessor of a node is called the left link, and that denoting its successor its right link



A Node in a Doubly Linked List

```
struct/class node {
    int value;
    struct node *next; //or lptr
    strcut node *prev; //or rptr
}
```





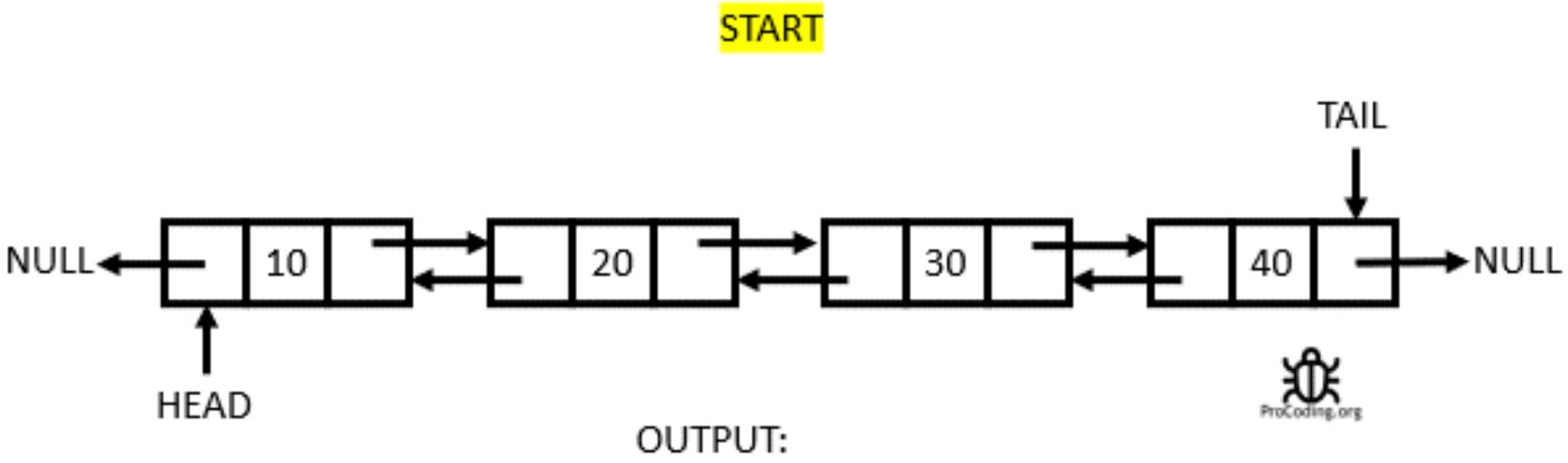
Basic Operation of Doubly Linked List

- Traverse: Go through the list starting from first and print.
- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

Traversal of a Doubly Linked list

- Here, we can traverse from both direction – forward or reverse!
- For forward traversal, we start from the head and move the temp node forward (using next). For reverse traversal, start from tail and move temp node backward (using prev).
- Print the values of each node as we go.

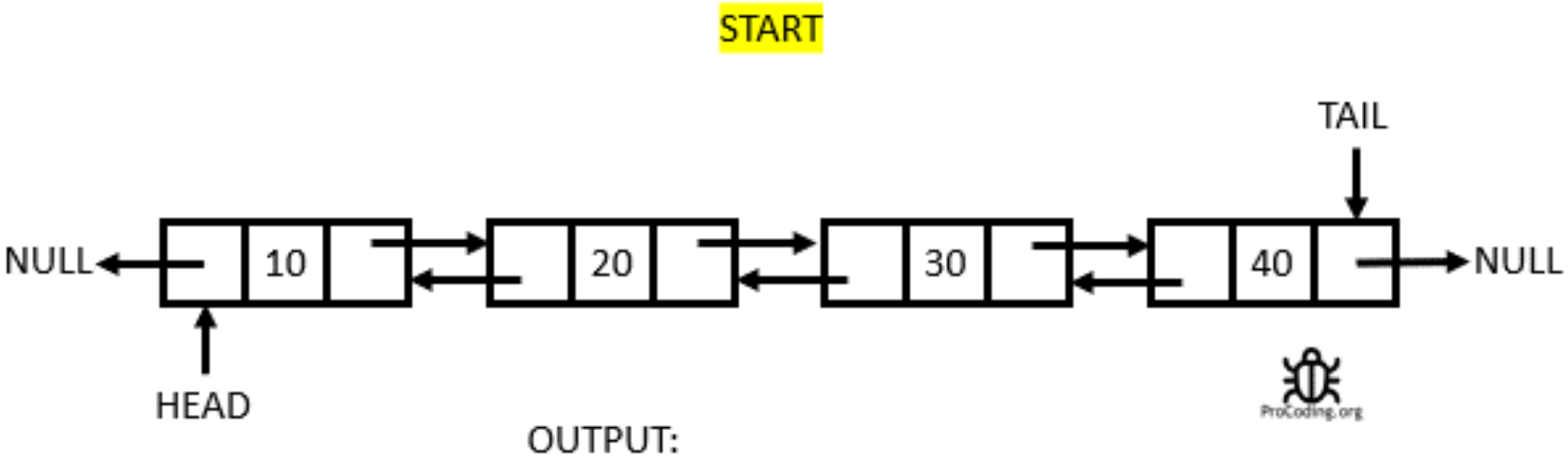
Forward Traversal



Traversal of a Doubly Linked list

- Here, we can traverse from both direction – forward or reverse!
- For forward traversal, we start from the head and move the temp node forward (using next). For reverse traversal, start from tail and move temp node backward (using prev).
- Print the values of each node as we go.

Reverse Traversal

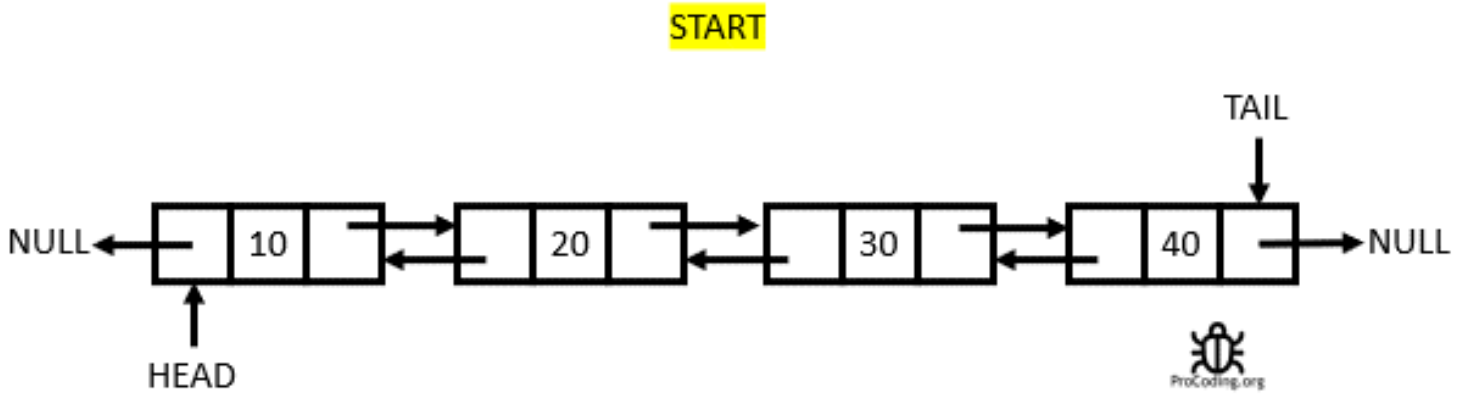


Insert First or Insert Last into a Doubly Linked List

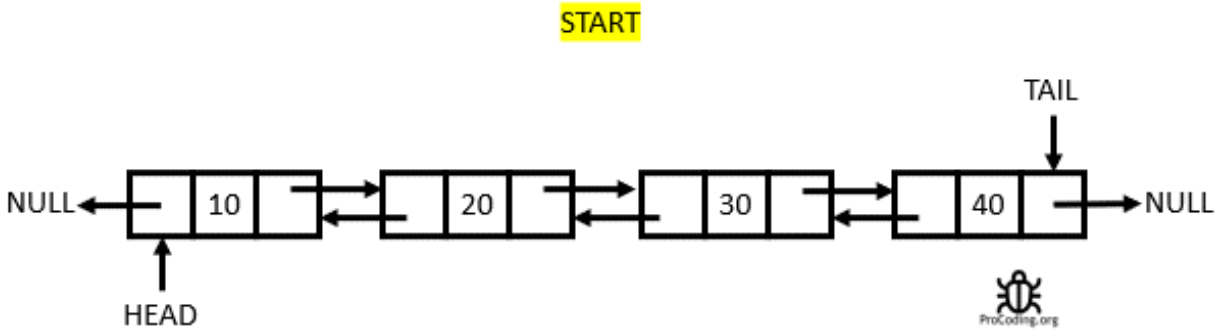
- **Insertion** is to add a new node into a linked list. It can take place anywhere -- the first, last, or interior of the linked list.
- To add a new node to the head and tail of a double linked list is similar to the linear linked list.
- First, we need to construct a new node that is pointed by pointer *newItem*.
- Then the newItem is linked to the left-most node (or right-most node) in the list. Finally, the *Left* (or *Right*) is set to point to the new node.

Insert First or Insert Last into a Doubly Linked List

Insert First



Insert Last



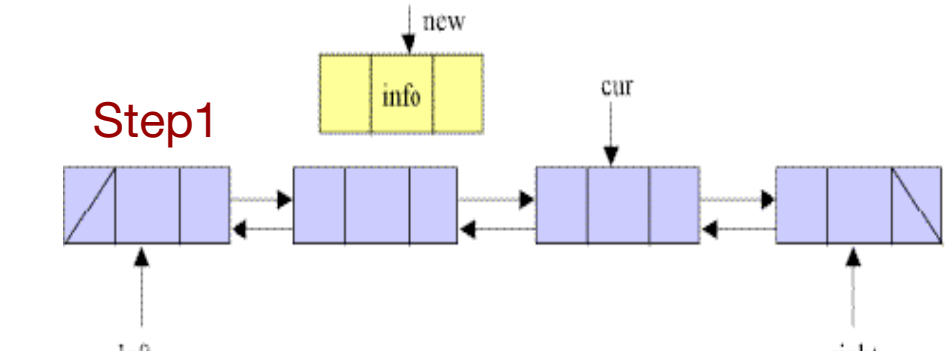
Insert to The Interior of Doubly Linked List

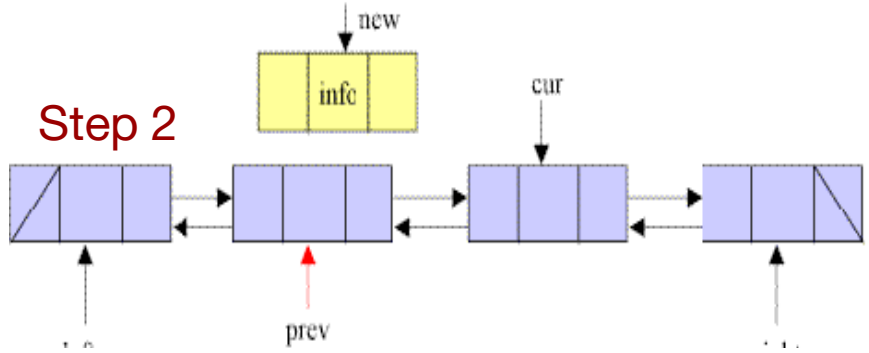
Step1. Create a new node that is pointed by *newItem. Create a node cur and traverse the list to find the target node. Point cur to it.*

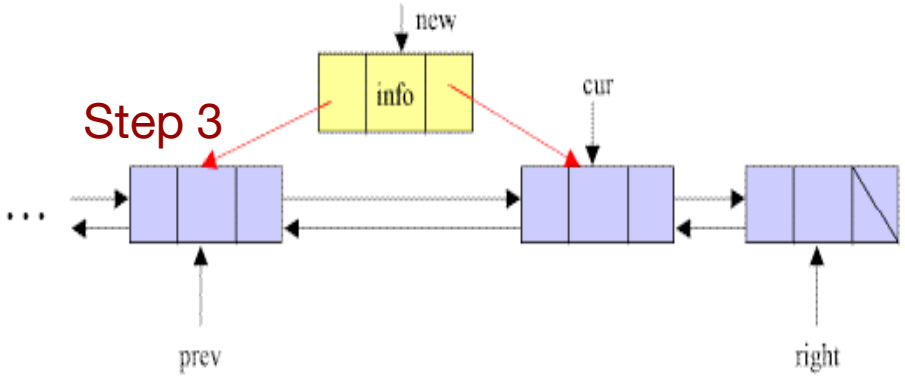
Step2. Set the pointer *prev* to point to the left node of the node pointed by *cur*.

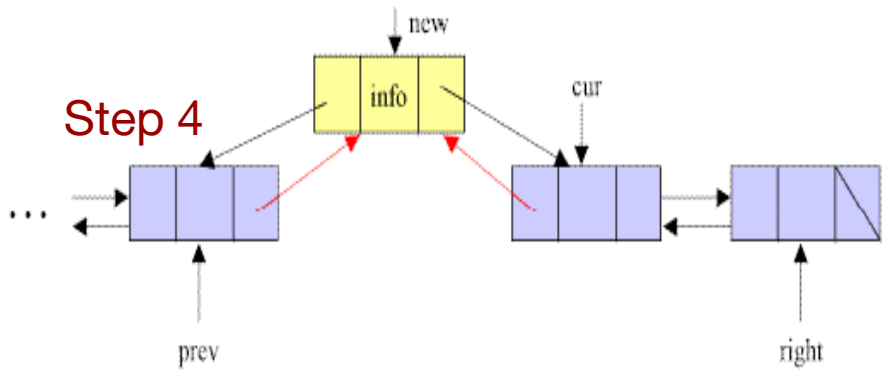
Step3. Set the left link of the new node to point to the node pointed by *prev*, and the right link of the new node to point to the node pointed by *cur*.

Step4. Set the right link of the node pointed by *prev* and the left link of the node pointed by *cur* to point to the new node.

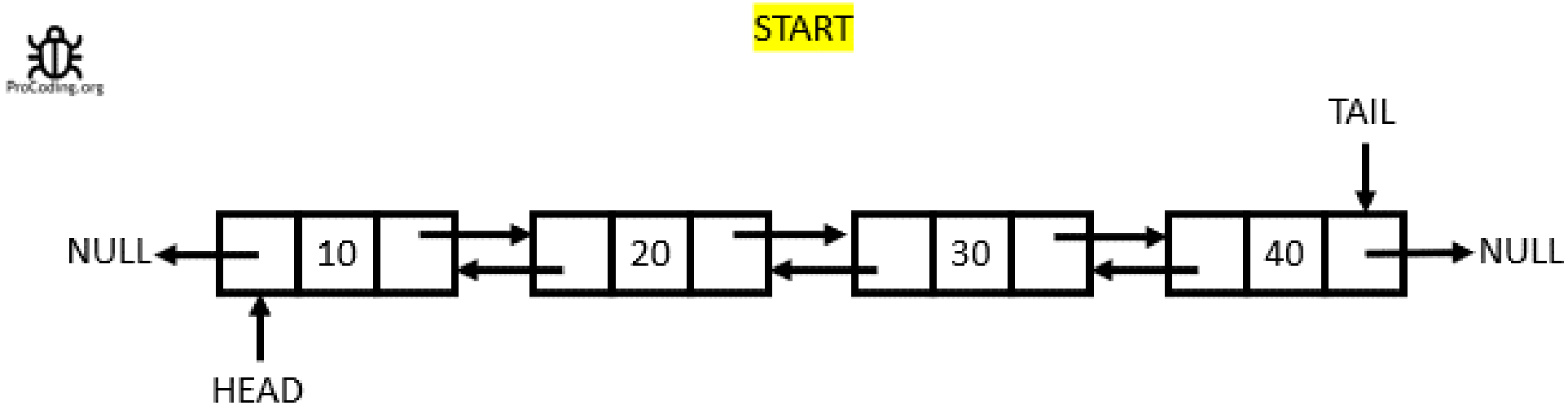








Insert to The Interior of Doubly Linked List



Insert (First) into a Doubly Linked List

```
struct node{
     struct node *prev;
     int value;
     struct node *next;
}*head, *last;

void insert_begning(int data){
    struct node *newItem,*temp;
     newItem=(struct node *)malloc(sizeof(struct node));
     newItem->value=data;
     if(head==NULL){ // when this is the first node
         head=newItem;                head->prev=NULL;
         head->next=NULL;             last=head;
     }
     else{ //when there are existing nodes
         temp=newItem;
         temp->prev=NULL;             temp->next=head;
         head->prev=temp;             head=temp;
     }
}
```

Insert (Last) into a Doubly Linked List

```
void insert_end(int data){
    struct node *newItem,*temp;
    newItem=(struct node *)malloc(sizeof(struct node));
    newItem->value=data;
    if(head==NULL){
        head=newItem;       head->prev=NULL;
        head->next=NULL;    last=head;
    }
    else{
        last=head;
        while(last != NULL){
            temp=last;
            last=last->next;
        }
    last=newItem;           temp->next=last;
    last->prev=temp;        last->next=NULL;
    }
}
```

Insert (Middle) into a Doubly Linked List

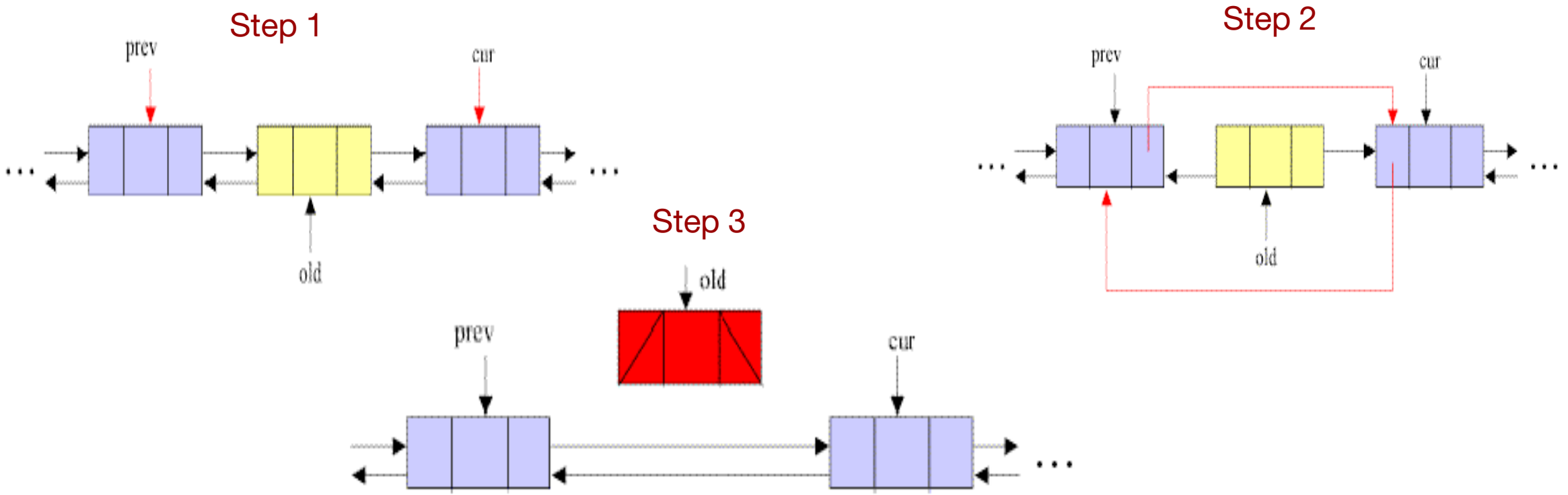
```
int insert_after(int data, int loc){
    struct node *temp,*newItem,*temp1;
    newItem=(struct node *)malloc(sizeof(struct node));
    newItem->value=data;
       if(head==NULL){
         head=newItem;    head->prev=NULL;     head->next=NULL;  }
    else{
         temp=head;
         while(temp!=NULL || temp->value!=loc)
              temp=temp->next;
         if (temp==NULL)
             printf("\n%d is not present in list ",loc);
         else{
         temp1=temp->next;   temp->next=newItem;   newItem->prev=temp;
        newItem->next=temp1;      temp1->prev=newItem;    }
    }
    last=head;
    while(last->next!=NULL)
         last=last->next;
}
```

Deletion of Doubly Linked List

- **Deletion** is to remove a node from a list. It can also take place anywhere -- the first, last, or interior of a linked list.
- To delete a node from a double linked list is easier than to delete a node from a linear linked list.
- For deletion of a node in a single linked list, we have to search and find the predecessor of the discarded node. But in the double linked list, no such search is required.
- Given the address of the node that is to be deleted, the predecessor and successor nodes are immediately known.

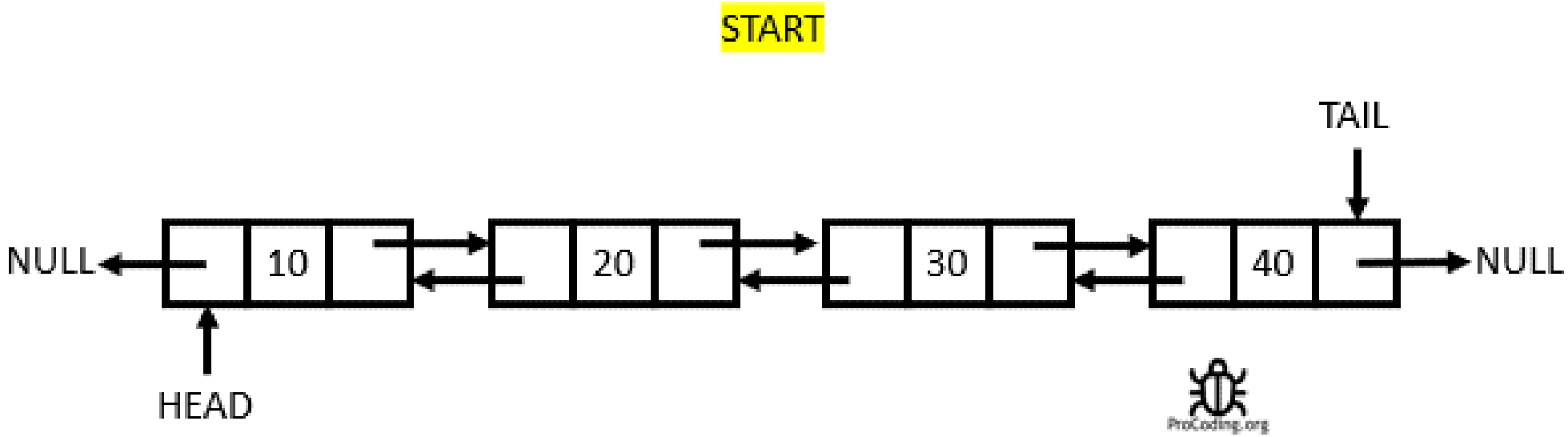
Deletion of Doubly Linked List (Cont.)

- Step1. Set pointer *prev* to point to the left node of *old* and pointer *cur* to point to the node on the right of *old*.
- Step2. Set the right link of *prev* to *cur*, and the left link of *cur* to *prev*.
- Step3. Discard the node pointed by *old*.



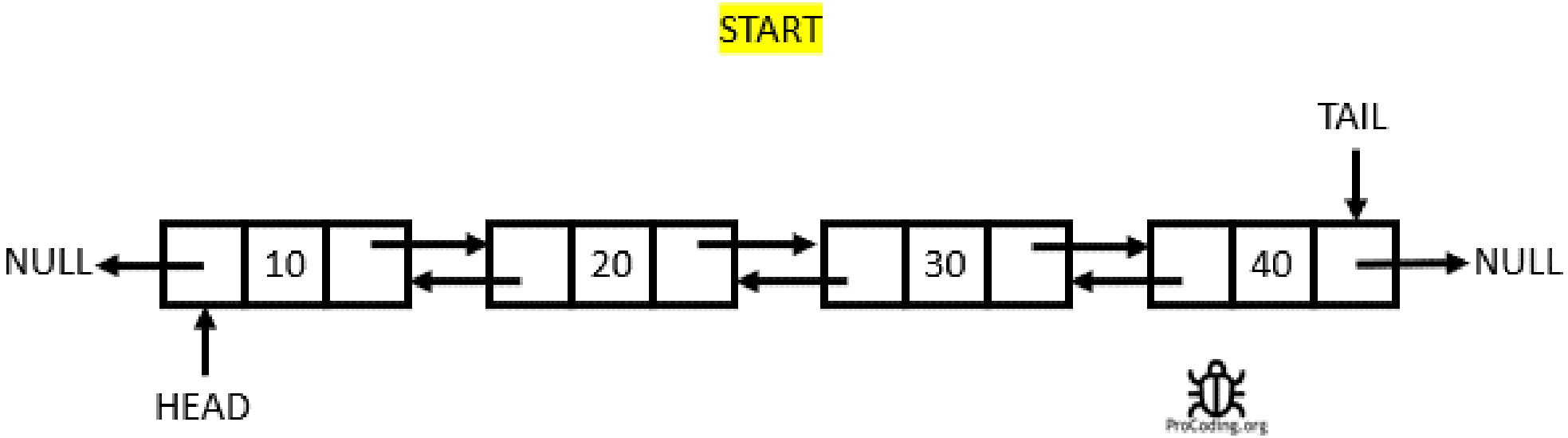
Deletion from a Doubly Linked List

Delete from Start



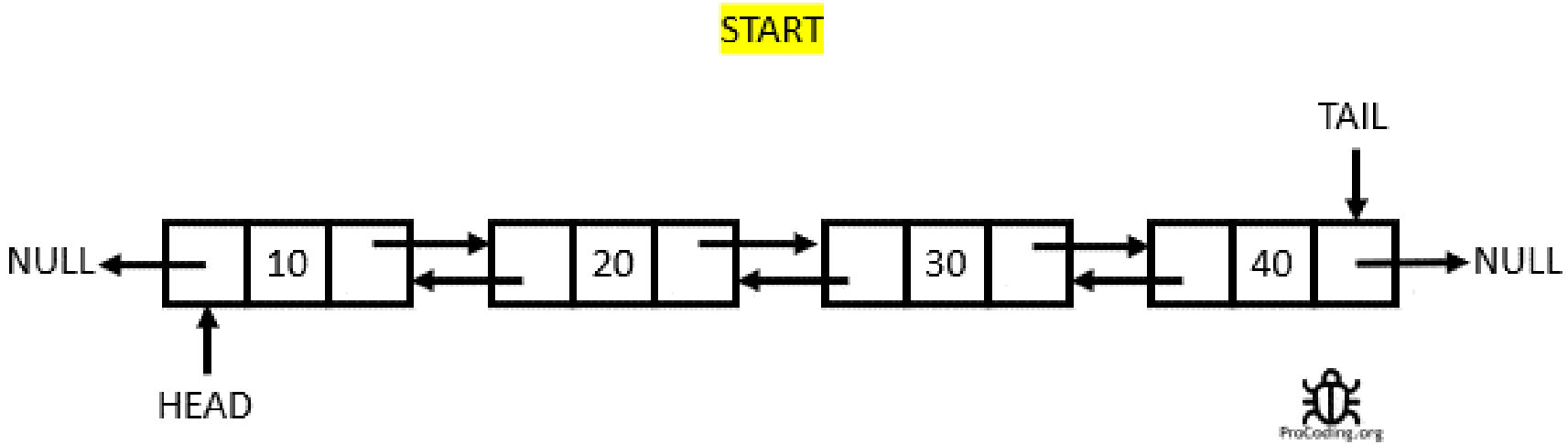
Deletion from a Doubly Linked List

Delete from End



Deletion from a Doubly Linked List

Delete from Given Position



Deletion of Doubly Linked List (Cont.)

```
struct dnode {
    struct dnode *prev;
    int value;
    struct dnode *next;
    };

void deleteNode (struct dnode *old) {
    if(head == old)     /* If node to be deleted is head node */
        head = old->next;

    /* Change next only if node to be deleted is NOT the last node */
    if(old->next != NULL)
        old->next->prev = old->prev;

    /* Change prev only if node to be deleted is NOT the first node */
    if(old->prev != NULL)
        old->prev->next = old->next;

    free(old);   /* Finally, free the memory occupied by old*/
    return;
}
```

Deletion of Doubly Linked List

Advantages of Doubly Linked List over Singly Linked List

1. Doubly Linked List can be traversed in both (forward and backward) directions
2. Only single temporary variable required for deletion of a node

Disadvantages of Doubly Linked List over Singly Linked List

1. Every node required an extra space for previous pointer
2. All operations take some time to adjust previous pointer

Thank you!

Stacks & Queues

CSE 203 (Data Structures and Algorithms) – Week 5

LEC RAIYAN RAHMAN
Dept of CSE, MIST
raiyan@cse.mist.ac.bd
Courtesy: Lec Shahriar Rahman Khan

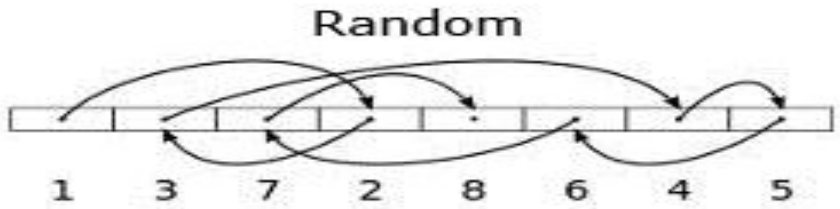
Linear Data Structures

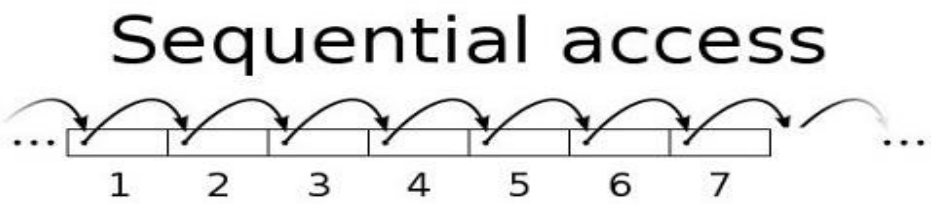
Array

- Random Access in a constant time

Linked List

- Sequential Access is possible





Operations We Can Do in SLL

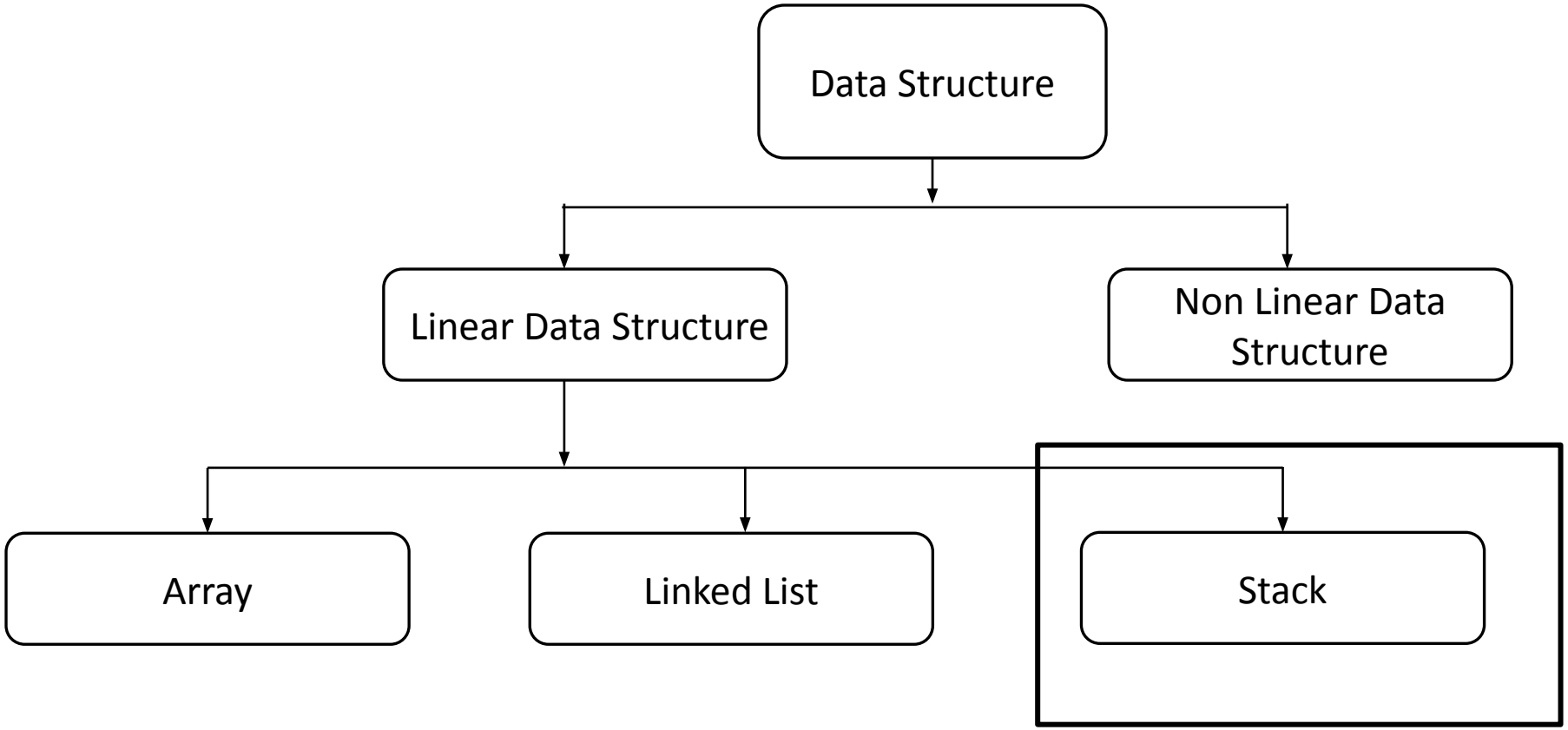

```
                Implementation of a Single Linked List
                Check the slide for Explanation
                CSE 203, Lec Raiyan

1.Insert First
2.Insert Last
3.Insert Middle (Any other pos)
4.Insert After a Target Val
5.Delete Head
6.Delete Tail
7. Delete from a Position
8.Delete a Value
9.Print
10. Add a function for practice
11.Exit

Enter Choice:
```

Maybe we don’t need to do all that!

You All After Linked List

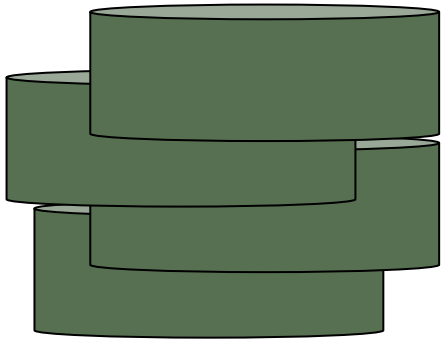
And I Agree

Difference Between Linear Data Structures



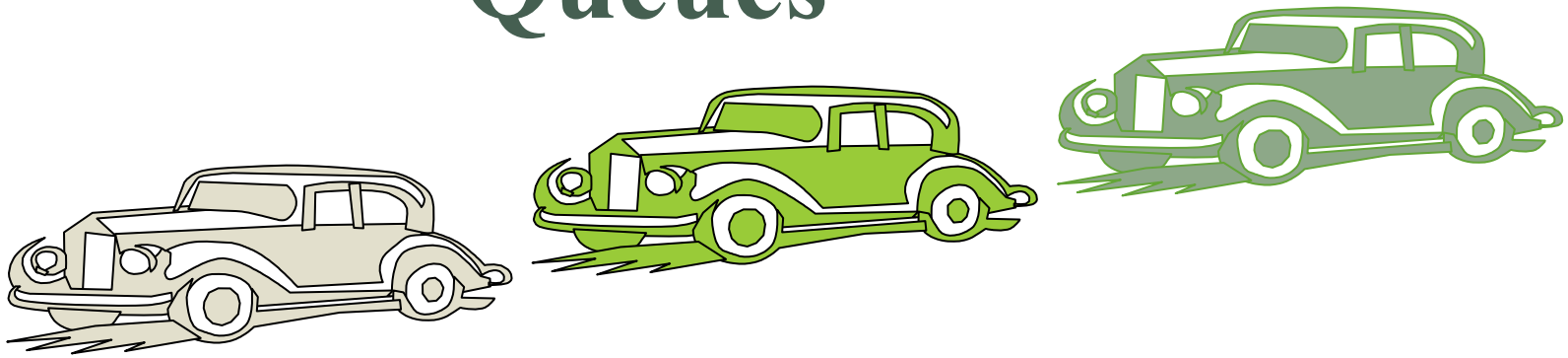
Let's do a little less (and specific) operations!

Stacks



Vs

Queues



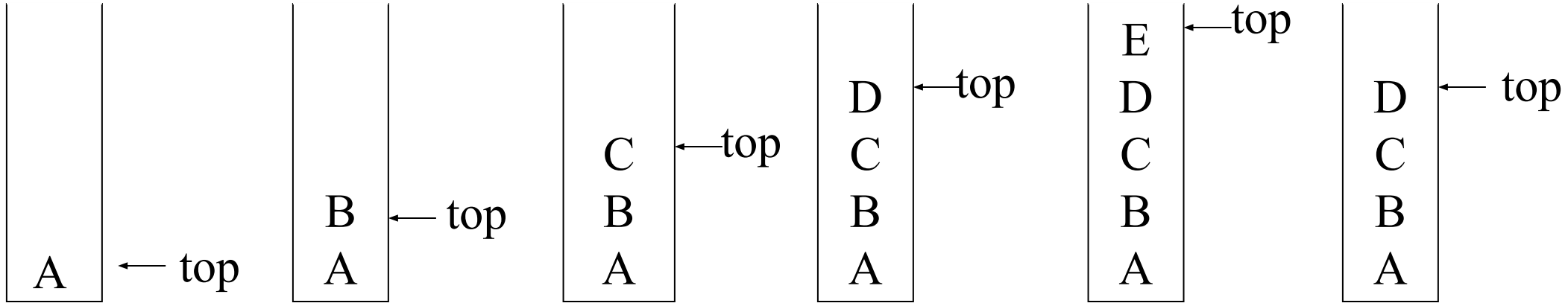
Stacks and Queues

- A stack is a **last in, first out (LIFO)** data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- A queue is a **first in, first out (FIFO)** data structure
  - Items are removed from a queue in the same order as they were inserted

See the difference? We're limiting when and where a value can be inserted to or deleted from. We can then apply those specific structures (stack and queues) where these specific operations are needed.

Stack: Last In First Out

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the *top* of the stack.
- The operations: **push** (insert) and **pop** (delete)



Applications of Stacks

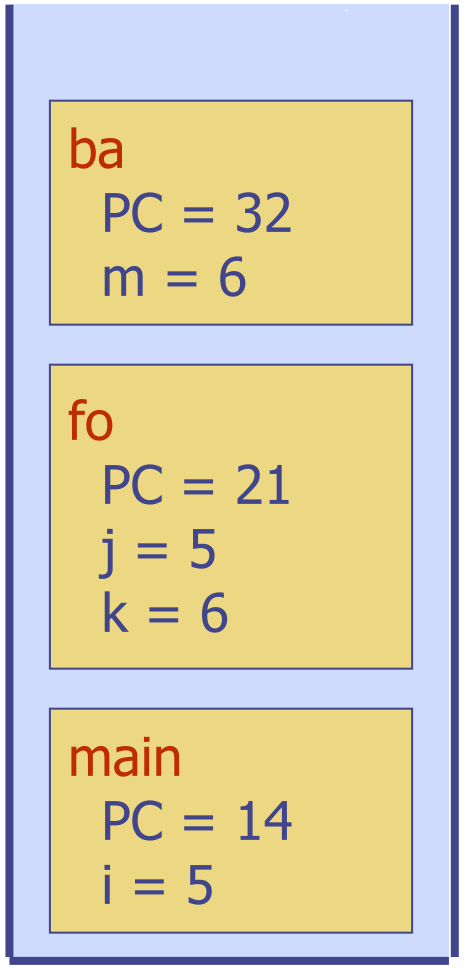
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor (CTRL-Z)
  - Saving local variables when one function calls another, and this one calls another, and so on.

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Run-time Stack of **C**

- The C run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the function on top of the stack

```
   main() {
         int i = 5;
14       fo(i);
       }

   fo(int j) {
         int k;
         k = j+1;
21       ba(k);
       }

32 ba(int m) {
         …
       }
```



What is Stack?

Stack??

Stack??

Stack??

Stack??

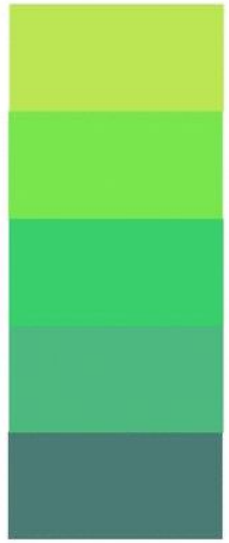
Stack??

Definition of Stack

A stack is a data structure in which items can be inserted only from one end and get items back from the same end.

top=5

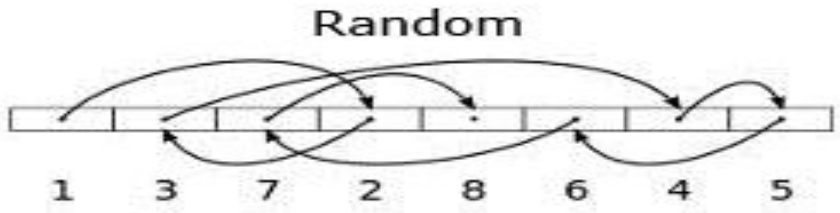
size = 5



Difference Between Linear Data Structures

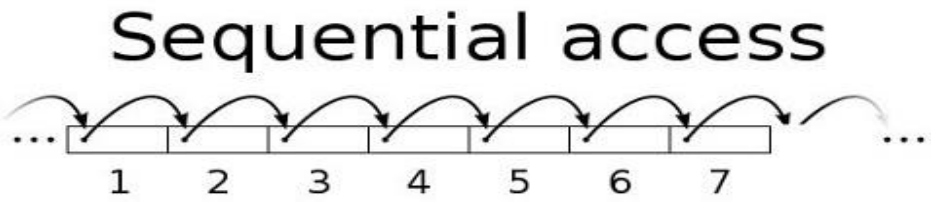
Array

- Random Access in a constant time



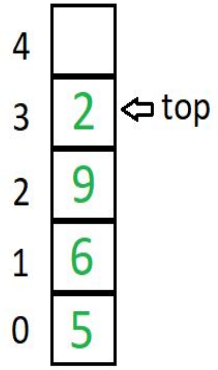
Linked List

- Sequential Access is possible



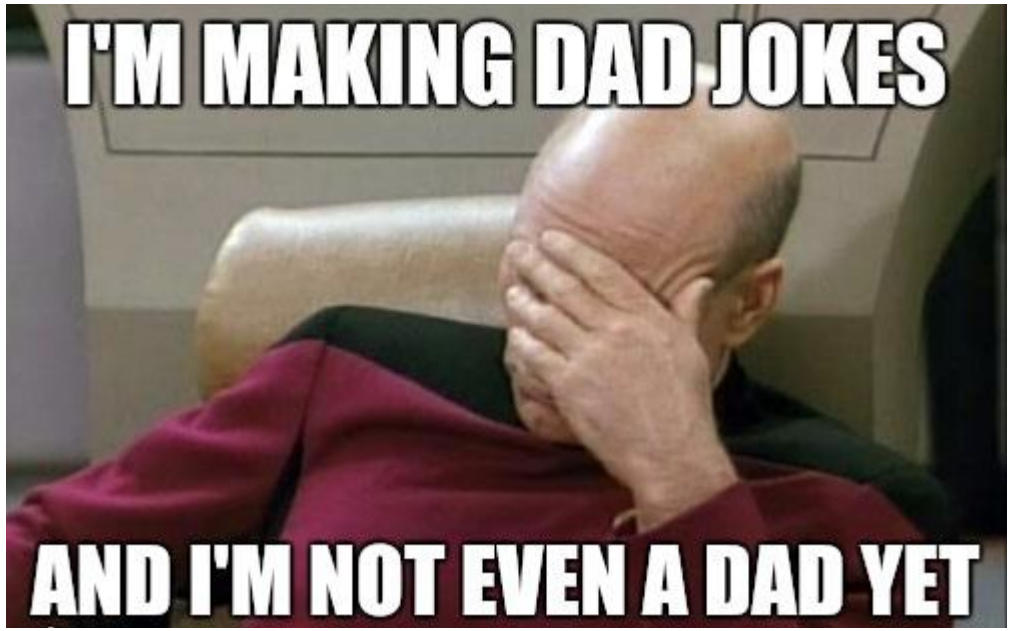
Stack

- Limited Access is possible



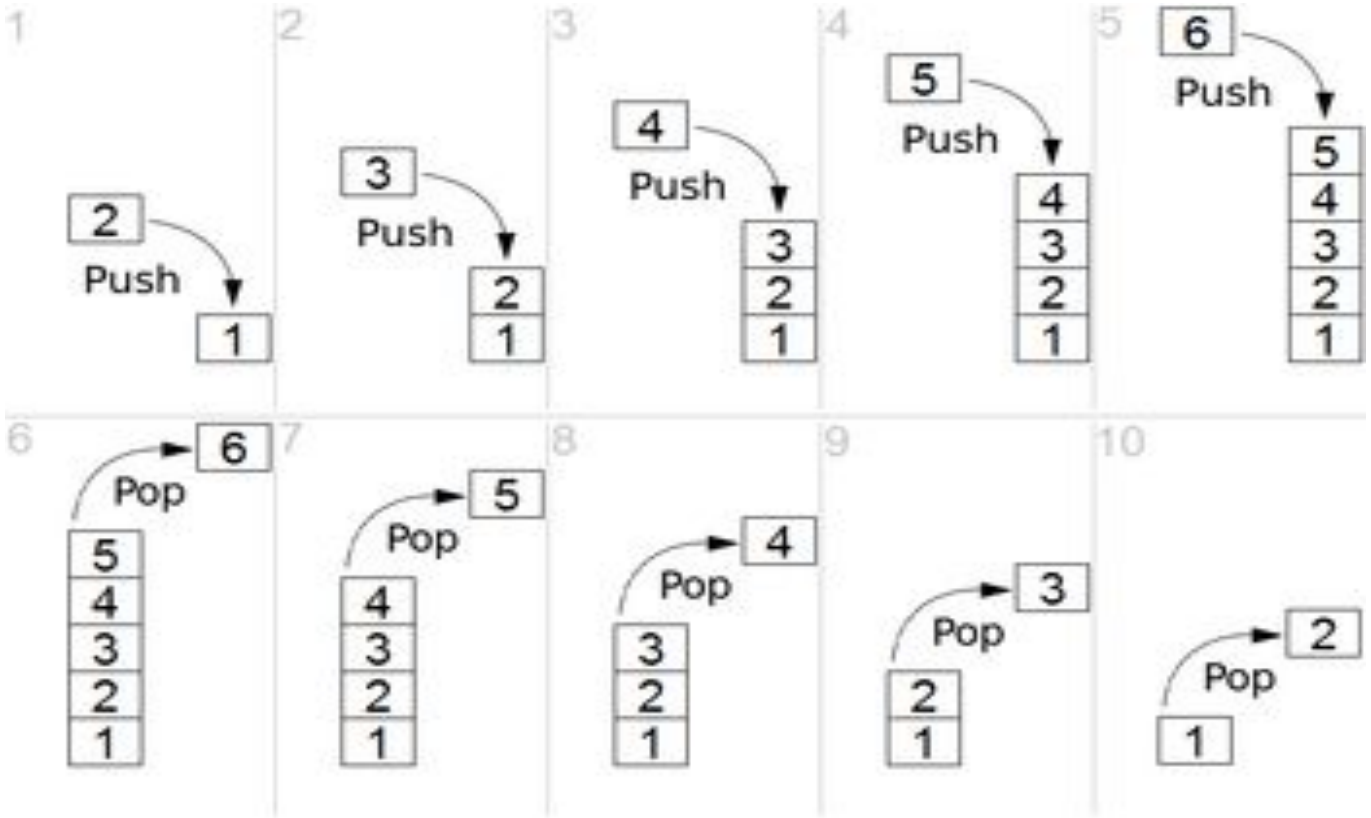
Now, Let’s get to the bottom of it!

(Of the Stack)



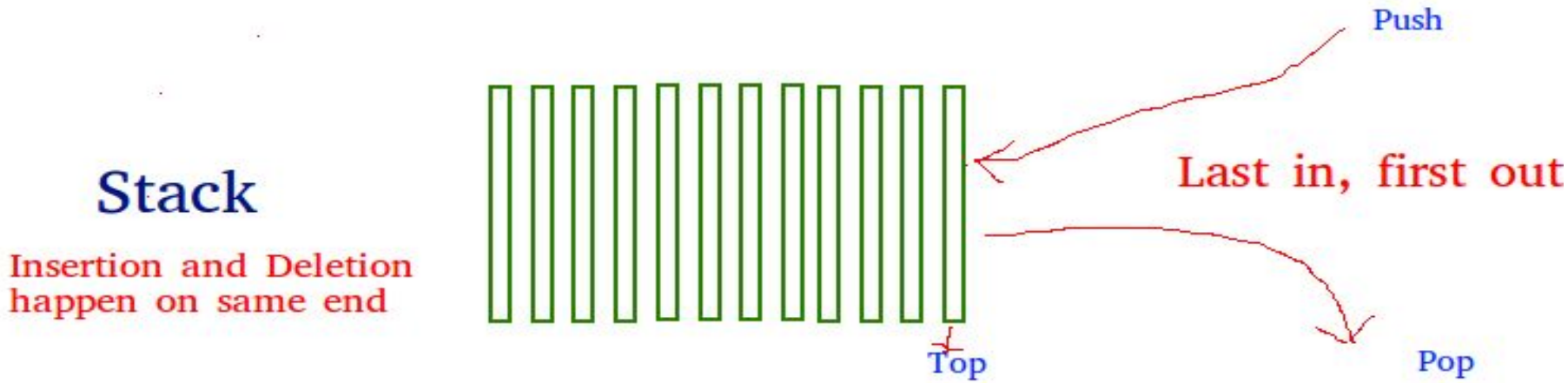
Stack Rules

Insertion and deletion is only possible from one end, that's from top



You can't insert or delete an element from bottom or left or right

Stack Order

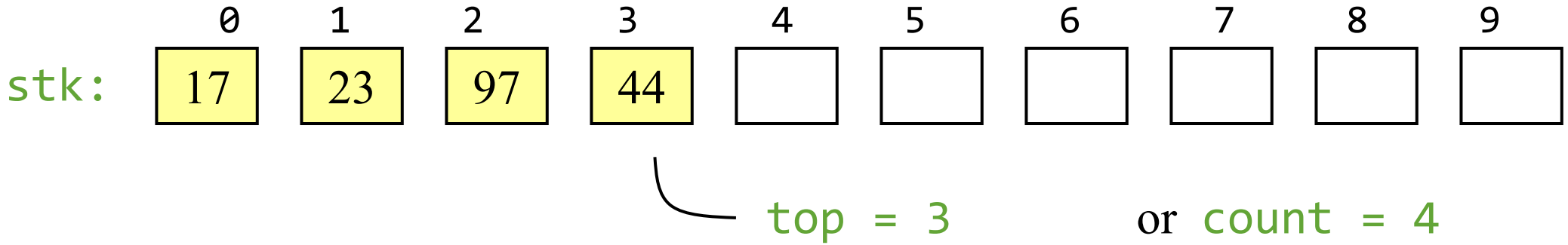


Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

Array Implementation of Stacks

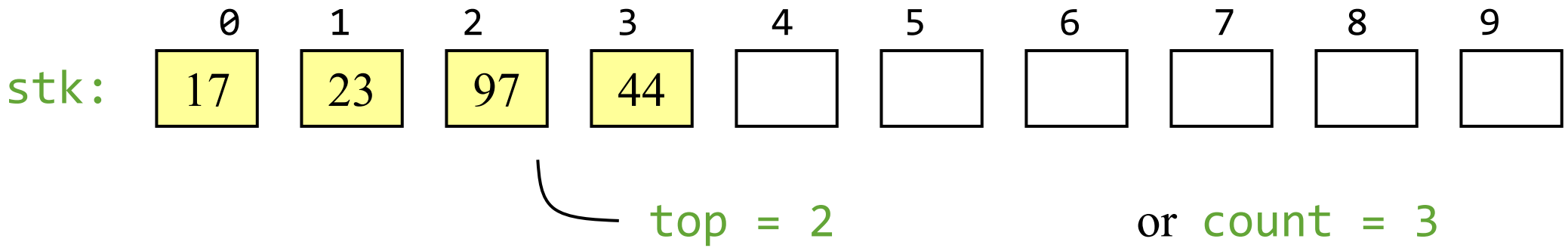
- To implement a stack, items are inserted and removed at the same end (called the top)
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack

Push and Pop



- If the bottom of the stack is at location `0`, then an empty stack is represented by `top = -1` or `count = 0`
- To add (push) an element, either:
  - Increment `top` and store the element in `stk[top]`, or
  - Store the element in `stk[count]` and increment `count`
- To remove (pop) an element, either:
  - Get the element from `stk[top]` and decrement `top`, or
  - Decrement `count` and get the element in `stk[count]`

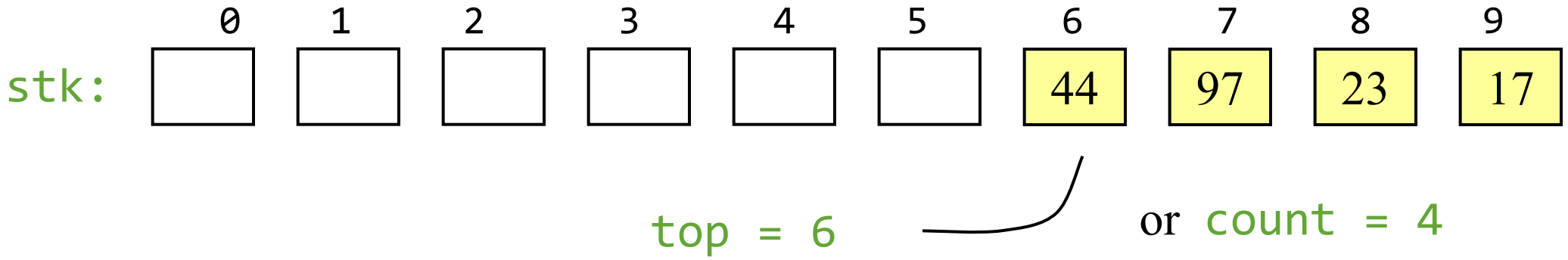
After Popping



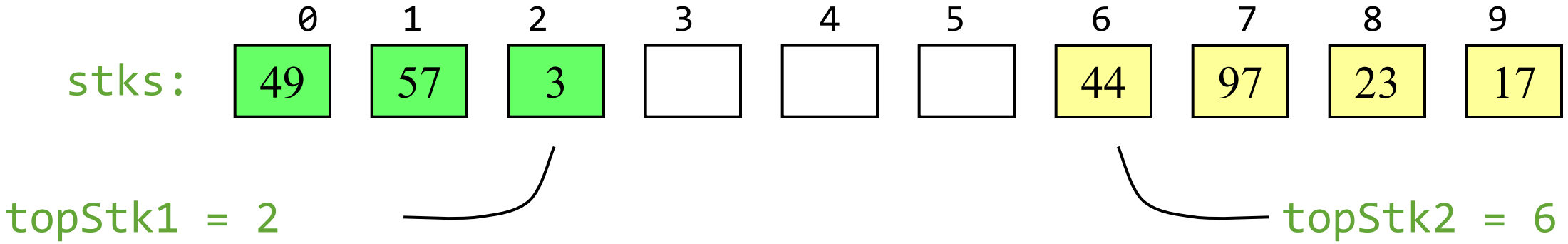
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, *"it depends"*
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to `null`
  - Why? To allow it to be garbage collected!

Sharing Space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*



Error Checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an `ArrayIndexOutOfBounds` exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

Stack Operations

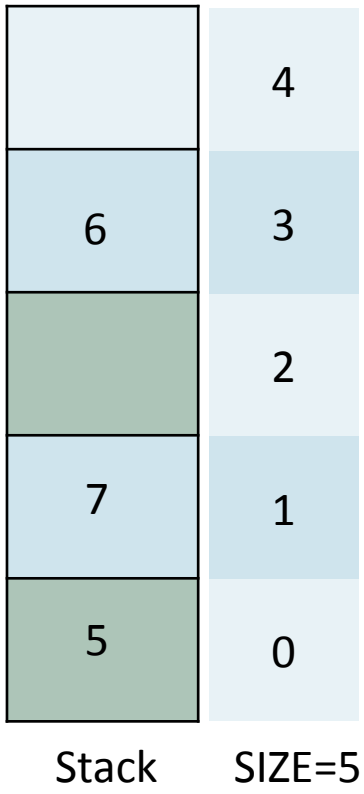
Now, we'll need at least the following operations:

- Push(x) - to insert element in the Stack.
- Pop() - to delete element from stack.
- Peek()/Top() - to display element at top.
- isEmpty() - to check if the Stack is empty or not.
- isFull() - to check if the Stack is full or not.

Except these basic operations, you can do many operations like Sorting array using Stacks, Reverse individual words, find the min or max value from the stack, reverse the stack etc.

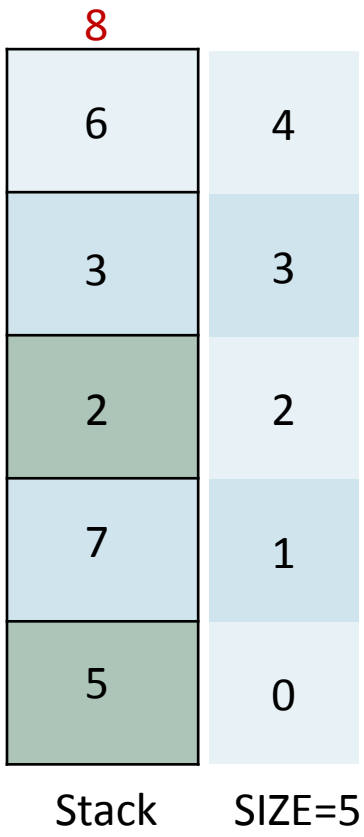
Push(X) Operation

- Let, This is our stack.
- Now if we push an element in this stack, what will happen?
- Let, we call the function push(5).
- It seems like, the element has dropped in the bottom of the stack.
- Then if we push second element, let's say push(7).
- It has dropped on top of the element 5.
- Now, is it possible that, if we want the next element to be at index 3?
- The answer is NO! The next element will always drop on the current top element.



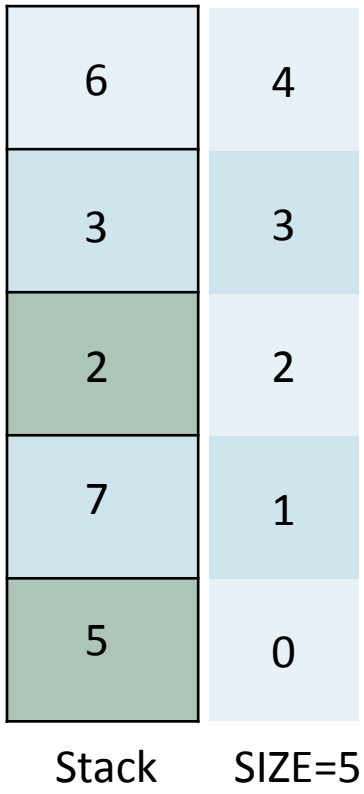
Overflow Condition

- Let, we call the function push(2).
- Then, we call the function push(3).
- Then, we call the function push(6).
- Now, if we push the 6th element, let's say push(8). What will happen?
- Yes, it will not be able to push in the stack, as the stack has reached it's maximum size, which is 5.
- This is called overflow condition of stack.



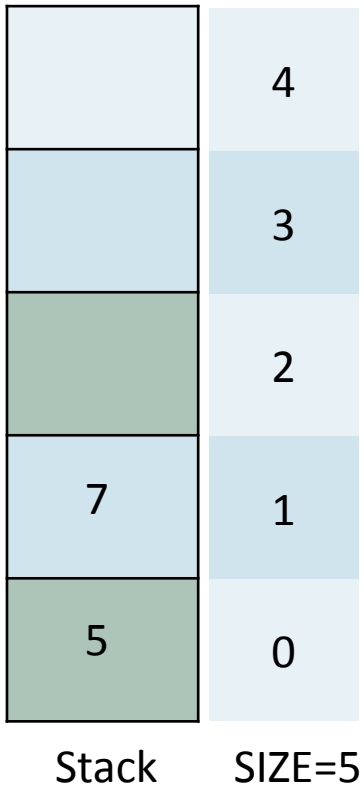
Pop() Operation

- Now let's say, we want to delete the element at index 2.
- Can we do that without hampering the elements at index 3 and index 4?
- No, we can't do that as long as this is stack!
- First, we have to delete/remove/ pop the element at index 4.
- Then, we have to delete/remove/ pop the element at index 3.
- Finally, we have to delete/remove/ pop the element at index 2.
- So, have you understood the concept LIFO – Last In First Out? We pushed the element 6 at last, but we are popping out element 6 at first. That's LIFO.



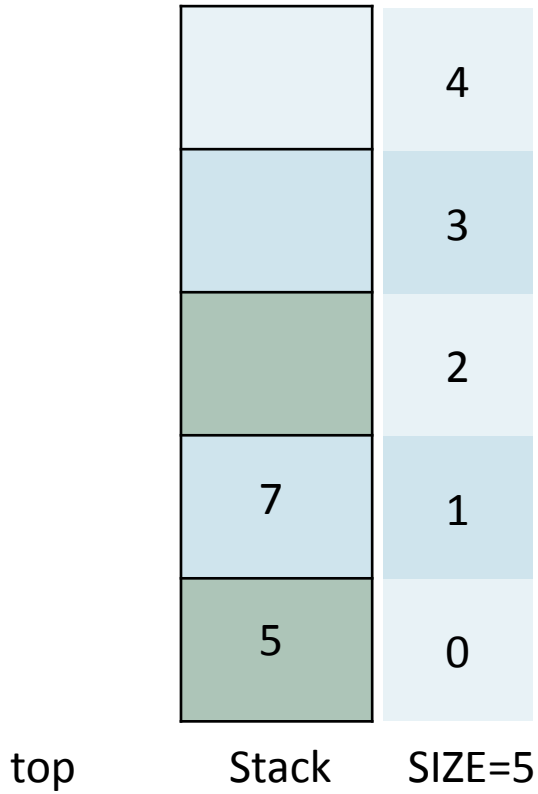
Underflow condition

- Now, let's pop out the element at index 1 and index 0.
- We call the function Pop(). That will pop out 7.
- Then, we again call the function Pop(). That will pop out 5 from the stack.
- Now, we try to call the function Pop() again. What do you think will happen?
- There is nothing in the stack, the stack is empty.
- So, if we try to pop out something from an empty stack, the situation will be called underflow Condition of a stack.
- I hope, you have understood the push and pop function and the overflow and underflow condition.



Top variable

- Now, how to track which element is on top of the stack?
- To do that, a variable named top is introduced.
- Initially, the value of the top = -1.
- Top = -1 means the stack is empty. So, before pushing an element into the stack, top must be incremented. That's top++.
- Now, if we insert the next element 7, what should be done?
- Two steps:
  - Increment the top variable: top++
  - Insert the element in the top position, array[top]=7



Top variable

- Now, when we pop something, what do we do?
- We delete the topmost element from the stack.

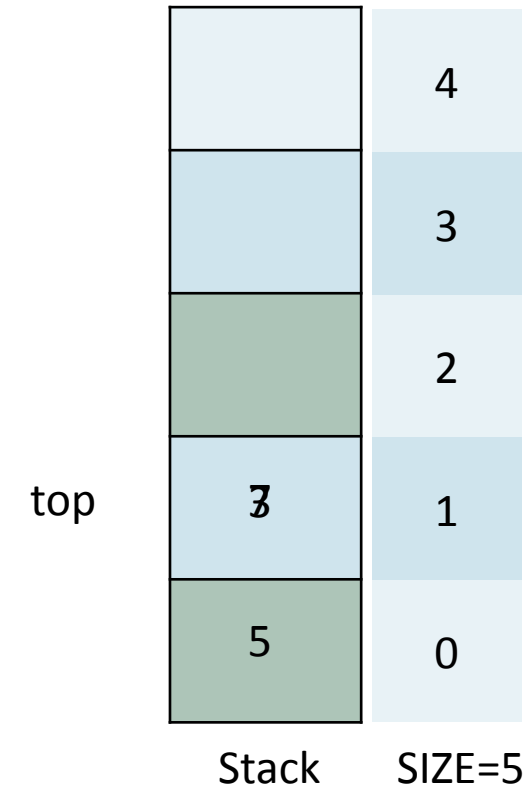
- If we want to return the topmost element, then the steps are:
  - Store the topmost value in a variable: temp = arr[top]
  - Decrement variable top: top --
  - Return temp

- If we don't want to return the topmost element, just want to delete it, then
  - We simply just decrement top: top --

- Suppose, we decrement top. What will happen to element 7?

- It will be treated as garbage value.

- Next time, when I will push an element (say 3) to the stack, it will overwrite 7.



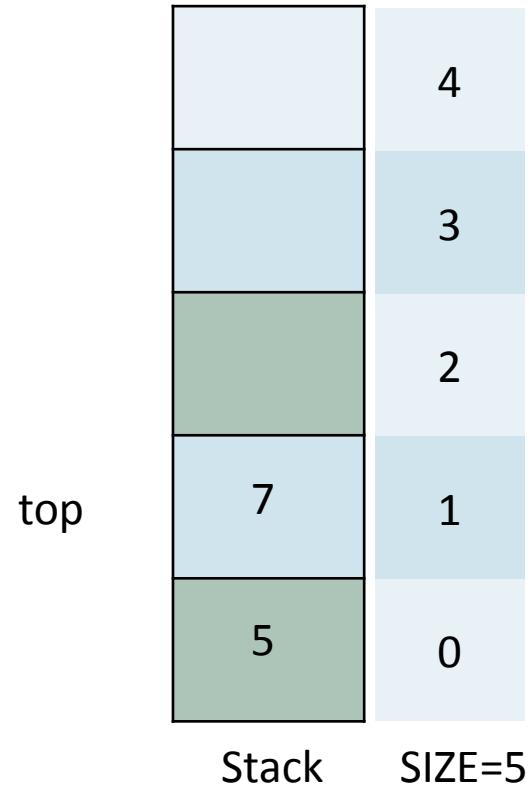
Understanding Memory Structure

- So far, we have talked about stack, but what do we need to create a stack?
  - An array to store elements of a stack
  - Variable named top
  - Size of the array

- Now, suppose you have to maintain 1000 stacks, will you declare these variables 1000 times?
- Then, what's the solution? Using class!

```
class Stack{
        int top;
        int capacity;
        int *arr;  };
```

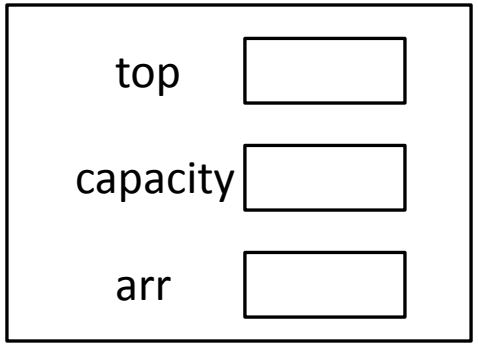
- Now, in future, whatever number of stacks we may need, we just have to declare a variable of Stack type, where we can access these 3 variables automatically.

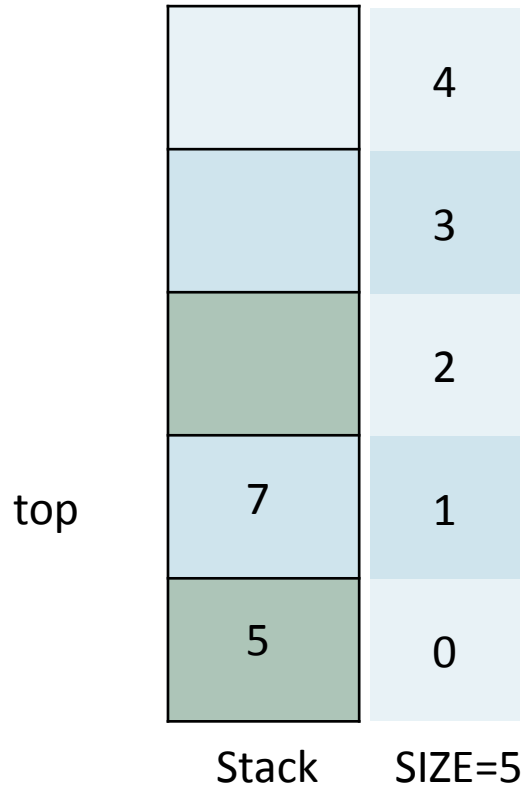


Understanding Memory Structure

- class Stack{
  int top;
  int capacity;
  int *arr; };

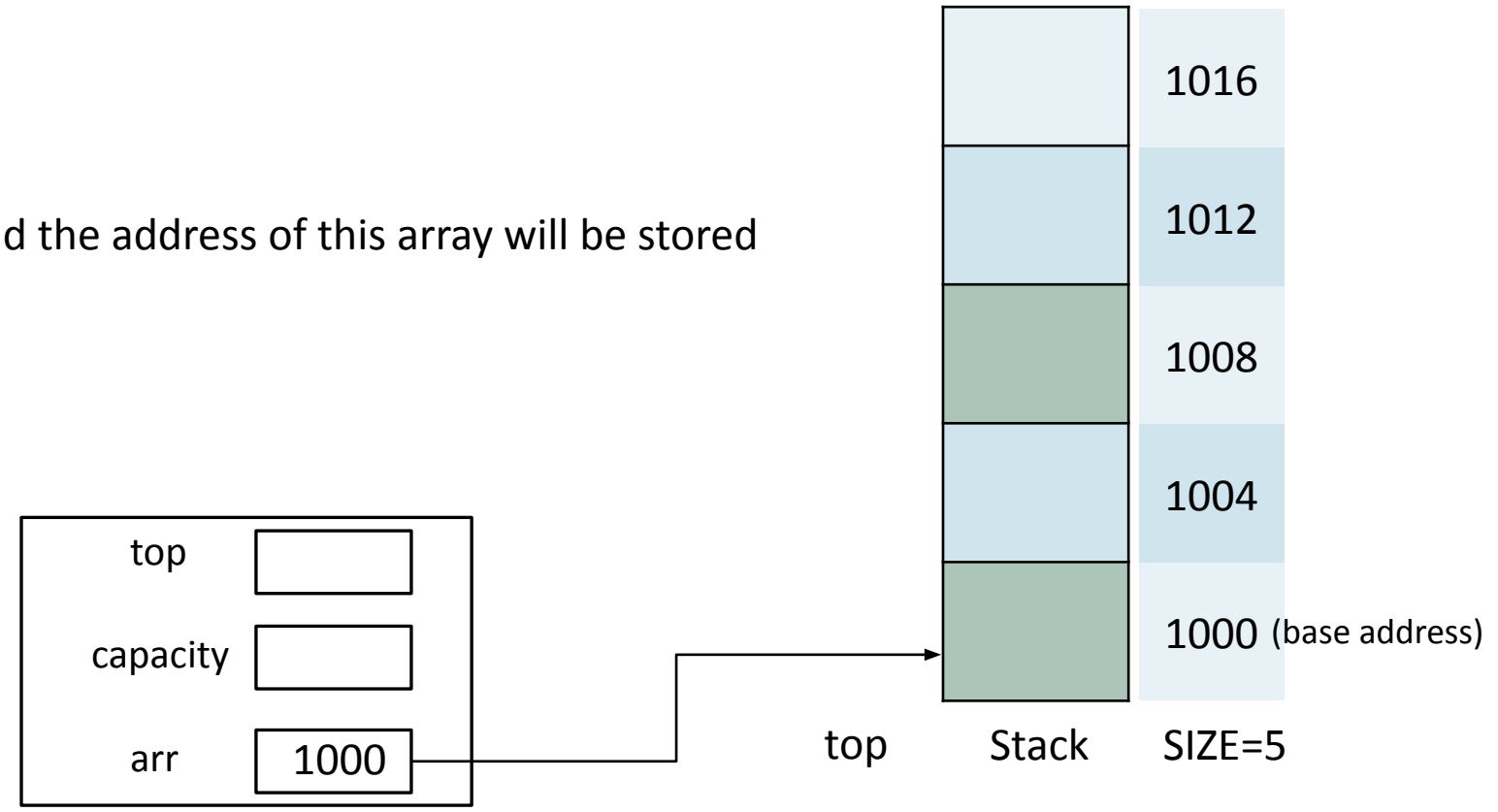
- Now for this Stack, we are declaring an object of this Stack class.
  Stack s1(5);





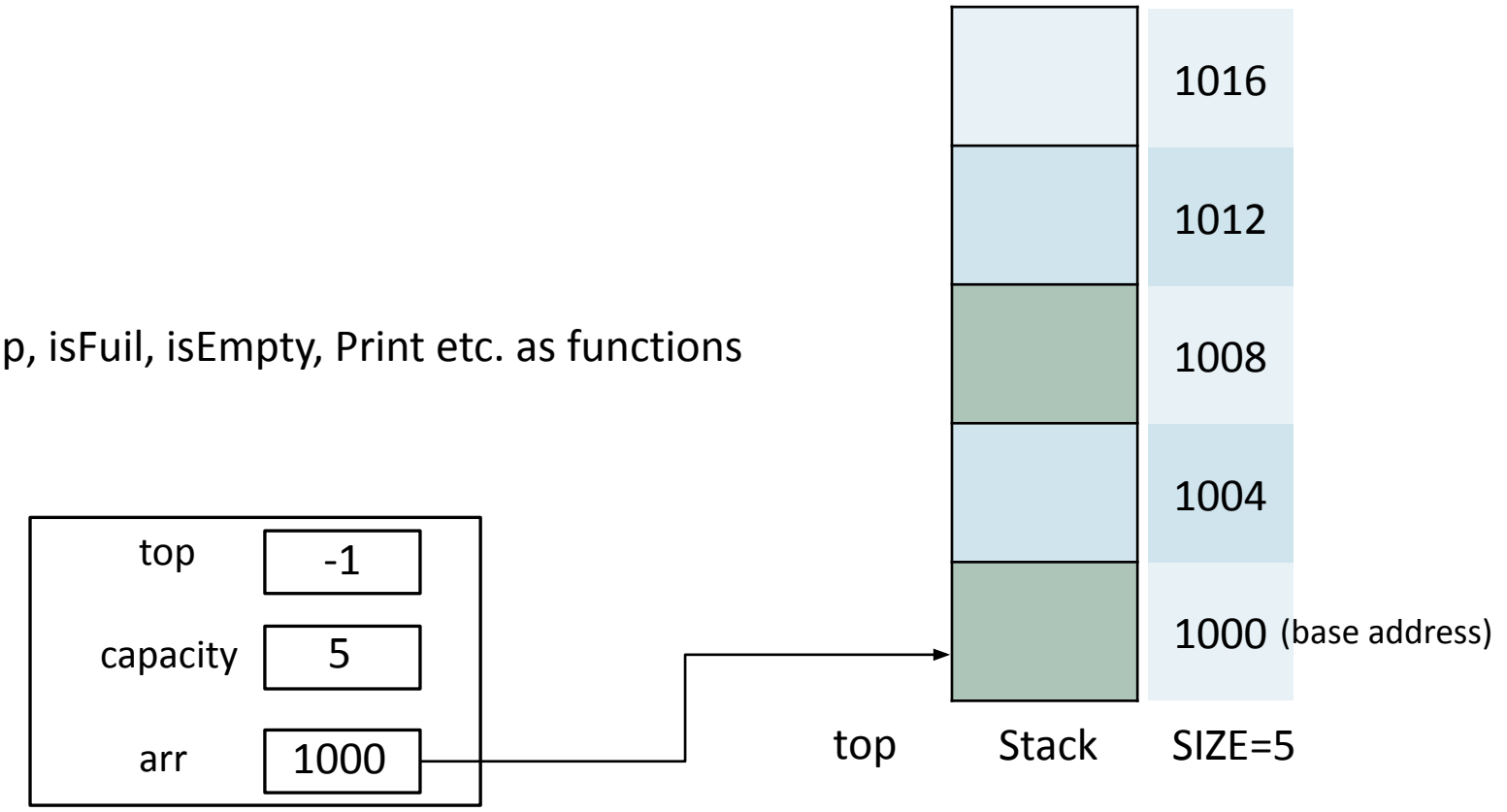
Understanding Memory Structure

- class Stack{
  int top;
  int capacity;
  int *arr; }
- Now we have to create this array, and the address of this array will be stored in *arr.
  arr = new int[capacity];



Declaration of Stack with a Class

```
Class Stack {
Stack(int n){
  top=-1;
  capacity=n;
  arr = new int[capacity];
}
//Add the necessary features (push, pop, isFuil, isEmpty, Print etc. as functions
}

int main(){
    Stack stk(5);
}
```



Tired? 

IF YES, THEN LET’S HAVE A BREAK!!

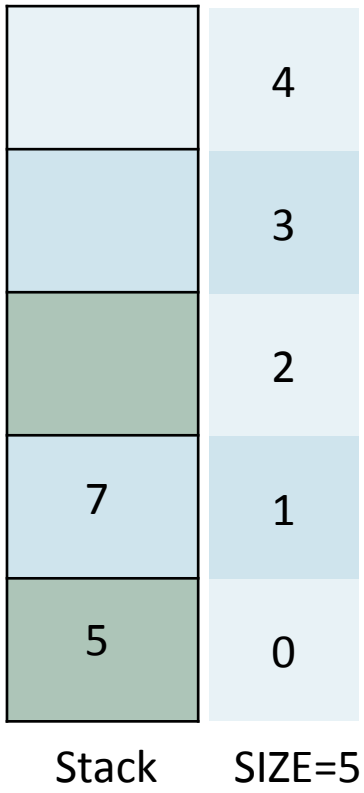
AND THEN LET’S DIVE INTO SOME CODING!

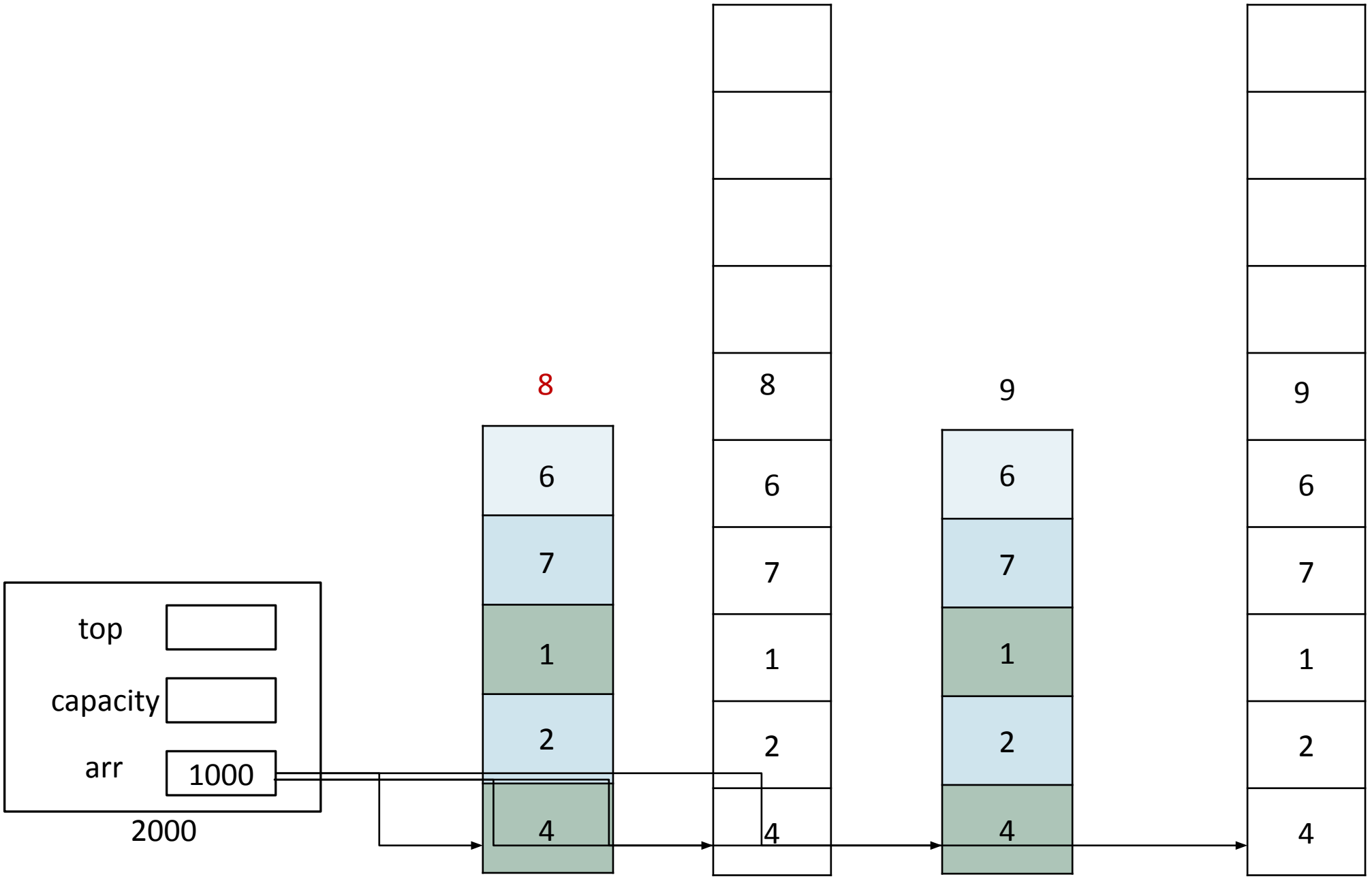
Stack Using Dynamic Array

A LITTLE BIT MORE CONCEPT

Basic of Dynamic Array

- In previous case, there was a condition called overflow condition, when stack reached it's maximum capacity.
- Now what's the solution to overcome this problem?
  - Yes, dynamic array!
- What we will do now is, we will follow these steps:
  1. We will keep pushing elements into the stack
  2. Check if the stack is full
  3. If full, then create an array of double capacity
     Copy all the element from previous array to new array
     Release the memory of previous array
     Go to step 1
- As we are increasing the size of array during runtime, that's why this is called stack using dynamic array.



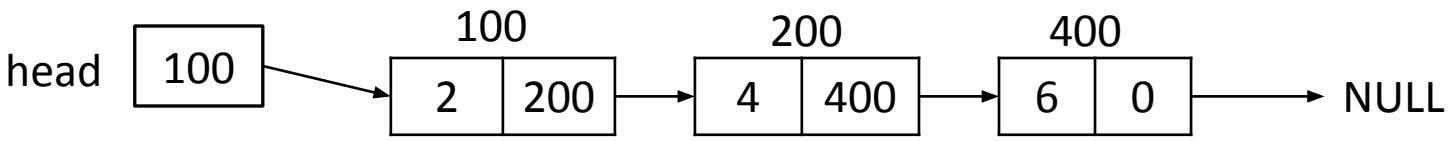

top
capacity
arr
1000
2000
8
6
7
1
2
4
8
6
7
1
2
4
9
6
7
1
2
4
9
6
7
1
2
4

Stack Using Linked List

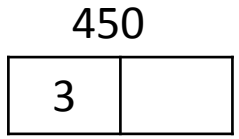
A LITTLE BIT MORE CONCEPT

Stack Using Linked List

- Here, we have to implement stack using linked list, but with restrictions.
- We know in stack, insertion and deletion is done only from one end!
- The behavior of stack can be easily implement in Linked List.



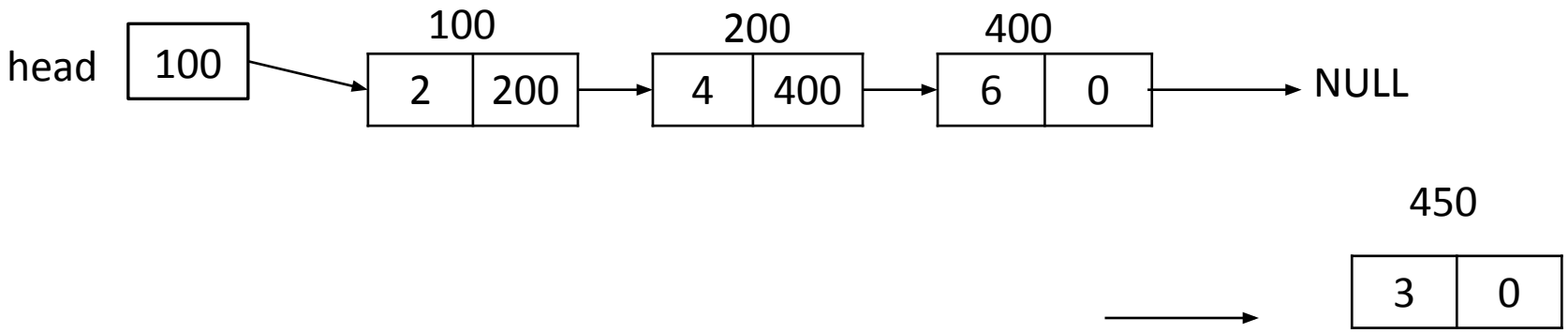
- We already know how to create a linked list.
- To insert a new node:
  - First we create a new node
  - It has an address



- Now we have to insert it in linked list from one end, only then it will perform as stack.
- It has 2 ways:
  - At end of the list (tail)
  - At beginning of the list (head)

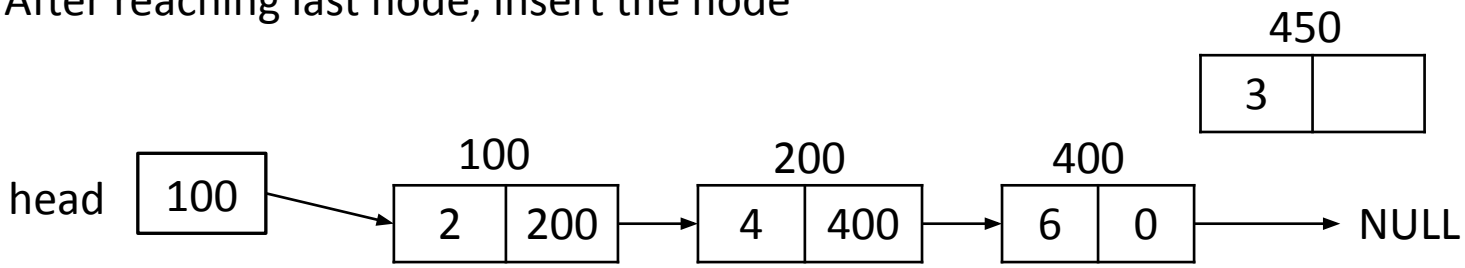
Stack Using Linked List

- Here, Inserting/deleting a node at end of the linked list takes O(n) time.
- But in stack, operations like push(x), pop(), isEmpty() these takes O(1) time.
- To insert a new node at end of list:
  - Go to first node, then next node, then next node
  - After reaching last node, insert the node



Stack Using Linked List

- Here, Inserting/deleting a node at end of the linked list takes O(n) time.
- But in stack, operations like push(x), pop(), isEmpty() these takes O(1) time.
- To insert a new node at end of list:
  - Go to first node, then next node, then next node
  - After reaching last node, insert the node

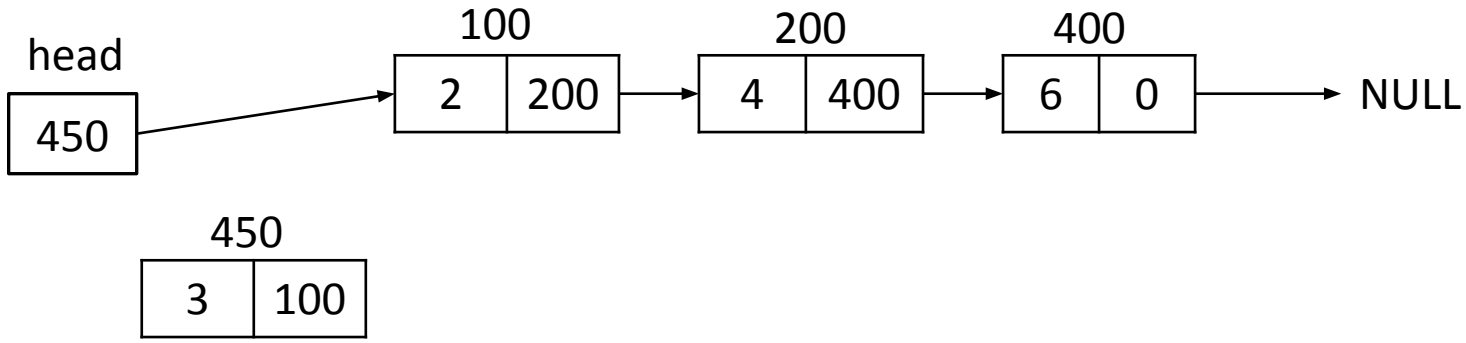


- To delete last node, it follows same process
  - Go to first node, then next node, then next node
  - After reaching the second last node, break the link with last node
  - Set the address field to 0
  - Release memory of the last node

Both operation takes O(n) time

Stack Using Linked List

- So, how to insert/delete from beginning of the linked list?



Your Task for Next Week

Don’t be afraid, it’s easy, if you have followed this class and previous classes carefully!!

1. Implement stack using Linked List
2. Change the code if necessary
3. Details of the assignment will be given soon

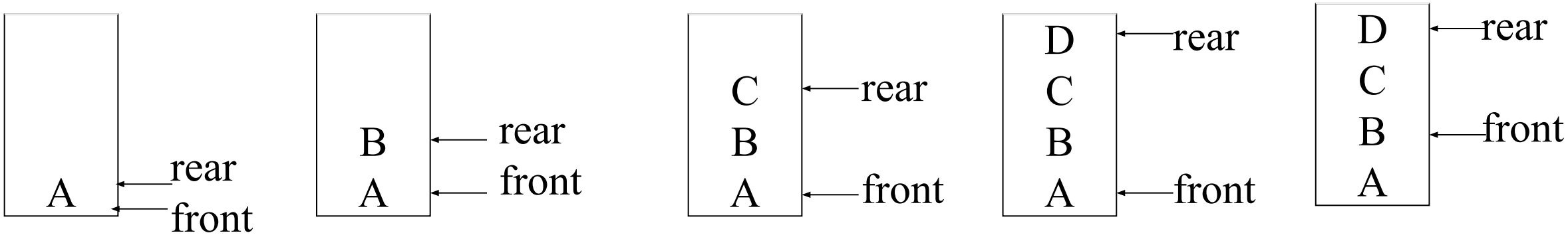
Queues

CSE 203 (Data Structures and Algorithms) – Week 5

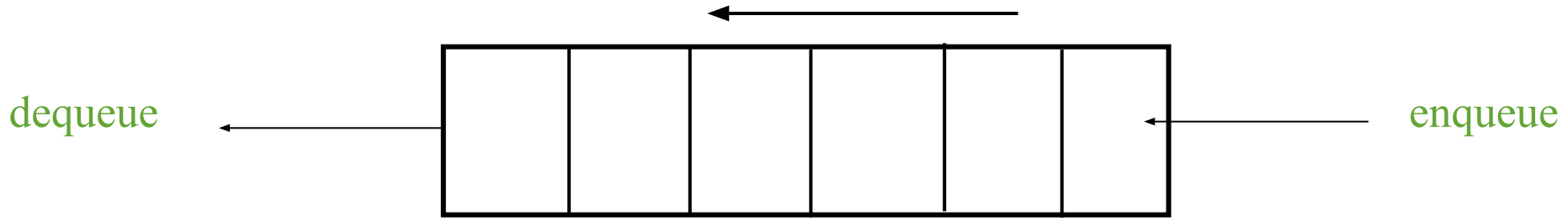
LEC RAIYAN RAHMAN
Dept of CSE, MIST
raiyan@cse.mist.ac.bd
Courtesy: Prof Dr. Abul Kashem Mia

Queue: First In First Out

- A **Queue** is an ordered collection of items from which items may be removed at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue).



- The operations: **enqueue** (insert) and **dequeue** (delete)

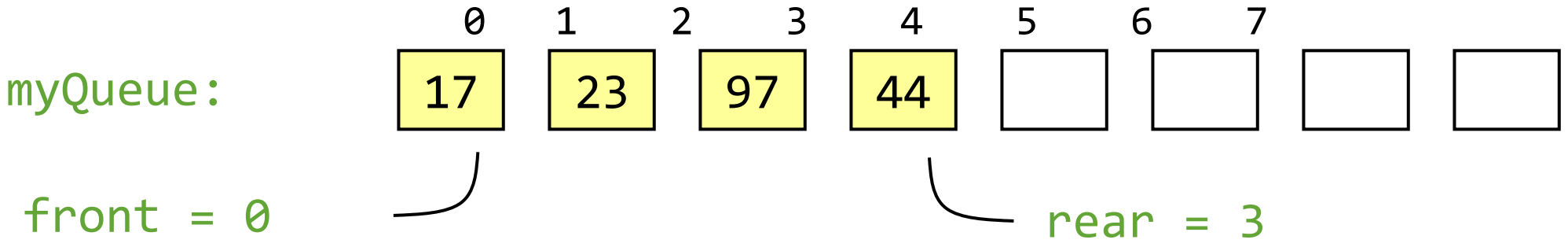


Applications of Queues

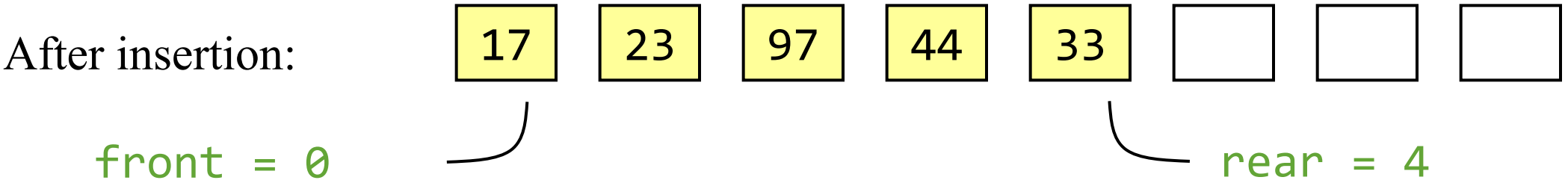
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array Implementation of Queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



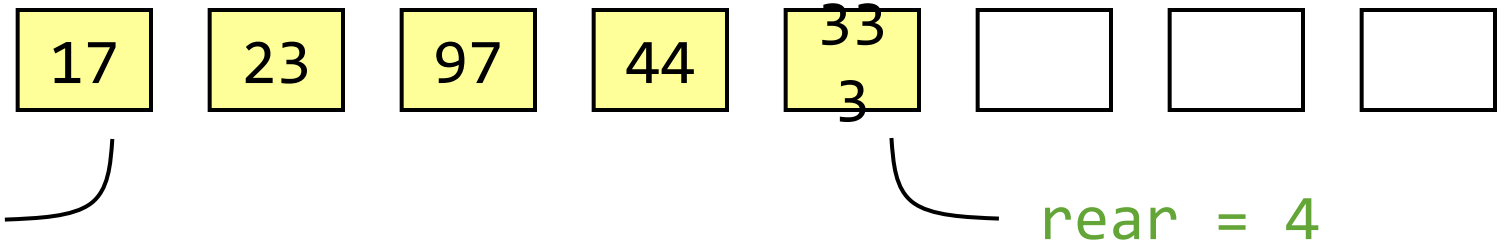
- **To insert:** put new element in location 4, and set rear to 4



Array Implementation of Queues

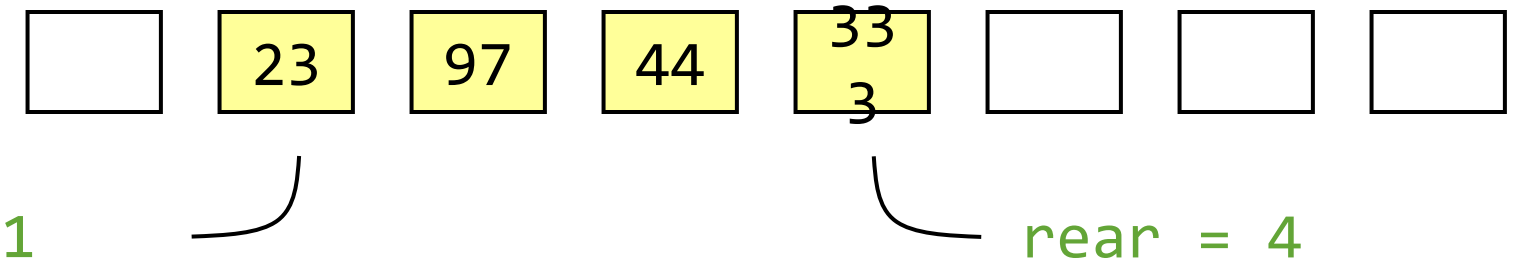
- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)

After insertion:

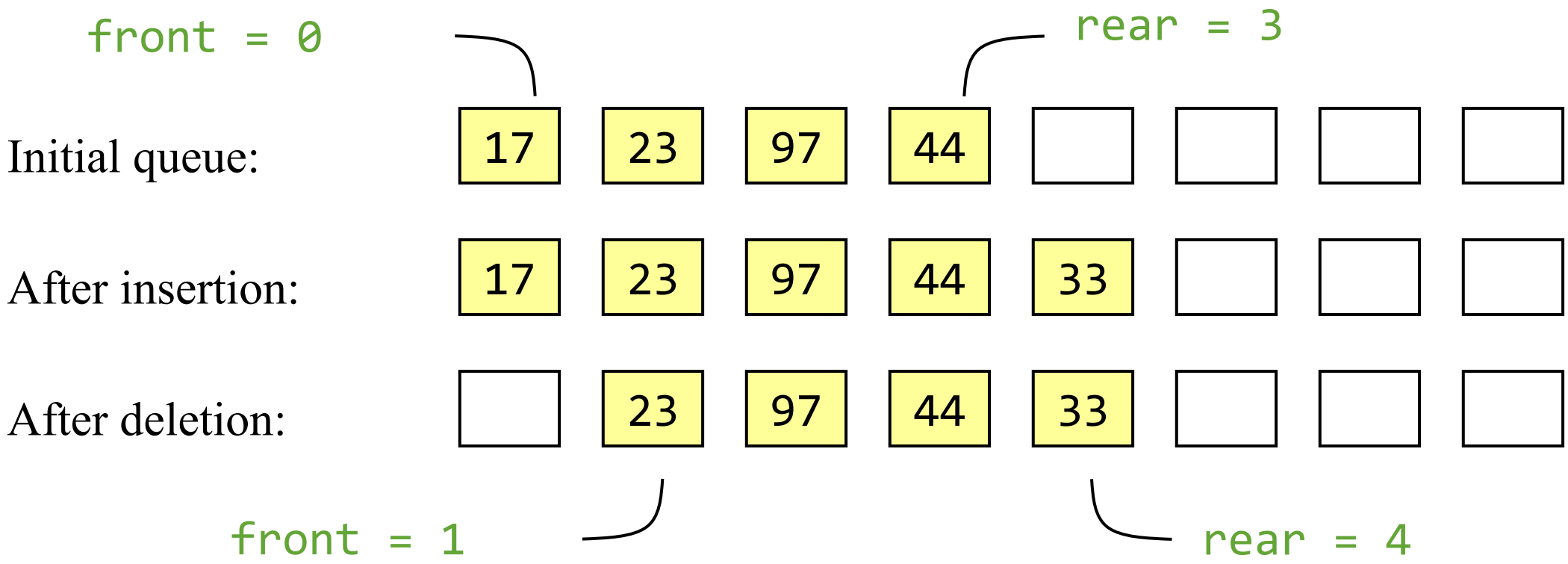


- **To delete:** take element from location 0, and set front to 1

After deletion:



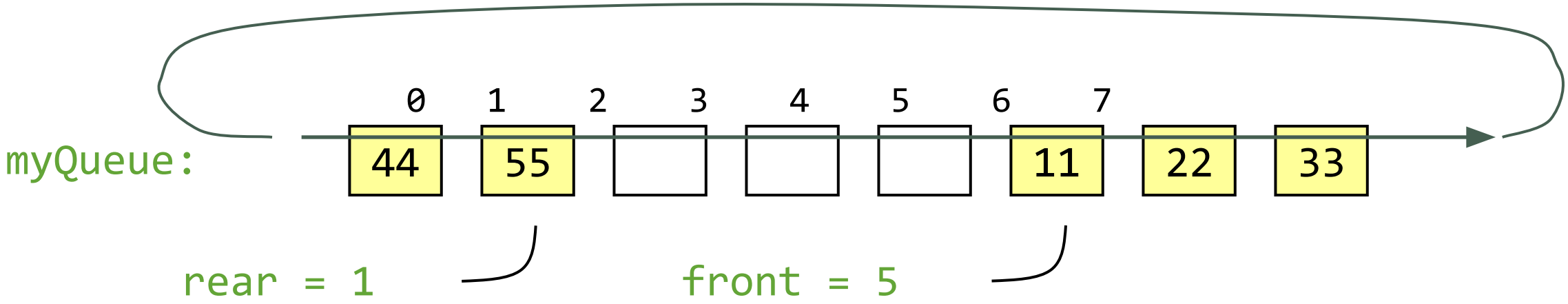
Array Implementation of Queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Circular Queues using Arrays

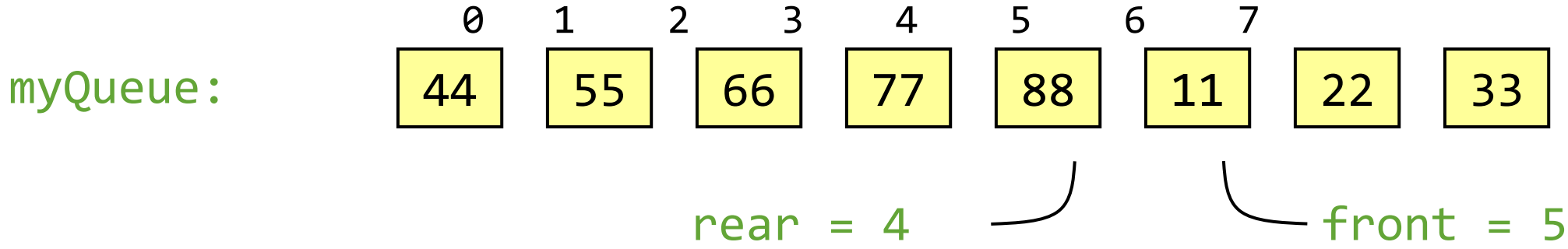
- We can treat the array holding the queue elements as circular (joined at the ends)



- Elements were added to this queue in the order `11`, `22`, `33`, `44`, `55`, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

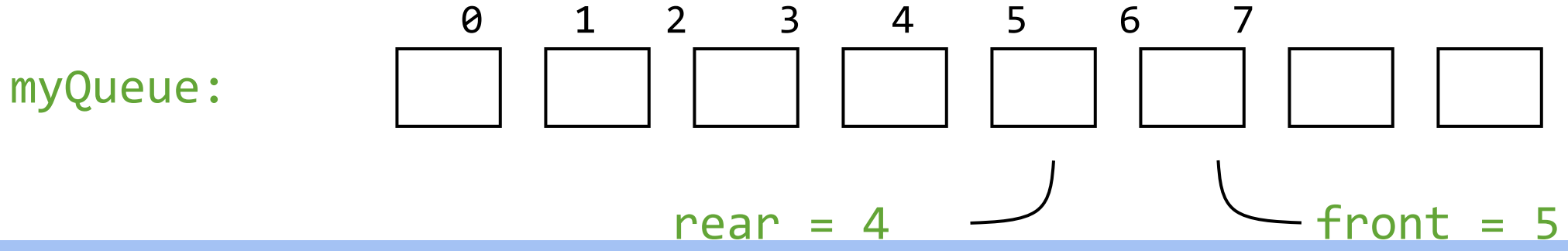
Full and Empty Queues

- If the queue were to become completely full, it would look like this:



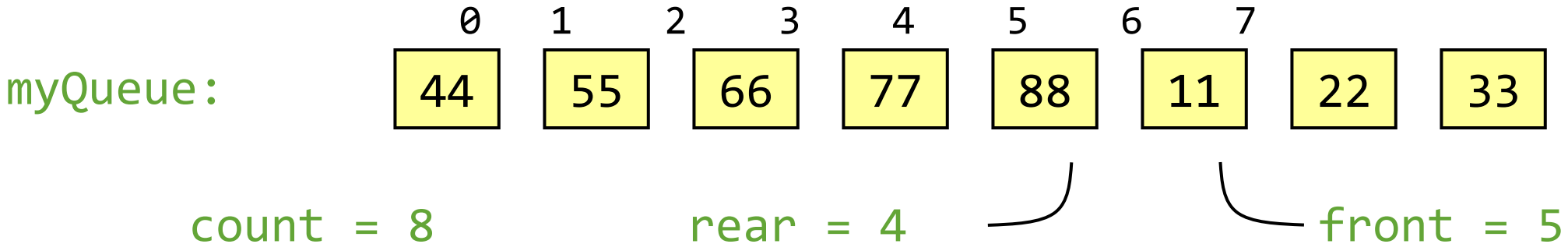
This is a problem!

- Again, if we were to remove all eight elements, making the queue completely empty, it would look like this:

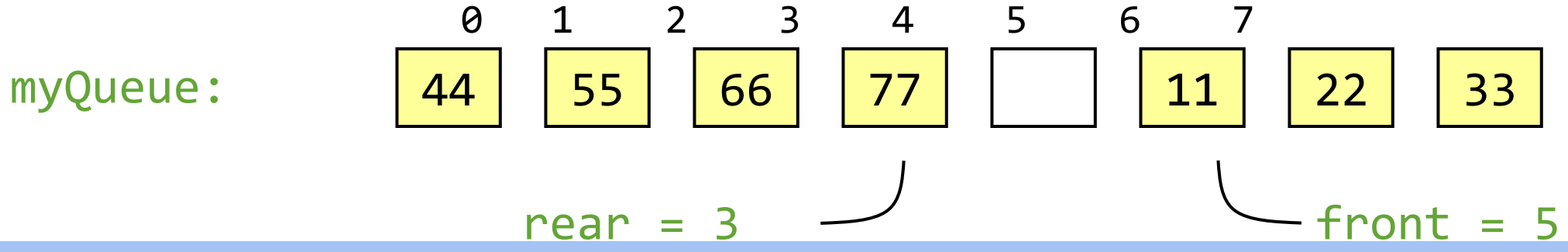


Full and Empty Queues: Solutions

- **Solution #1:** Keep an additional variable



- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements



Linked-list Implementation of Queues

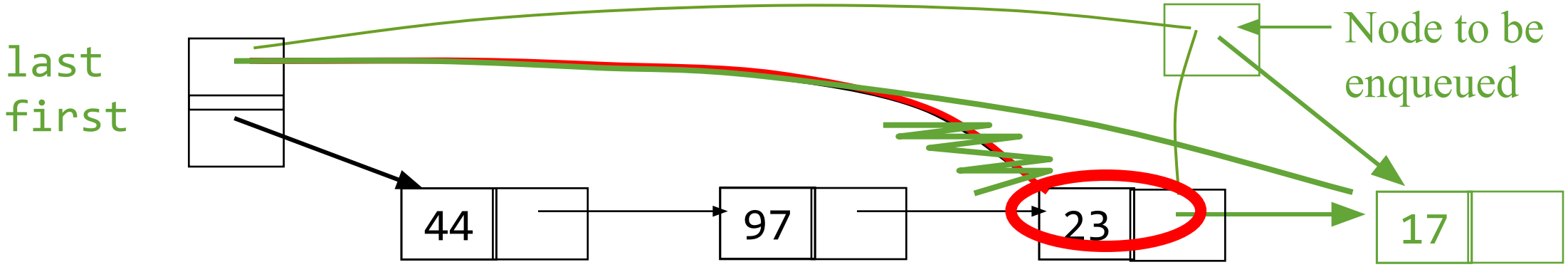
- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are $O(1)$, but at the other end they are $O(n)$
  - Because you have to find the last element each time

- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in $O(1)$ time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list

SLL Implementation of Queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Enqueueing a Node



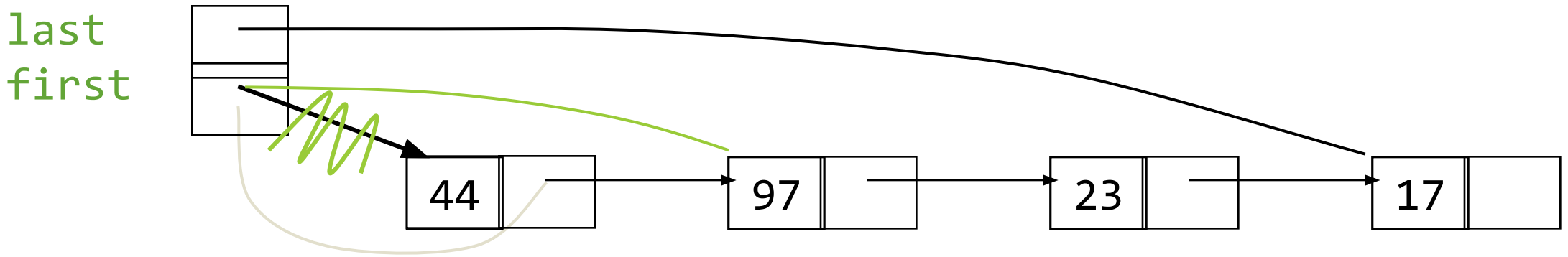
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the `last` pointer in the list header

Dequeueing a Node

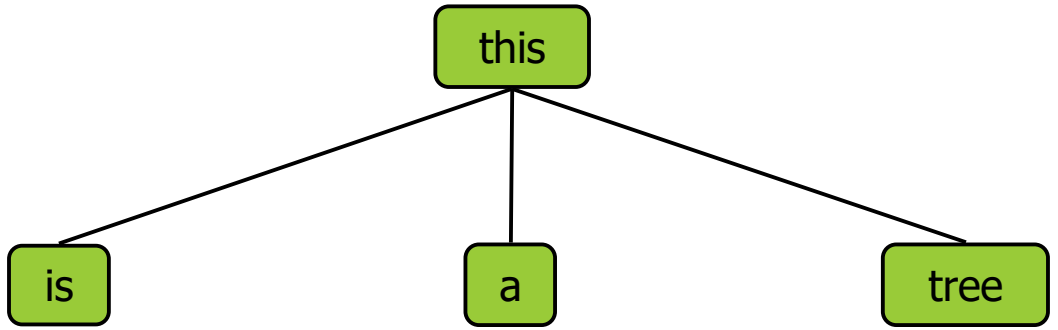


- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

Queue Implementation Details

- With an array implementation:
  - you can have both overflow and underflow
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition


عيد
Mubarak



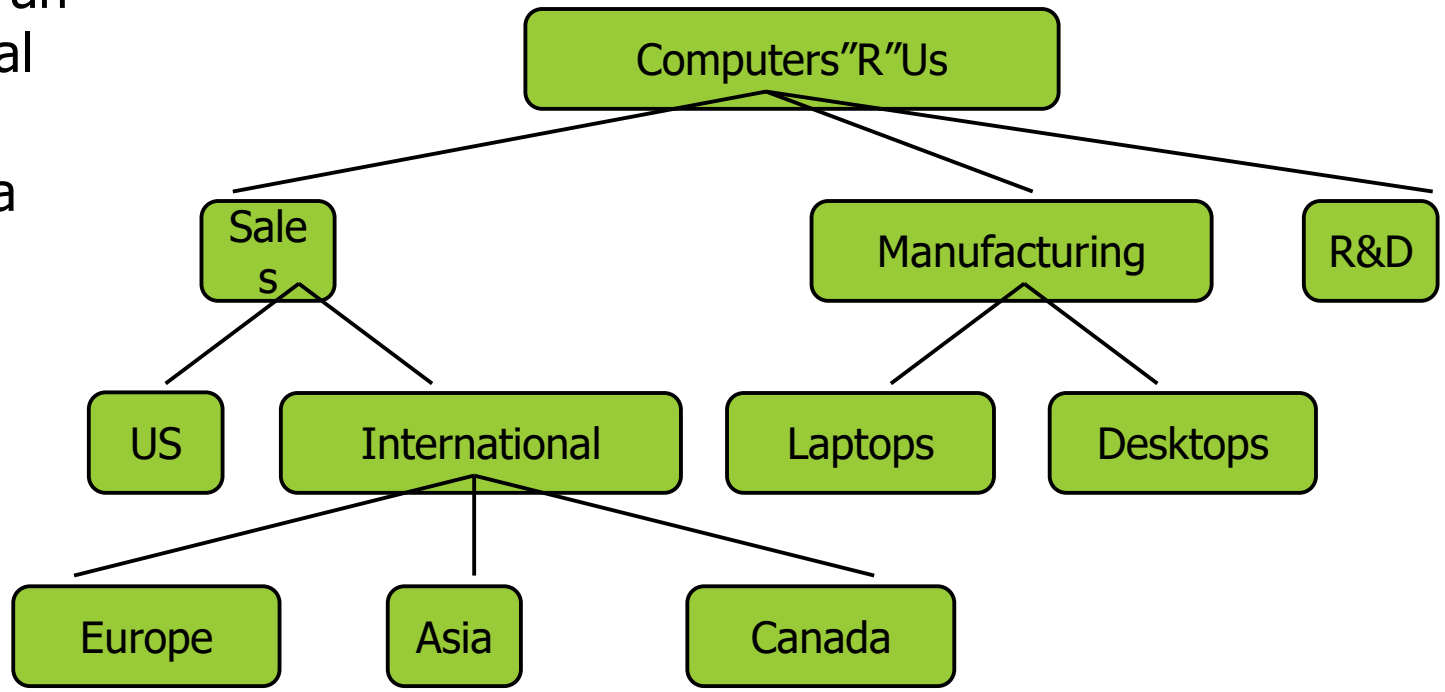
Graphs and Trees

CSE 203 (Data Structures and Algorithms) – Week 7

LEC RAIYAN RAHMAN
Dept of CSE, MIST
raiyan@cse.mist.ac.bd
Courtesy: Lec Shahriar Rahman Khan

What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relationship
- Applications:
  - Organization charts
  - File systems
  - Programming environments



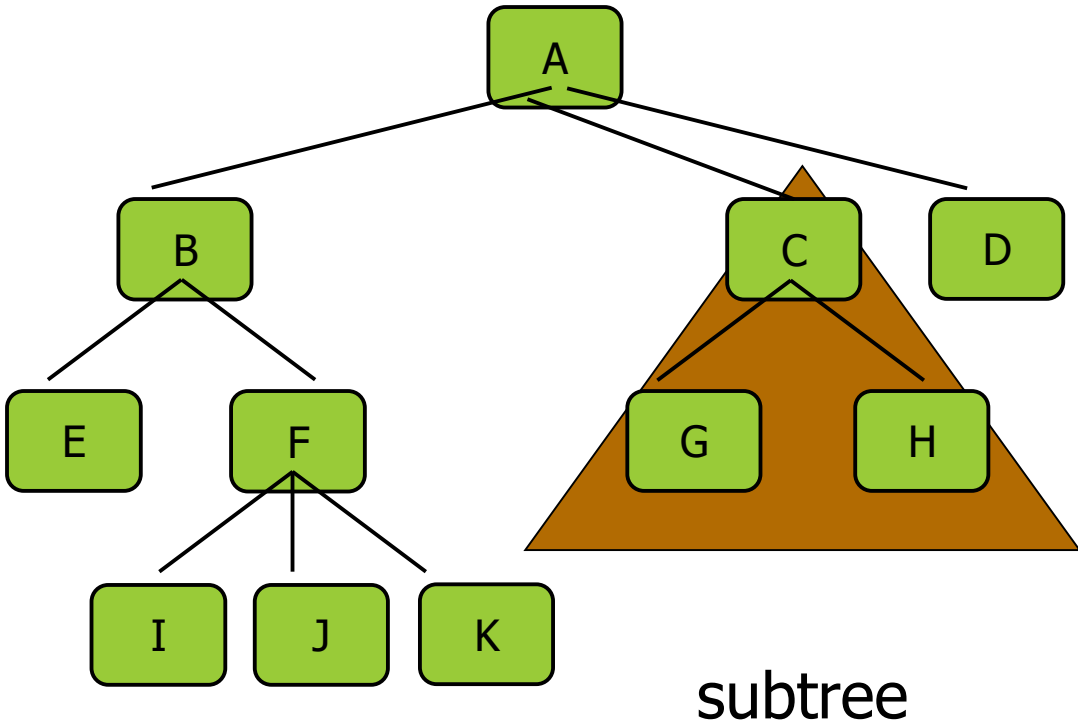
What is a Tree

- A **connected** **acyclic** graph is a tree.
- A tree T is a set of nodes in a parent-child relationship with the following properties:
  - T has a special node r, called the root of T, with no parent node
  - Each node v of T, different from r, has a unique parent node u
- A tree cannot be empty, since it must have at least one node – the root.

Graph/Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (Leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendants of a node: child, grandchild, grand-grandchild, etc.
- Depth of a node: number of ancestors, excluding the node itself
- Height of a tree: maximum depth of any node (3)
- Siblings: two nodes that are children of the same parent

- Subtree: tree consisting of a node and its descendants



Depth and Height

- The depth of a node v can be recursively defined as follows
  - If v is the root, then the depth of v is 0.
  - Otherwise, the depth of v is one plus the depth of the parent of v

```
Algorithm depth(T, v)
    if T.isRoot(v) then
      return 0
    else
      return 1 + depth(T, T.parent(v))
```

Running time: $O(1 + d_v)$, d_v is depth of v in T

In worst case $O(n)$, n is the number of nodes in T

Depth and Height

- The height of a node *v* can be recursively defined as follows
  - If v is a leaf node, then the height of v is 0.
  - Otherwise, the height of v is one plus the maximum height of a child of v

The height of a tree *T* is the height of the root of *T*

The height of a tree *T* is equal to the maximum depth of a leaf node of *T*

Depth and Height

```
Algorithm height1(T)
    h = 0
    for each v ε V(T) do
      if T.isExternal(v) then
          h = max(h, depth(T, v))
    return h
```

Running time: $O(n + \Sigma_{v \varepsilon E} (1 + d_v))$, where d_v is depth of v in T, E is the set of leaves in T

In worst case $O(n^2)$, n is the number of nodes in T

Depth and Height

```
Algorithm height2(T, v)
    if T.isExternal(v) then
        return 0
    else
        h = 0
        for each w ε T.children(v) do
            h = max(h, height2(T, w))
        return 1 + h
```

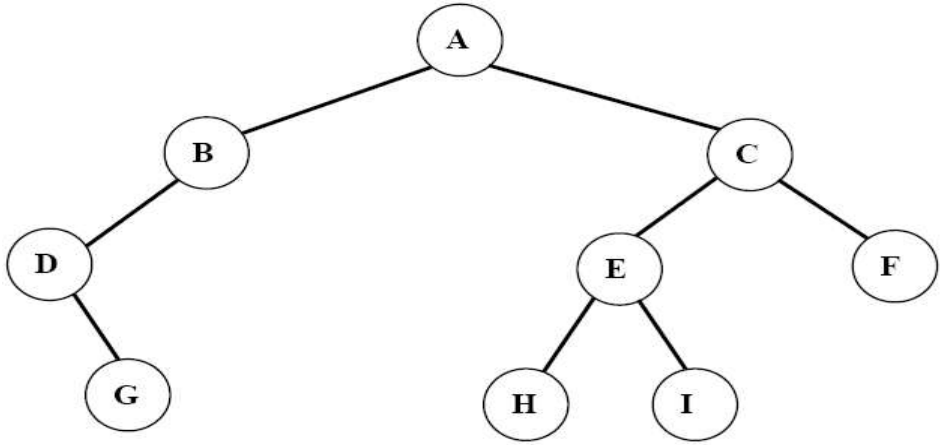
Running time: $O(\Sigma_{v \varepsilon T} (1 + c_v))$, where c_v is the number of children of v in T

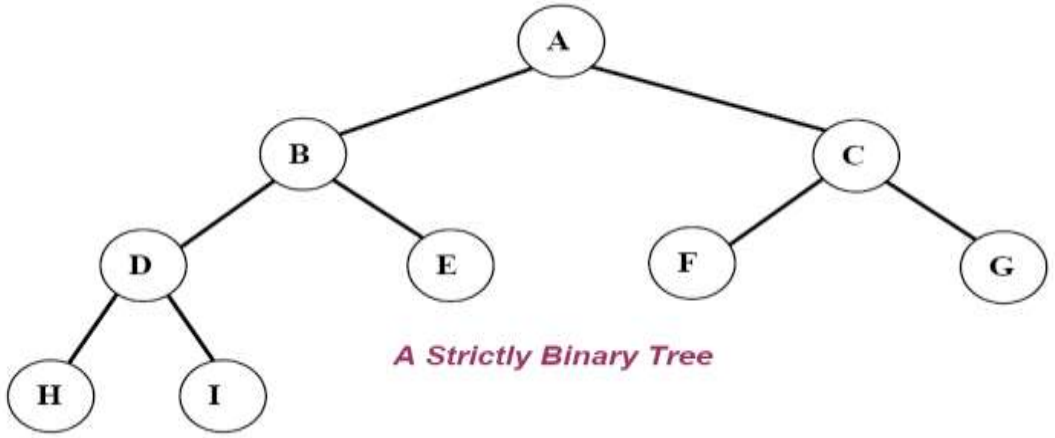
Since $\Sigma_{v \varepsilon T} c_v = n - 1$, we have $O(\Sigma_{v \varepsilon T} (1 + c_v)) = O(n)$.

- The height of tree T is obtained by calling height2(T, r).

Ordered Trees

- A tree is ordered if there is a linear ordering defined for each child of each node.
- A binary tree is an ordered tree in which every node has **at most** two children.
- If each node of a tree has either zero or two children, the tree is called a proper (strictly) binary tree.





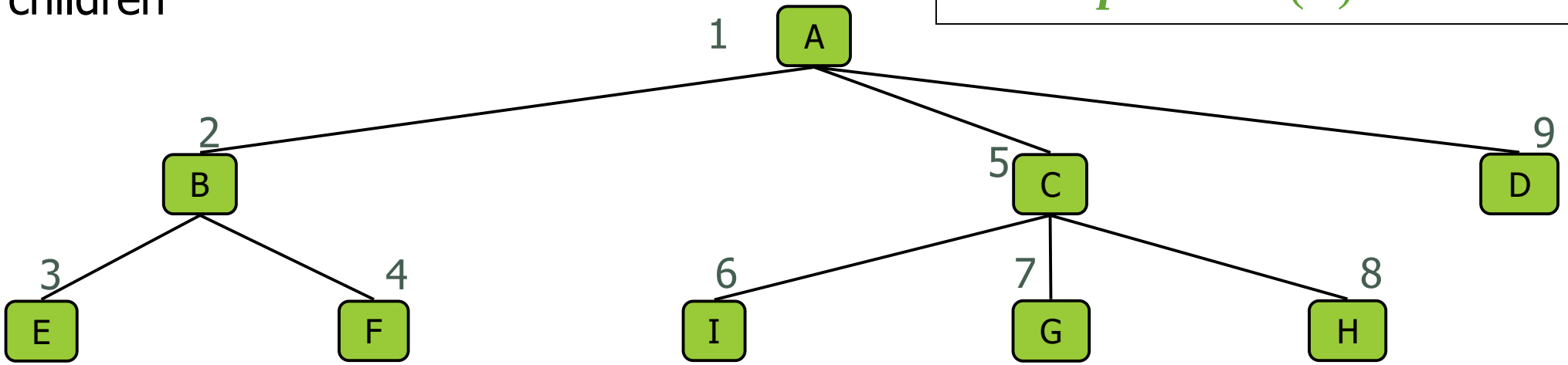
Traversal of Trees

- A traversal of a tree T is a systematic way of visiting all the nodes of T
- Traversing a tree involves visiting the root and traversing its subtrees
- There are the following traversal methods:
  - Preorder Traversal
  - Postorder Traversal
  - Inorder Traversal (of a binary tree)

Preorder Traversal

- In a preorder traversal, a node is visited before its descendants
- If a tree is ordered, then the subtrees are traversed according to the order of the children

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)

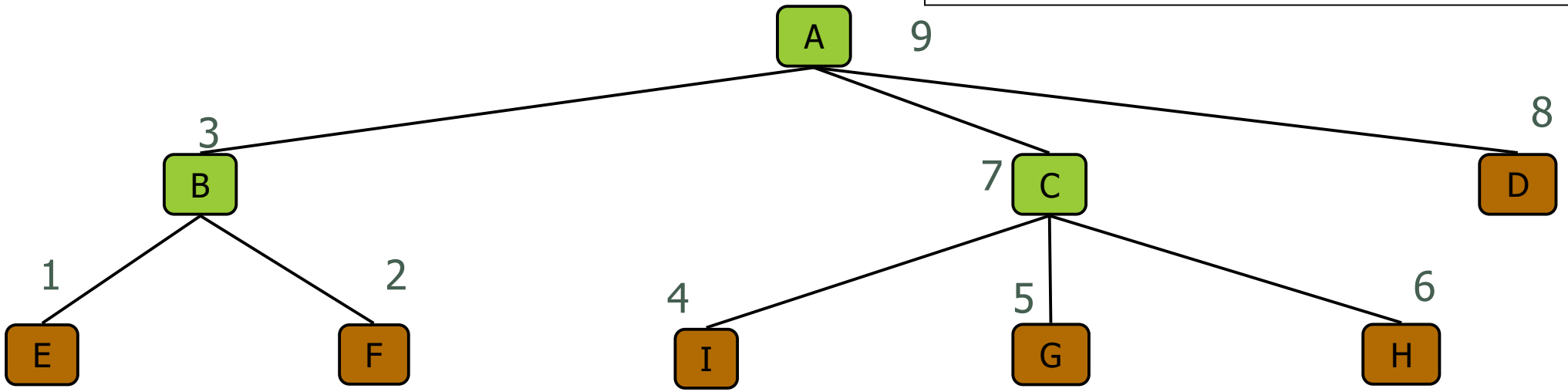


Preorder: *ABEFCIGHD*

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants

Algorithm *postOrder*(*v*)
  for each child *w* of *v*
    postOrder (*w*)
  visit(*v*)

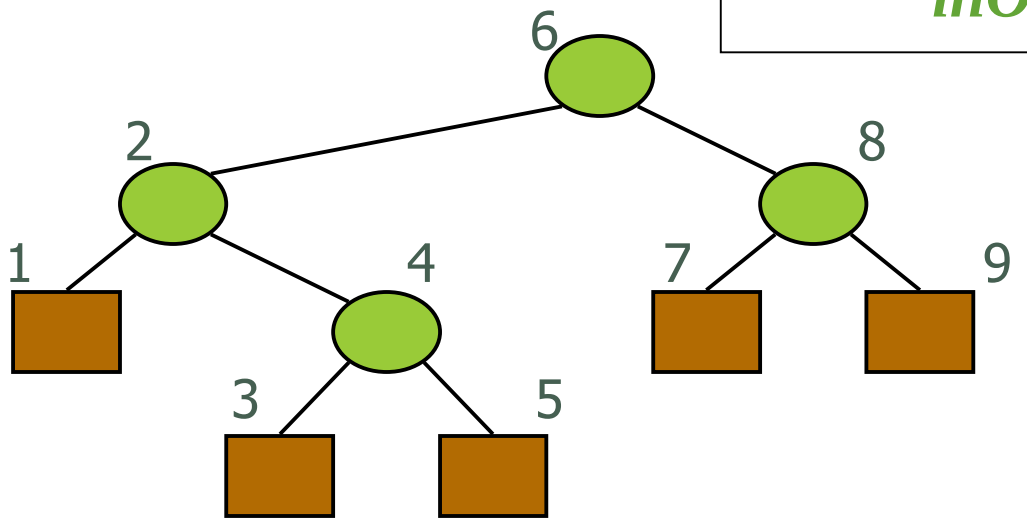


Postorder: *EFBIGHCDA*

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

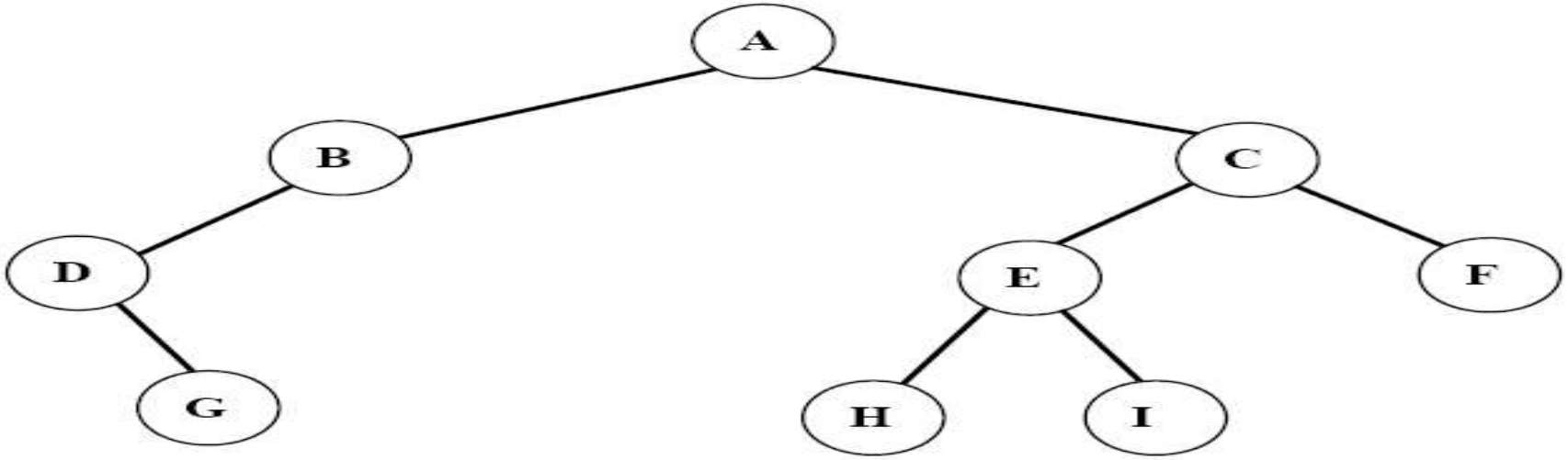
```
Algorithm inOrder(v)
    if isInternal (v)
        inOrder (leftChild (v))
    visit(v)
    if isInternal (v)
        inOrder (rightChild (v))
```



Inorder Traversal

Traversing a binary tree in *inorder*

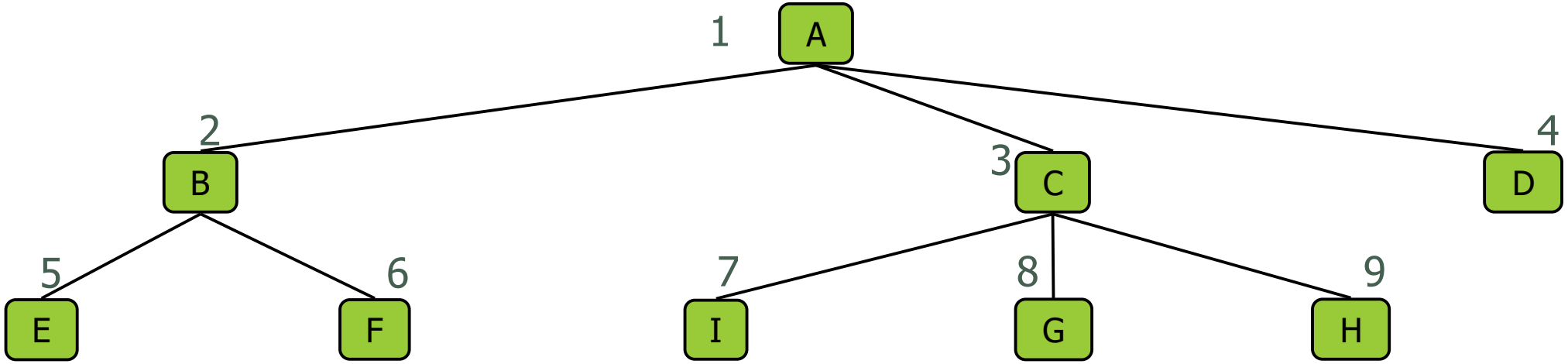
1. Traverse the ***left subtree*** in inorder.
2. Visit the ***root***.
3. Traverse the ***right subtree*** in inorder.



Inorder*: DGBAHEICF*

Level Order Traversal

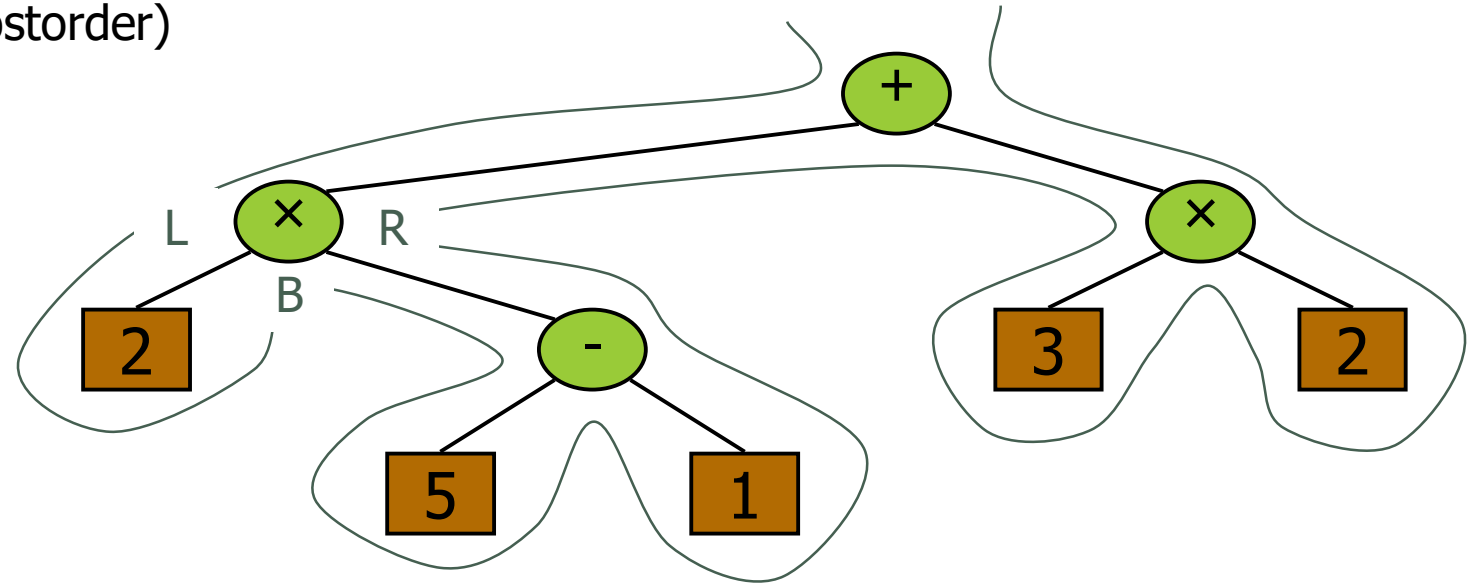
- In a level order traversal, every node on a level is visited before going to a lower level



Level order*: ABCDEFIGH*

Euler Tour Traversal

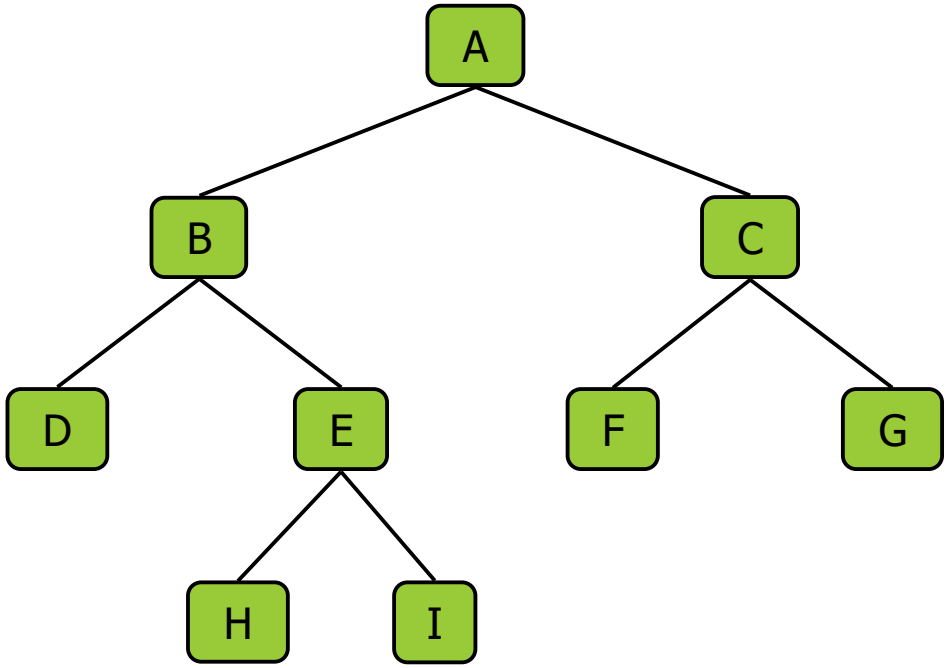
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



(Proper) Binary Tree

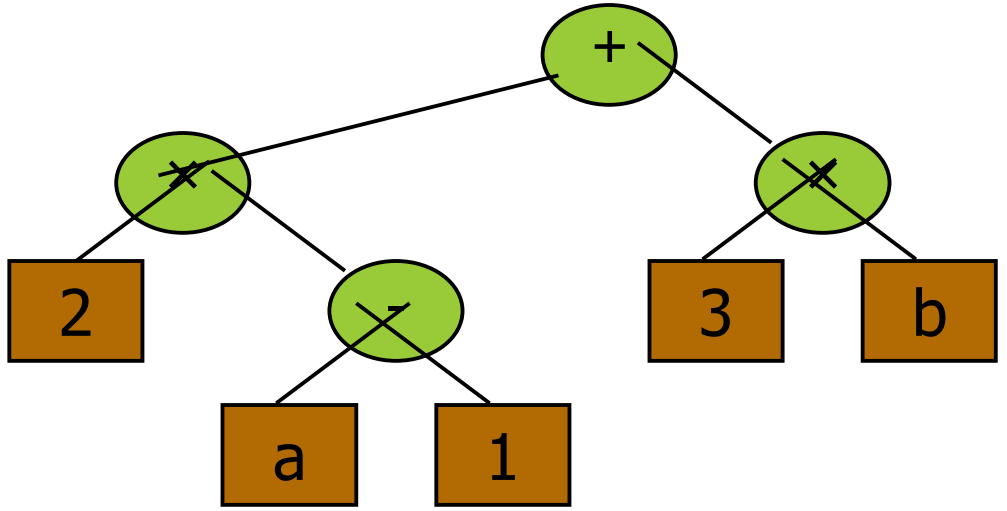
- A (proper) binary tree is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a (proper) binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a proper binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



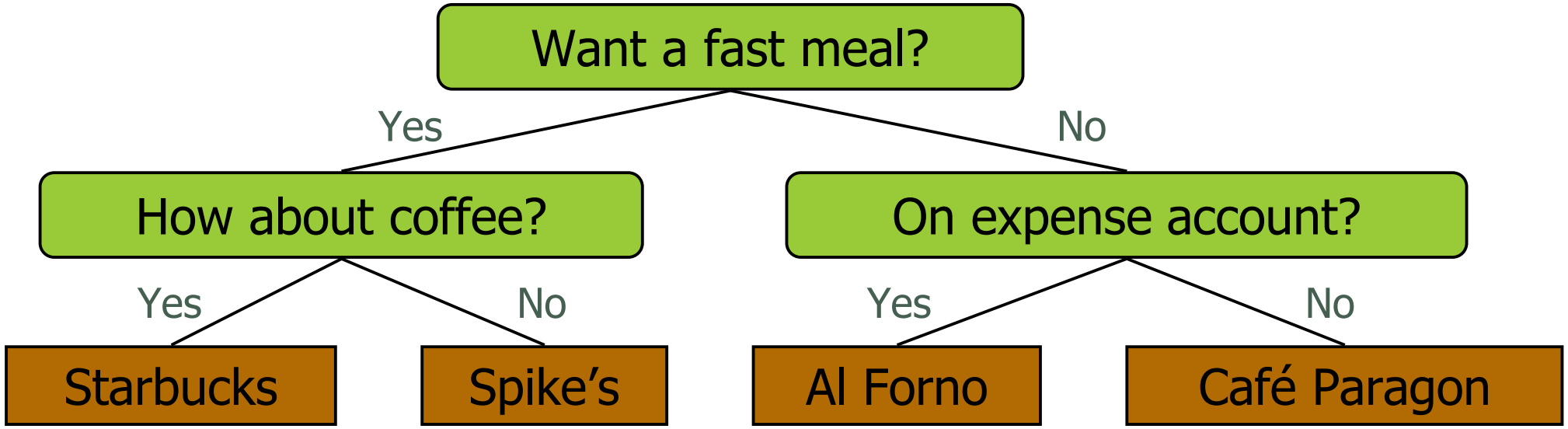
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression (2 × (a - 1) + (3 × b))



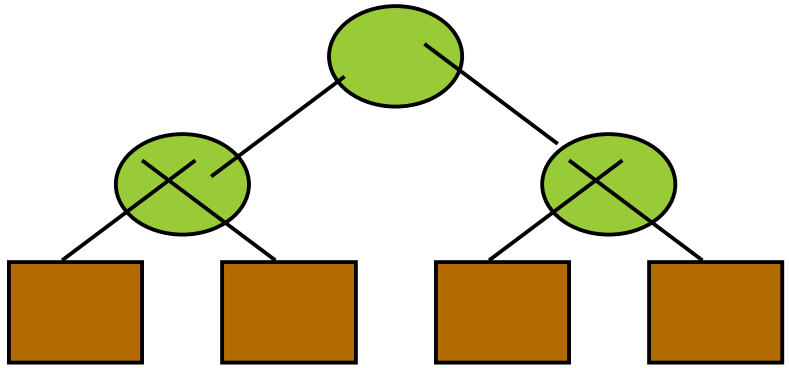
Decision Tree

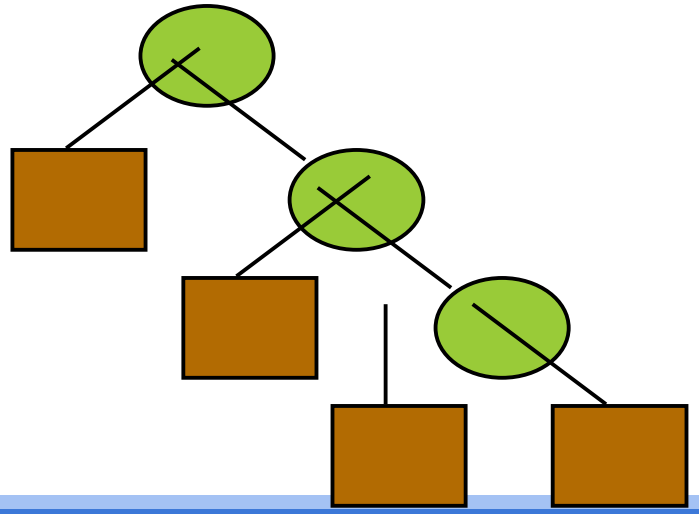
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Binary Trees

- Let T be a (proper) binary tree with n nodes, and let h denote the height of T. Then T has the following properties.
  - The number of external (leaf) nodes in T is at least h+1 and at most 2^h.
  - The number of internal nodes in T is at least h and at most 2^h-1.
  - The number of nodes in T is at least 2h+1 and at most $2^{h+1}-1$.
  - The height h of T satisfies $\log(n+1)-1 \leq h \leq (n-1)/2$.
- Proof:

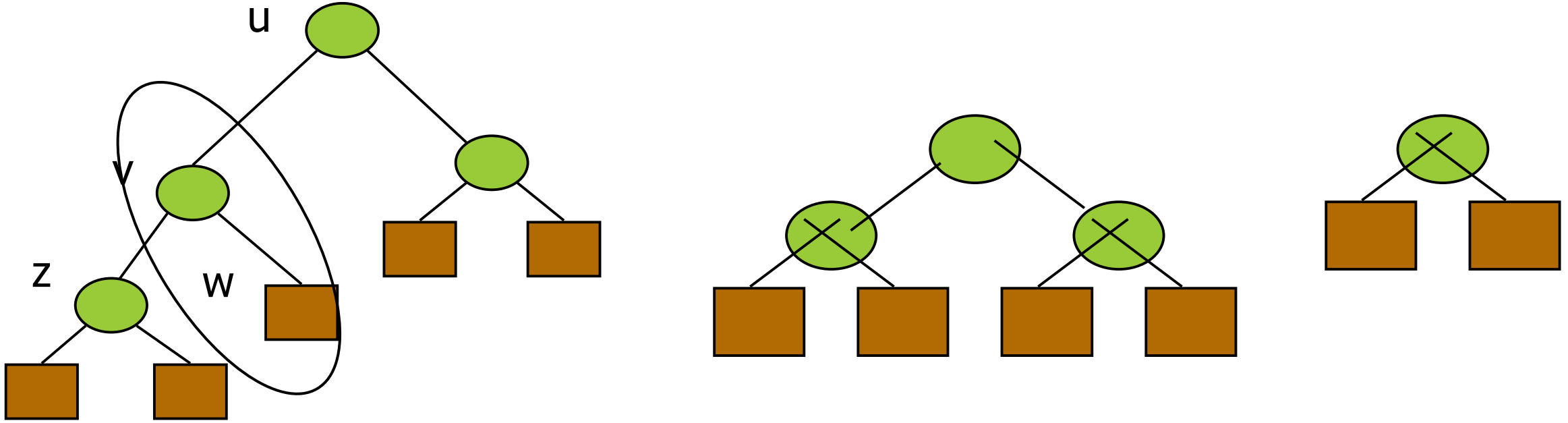




Properties of Binary Trees

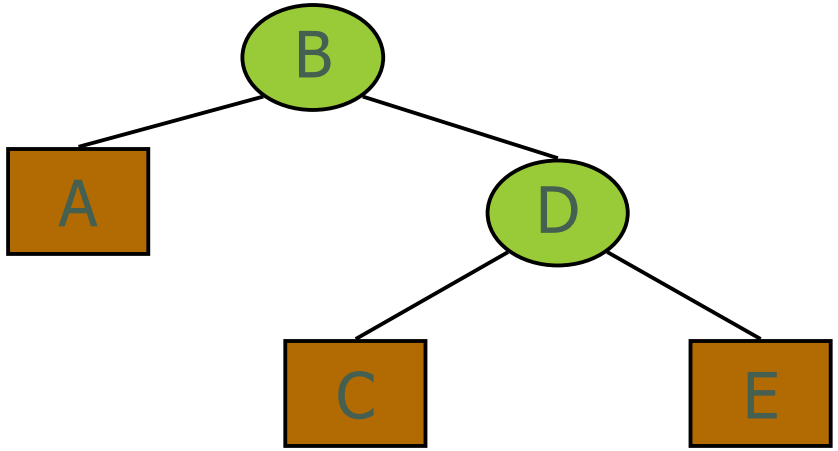
- In a (proper) binary tree T, the number of external nodes is 1 more than the number of internal nodes.

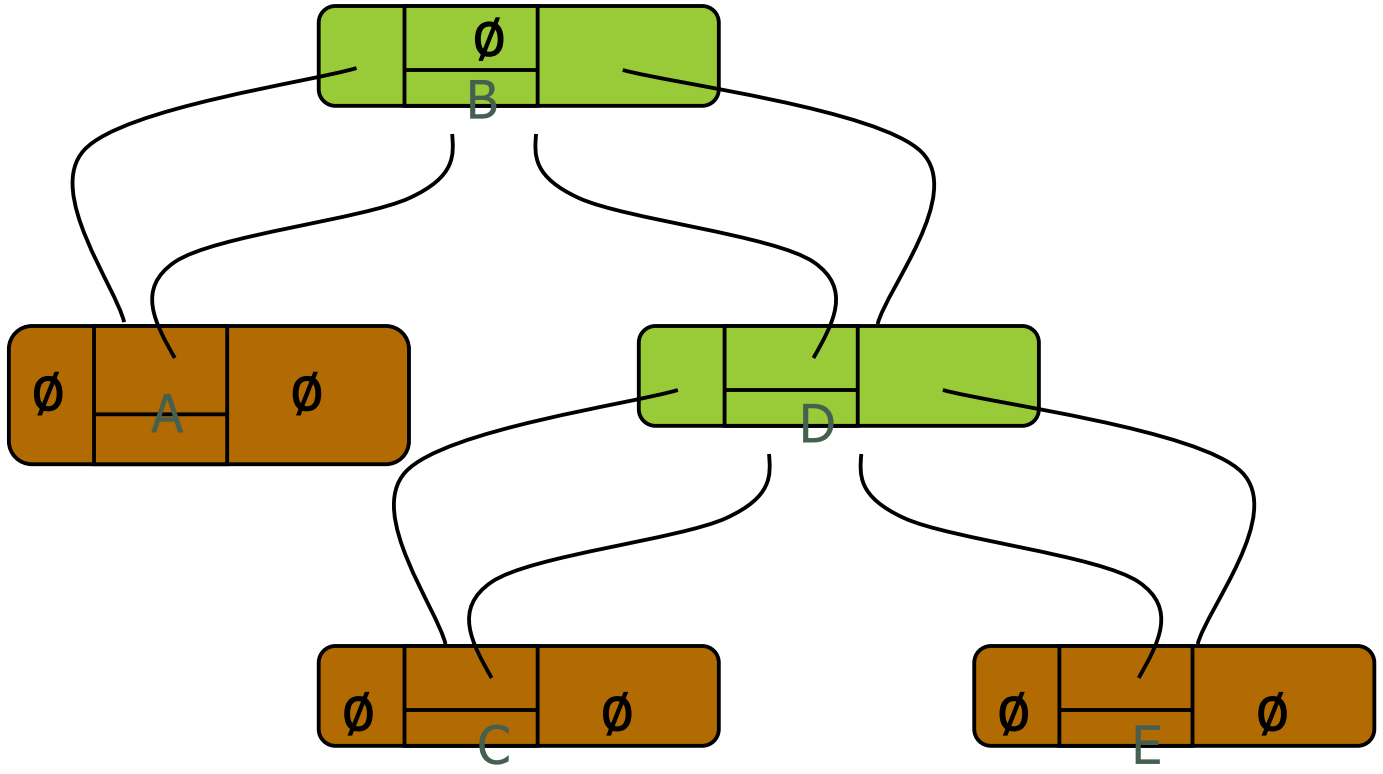
- Proof:



Linked Structure for Binary Trees

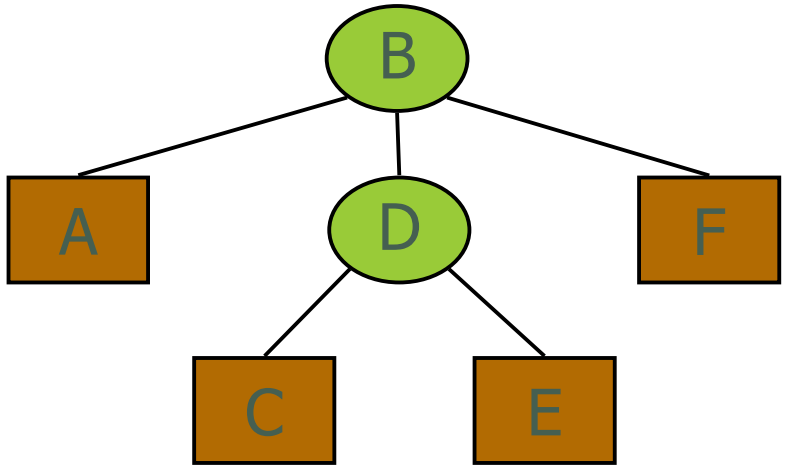
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node

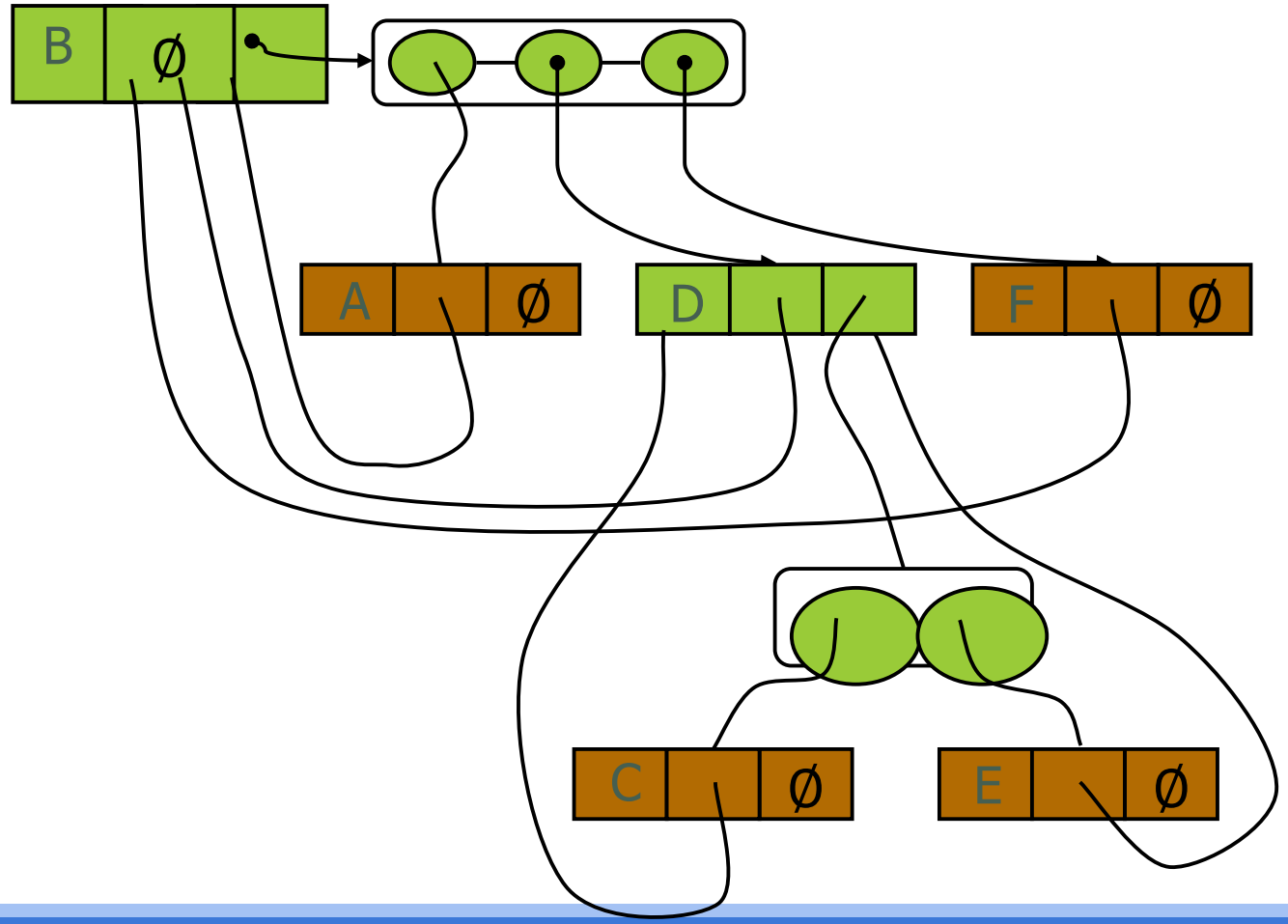




Linked Structure for General Trees

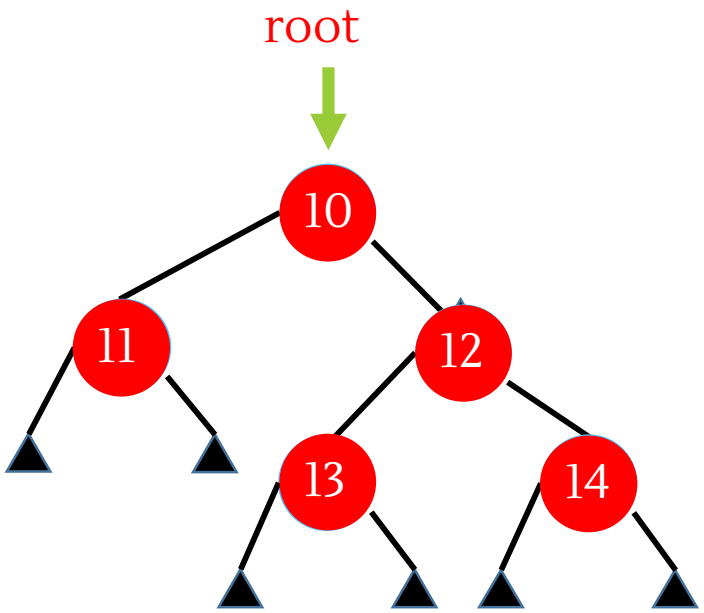
- A node is represented by an object storing
  - Element
  - Parent node
  - Children Container: Sequence of children nodes

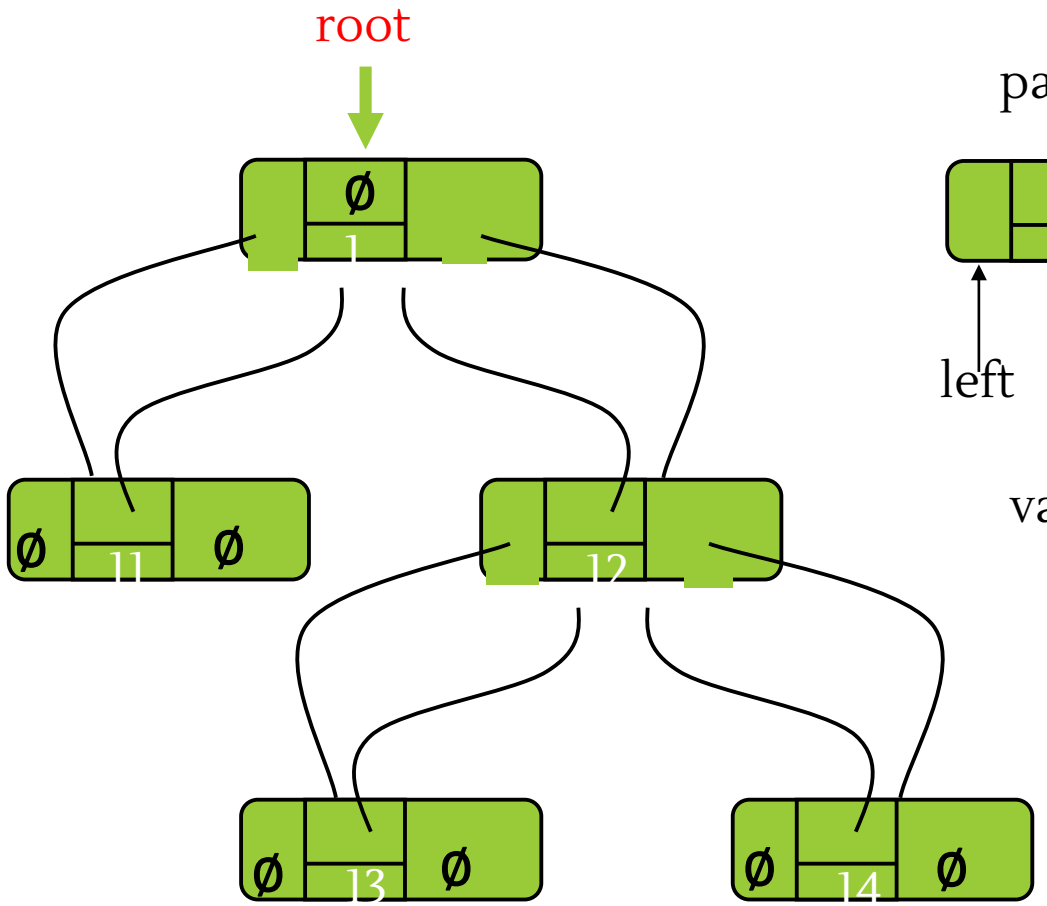


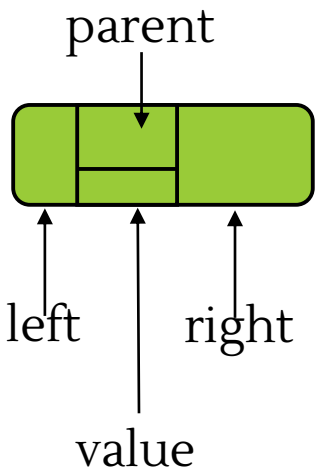


Tree Representation

Representation of Binary Tree








```
struct  Node {
    int value;
    Node *left, *right, *parent;
}
```

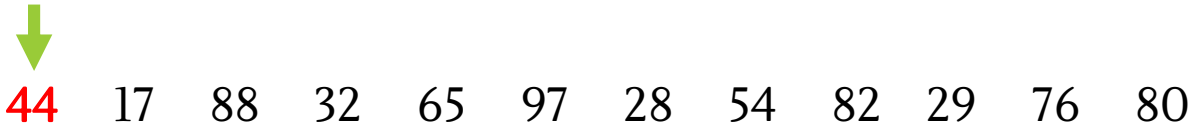
Binary Search Tree

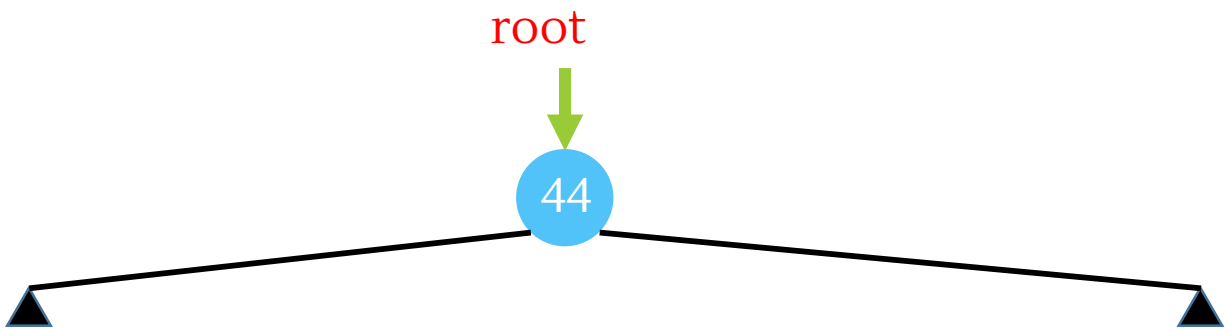
Characteristic

Left < Parent <= Right

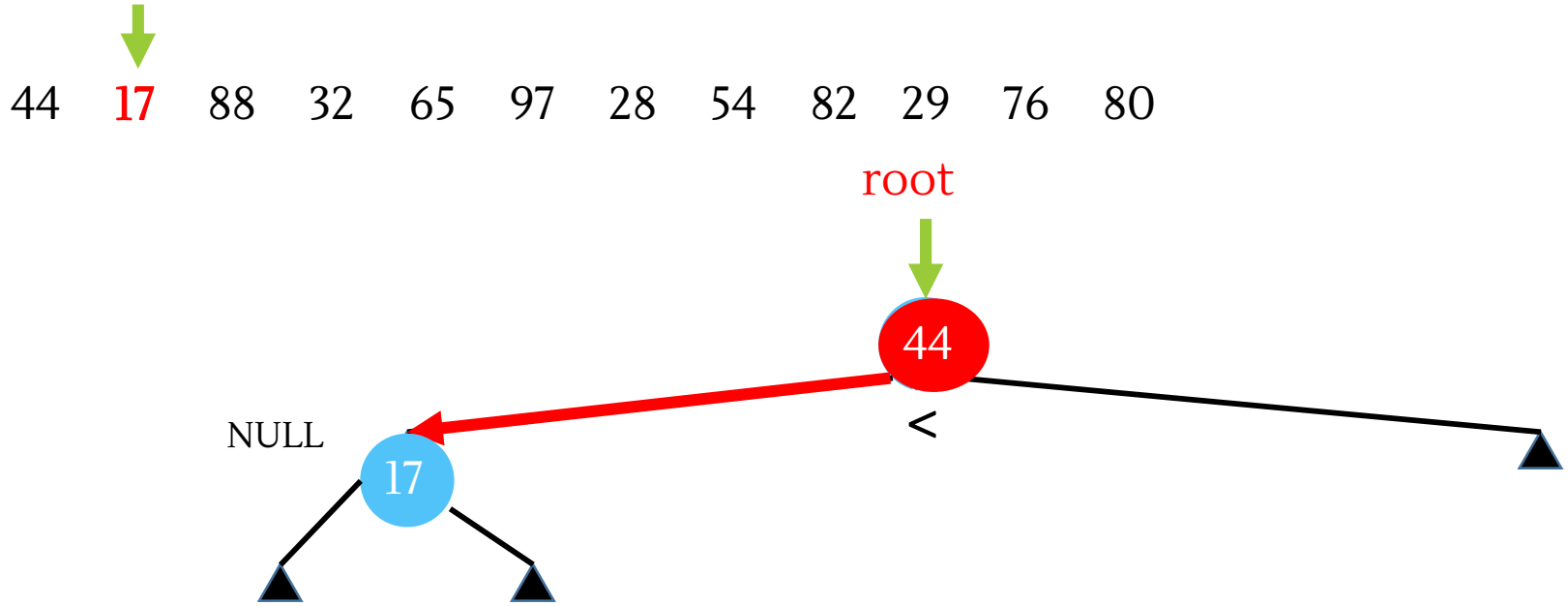
Each sub tree is a Binary Search Tree

Insertion

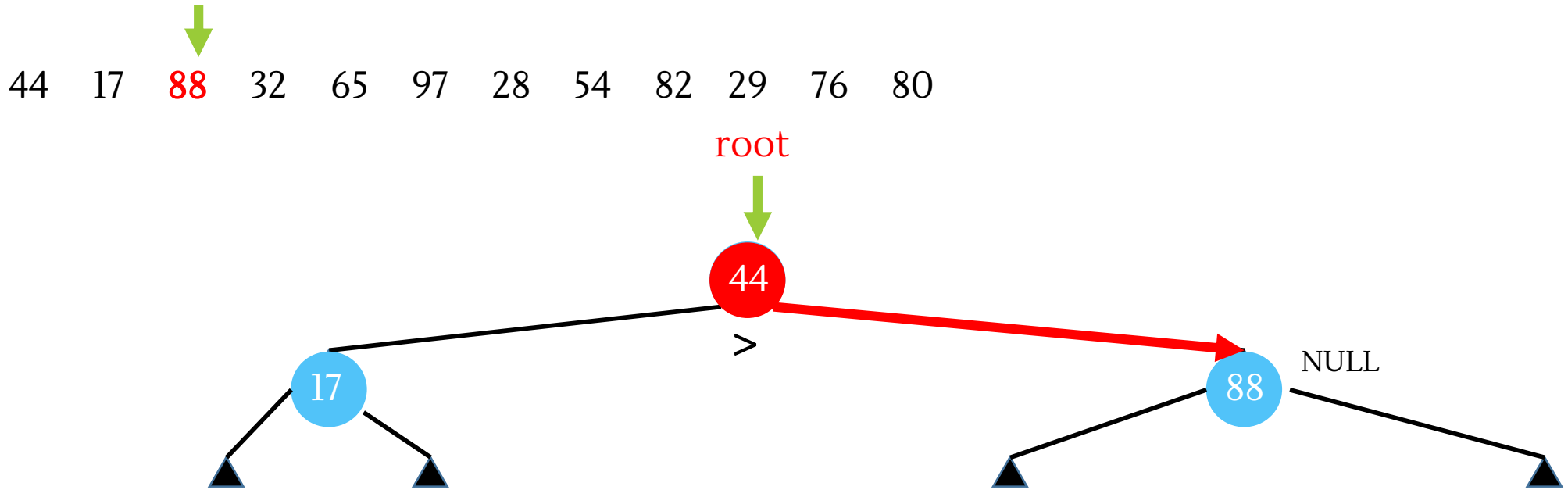




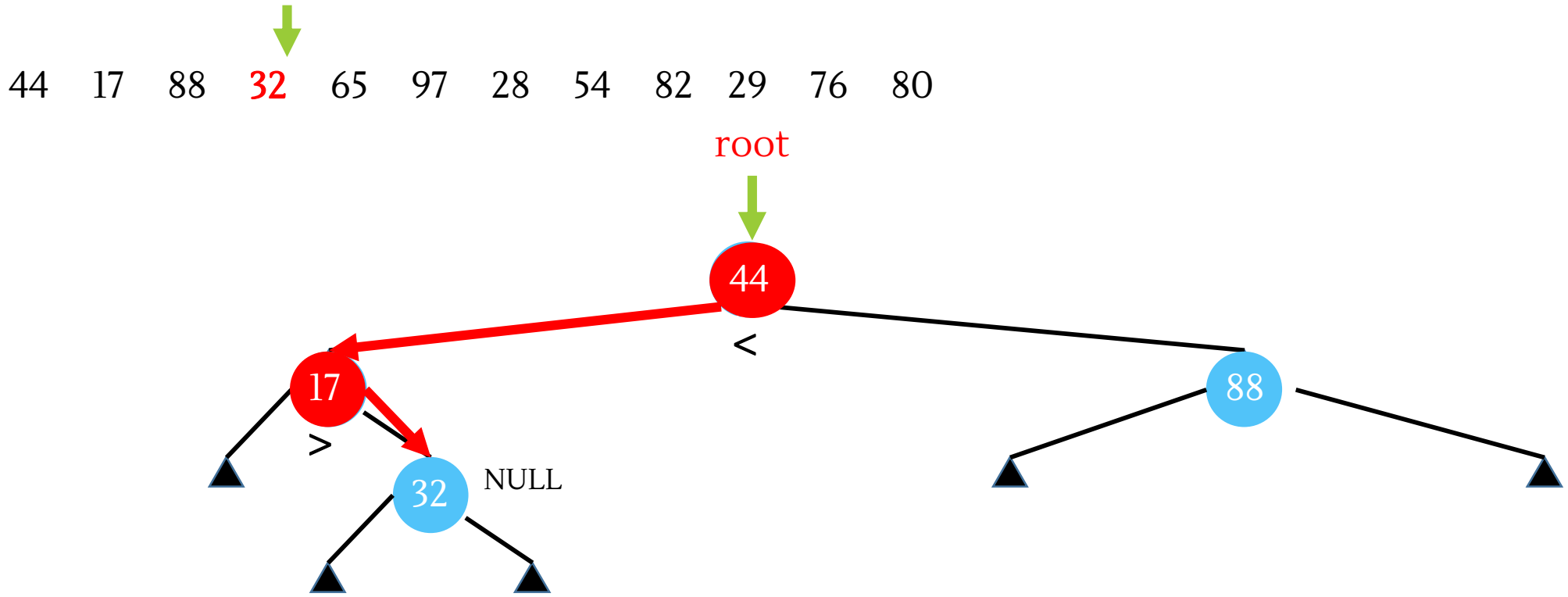
Insertion



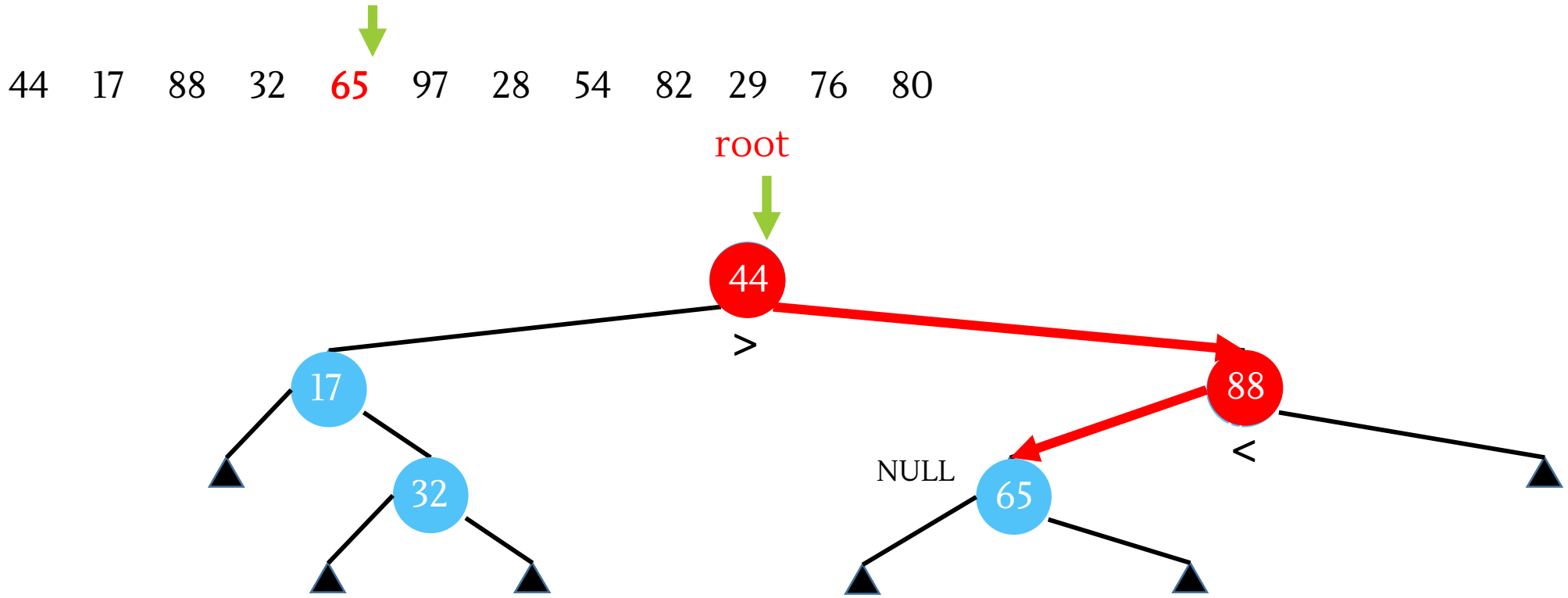
Insertion



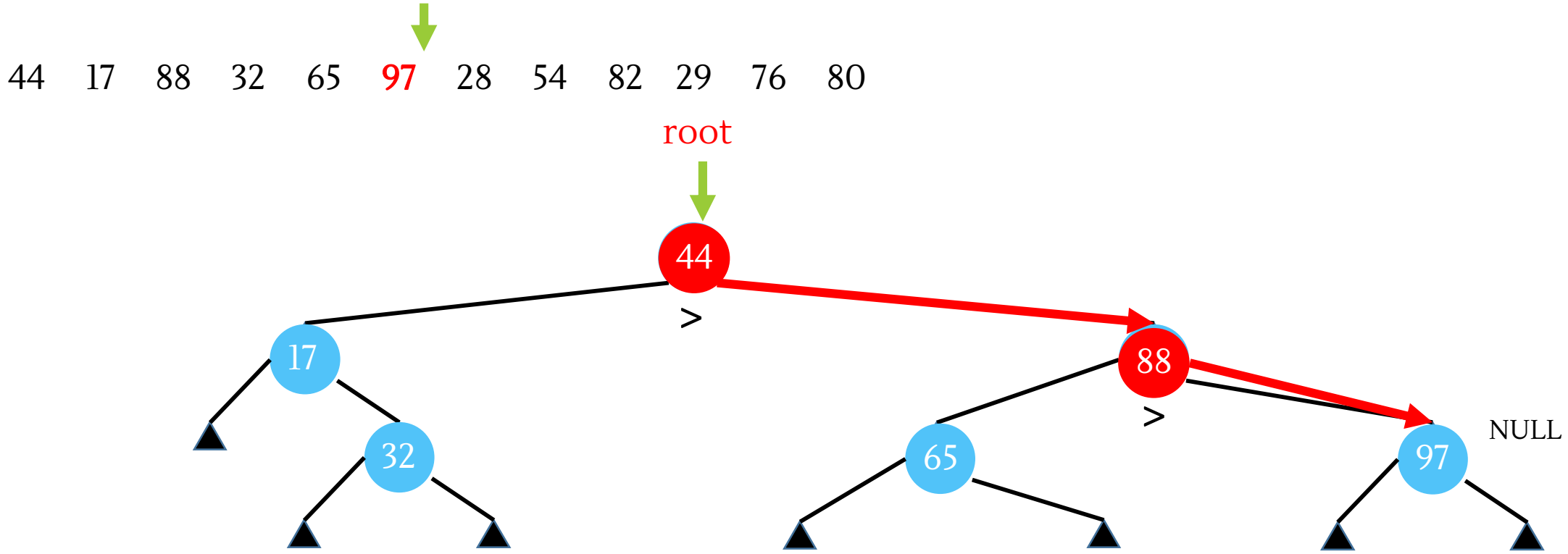
Insertion



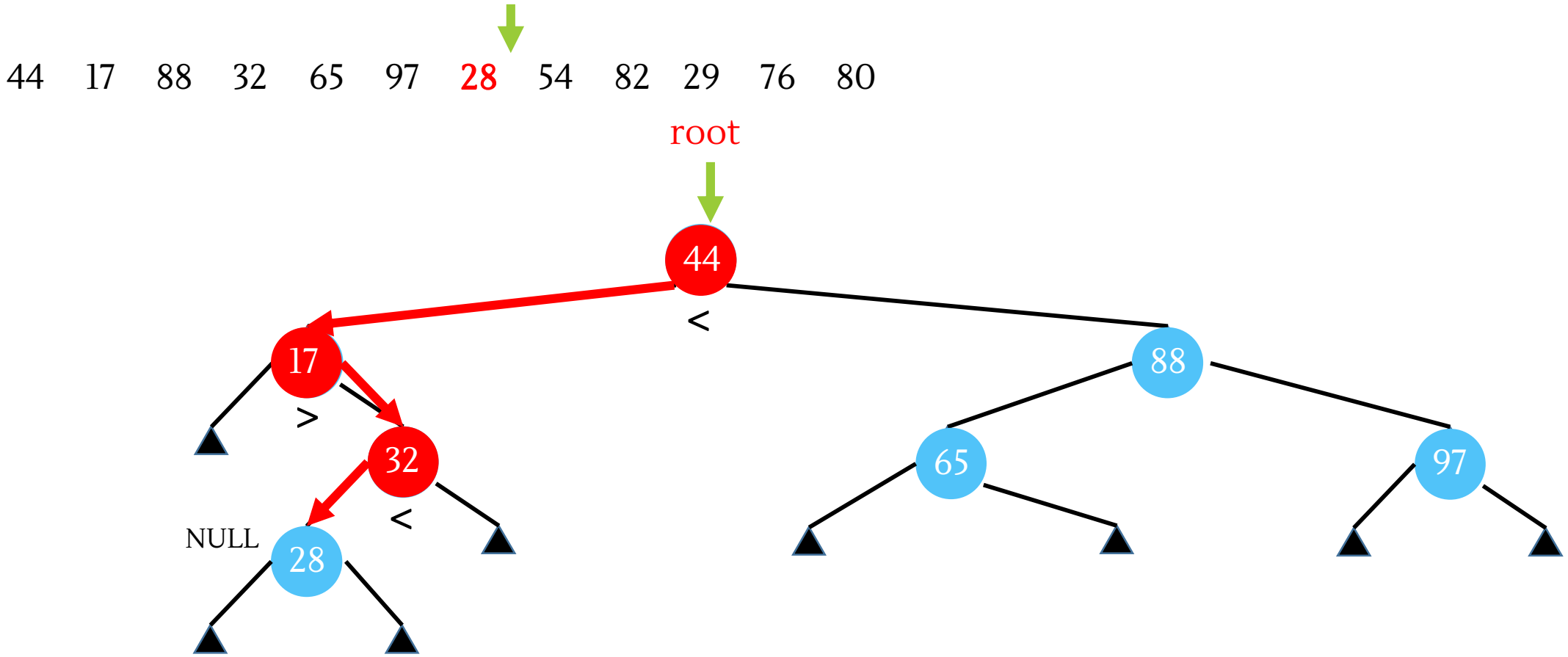
Insertion



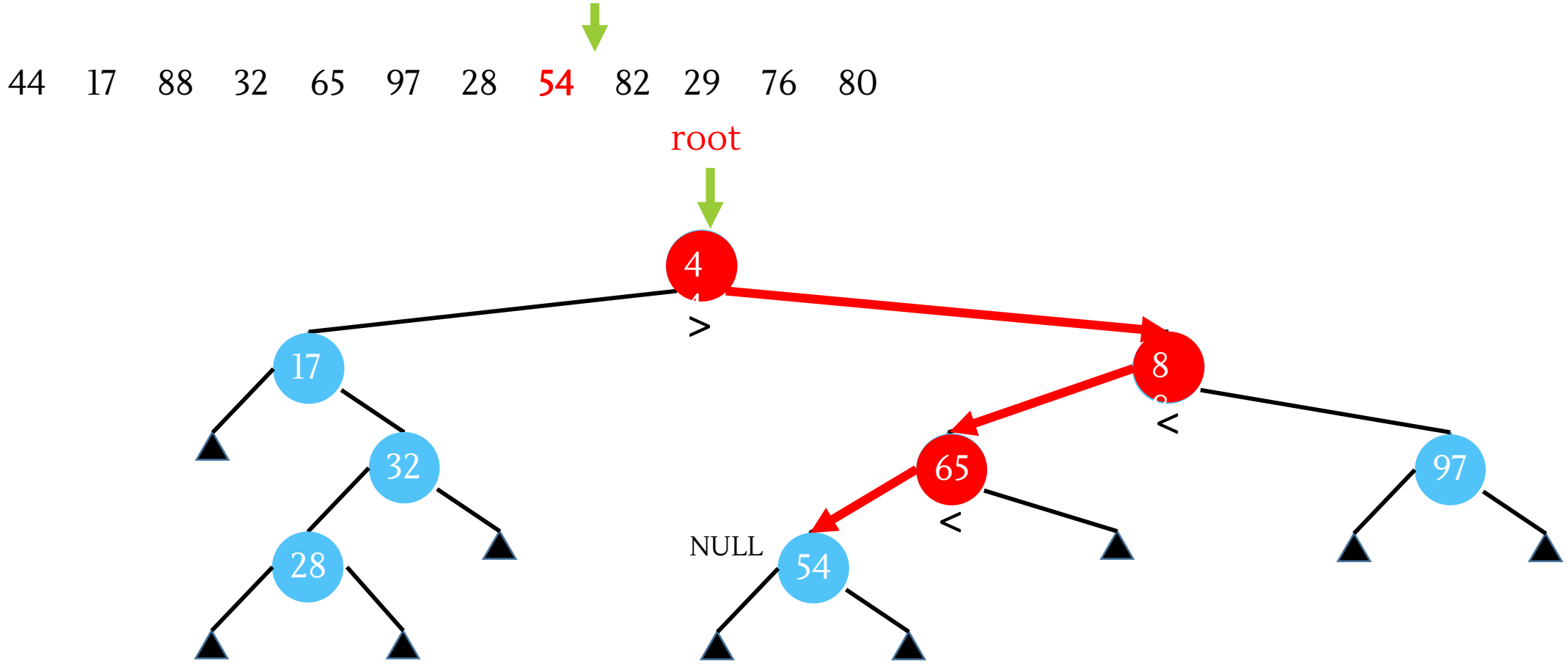
Insertion



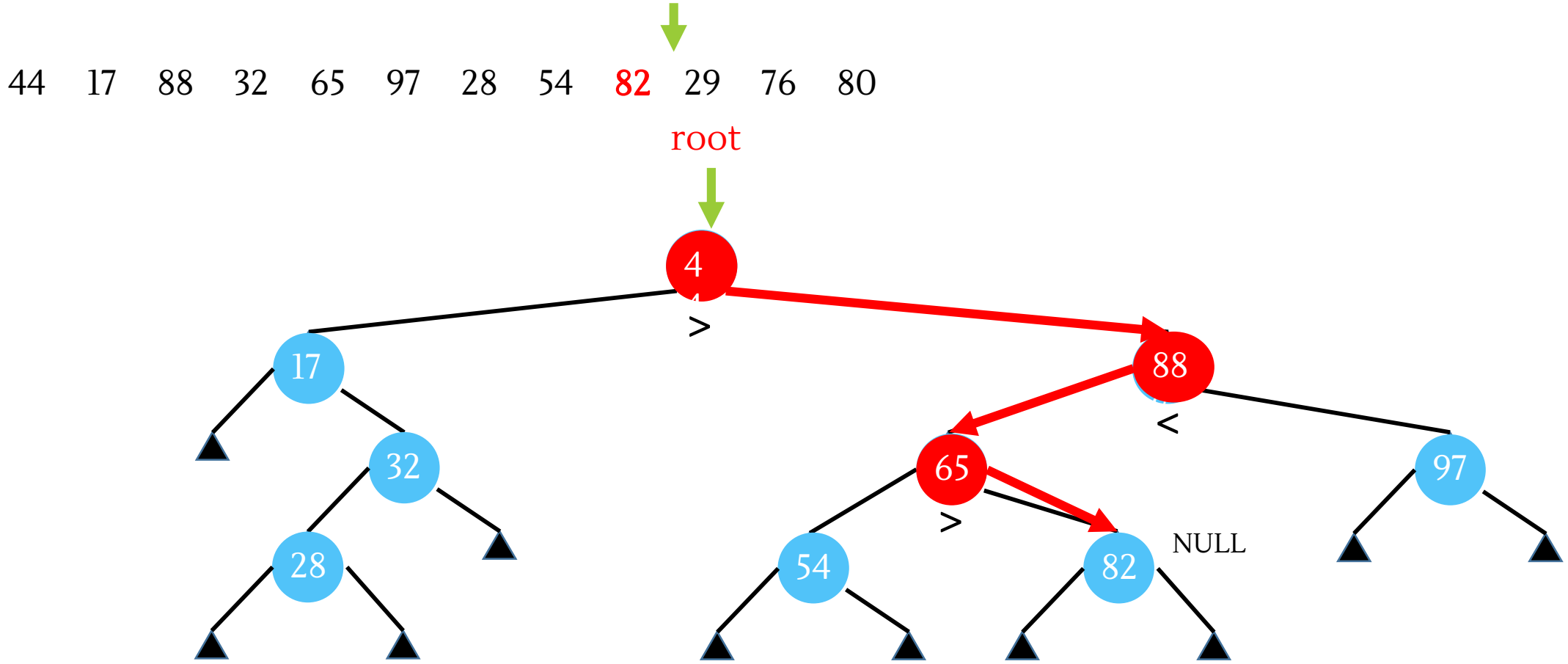
Insertion



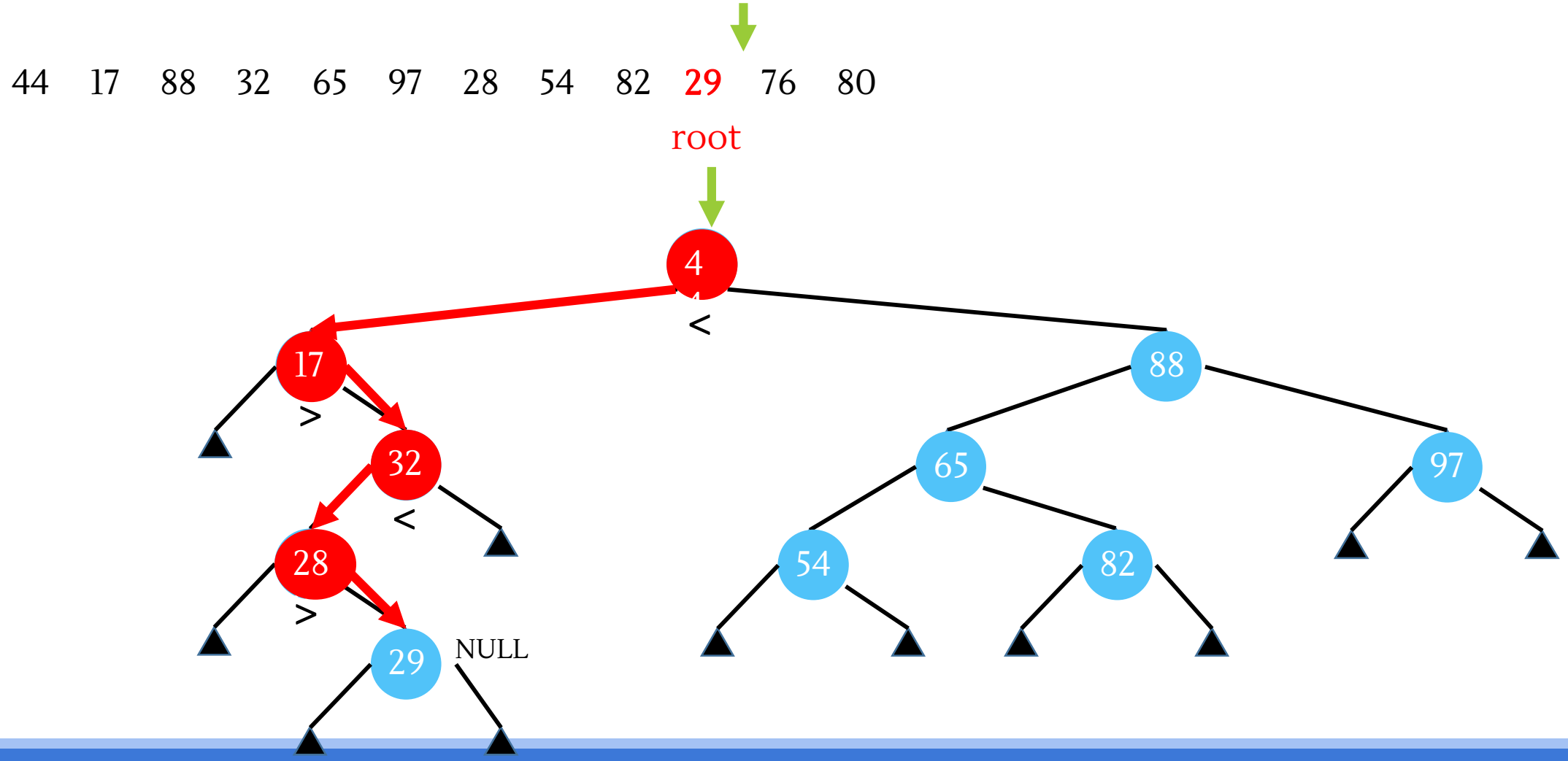
Insertion



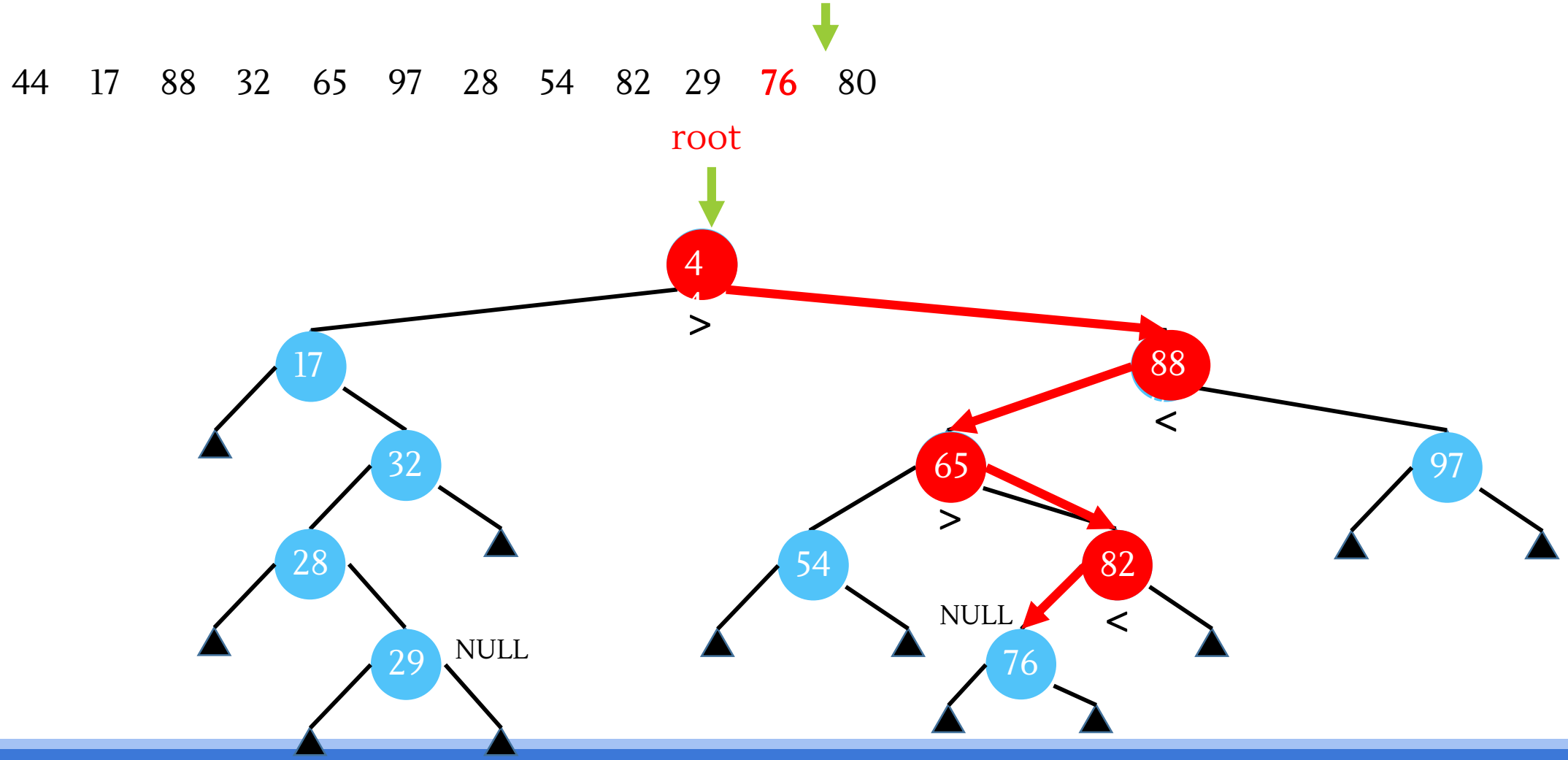
Insertion



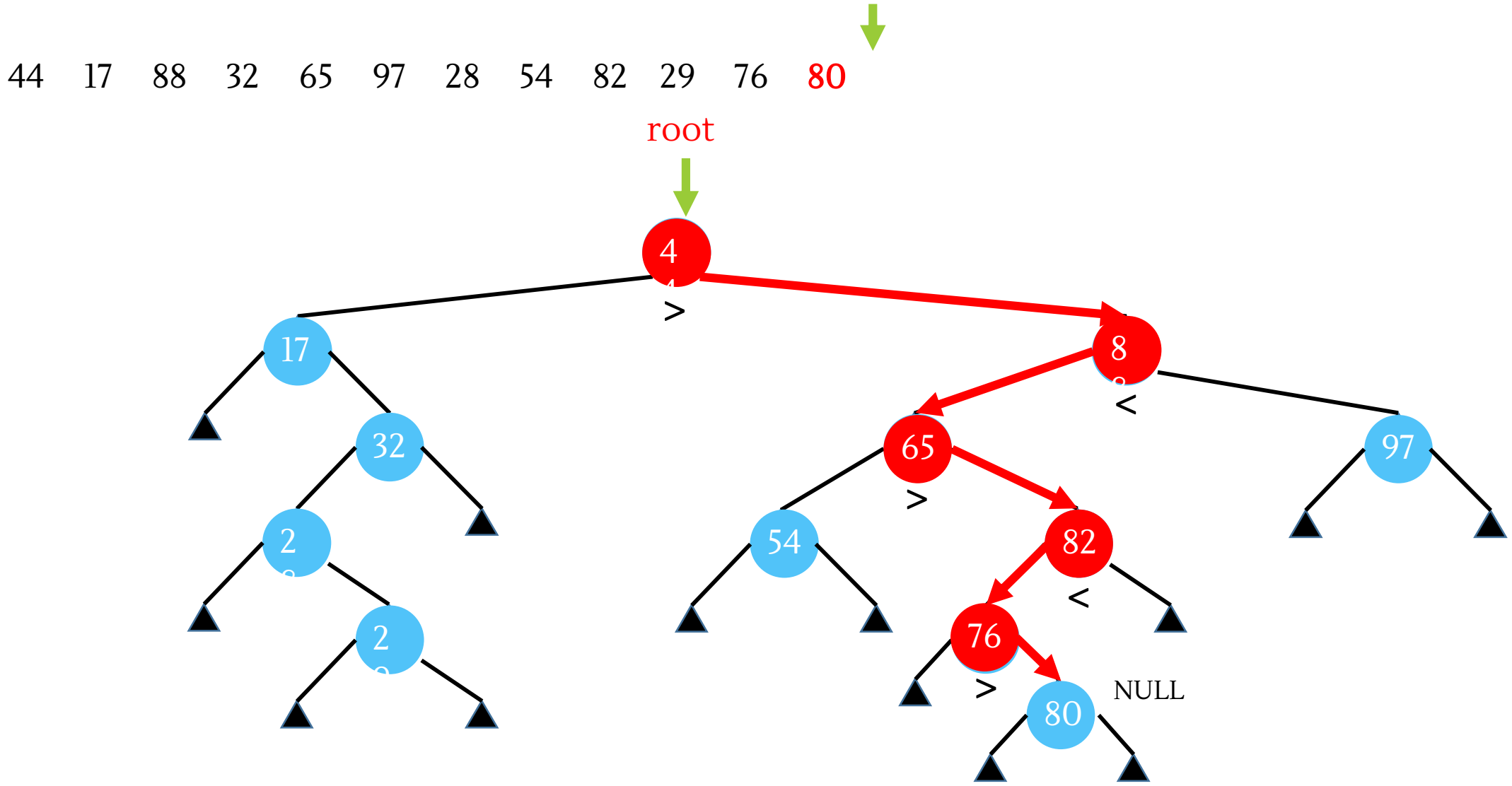
Insertion


44 17 88 32 65 97 28 54 82 29 76 80
root
4
<
17
>
88
32
<
65
97
28
>
54
82
29
NULL

Insertion


44 17 88 32 65 97 28 54 82 29 76 80
root
4
<
>
17
88
<
32
65
97
>
28
54
82
NULL
<
29
NULL
76

Insertion


44 17 88 32 65 97 28 54 82 29 76 80
root
4
>
17
8
<
32
65
97
>
2
54
82
<
2
76
>
NULL
80

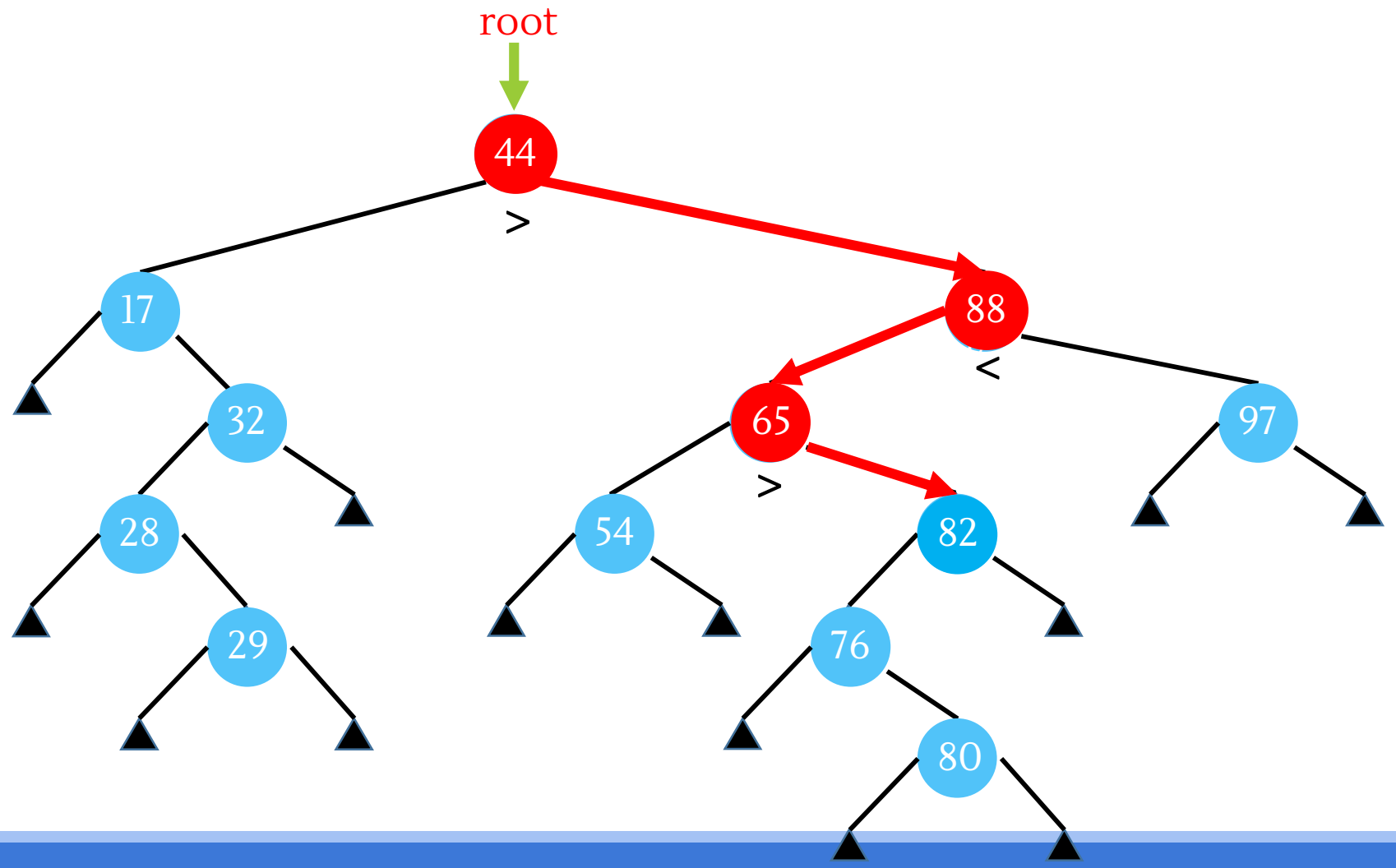
Insertion Sequence

- What if a sorted input sequence is inserted sequentially in a BST?
  - The Binary Search Tree will turn into a linear list
  - Height of the BST will be n [n=no of elements]
  - Searching complexity will be O(n)
  - This is the worst case scenario for a BST

- Insert the following sorted sequence into a BST and prove the statements stated above.
  - 1 4 9 17 20 28 35

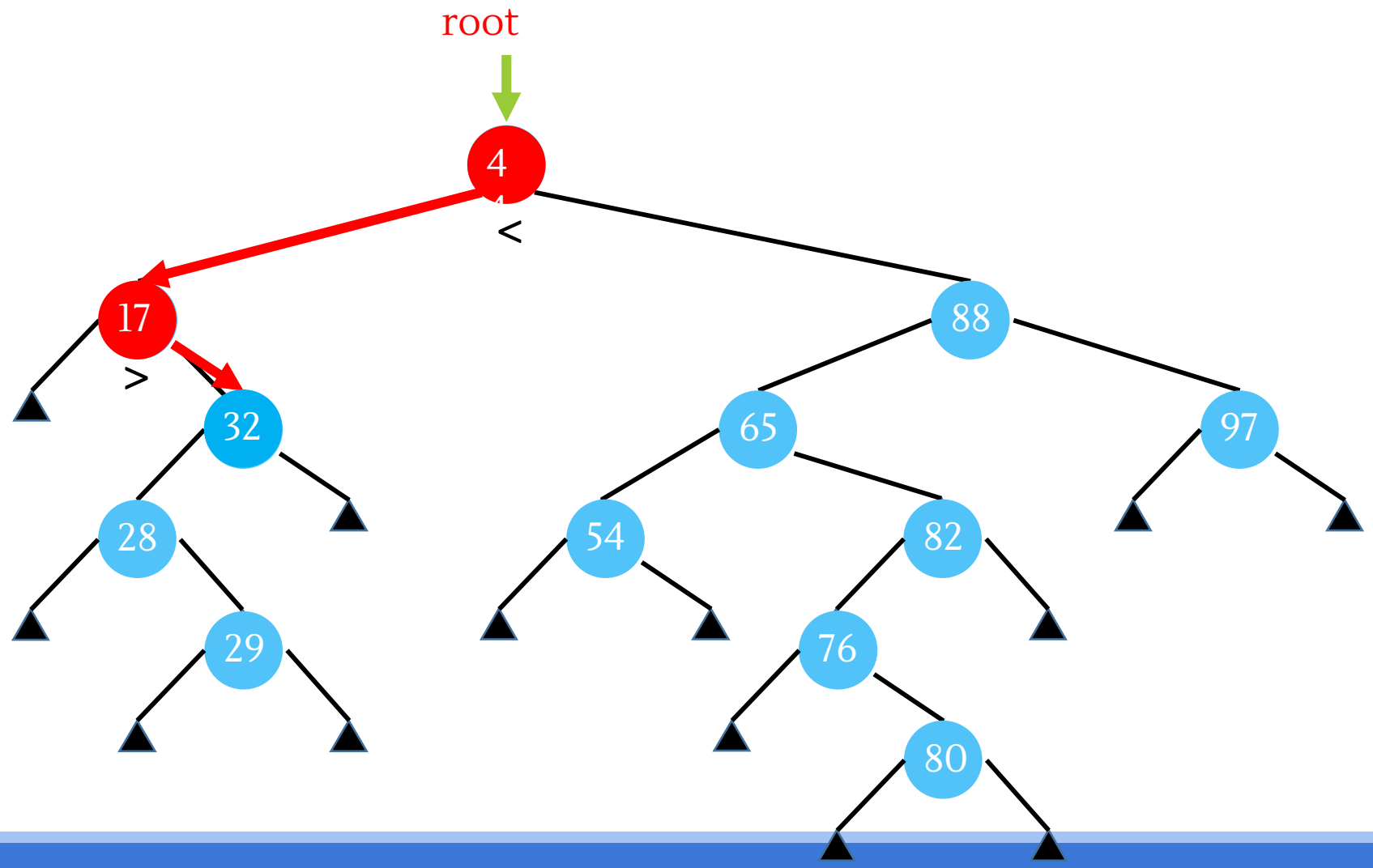
Searching in BST

Search:
82



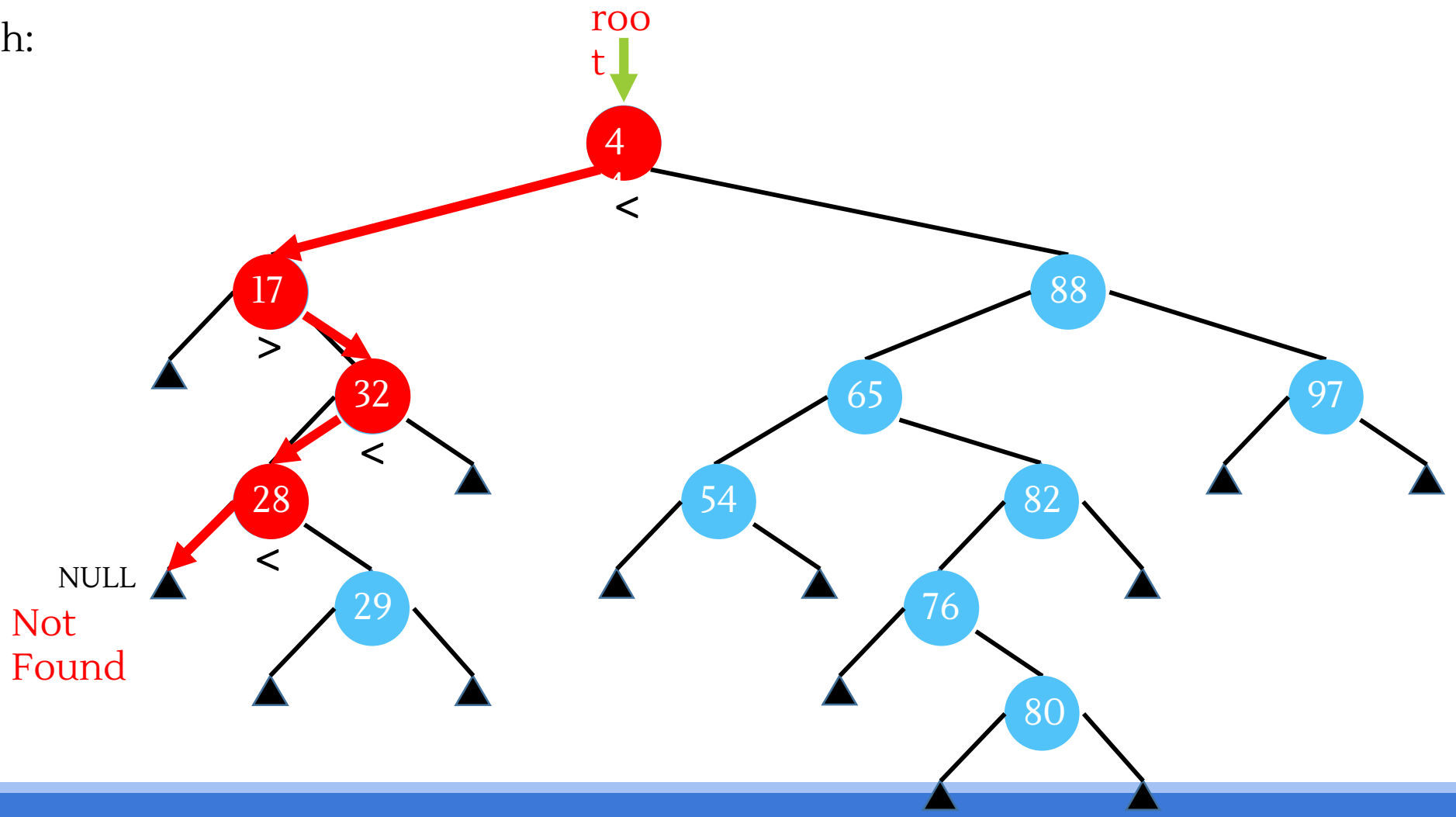
Searching in BST

Search:
32



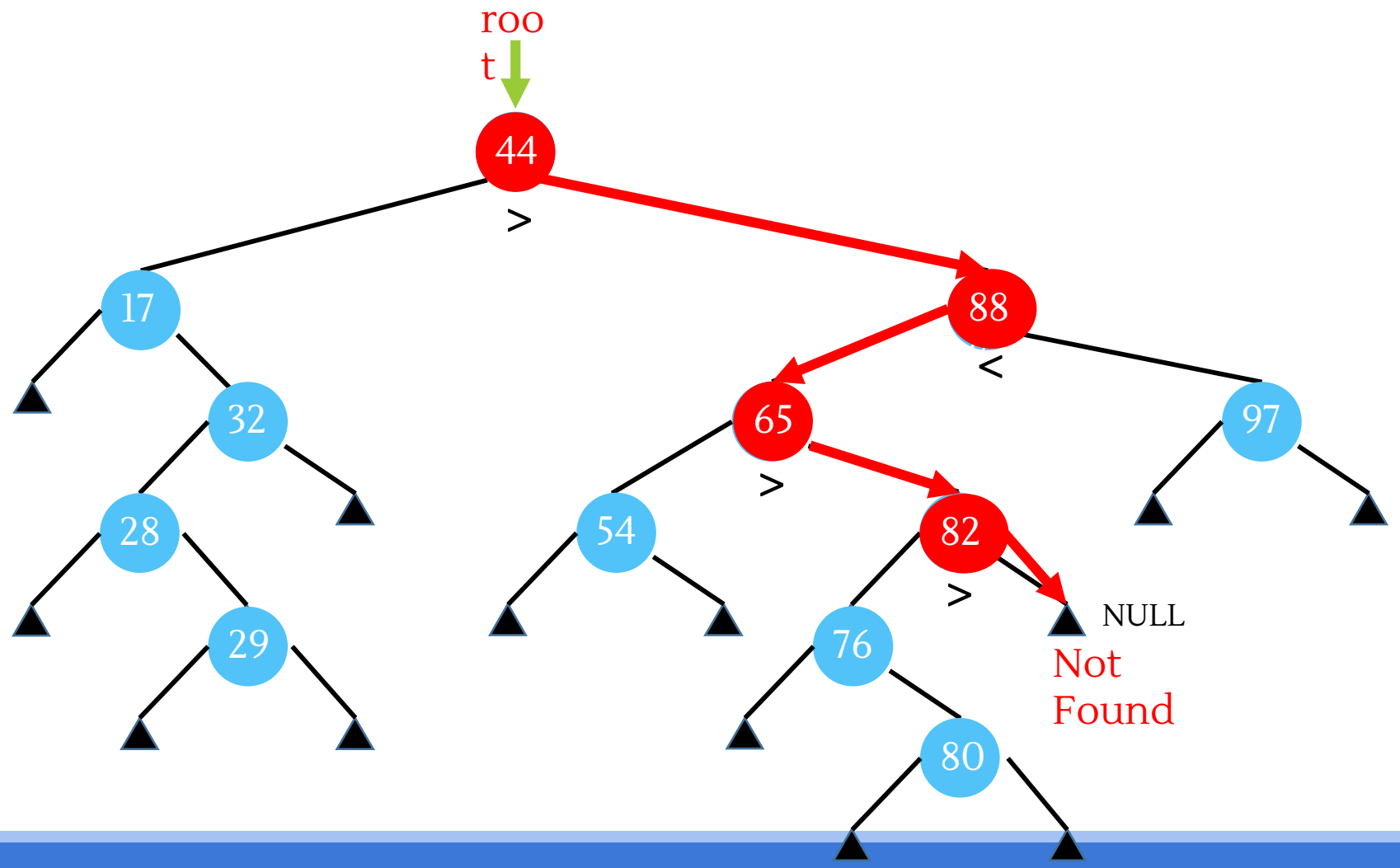
Searching in BST

Search:
27

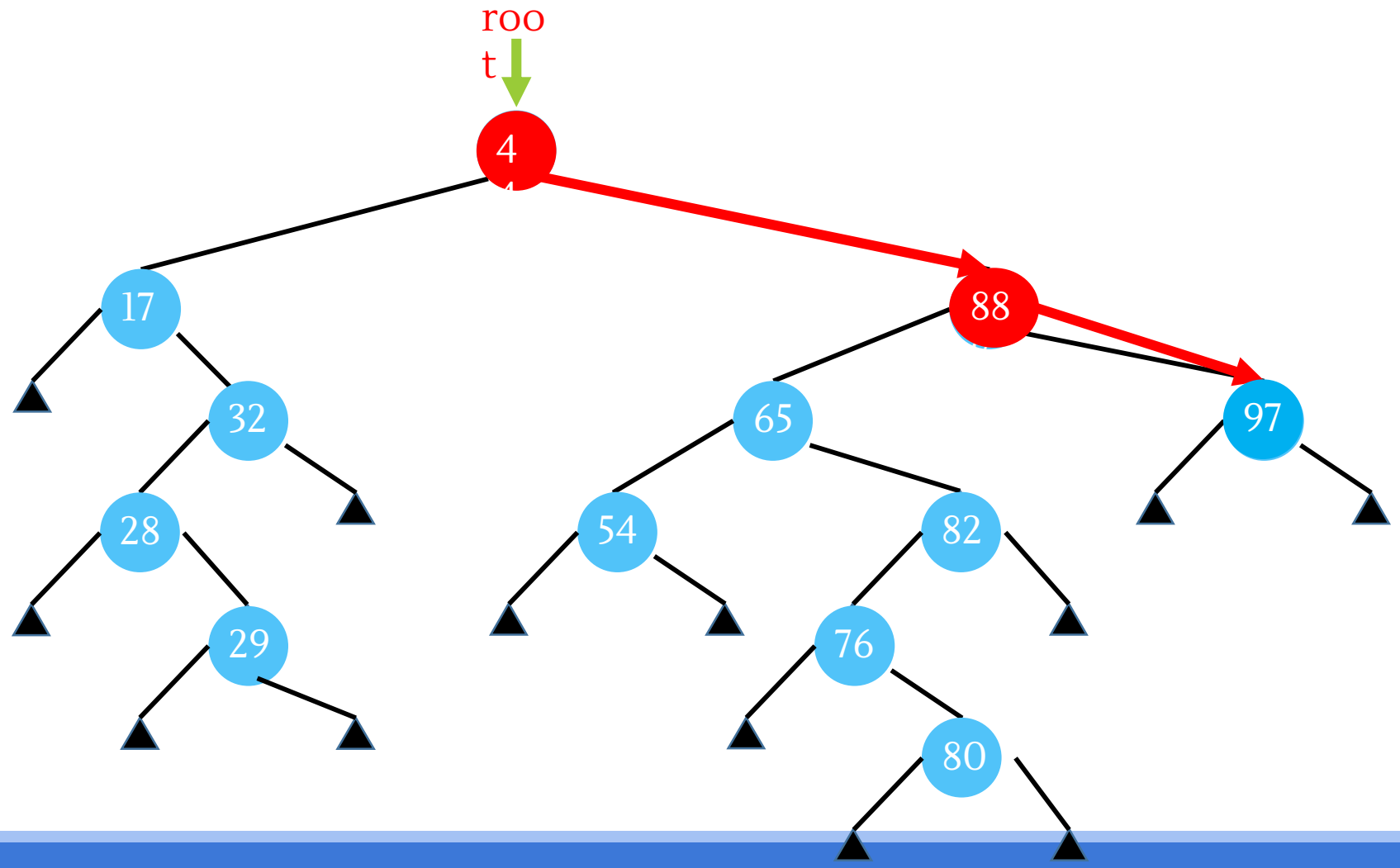


Searching in BST

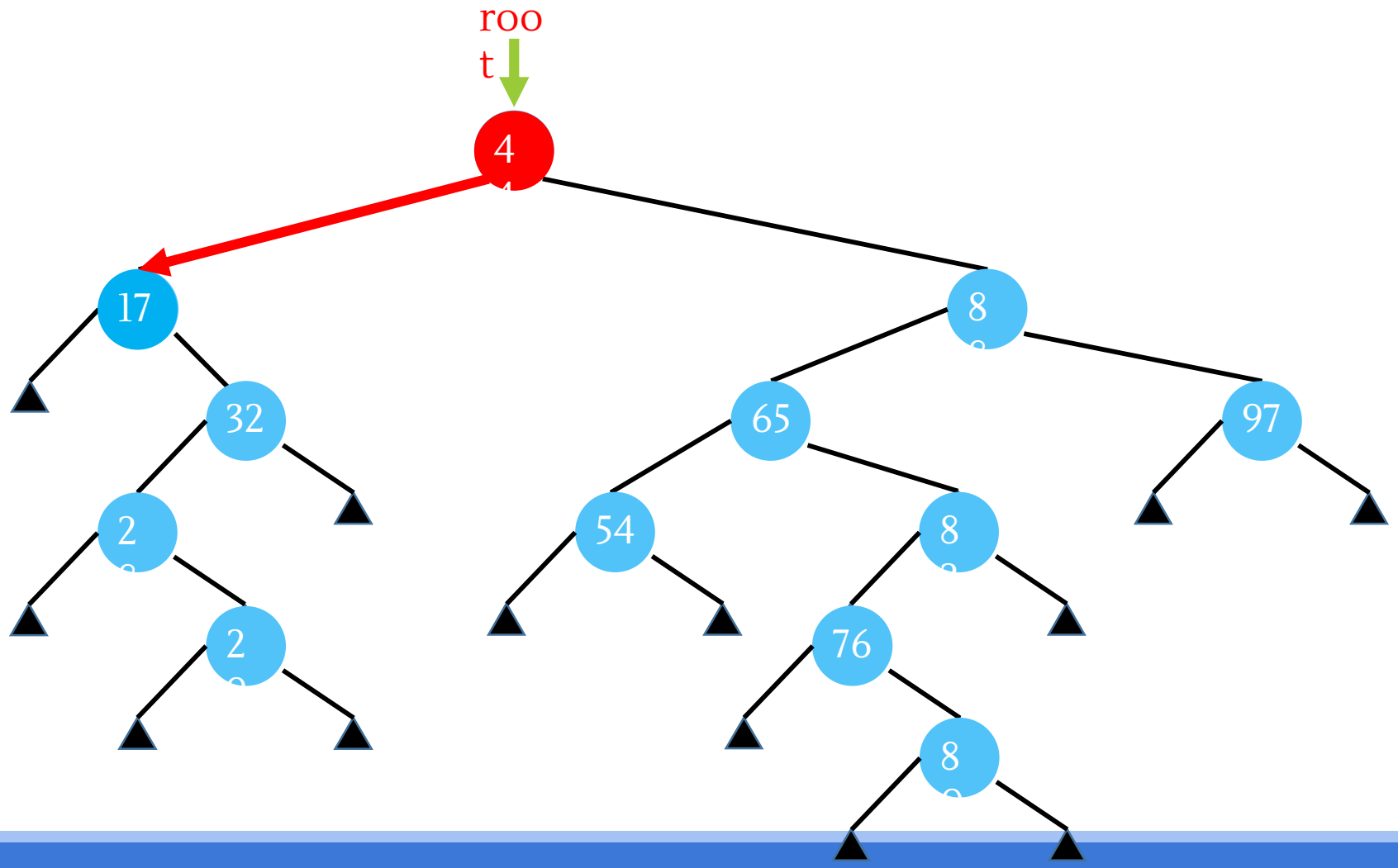
Search:
85



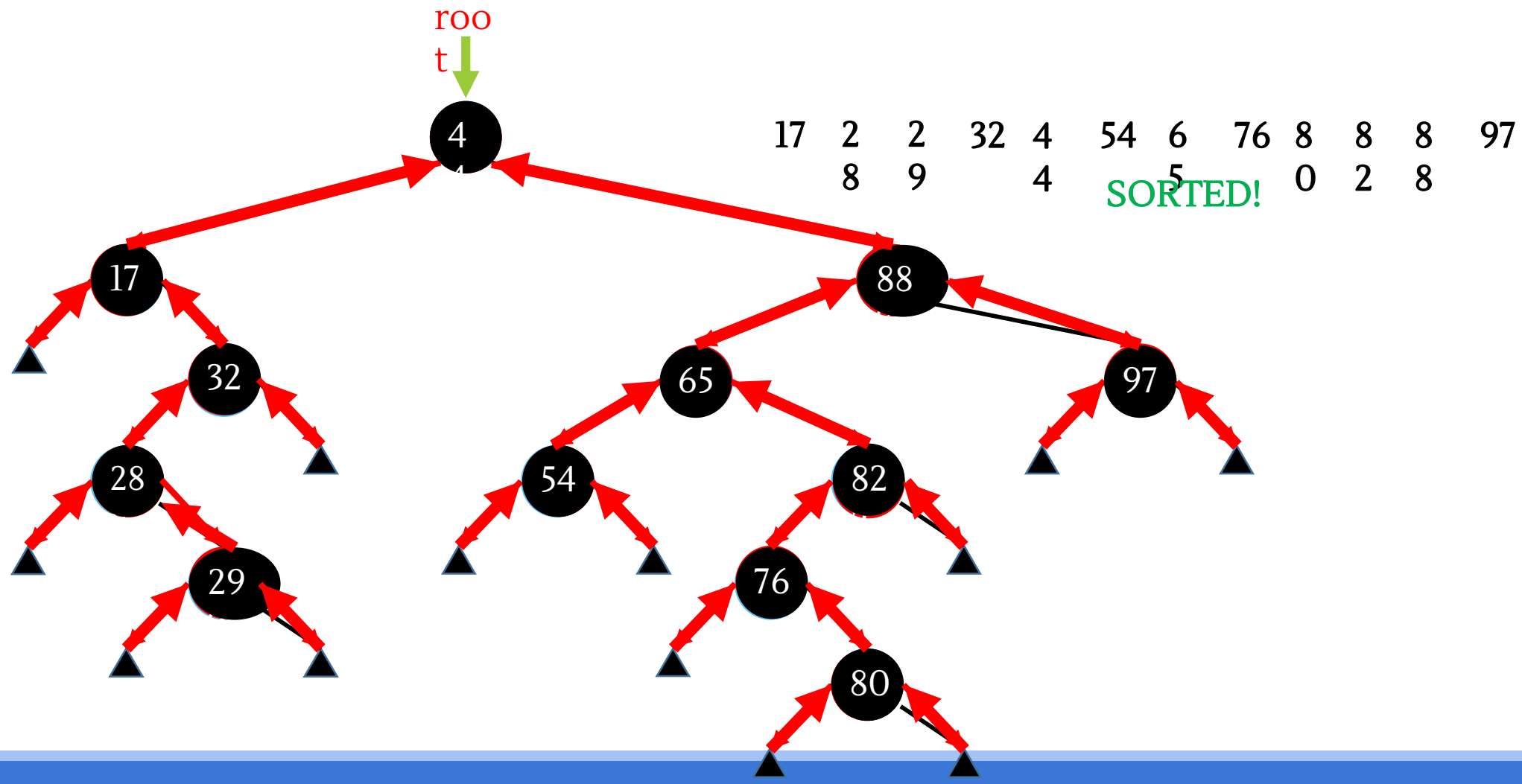
Maximum in BST



Minimum in BST

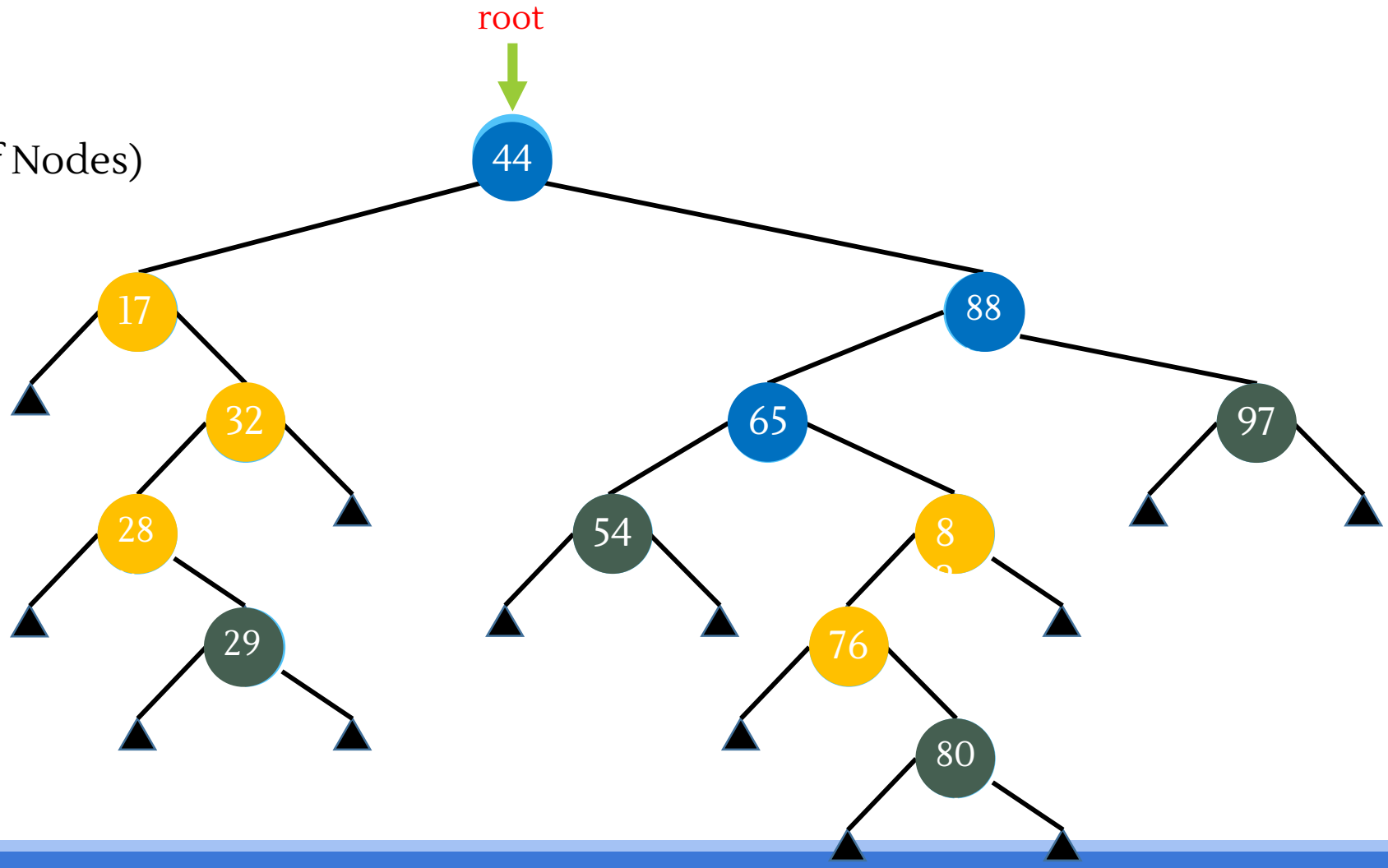


In Order Traversal in BST



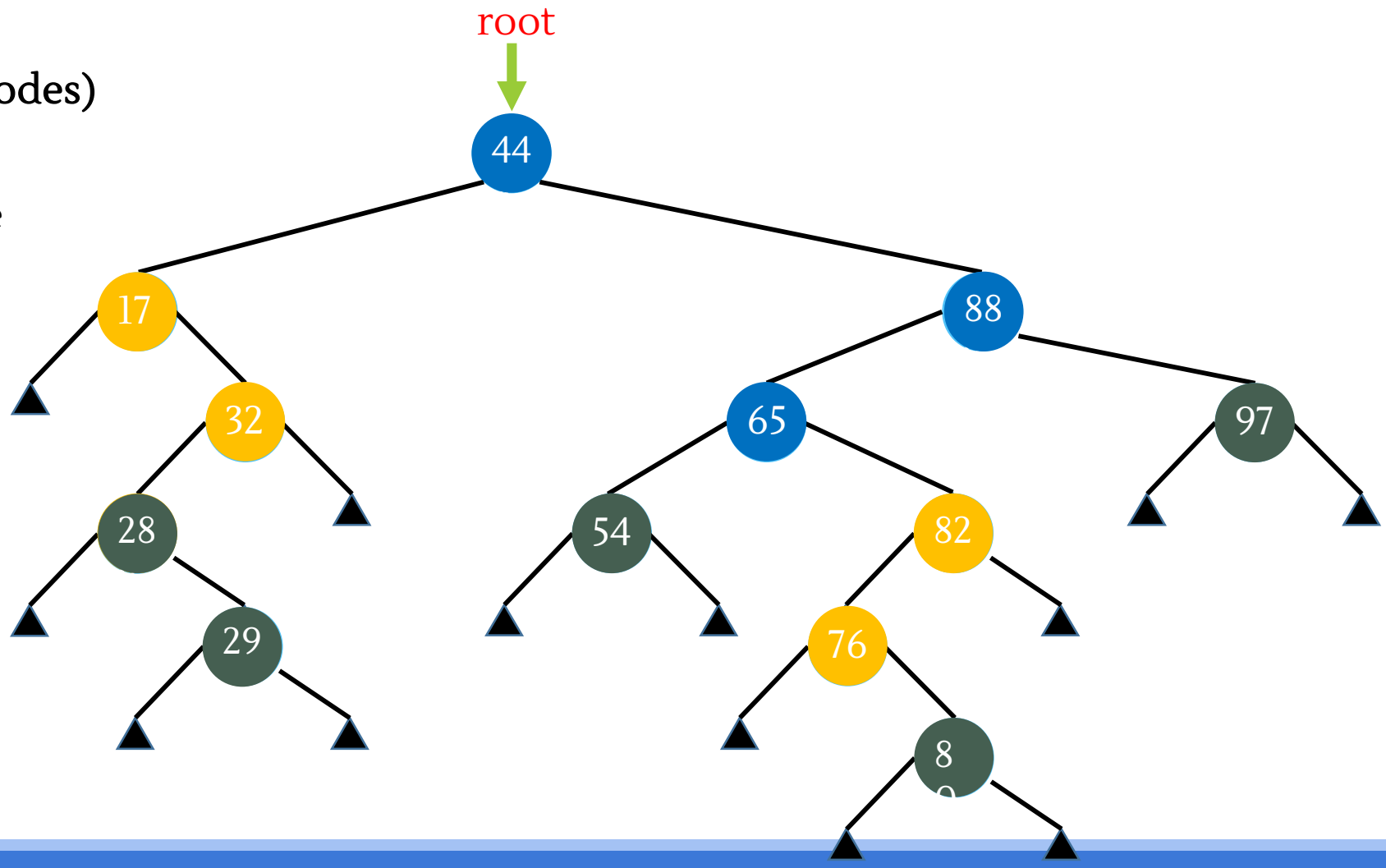
Delete in BST

- 3 Cases
  - Nodes with 0 child (Leaf Nodes)
  - Nodes with 1 child
  - Nodes with 2 children



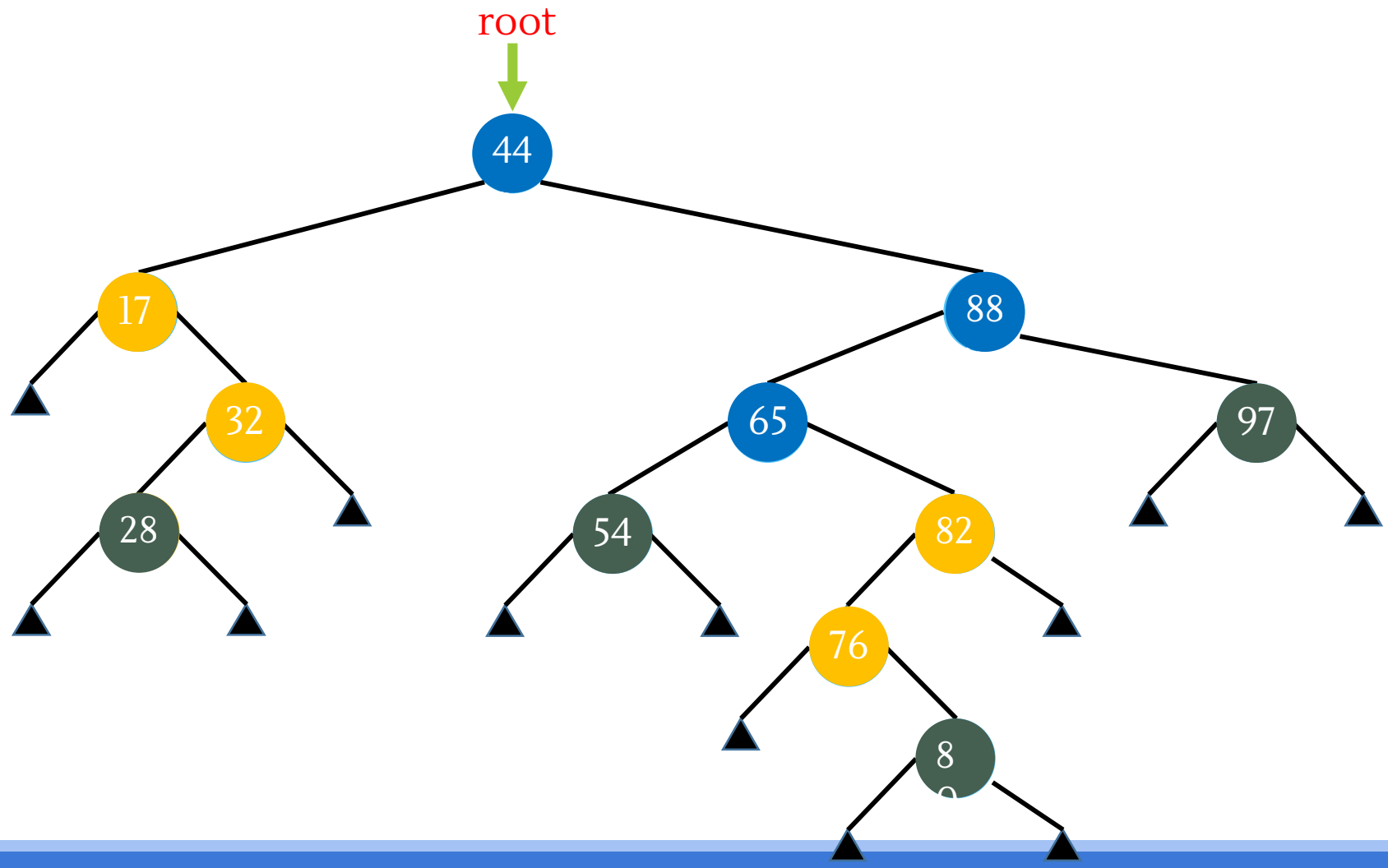
Delete in BST (Case 1)

- Nodes with 0 child (Leaf Nodes)
  - Delete: 29
  - Simply remove the node



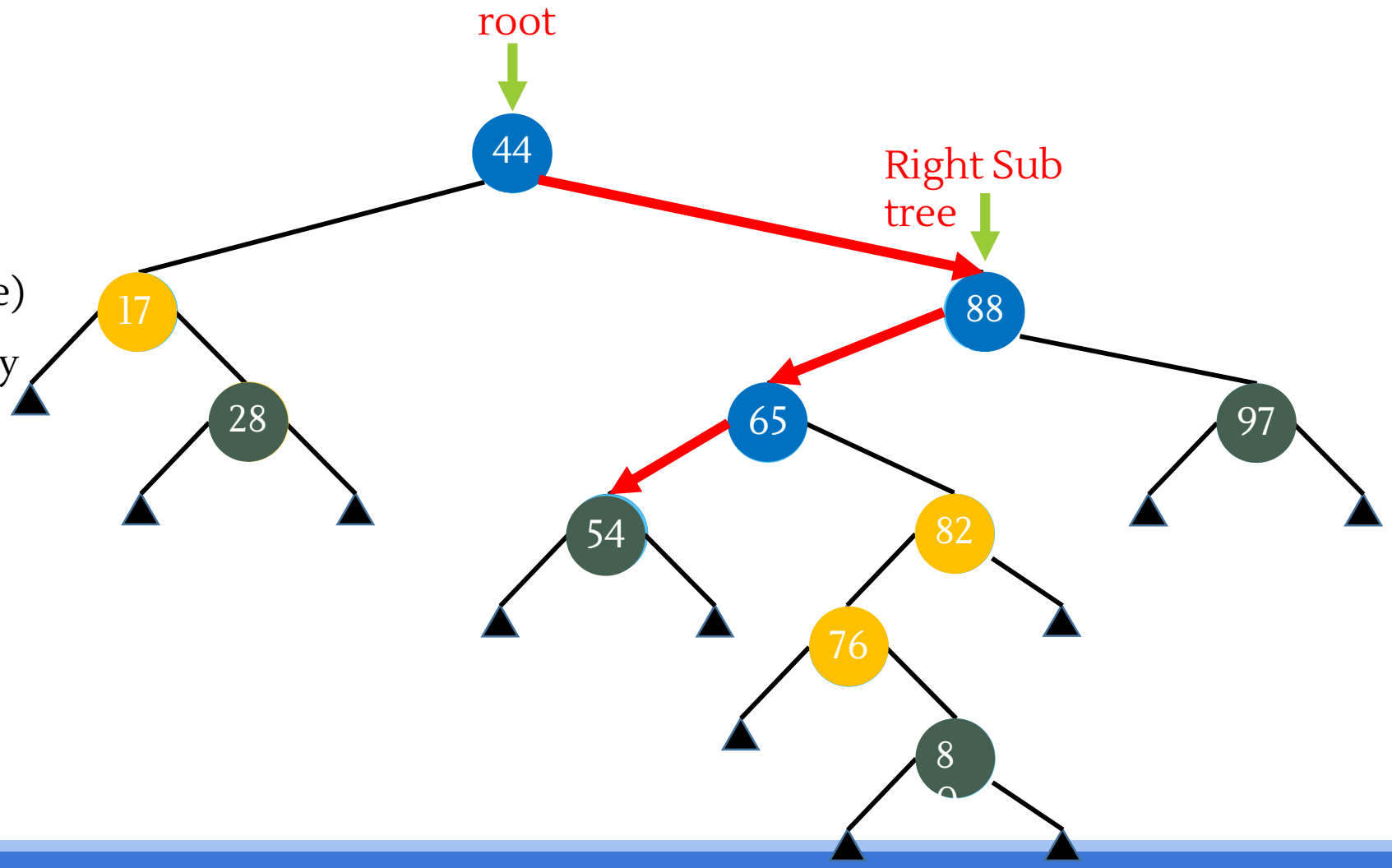
Delete in BST (Case 2)

- Nodes with 1 child
  - Delete: 32
  - Splice out the node



Delete in BST (Case 3)

- Nodes with 2 children
  - Delete: 44
  - Find the in order successor (Minimum node of right sub tree)
  - Replace the key of the node by the key of it's successor (88 by 54)
  - Delete the successor (54)




thank
you