IS2020 COMP 2540: Data Structures and Algorithms

Lecture 02: Linked List

Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

May 27, 2020

Use of a List: Motivation

- What are the advantages of arrays?
  - Arrays are simple and easy to use.
  - Faster access time to the element O(1).

- What are the disadvantages of arrays?
  - Pre-allocates memory up front (why is that bad thing?)
  - Fixed Size (Problem when inserting new element)
  - Contiguous memory allocation
  - Insertion and deletion are in general expensive.

Use of a List: Motivation

- C

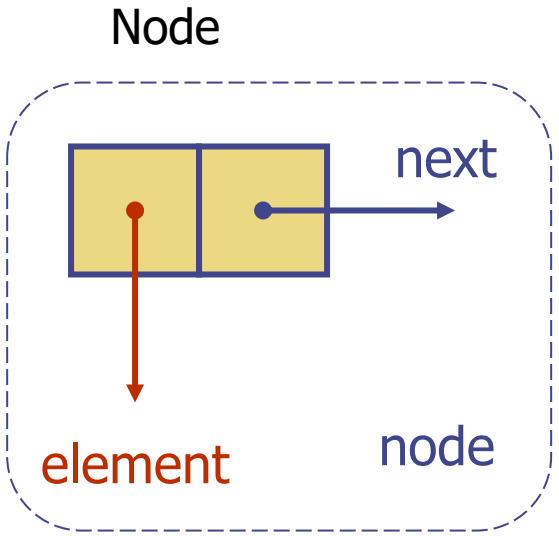
```
public class ArrayInsert {
    public static void main(String arg[]){
      int MAX=10;
      int [] my_array = new int[MAX];
        my_array[0]=10;
        my_array[1]=20;
        my_array[2]=30;
        my_array[3]=40;
        my_array[4]=50;
        my_array[5]=60;

// Insert an element in 3rd position of the array (index->2, value->5)
    int Index_position = 2;
    int newValue    = 5;

    System.out.println("Original Array : "+Arrays.toString(my_array));
    for(int i=my_array.length-1; i > Index_position; i--){
        my_array[i] = my_array[i-1];
    }
    my_array[Index_position] = newValue;
    //print the array after inserting an element
    System.out.println("New Array: "+Arrays.toString(my_array));

}
}
```

Use of a List: Motivation

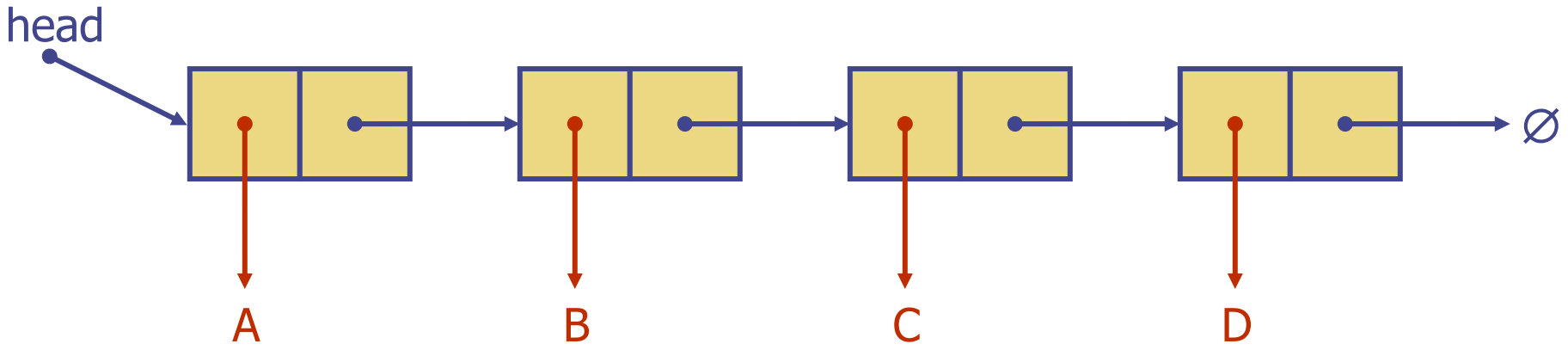
- Can we create dynamic arrays?
  - **No,** we need an efficient linear data structure.

Linked List

- **List** is one of the most basic types of data collection
  - For example, list of groceries, list of modules, list of friends, etc.
  - In general, we keep items of the same type (class) in one list



- **Definition:**
  - A collection of **nodes** where each node has an **element** and points to the **next** node in the list.



Linked List Properties

- Pointers connect successive elements.
- A Linked list has a **dynamic size**. It can grow or shrink during execution time.
- It can grow as much as we want or until the system run-out of memory.
- It does not waste memory space.
- The pointer of the last element in the list points to **null**
- It require a special pointer "**head**" that always point to the beginning of the list.

How a Linked List stored in memory?

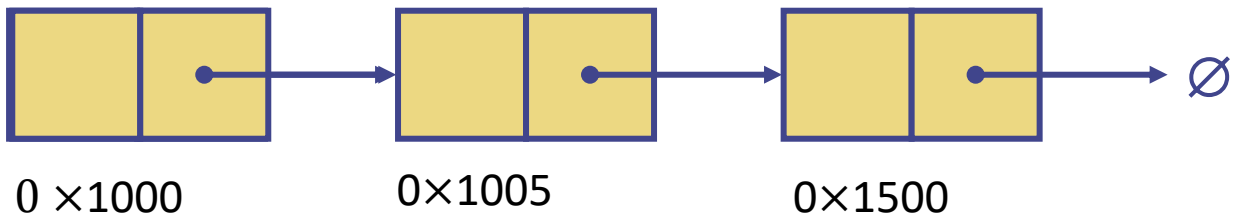
- Arrays in Memory: Continuous Allocation

Memory
0 ×1000
0 ×1001
0 ×1002
0×1003
0 ×1004

myArray

[0]	“A”
[1]	“P”
[2]	“P”
[3]	“L”
[4]	“E”

- Linked List in Memory: Random Allocation



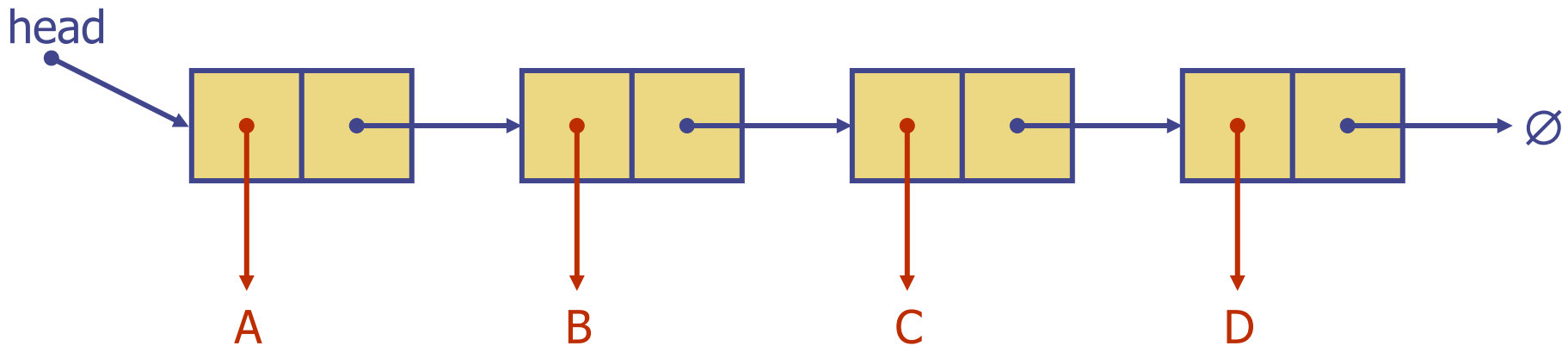
Linked List as ADT

- Linked List ADT = Linked List + Linked List Operations

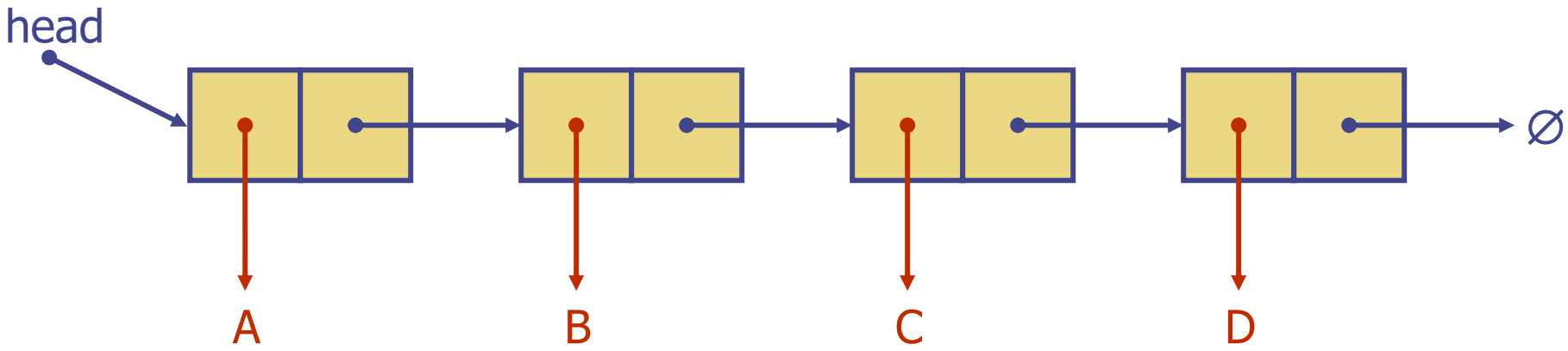
- Linked List Operations:
  - **Traverse**: visit each node (element) in the list in a sequential order.
  - **Insert**: add new node to the list
  - **Delete**: delete an existing node from the list
  - **Search:** find the index of a node with given data value
  - **Sort**: sort the nodes in the list based on some order
  - **Merge**: combine two or more list into one list

Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer.



Singly Linked Lists




```
public class Node {
    Node next;  //pointer to the next node
    int data;   // value of the data

    //Constructor
    public Node(int data)
    {
        this.data=data;
    }
}
```

Linked List ADT Operations: Traversing

- Traversing a Linked List
  1. Start from the head pointer
  2. Follow the next pointers
  3. Process every node as required (e.g. print the node date, count the size of the node, find the node with maximum data value, etc)
  4. Stop when the pointer next points to NULL.

  The time complexity is O(n)

Singly Linked Lists

```
public class LinkedList {
        Node head; // head of the list - first element
        // Insert at the rear
        public void insertData(int data){
            //check whether you are inserting to the head or node
            if(head==null) {
                head = new Node(data);
                return;
            }
            Node current = head;
            while(current.next!=null)    //check until last node
            {
                current=current.next;    // move pointer one by one
            }
            current.next=new Node(data); // when the pointer reaches the null
pointer, insert the data.
        }
```

Singly Linked Lists

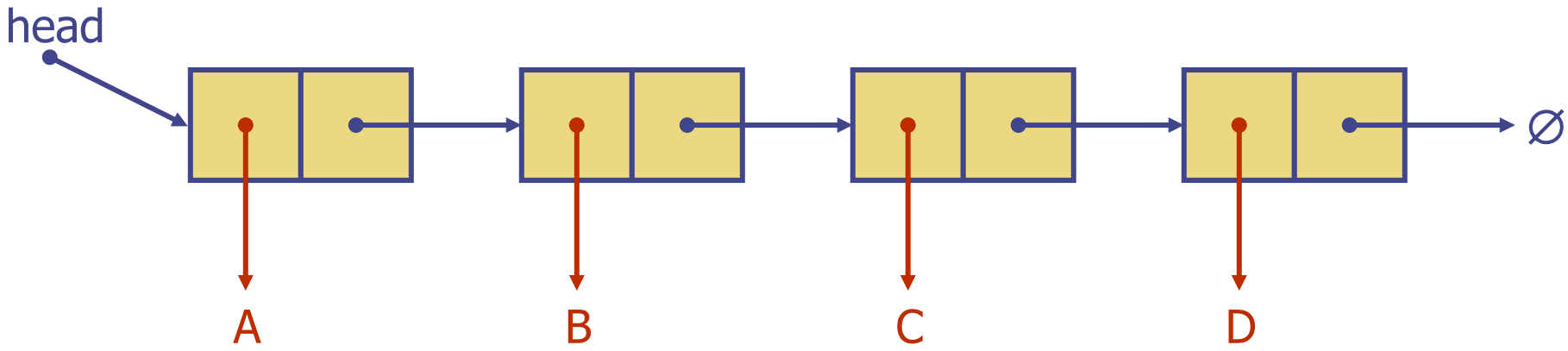
```
public class LinkedList {
        Node head; // head of the list - first element
        // Insert at the rear

        // print element in the linkedlist
        public void displayData()
        {
            Node current = head;
            while(current.next!=null)
            {
                System.out.println(current.data);
                current=current.next;
            }
            System.out.println(current.data);
        }
    }
```

Linked List ADT Operations: Inserting

- When we are inserting a node into a linked list we have three cases:
  1. Inserting a node at the beginning of the list
  2. Inserting a node at the end of the list
  3. Inserting a node at the middle of the list

Linked List ADT Operations: Inserting

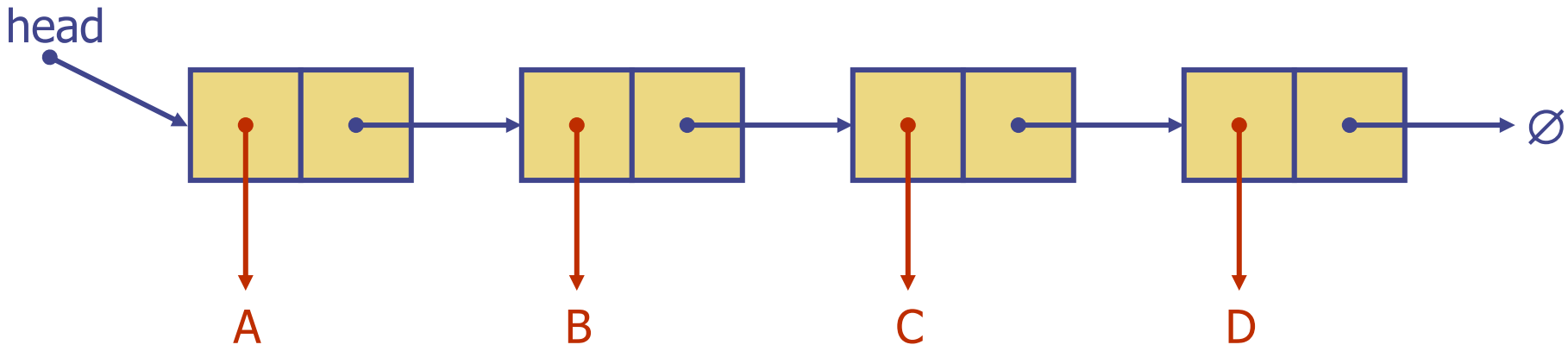
- CASE 1: Inserting at the beginning




```
public void insertFirst(int data)
{
     Node newHead=new Node(data);   // crate the new node
     newHead.next=head;          // assign the existing head at next
     head = newHead;            //  assign the new node at head
}
```

Linked List ADT Operations: Inserting

- CASE 2: Inserting a node at the end of the list




```
// Insert at the rear
public void insertData(int data){
    //check whether you are inserting to the head or node
    if(head==null) {
        head = new Node(data);
        return;
    }
    Node current = head;
    while(current.next!=null)    //check until last node
    {
        current=current.next;    // move pointer one by one
    }
    current.next=new Node(data); // when the pointer reaches the null pointer, insert the data.
}
```

Linked List ADT Operations: Inserting

- CASE 3: Inserting a node at the middle of the list

```
public void insertAt(int position,int data)
{
    if (head==null) return;
    Node newNode=new Node(data);  //create a new node
    Node previous=head;       // temporary node to point head
    int count=1;
    // find the position to insert data
    while(count<position-1)
    {
        previous=previous.next;
        count++;
    }

    Node current=previous.next;      // temporary node
    newNode.next=current;
    previous.next=newNode;
}
```

Linked List ADT Operations: Deleting

- When we are deleting a node into a linked list we have three cases:
  1. Delete from the beginning (first node)
  2. Delete from the end (delete the last node)
  3. Delete from the middle.

Linked List ADT Operations: Deleting

- CASE 1: Deleting at the beginning

Or

```
public void deleteFirst()
{
     head=head.next;
}
```

```
public void deleteFirst()
{
     Node current= head;
     head=head.next;
     current.next=null;
}
```

Linked List ADT Operations: Deleting

- CASE 2: Deleting at the end

```
public void deleteData()
{
     Node current = head;
     Node previous=null;

     while(current.next!=null)
     {
          previous=current;
          current=current.next;
     }
     previous.next=null;

}
```

Linked List ADT Operations: Deleting

- CASE 3: Delete from the middle

```
public void deleteAt(int position)
{
    if (head==null) return;
    Node previous=head;       // temporary node to point head
    int count=1;
    // find the position to delete data
    while(count<position-1)
    {
        previous=previous.next;
        count++;
    }
    Node current=previous.next;      // temporary node
    previous.next=current.next;
    current.next=null;
}
```

Advantages and Disadvantages of Singly Linked List