Data Structures

Abstract Data Types and an Introduction to lists

DATE : 19/10/2022

Learning Outcomes

Learning Outcomes

- By the end of this lecture, you will be able to understand:
  - what an abstract datatype is and how it relates to data structures and OOP classes.
  - what a list is and why arrays pose a limitation for implementing them
  - what linked lists are, their types, their structure and how to implement them

Abstract Data Types and Data Structures

Data types and Data structures

- **Simple (scalar) data types**:
  - can hold only one piece of data at a time the value of which can vary over some range of ordered values. For any given simple data type, there is a set of operations that can be performed on its values.
  - are often called **primitive** data types or fundamental data types. Scalar data types include int, char, float and double.
- **Structured data types:**
  - hold a collection of data values. Examples include arrays, records (structs), classes and files.
- Primitive and structured data types are treated differently

Data types and Data structures – *cont'd*

- **A data structure:** is a collection (grouping) of simple or structured data types and **a set of rules** (operations) for organizing and accessing them.
- Examples include arrays, lists, stacks, queues, trees and graphs.
- Taking a **list** as an example, typical list operations allow you to:
  - create an empty list.
  - add an item to the list.
  - remove an item from the list.
  - display the contents (traverse the list) in order.
  - Modify/update an entry in the list.

Abstract data types

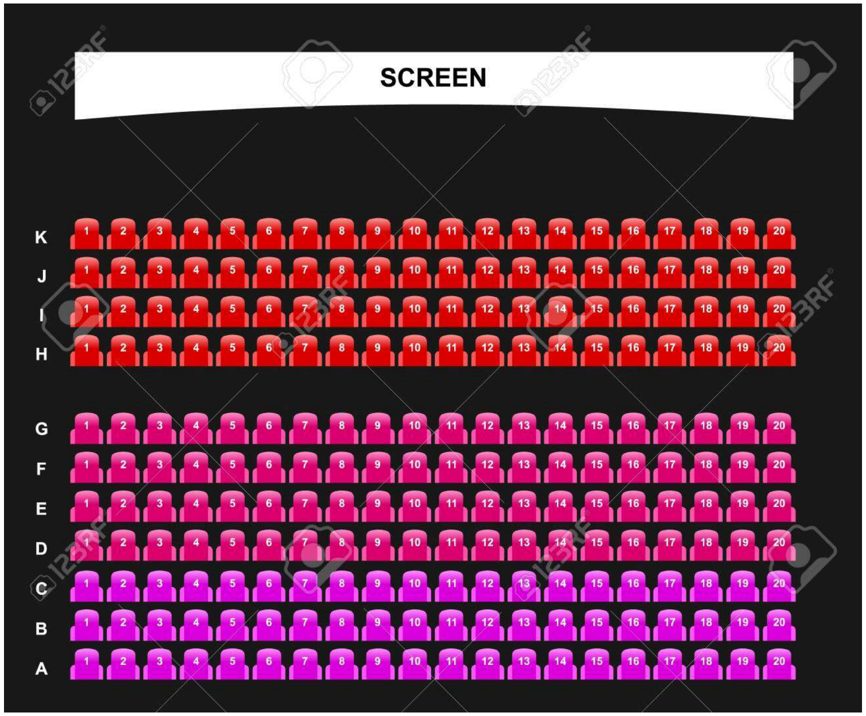
- An Abstract Data Type (ADT) is a well-specified collection of data and a group of operations that can be performed upon the data. The ADT's specification describes what data can be stored (the characteristics of the ADT), and how it can be used (the operations), but not how it is implemented or represented in the program.
- A list can be described by the type of information that it holds and by the operations that can be performed on the list. In this sense the list (which is a data structure) is an example of an ABSTRACT DATA TYPE. It is possible to think about a list without knowing the details of how it is implemented.

Relationship between Classes and ADTs

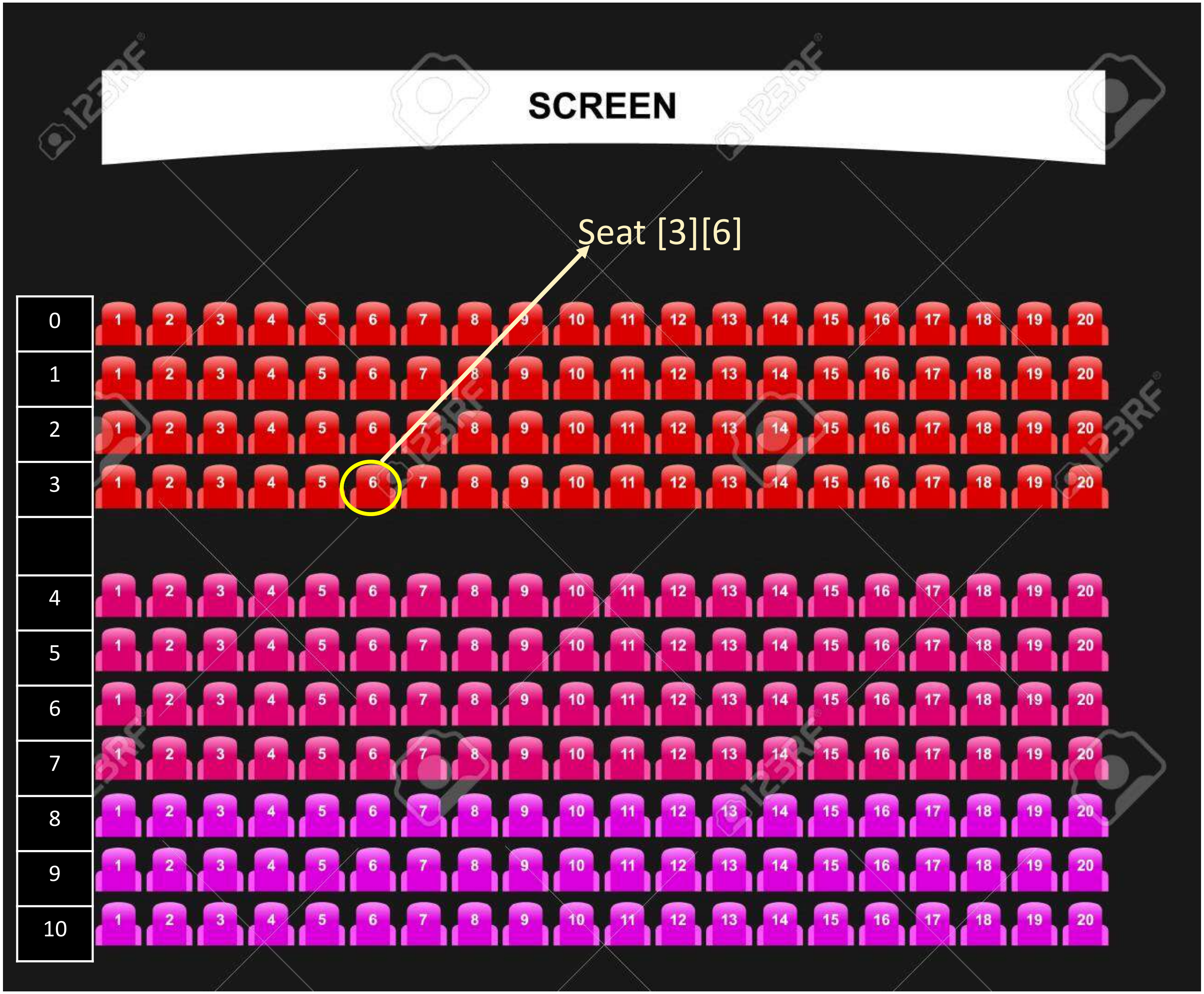
- An abstract data type can be implemented as a class
- A class is a user-defined data structure that consists of:
  - Data members that define the class structure
  - Function members that operate on the data members. Function members are called methods. They define the rules/operations for organizing and accessing them
- Classes are used to define objects, so they serve as templates from which ‘similar’ objects can be created.
- Defining a class does not create an object.

Why study data structures?

- There is almost no real life application that does not make use of one or more data structures.
- Lets say that you want to build a cinema booking application, how would you represent the seats shown to the user internally?
    - Use a 2D array



A closer look



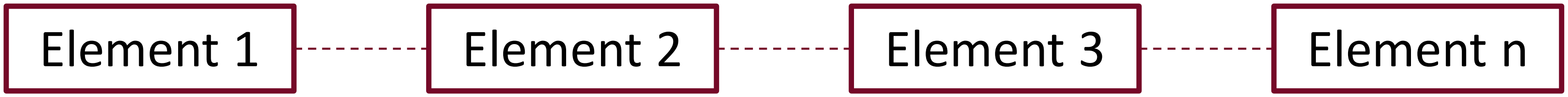
Other examples

- How do you think the undo functionality in editors is implemented?
  - Using stacks. Stacks are data structures that can be used to retrieve items from newest to oldest.
- In an office space, how do you think a printer deicides which job it should print first from the many print requests it get from around the office?
  - Using Queues. Queues are data structures that can be used to retrieve items from oldest to newest. This emulates a real life queue where the person who arrives first and stands in line first, gets served first.
- Can you think of how you can replicate the functionality of storing contacts on your mobile phone?

Lists

List Definition

- A list is sequence of elements where there is a first and last element and each element has a previous element and a next element, except that there is:
  - Nothing before the first element
  - Nothing after the last element



Element 1

Element 2

Element 3

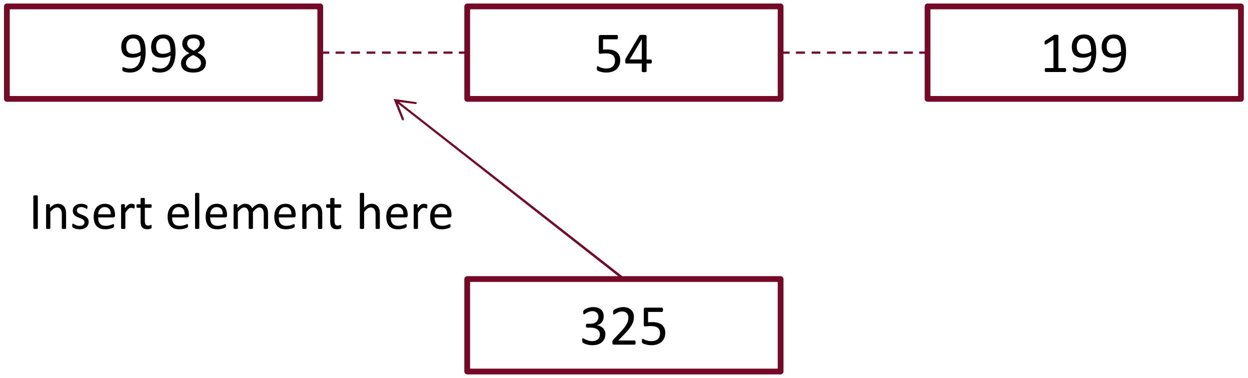
Element n

List Operations

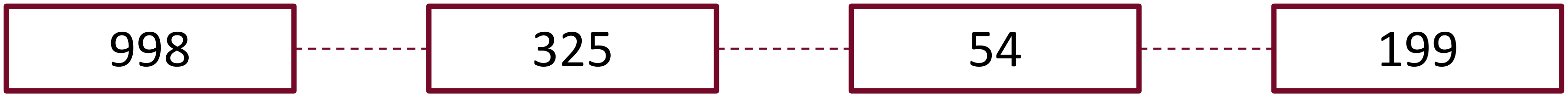
- There are many different operations that can be applied to a list regardless of the implementation, a few example operations are:
  - Basic operations:
    - Initializing a list
    - Destroying a list
  - Deletion and insertion operations
  - Traversal operations:
    - Printing all list elements
    - Finding an element
    - Counting the elements

List Operations Example

- Inserting an element at position 2
- List before insertion:



- List after insertion:

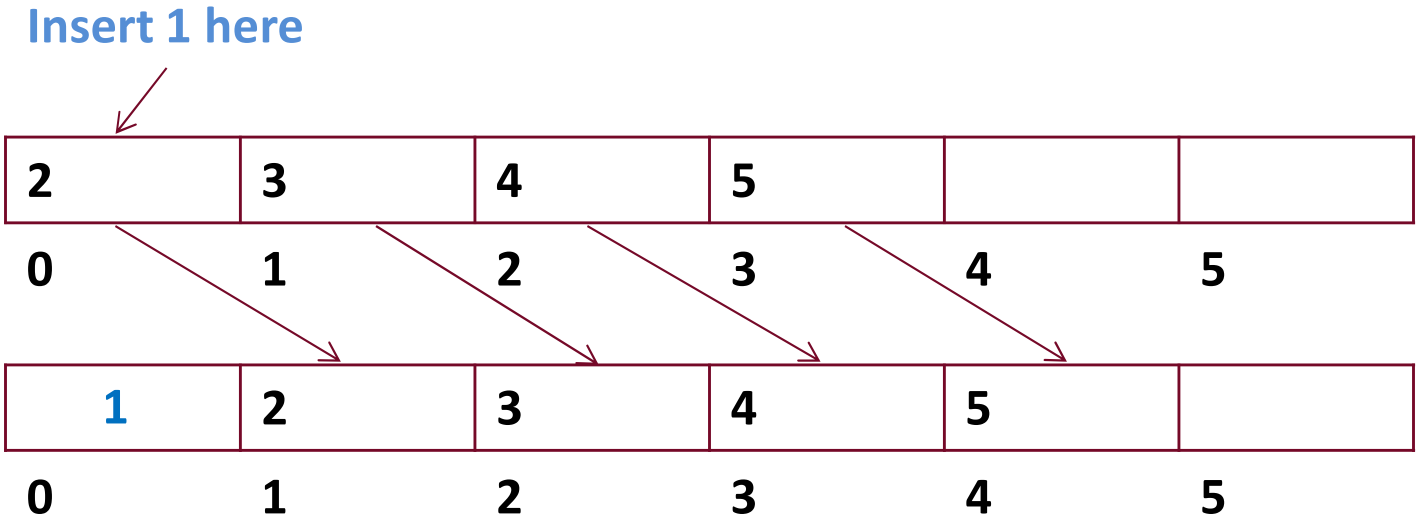


List Implementations

- There are two ways of implementing a list:
  - The first is using arrays
  - The second and most common way is using linked lists

Array-based Lists

- While arrays are a very useful data structure provided in all programming languages, they have a certain challenge:
  - Manipulating the array-based list, either by adding or removing an element either at the beginning or in the middle of the list requires shifting all or part of the contents of the array.



Insert 1 here

2	3	4	5		
0	1	2	3	4	5

1	2	3	4	5	
0	1	2	3	4	5

More Limitations of Arrays

- Arrays are not self- contained objects
  - An operation on an array needs the array as well as its size
  - The size of an array is not contained in the array object
- The programmer is responsible for indexing an array within its index range. If he/she doesn't, a runtime error can easily take place
- Arrays cannot grow or shrink during program execution (this is solved by linked lists)
  - Once an array is allocated, its size is fixed
  - To increase the size of an array, a new larger array should be allocated then:
    - Content of smaller array should be copied into larger array
    - Space taken up smaller array should be freed

Linked Lists

Linked List Definition

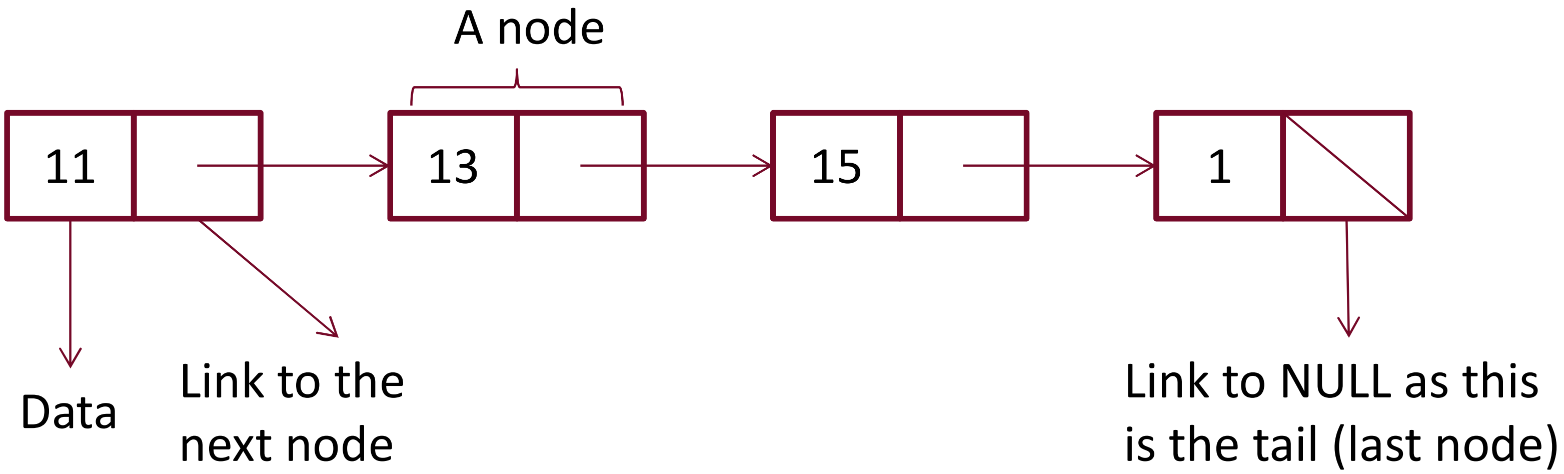
- A linked list is a structure for storing a list of items.
- It is made of any number of pieces of memory (nodes) and each node contains whatever data you are storing along with a pointer (a link) to another node.
- Memory is allocated dynamically for new nodes to be inserted and freed whenever a node is deleted.

Linked List Types

- There are several types of linked lists, each with different structure and properties:
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List
  - Doubly Circular Linked List

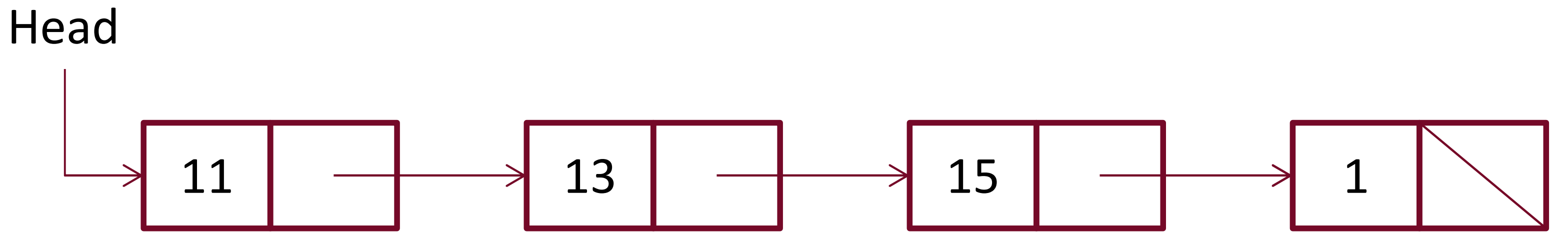
Singly Linked List Structure

- Nodes:
  - A list node is a container for data
  - A node can have at most one predecessor and one successor node
  - We can insert and delete nodes in any order



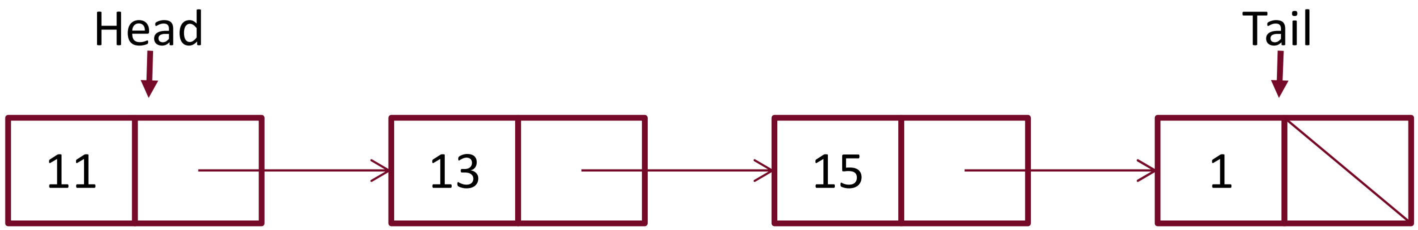
Finding the first node

- Each linked list, has a special link, called the head.
- The head stores the link to the first node, if the list is not empty, or points to nothing to indicate that the list is empty.



Marking the last node

- Sometimes it’s useful, to also have a pointer to last element in a list
- When such a pointer is implemented, it is usually called the ‘tail’.



Defining an integer node as a struct

```
struct Node {
  int data;
  Node *next;
};
```

Defining an integer singly linked list

```
class LinkedList {
    private:
        Node *head, *tail;
    public:
        LinkedList(){
            head=tail=NULL;
        }
  //Put other operations we want the list
  //to support here
};
```

What other operations do we want to support?

- Adding an element (where at the beginning?, at the end?, both?)
  - Let’s assume we want to support both, so we are going to define a method for adding at the beginning and another at the end:

    ```
    void addToStart(int d)
    void addToTail(int d)
    ```
- Checking if the list is empty

  ```
  bool isEmpty()
  ```
- Printing elements in the list

  ```
  void printList()
  ```
- Deleting an element ? Should we know value of the element we want to delete?

  ```
  void delElement(int e)
  ```

Defining an integer singly linked list

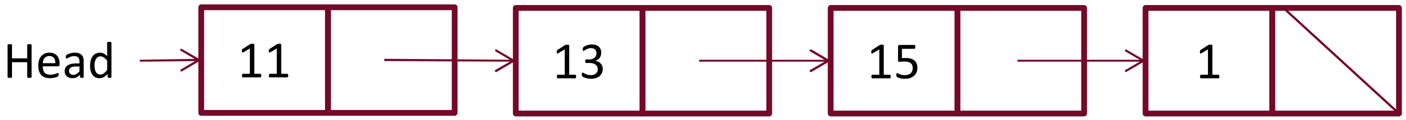
```
class LinkedList {
    private:
        Node *head, *tail;
    public:
       LinkedList(){
            head=tail=NULL;
       }
        ~LinkedList();   //destructor
       void addToStart(int d);
       void addToTail(int d);
       bool isEmpty(){
              return head==NULL;
       }
       void printList();
       int pop_front(); //deletes first element and returns its value
};
```

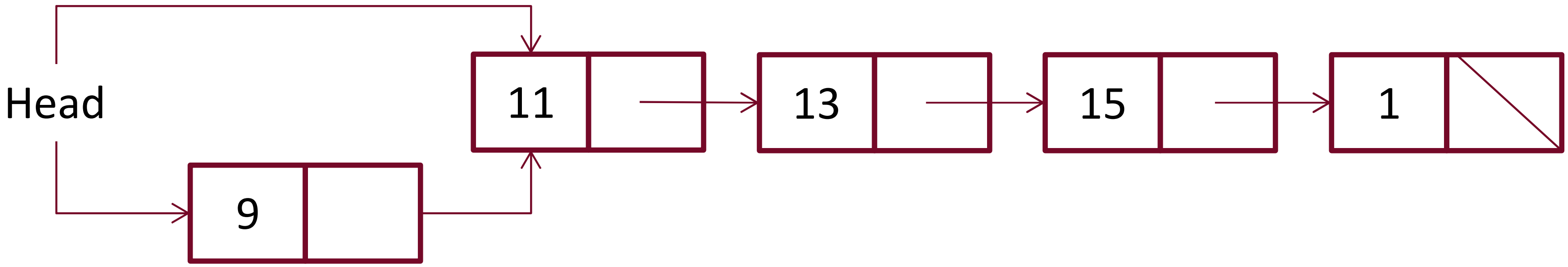
Adding a node at the beginning of the linked list

- Inserting a node at the start of linked list can be done in 4 steps, as follows:
  - Create a new node.
  - Put the data value into the new node.
  - Copy the link from the head node so that the new node is pointing to the old first node.
  - Change the link in the head node to point to the new node.

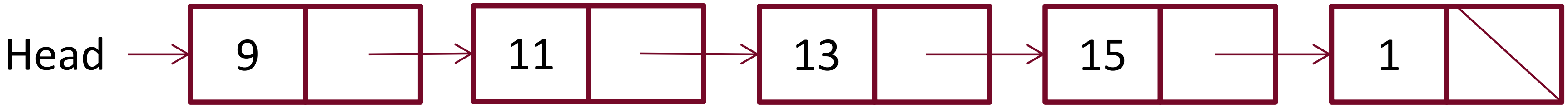
Example: adding a node at the beginning of the linked list

- The linked list originally:





- The linked list after the insertion:



Code for adding a node at the beginning of the linked list

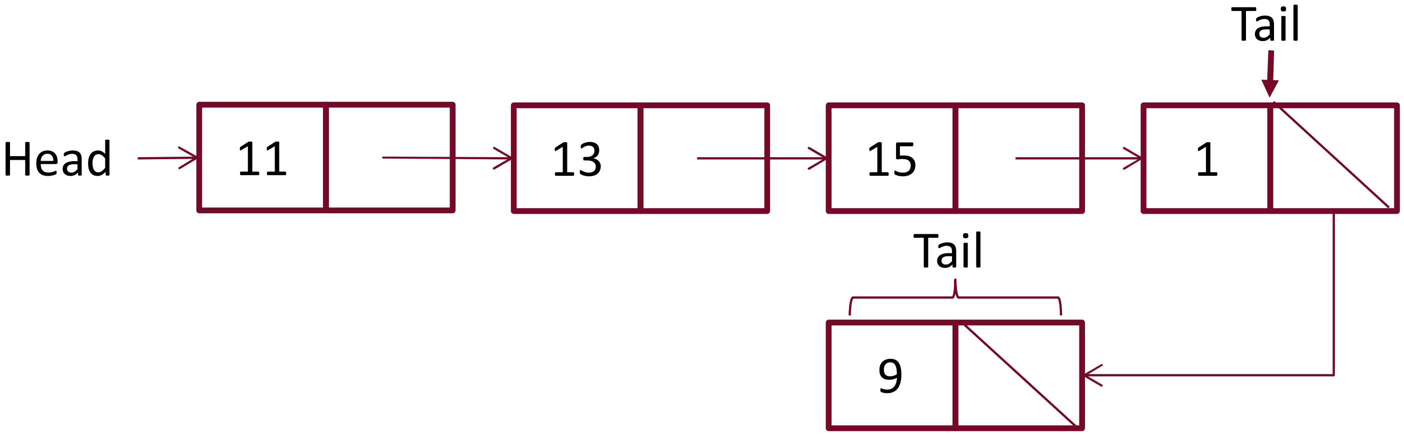
```
void LinkedList::insertAtStart(int d){
    //Create a new node
    Node *newNode= new Node();
    //Put the data value into the new node.
    newNode->data = d;
    //Copy the link from the head node so that the new node is pointing to
    the old first node.
    newNode->next = head;
    //Change the link in the head node to point to the new node.
    head = newNode;
   //If no elements had been added before, update tail to point to the first
   element in the list
    if(tail == 0)
          tail = head;
}
```

Adding a node at the end of the linked list

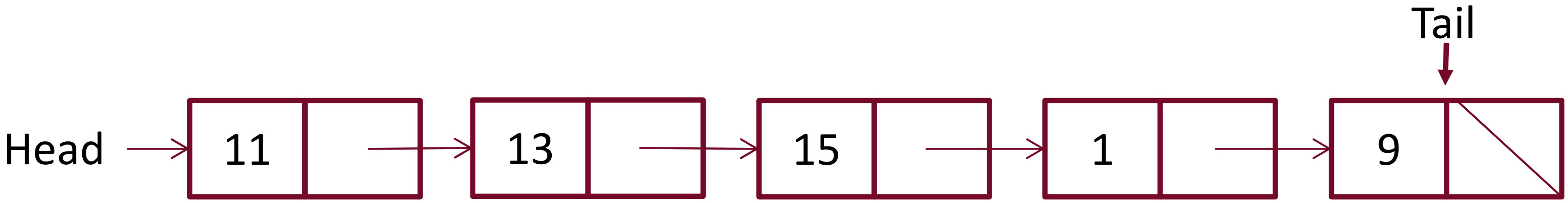
- Inserting a node at the end of linked list can be on 5 steps, as follows:
  - Create a new node.
  - Put the data value into the new node
  - Change the 'link' (next) field pointed to by the tail so that it would point to the new node.
  - Mark the next field in the new node to indicate that it is the end of the list, as in, make it point to nothing (NULL).
  - Let the tail pointer point to the new node.

Example: adding a node at the end of the linked list

- The linked list originally:



- The linked list after the insertion:



Code for adding a node at the end of the linked list

```
void LinkedList::insertAtEnd(int value){
    //Create a new node.
    Node *newNode= new Node();
    //  Put the data value into the new node
    newNode->data = d;
    if (head == 0) // list is empty -- set head and tail to
    point to new node
        head = tail =  newNode;
    else {
        // Change the next) field pointed to by the tail so that it would point to the new
    node.
        tail->next = newNode;
        // Let the tail pointer point to the new node.
        tail = newNode;
    }
    //Mark the next field in the new node to indicate that it is the end of the list
    newNode->next = 0;
    }
```

Code for Deleting First Element in the List

```
int LinkedList::pop_front() {
    int data = head->data;
    Node *tmp = head;
    if (head == tail) // if only one node in the list;
         head = tail = 0;
    else head = head->next;
    delete tmp;
    return data;
}
```

There is something missing from this function..

What if the list was empty?

Code for printing all elements in a list

```
int LinkedList::printList() const{
     Node *p;
     for(p=head;p!=NULL;p=p->next){
        cout << p->data <<endl;
     }
   }
```

Code for the Destructor

```
LinkedList::~LinkedList() {
    for (Node *p; !isEmpty(); ) {
        p = head->next;
        delete head;
        head = p;

    }
```

Defining an integer node as a class

```
class Node {
   public:
      int data;
      Node *next;
      Node(int value=0, Node *ptr=NULL){
         data=value;
         next=ptr;
      }
};
```

Summary

Summary

- In this lecture you’ve learned:
  - what an abstract datatype is and how it relates to data structures and OOP classes.
  - what a list is and why arrays pose a limitation for implementing them
  - What linked lists are, their types, their structure and how to implement them
  - Some linked list operations and methods and how to implement them