DTS203TC
Design and Analysis of Algorithms

Lecture 5: Elementary Data Structure

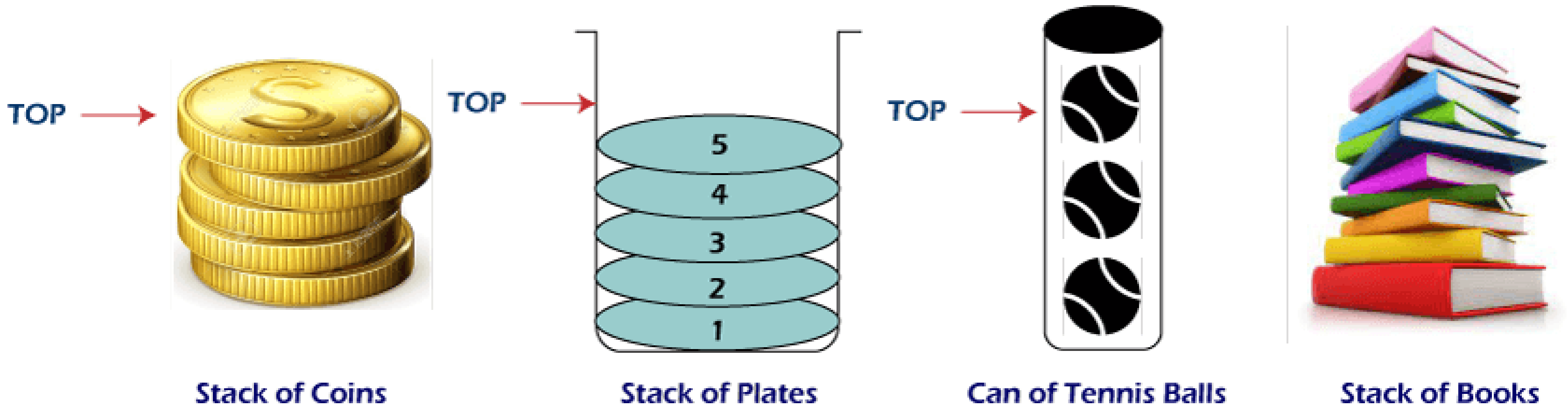
Dr. Qi Chen
School of AI and Advanced Computing

Learning outcomes

- Stack
- Queue
- LinkedList
  - Doubly Linked List
- Tree
  - Binary Tree

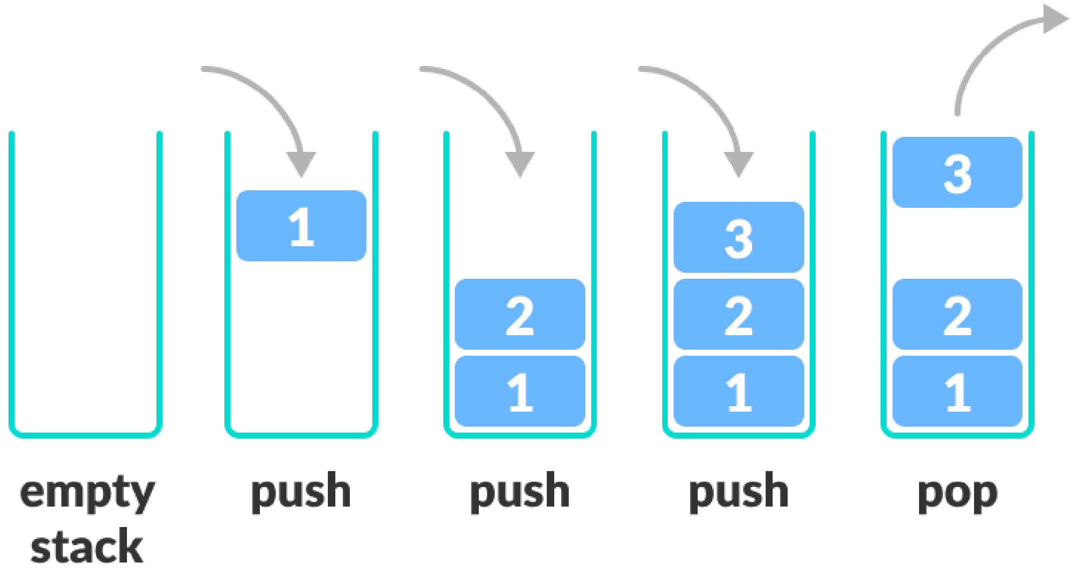
Stack

- A stack is a linear data structure that follows the principle of **Last In First Out (LIFO).**
  - the last element inserted inside the stack is removed first.



Basic Operations of Stack

- **Push:** Add an element to the top of a stack
- **Pop:** Remove an element from the top of a stack
- **Peek:** Get the value of the top element without removing it
- **IsEmpty:** Check if the stack is empty



IsEmpty

- A pointer called TOP is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.

```
IsEmpty(S)
if S.TOP == -1
    return True
else
    return False
```

Push

- On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.

```
Push(S,x)
S.TOP = S.TOP+1
S[S.TOP] = x
```

Pop

- On popping an element, we return the element pointed to by TOP and reduce its value.

```
Pop(S)
if isEmpty(S)
    error "underflow"
else
    S.TOP = S.TOP-1
    return S[S.TOP+1]
```

Stack implementation – Python

- We usually use arrays to implement Stack in Java and C/++. In the case of Python, we use lists.

```
class Stack:

    def __init__(self):
        self.stack = []

    # check empty
    def isEmpty(self):
        return len(self.stack) == 0

    # Adding items into the stack
    def push(self, item):
        self.stack.append(item)
        print("pushed item: " + item)

# Removing an element from the stack
    def pop(self):
        if (self.isEmpty()):
            return "underflow"
        return self.stack.pop()

    # Display the stack
    def display(self):
        print(self.stack)
```

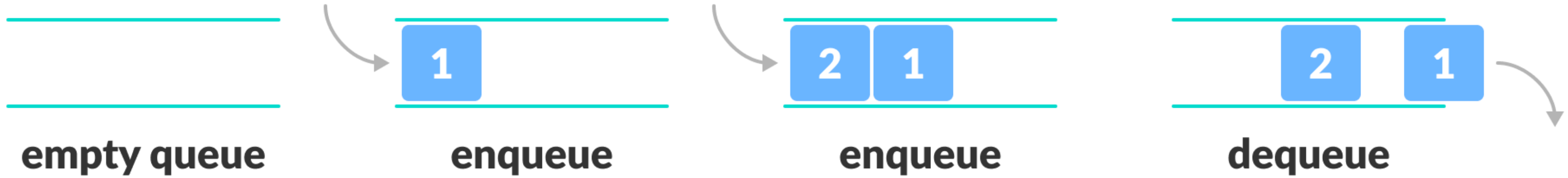
Stack implementation – Python

```
stack = Stack()
stack.push(str(1))
stack.push(str(2))
stack.push(str(3))
stack.push(str(4))
print("popped item: " + stack.pop())
print("stack after popping an element: ")
stack.display()
```

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element:
['1', '2', '3']
```

Queue

- Queue follows the First In First Out (FIFO) rule
  - the item that goes in first is the item that comes out first.



Basic Operations of Queue

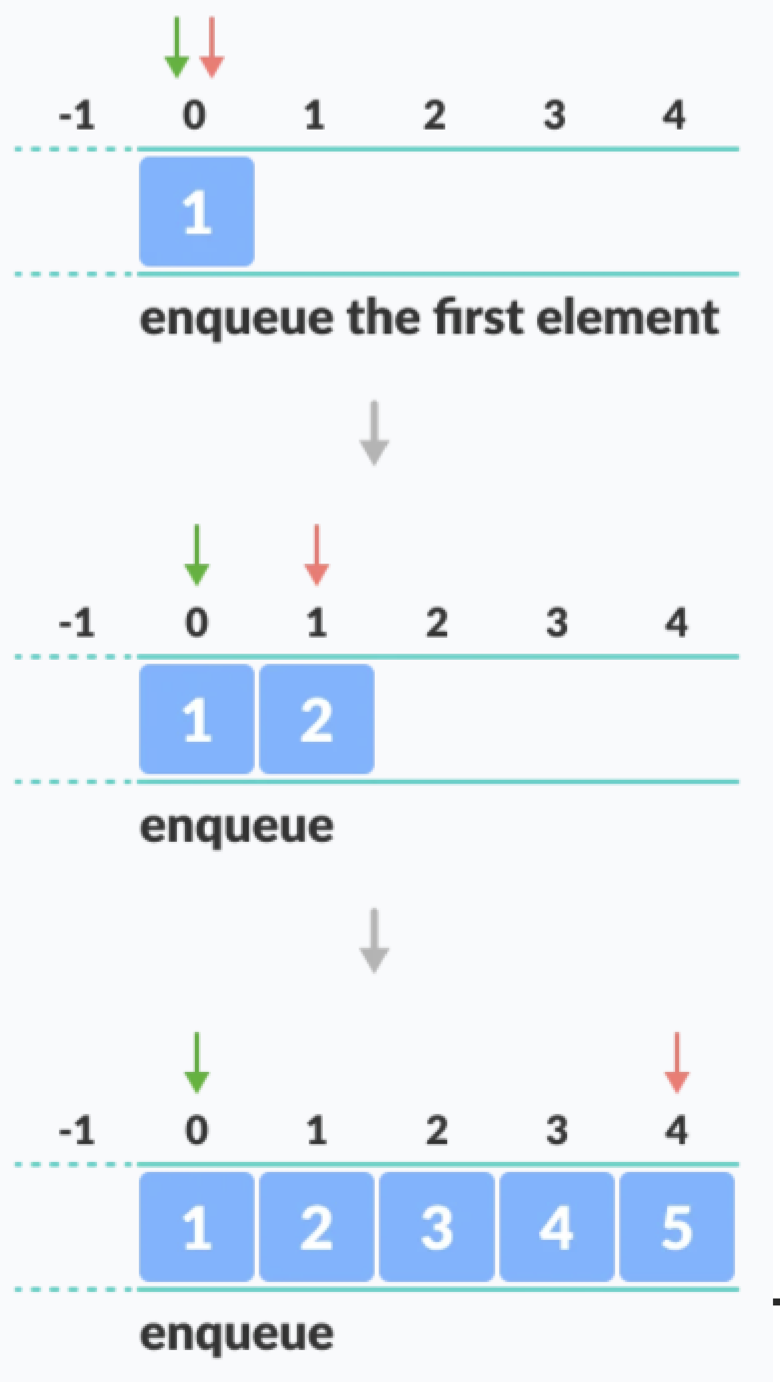
- Enqueue: Add an element to the end of the queue
- Dequeue: Remove an element from the front of the queue
- IsEmpty: Check if the queue is empty
- Peek: Get the value of the front of the queue without removing it

Working of Queue

- Queue operations work as follows:
  - two pointers FRONT and REAR
  - FRONT track the first element of the queue
  - REAR track the last element of the queue
  - initially, set value of FRONT and REAR to -1

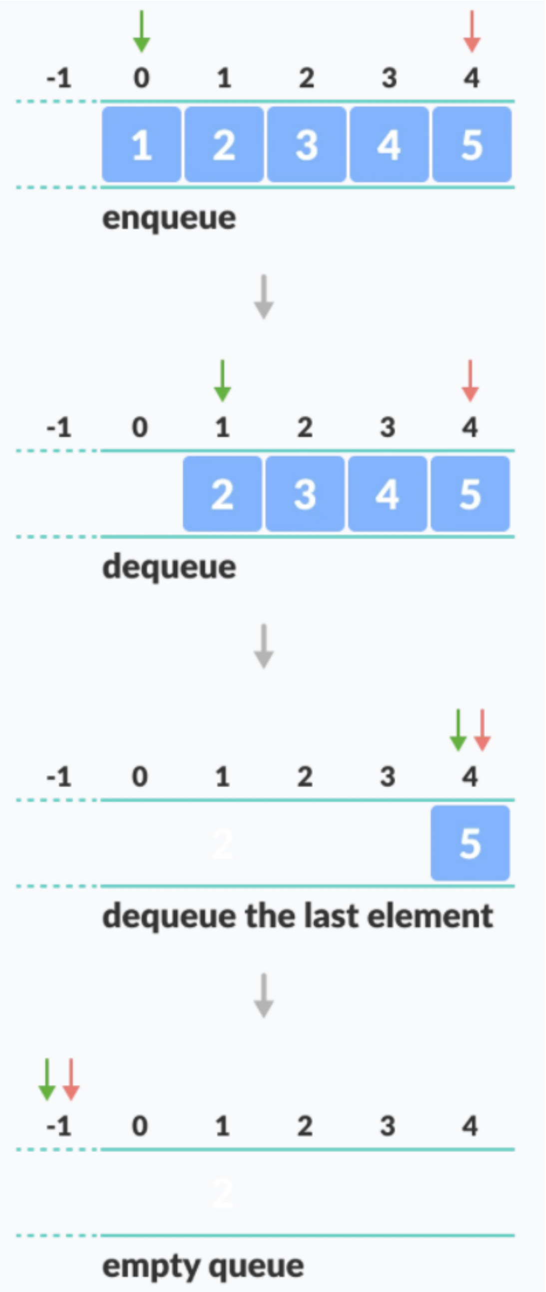
Enqueue

- For the first element, set the value of FRONT to 0
- Increase the REAR index by 1
- Add the new element in the position pointed to by REAR



Dequeue

- Check if the queue is empty
- Return the value pointed by FRONT
- Increase the FRONT index by 1
- For the last element, reset the values of FRONT and REAR to -1



Queue implementation – Python

- We usually use arrays to implement queues in Java and C/++. In the case of Python, we use lists.

```
class Queue:

    def __init__(self):
        self.queue = []

    # Add an element
    def enqueue(self, item):
        self.queue.append(item)

    # Remove an element
    def dequeue(self):
        if (self.isEmpty()):
            return "underflow"
        return self.queue.pop(0)

    def isEmpty(self):
        return len(self.queue) == 0

    # Display the queue
    def display(self):
        print(self.queue)
```

List Implementation - Python

```
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.display()
q.dequeue()
print("After removing an element")
q.display()
```

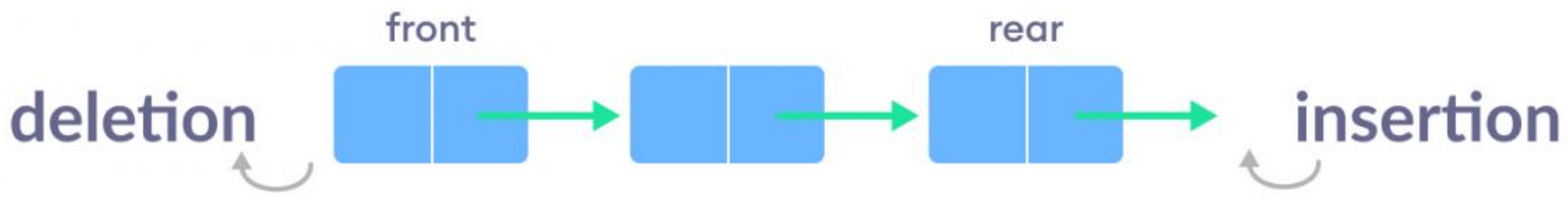
```
[1, 2, 3, 4]
After removing an element
[2, 3, 4]
```

Types of Queues

- There are four different types of queues:
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Double Ended Queue

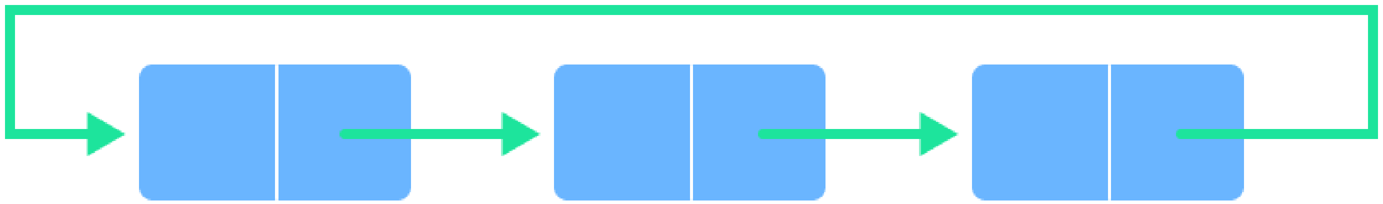
Simple Queue

- In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



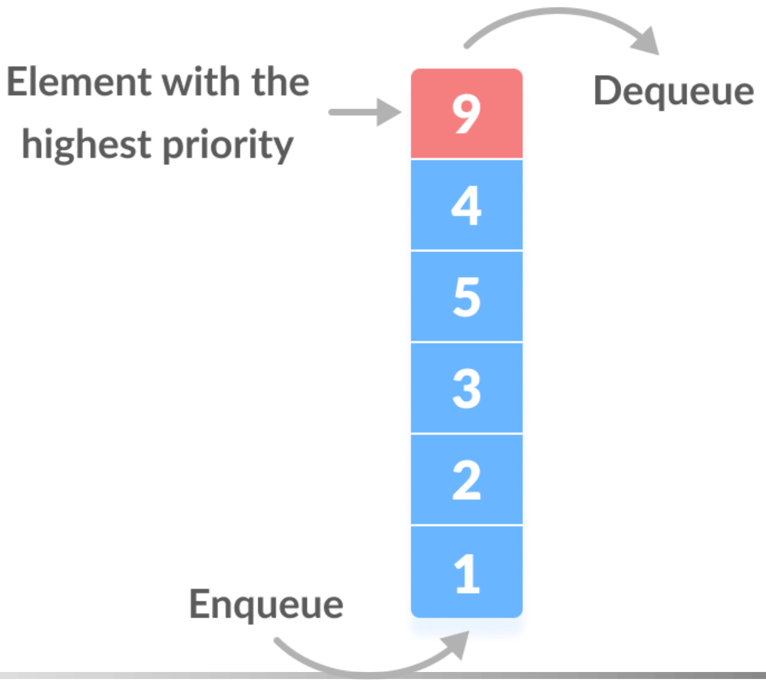
Circular Queue (circular buffer)

- In a circular queue, the last element points to the first element making a circular link.
  - The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position.



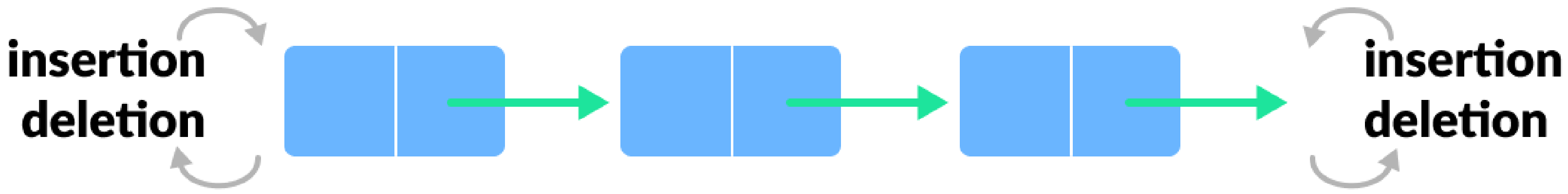
Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first.



Double Ended Queue

- In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



Linked List

- A linked list is a linear data structure that includes a series of connected nodes.
- Unlike an array, the order in a linked list is determined by a pointer in each object.
- Linked lists can be of multiple types: singly, doubly, and circular linked list.
  - In this Lecture, we will focus on the doubly linked list.

Doubly Linked List

- A doubly linked list is a type of linked list in which each node consists of 3 components:
  - prev - address of the previous node
  - data - data item
  - next - address of next node

- head points to the first node of the linked list.
- next pointer of the last node is NULL.

Operations of Linked List

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list

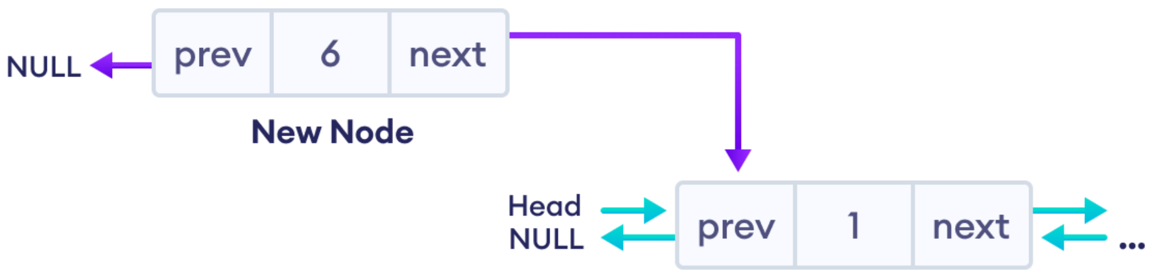
Insertion on a Doubly Linked List

- We can insert elements at 3 different positions of a doubly-linked list:
  - Insertion at the beginning
  - Insertion in-between nodes
  - Insertion at the End

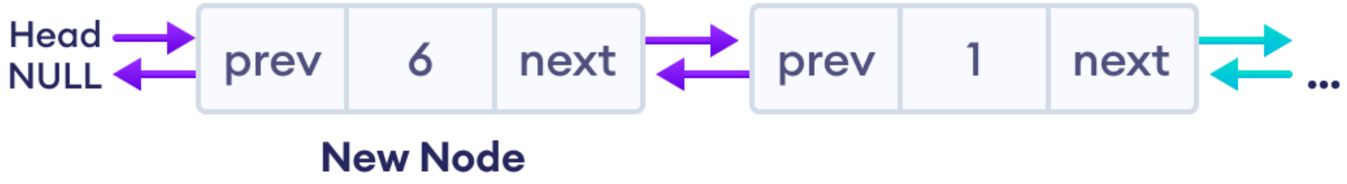
Insertion at the Beginning

- 1. Create a new node

- 2. Set prev and next pointers of new node



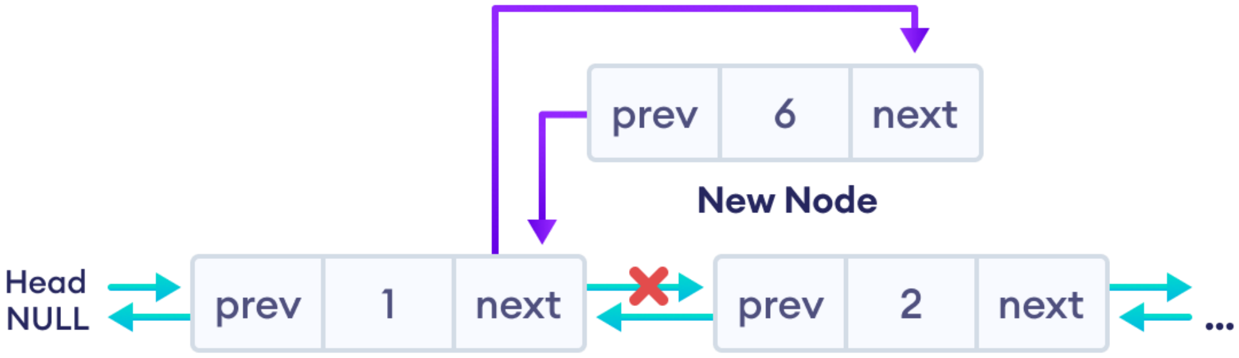
- 3. Make new node as head node



Insertion in between two nodes

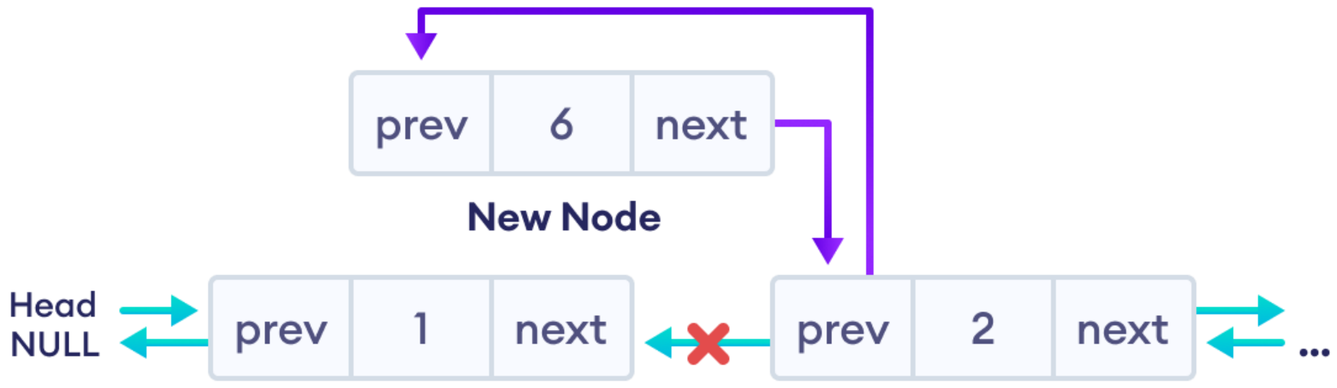
- 1. Create a new node

- 2. Set the next pointer of new node and previous node

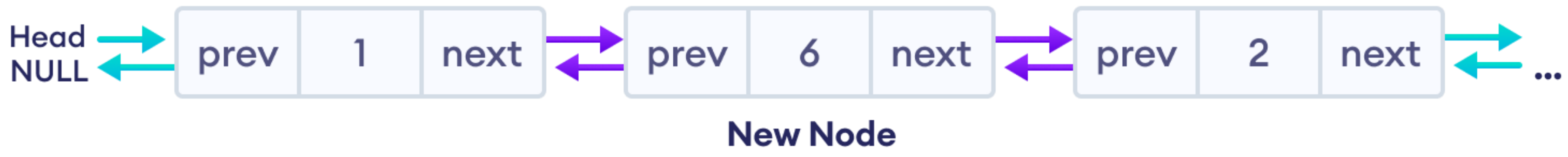


Insertion in between two nodes

- 3. Set the prev pointer of new node and the next node



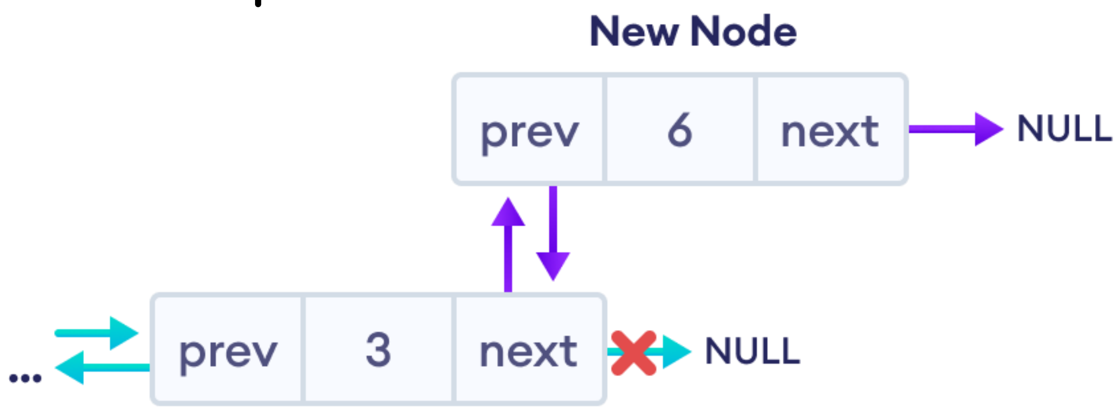
- The final doubly linked list is after this insertion is:



Insertion at the End

- 1. Create a new node

- 2. Set prev and next pointers of new node and the previous node



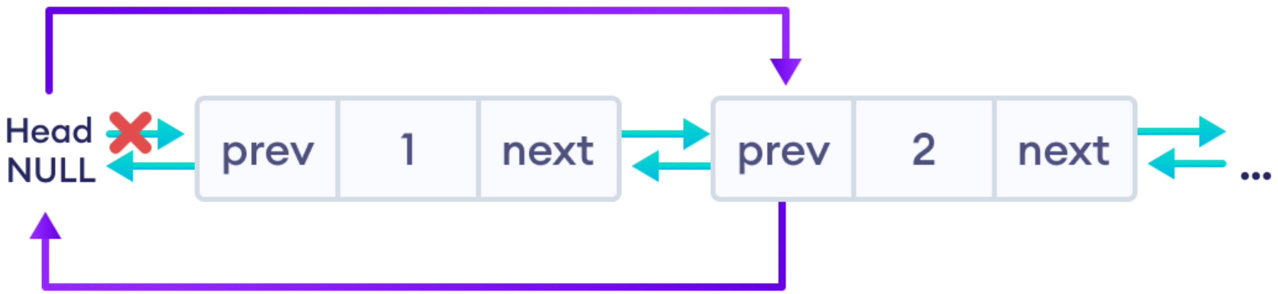
- The final doubly linked list looks like this.

Deletion from a Doubly Linked List

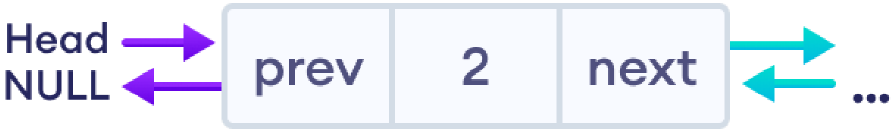
- Similar to insertion, we can also delete a node from 3 different positions of a doubly linked list.
  - 1. Delete the First Node of Doubly Linked List
  - 2. Deletion of the Inner Node
  - 3. Delete the Last Node of Doubly Linked List

Delete the First Node of Doubly Linked List

- If the node to be deleted (i.e. del_node) is at the beginning.
  - Reset value node after the del_node (i.e. node two)



  - Free the memory of del_node, and the linked list will look like this.

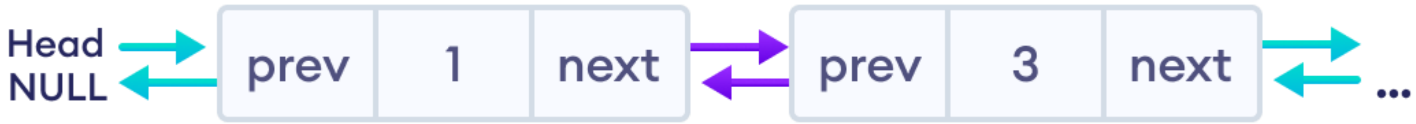


Free the space of the first node

Deletion of the Inner Node

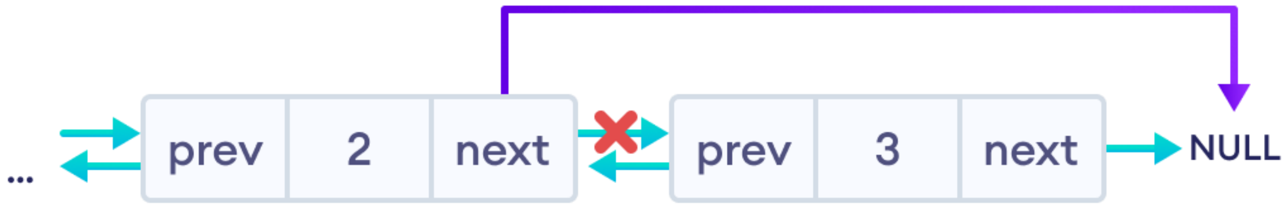
- If del_node is an inner node
  - reset the value of next and prev of the nodes before and after the del_node.

  - Free the memory of del_node, and the linked list will look like this.

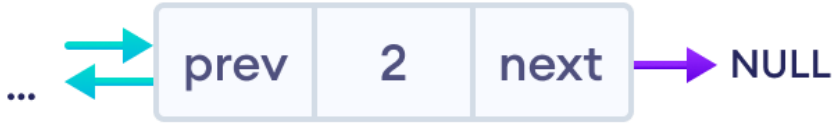


Delete the Last Node of Doubly Linked List

- If the node to be deleted (i.e. del_node) is at the End.
  - simply delete the del_node and make the next of node before del_node point to NULL.

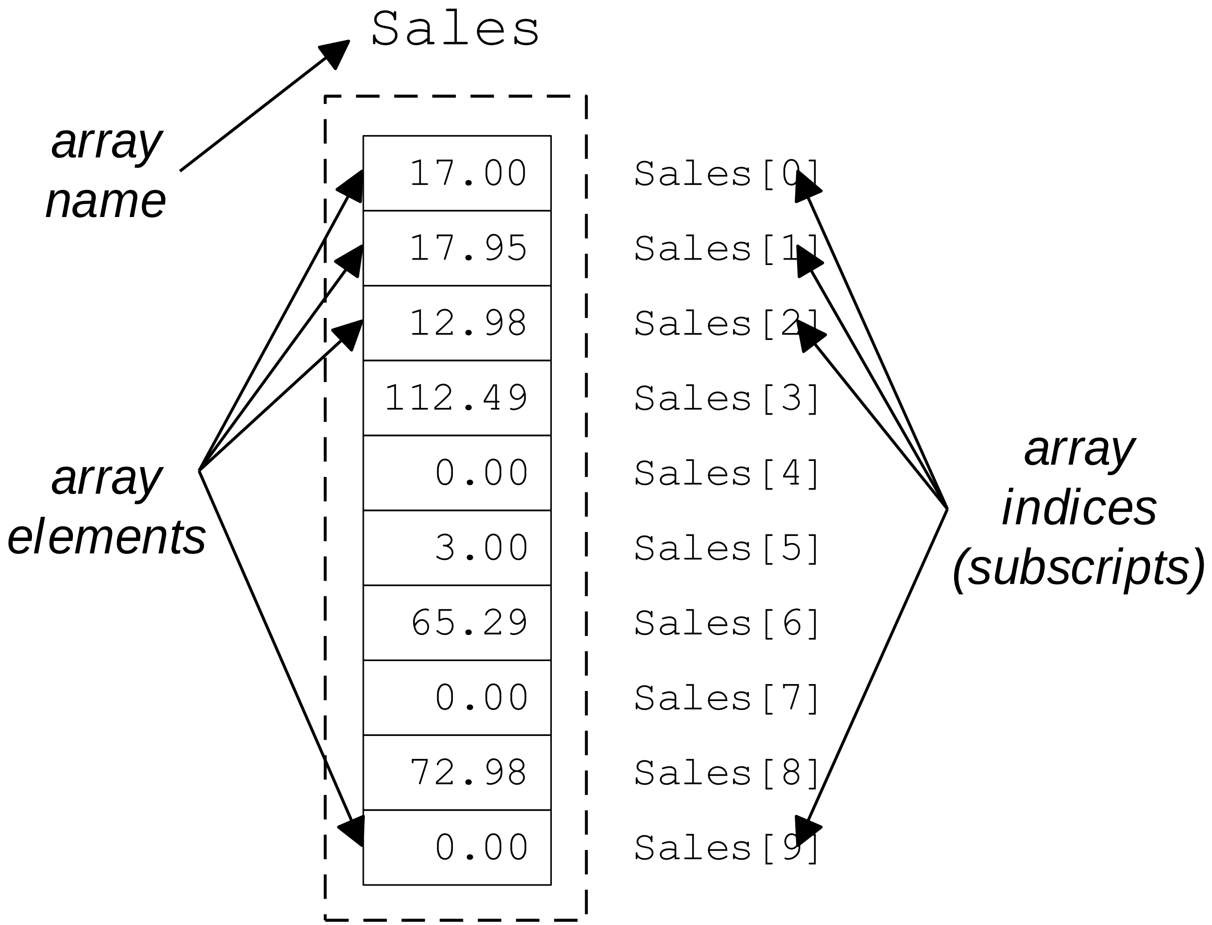


  - Free the memory of del_node, and the linked list will look like this.



Array

- Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index.



Array vs. Linked List

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

Singly Linked List

Singly Linked List vs. doubly Linked List

	Singly Linked List (SLL)	Doubly Linked List (DLL)
Fields	data and next.	data, prev and next.
Traversal direction	one direction	both directions
Memory	Less memory	More memory (3 fields)
Complexity of: insertion and deletion at a given position	O(n)	O(n)
Find next element	O(1)	O(1)
Find previous element	O(n)	O(1)
deletion with a given node	O(n)	O(1)
Insert a new node before a given node	O(n)	O(1)

Tree Data Structure

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

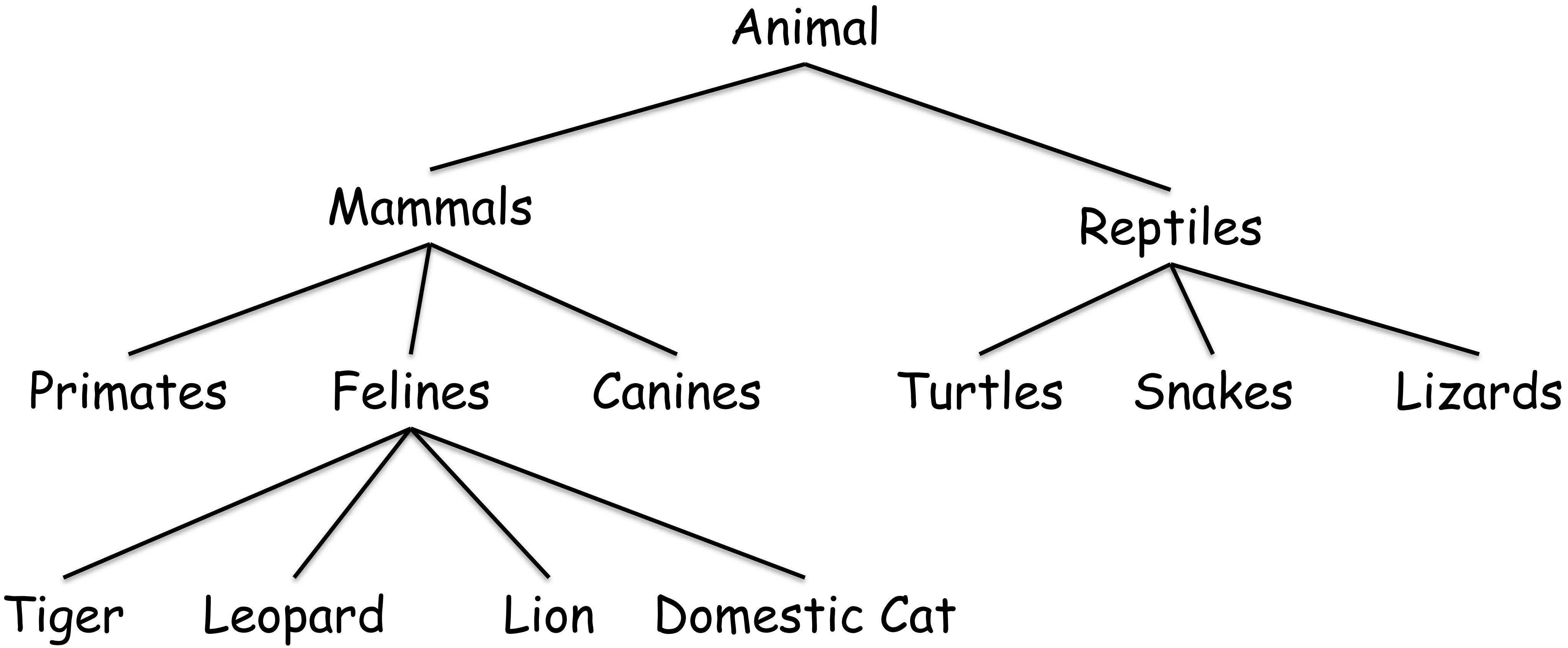
- Why tree data Structure?
  - Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
  - Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree

- Some data are not linear (it has more structure!)
  - Family trees
  - Organizational charts
  - …

- Trees offer an alternative
  - Representation
  - Implementation strategy
  - Set of algorithms

Example: Taxonomy Tree



How many types of tree are there?

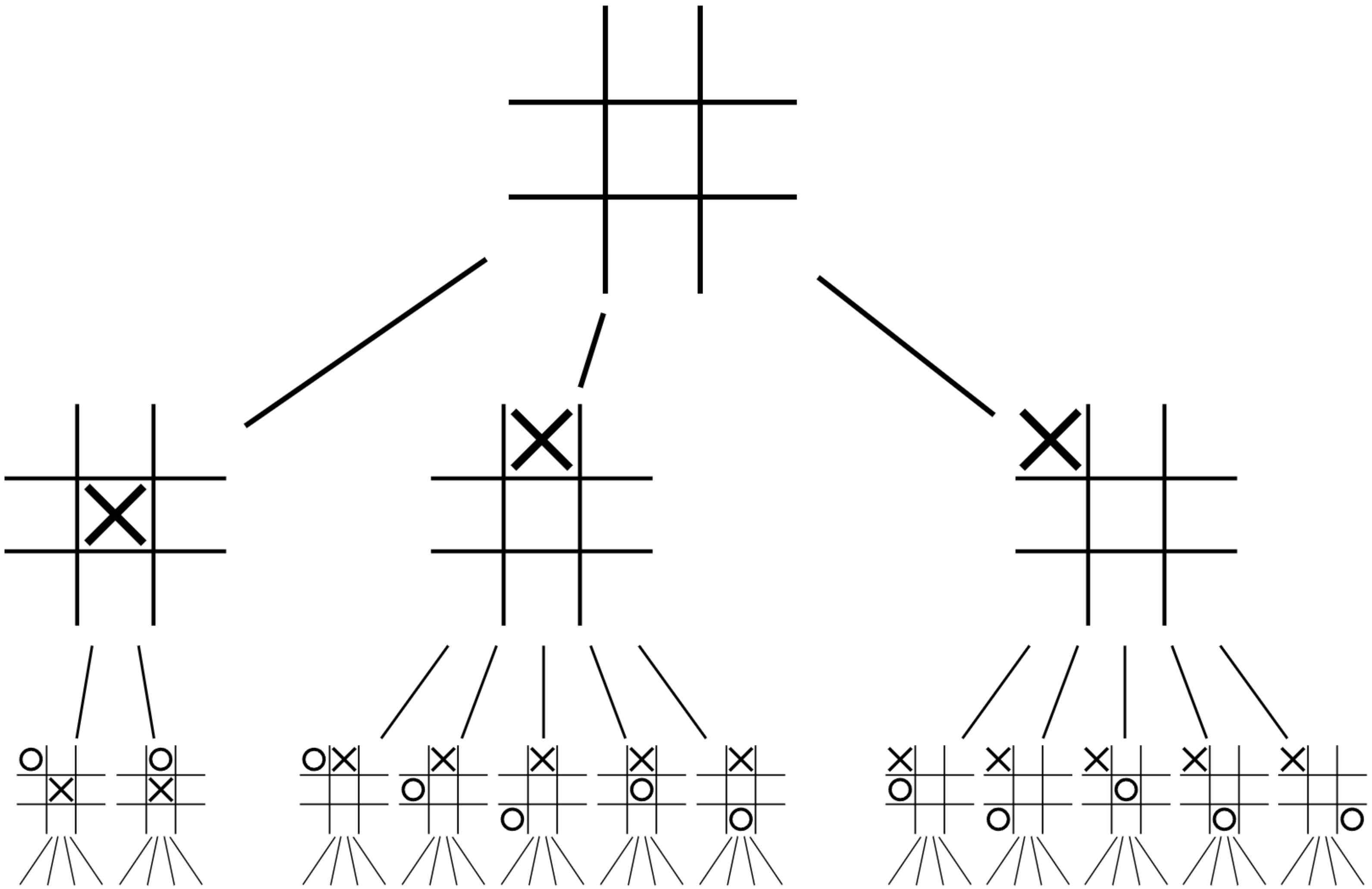
- Far too many:
  - General Tree
  - Binary Tree
  - Red-Black Tree
  - AVL Tree
  - B+ Tree
  - …
- Different types are used for different things
  - To improve speed
  - To improve the use of available memory
  - To suit particular problems

What is a tree useful for?

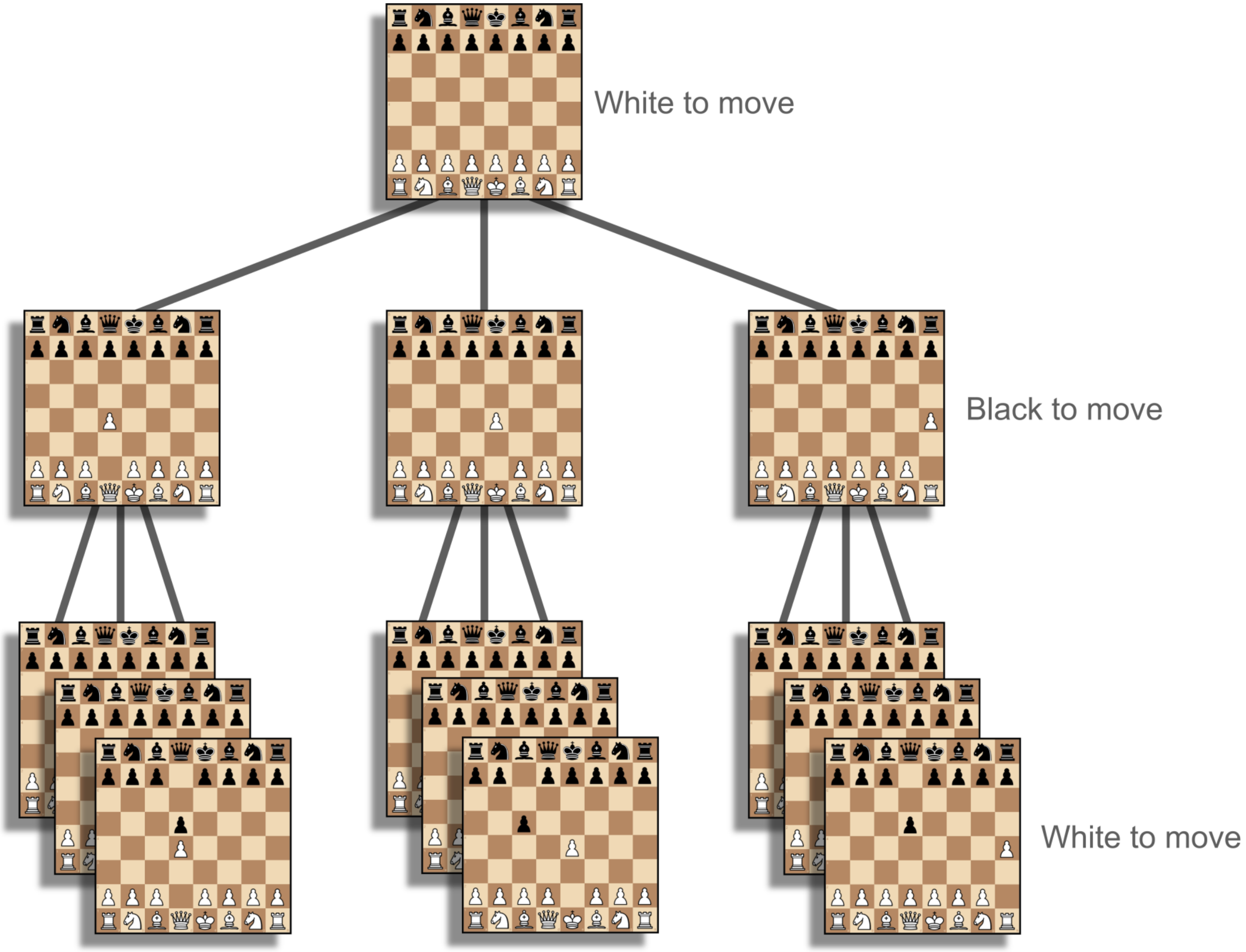
- Artificial Intelligence – planning, navigating, games

- Representing things:
  - Simple file systems
  - Class inheritance and composition
  - Classification, e.g., taxonomy
  - HTML pages
  - Parse trees for languages
  - Essential in compilers like java
  - Etc.

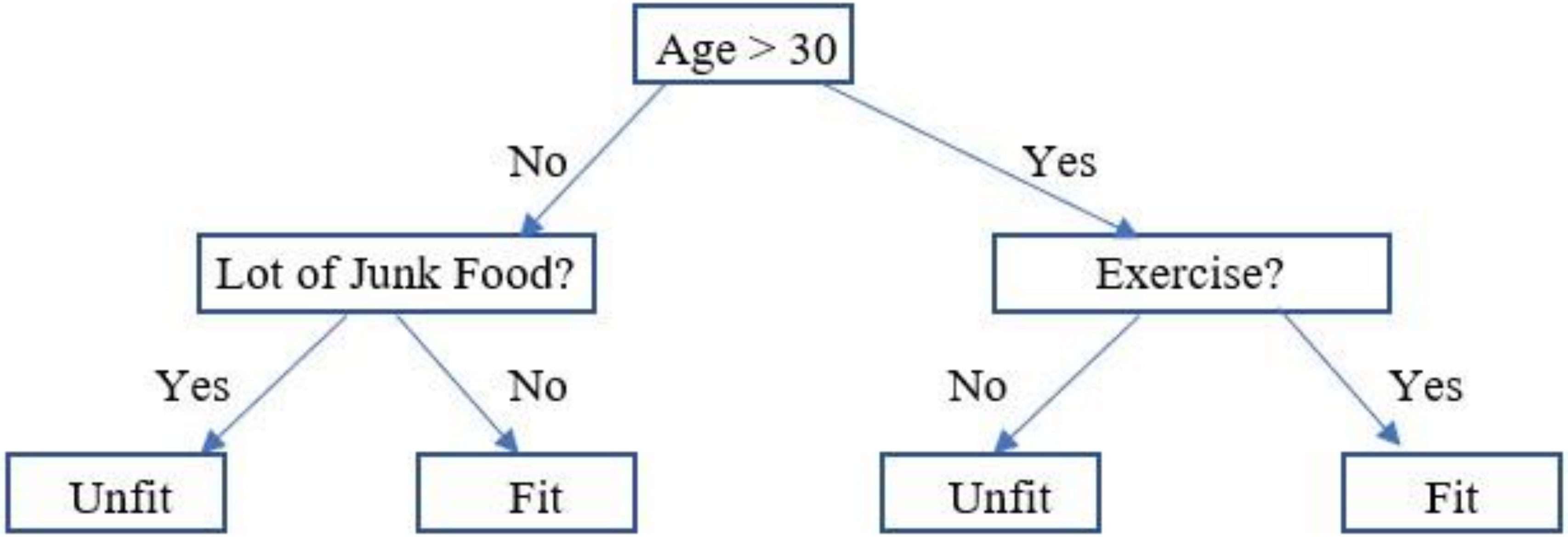
Example: Tic Tac Toe



Example: Chess



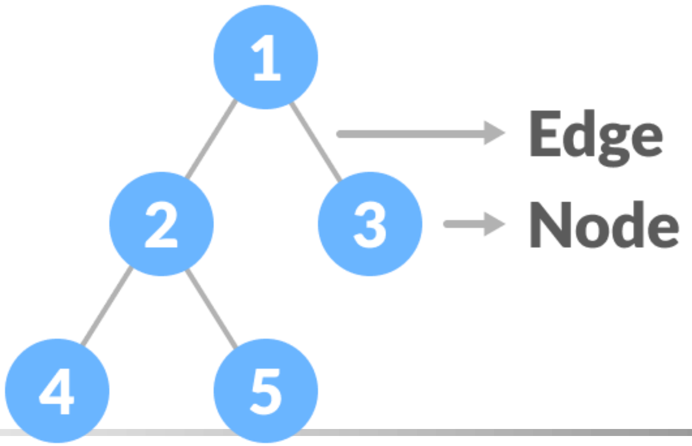
Example: Decision Tree



Tree Terminologies

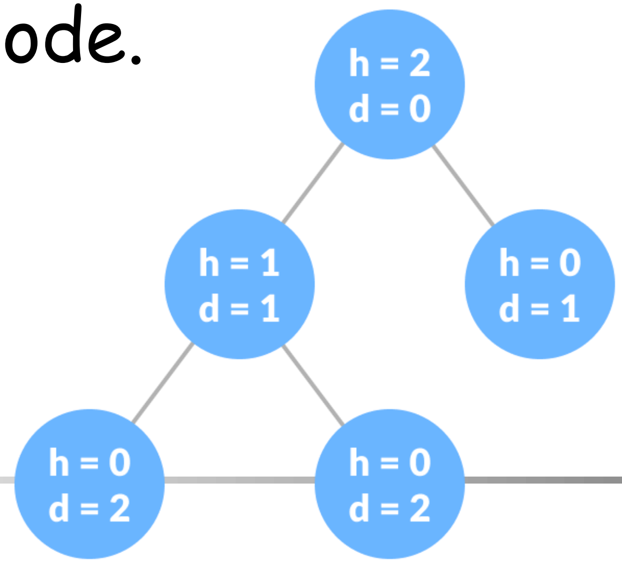
- Node
  - A node is an entity that contains a key and pointers to its child nodes.
  - The last nodes of each path are called leaf nodes or external nodes that do not contain a pointer to child nodes.
  - The node having at least a child node is called an internal node.

- Edge is the link between any two nodes.



Tree Terminologies

- Root is the topmost node of a tree
- The Height of a node is the number of edges from the node to the deepest leaf
- The Depth (or Level) of a Node is the number of edges from the root to the node.
- The height of a Tree is the height of the root node or the depth of the deepest node.



Tree Terminologies

- The degree of a node is the total number of branches of that node.
  - Bianry tree: each node has at most two children.

- A collection of disjoint trees is called a forest.

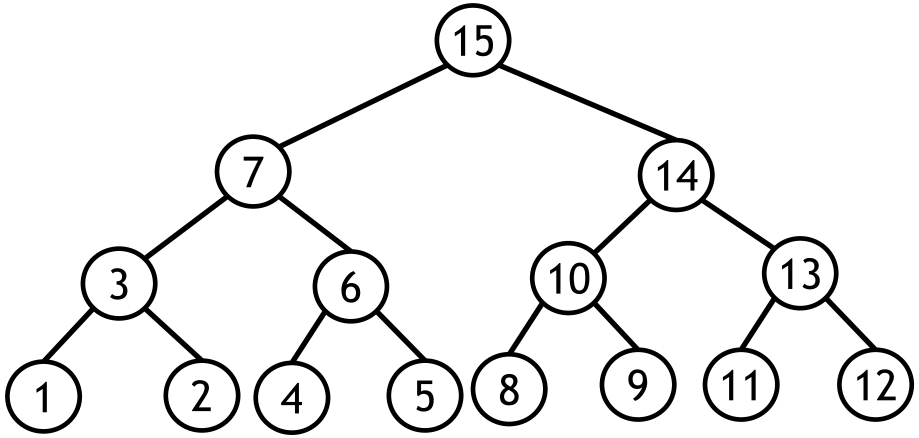
Binary Tree

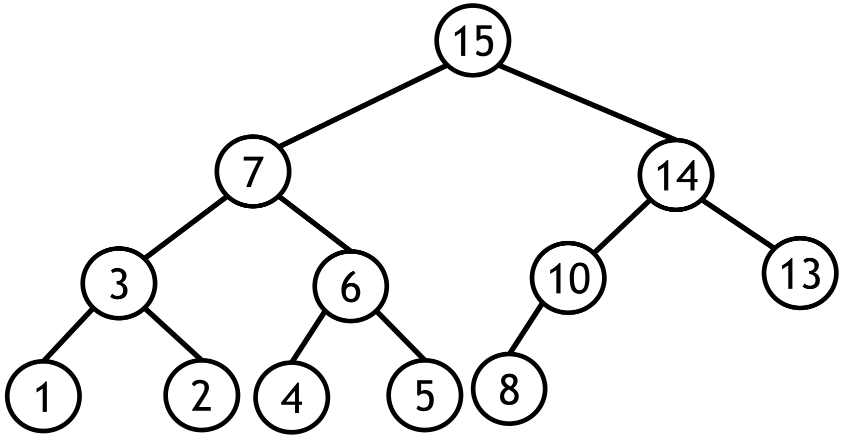
- Binary tree is a tree data structure, each node has at most two children (*left child* and *right child*).

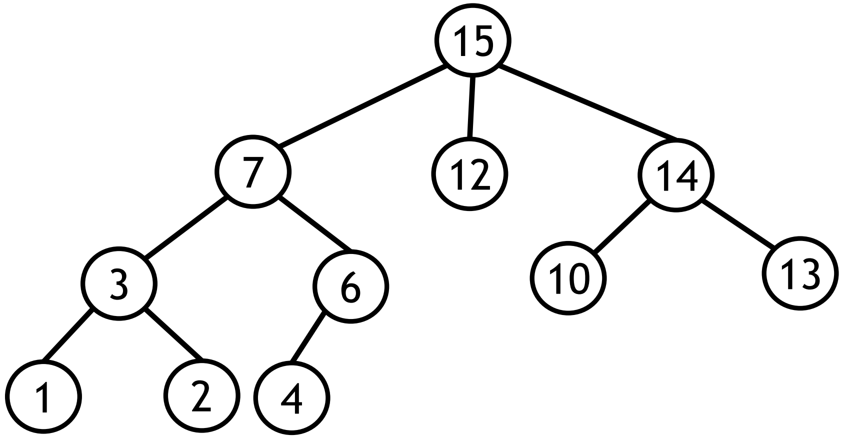
- Complete (perfect) binary tree
  - all interior nodes have two children
  - all leaves have the same depth or same level.

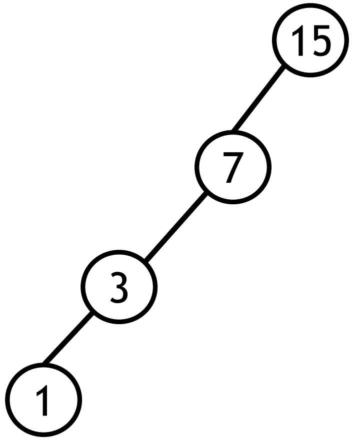
- nearly complete binary tree
  - every level, except possibly the last, is completely filled.
  - all nodes in the last level are as far left as possible.

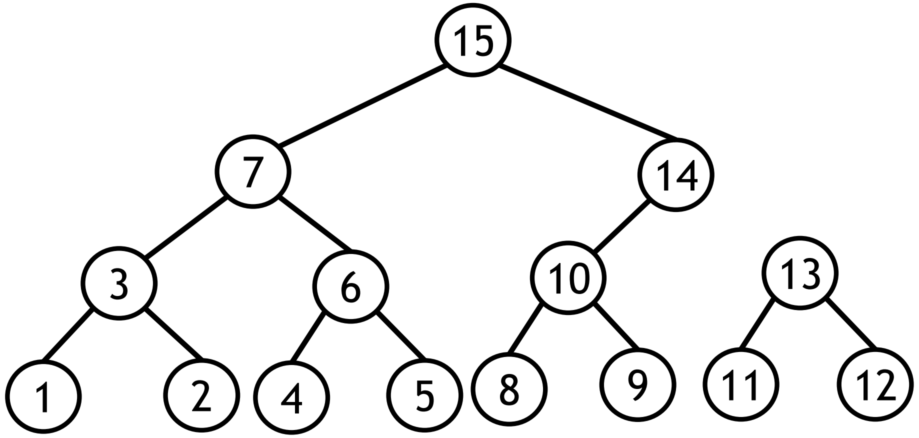
Is this Binary Tree?







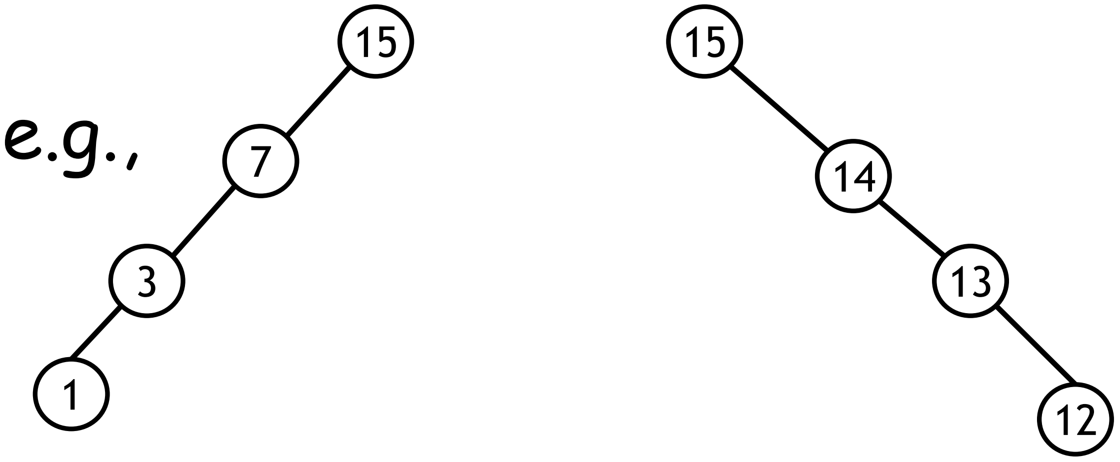




Worst Case

- Operations can degenerate to O(n) – worst case!

- Degenerates to a linked list, e.g.,
  - All nodes are to the left
  - All nodes are to the right



- Balance the tree to guarantee that height is O(logn).

Learning outcomes

- Stack
- Queue
- LinkedList
  - Doubly Linked List
- Tree
  - Binary Tree