Python for Data Scientists
L10: Non-Linear Data Structures

Shirin Tavara

Outline of the lecture

Recap

- Stack and Queue

Local search

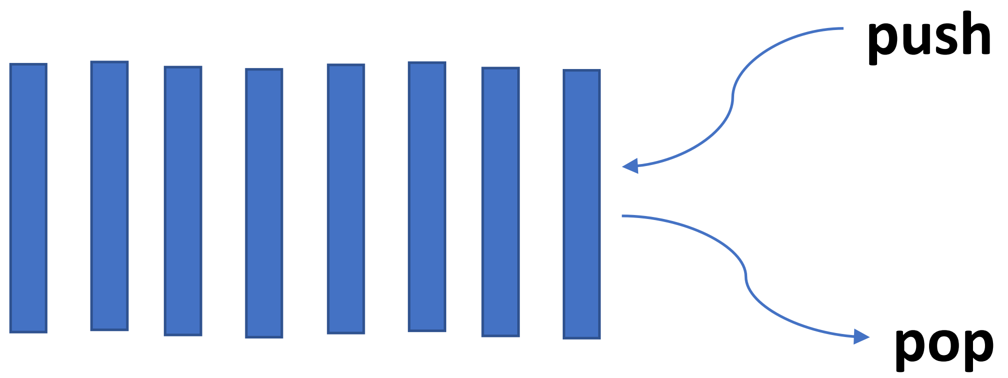
- Using arrays and linked lists

Non-linear data structures

- Graphs and trees

Stack

- Last-In-First-Out (LIFO)
- Has Push(key), Pop(), Top(), Size(), Get(), and IsEmpty() operations.



Stack implementation using list

```
stack = []

# pushing elements in the stack
stack.append(‘1')
stack.append(‘2')
stack.append(‘3')

print(stack)

# popping elements from stack LIFO
print('pop first element',stack.pop())
print('pop second element',stack.pop())
print('pop third element',stack.pop())

print(stack)
```

The items in list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently hold it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

```
>> [‘1’, ‘2', ‘3’]

>> pop first element 3
>> pop second element 2
>> pop third element 1

>>[]
```

Stack implementation using list

Stack class

```
class Stack:
    def __init__(self):
        self.items = ?

    def is_empty(self):
        return self.items == ?

    def push(self, data):
        ?

    def pop(self):
        return ?
```

```
if __name__=='__main__':


    stack = Stack()
    stack.push('1')
    stack.push('2')
    stack.push('3')

    stack.pop()
    stack.pop()
    stack.pop()
```

Stack implementation using collections.deque

```python
from collections import deque

stack = deque()

# pushing elements in the stack
stack.append('1')
stack.append('2')
stack.append('3')

print(stack)

# poping elements from stack LIFO
print(‘pop first element',stack.pop())
print(‘pop second element',stack.pop())
print(‘pop third element',stack.pop())

print(stack)
```

Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.

```
>> deque(['1', '2', '3’])

>> pop first element 3
>> pop second element 2
>> pop third element 1

>> deque([])
```

Stack implementation using queue module

```
from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize = 3)

# qsize() show the number of elements
in the stack
print(stack.qsize())

# pushing elements in the stack
stack.put('1')
stack.put('2')
stack.put('3')

#Return True if there are maxsize
items in the queue.
print("Full: ", stack.full())
print("Size: ", stack.qsize())
```

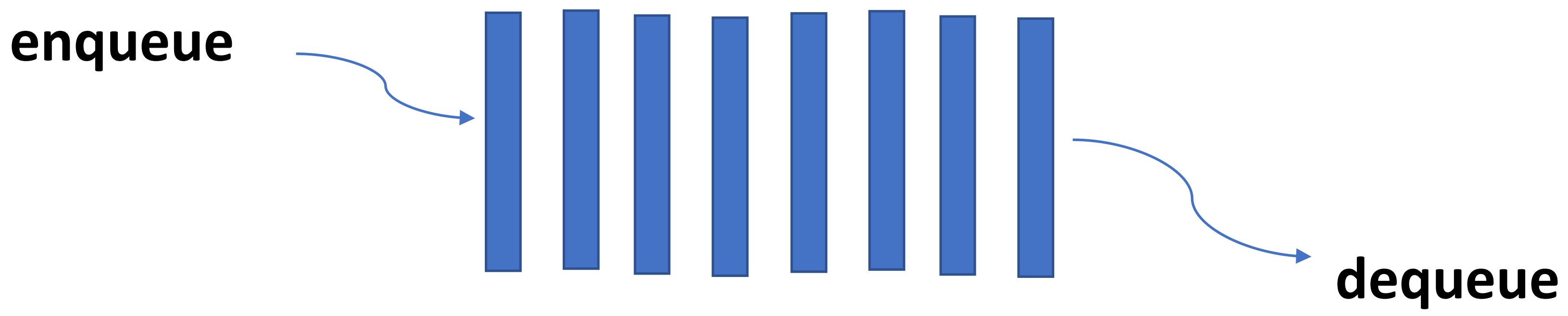
```
# popping elements from stack LIFO
print(‘pop first element',stack.get())
print(‘pop second element',stack.get())
print(‘pop third element',stack.get())

# return True if the queue is empty,
False if not
print("Empty: ", stack.empty())
```

```
>> 0
>> Full:  True
>> Size:  3
>> pop first element 3
>> pop second element 2
>> pop third element 1
>> Empty:  True
```

Queue

- It stores items in a First-In/First-Out (FIFO) manner
- Some of the operations are enqueue() and dequeue()



Queue implementation using list

```python
queue = []

# Adding elements to the queue
queue.append('1')
queue.append('2')
queue.append('3')

print(queue)

# Removing elements from the queue
print('dequeue first element', queue.pop(0))
print('dequeue second element', queue.pop(0))
print('dequeue third element', queue.pop(0))

print(queue)
```

```
>> ['1', '2', '3']

>> dequeue first element 1
>> dequeue second element 2
>> dequeue third element 3

>> []
```

Queue implementation using collections.deque

```
from collections import deque

q = deque()

# Adding elements to a queue
q.append('1')
q.append('2')
q.append('3')

print(q)

# Removing elements from a queue
print('dequeue first element', q.popleft())
print('dequeue second element',
q.popleft())
print('dequeue third element', q.popleft())

print(q)
```

>> deque(['1', '2', '3’])

>> dequeue first element 1
>> dequeue second element 2
>> dequeue third element 3

>> deque([])

Queue implementation using queue module

```
from queue import Queue

q = Queue(maxsize = 3)

print(q.qsize())                              >> 0

# Adding of element to queue
q.put('1')
q.put('2')
q.put('3')

print("Full: ", q.full())                     >> Full:  True

# Removing element from queue
print('dequeue first element', q.get())       >> dequeue first element 1
print('dequeue second element',               >> dequeue second element 2
q.get())
print('dequeue third element', q.get())       >> dequeue third element 3
```

Non-Linear Data Structures

Why we need non-linear data structures?

What are the examples that we can solve using a non-linear data structure?

Local Search Problems

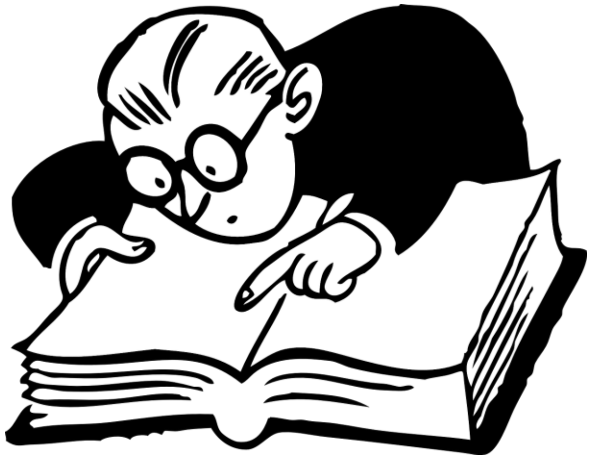
Dictionary search:

We want to find all the words with letter "a".

Local Search Problems

Range searches:

- Find a prime number between 2 and 20
- Find all emails received in July



Local Search Problems

Nearest Neighbor:

Find the house in your neighborhood which has the height closest to your house.

Local Search

We want to have a data structure that can

- Store elements with keys from a linearly ordered set.

  Examples of such sets:

  - A word sorted by alphabetical order
  - A date or height

- Support operations
  - RangeSearch(a,b): returns all elements whose keys are between a and b.
  - NearestNeighbour(c): returns all elements closest to c on either side in the data structure.

Local Search - Example

5	8	12	23	39	41	46	59	75

RangeSearch(10,50):

10 <   < 50

5	8	12	23	39	41	46	59	75

NearestNeighbour(30):

30

5	8	12	23	39	41	46	59	75

Local Search – Desired Property

We want to have a data structure that is **Dynamic.**

We want the possibility to modify the data structure and have the support for operations such as

- add(a)/insert(a): Adds an element with key "a"
- remove(b)/delete(b): deletes the element with key "b"

Local Search - Example

5	8	12	23	39	41	46	59	75

Insert(24):

5	8	12	23	24	39	41	46	59	75

delete(41):

5	8	12	23	24	39	46	59	75

Question – Poll

Given the following queries for an empty data structure:

insert(3),

inset(13),

insert(32),

insert(9),

delete(13),

insert(20),

What will be the result of NearestNeighbour(10)?

Question – Poll

Given the following queries for an empty data structure:

insert(3),

inset(13),

insert(32),

insert(9),

delete(13),

insert(20),

What will be the result of NearestNeighbour(10)?

3	9	~~13~~	20	32

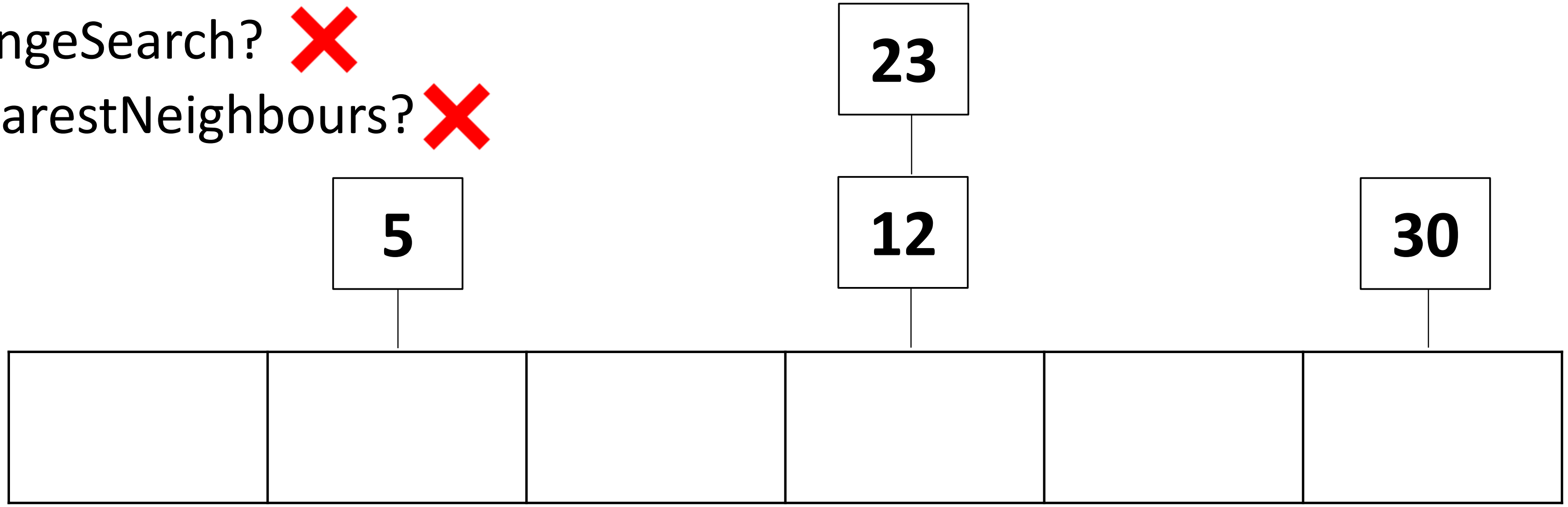
Local Search using Hash Tables

Storing and looking up the elements are very quick in hash tables

- Insert -> $O(1)$ ✓
- delete -> $O(1)$ ✓

What about searching?

- RangeSearch? ✗
- NearestNeighbours? ✗



Local Search using Arrays

Searches in an array: Possible but slow!

- RangeSearch:
  - Scan through the array and find the elements in the range that we want
  - e.g., RangeSearch(6,22)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: O(n)
  - e.g., RangeSearch(6,22)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: $O(\text{n})$
- NearestNeighbour:
  - Like the range search, we need to scan through the entire array
  - e.g., NearestNeighbour(6)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: $O(n)$
- NearestNeighbour: $O(n)$
  - e.g., NearestNeighbour(6)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: $O(\text{n})$
- NearestNeighbour: $O(\text{n})$
- Insert:
  - E.g., insert(9)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: $O(n)$
- NearestNeighbour: $O(n)$
- Insert: $O(1)$
  - If we have an expandable array, we can add elements to it
  - E.g., insert(9)

8	**12**	**5**	**20**	**1**	**23**	**9**

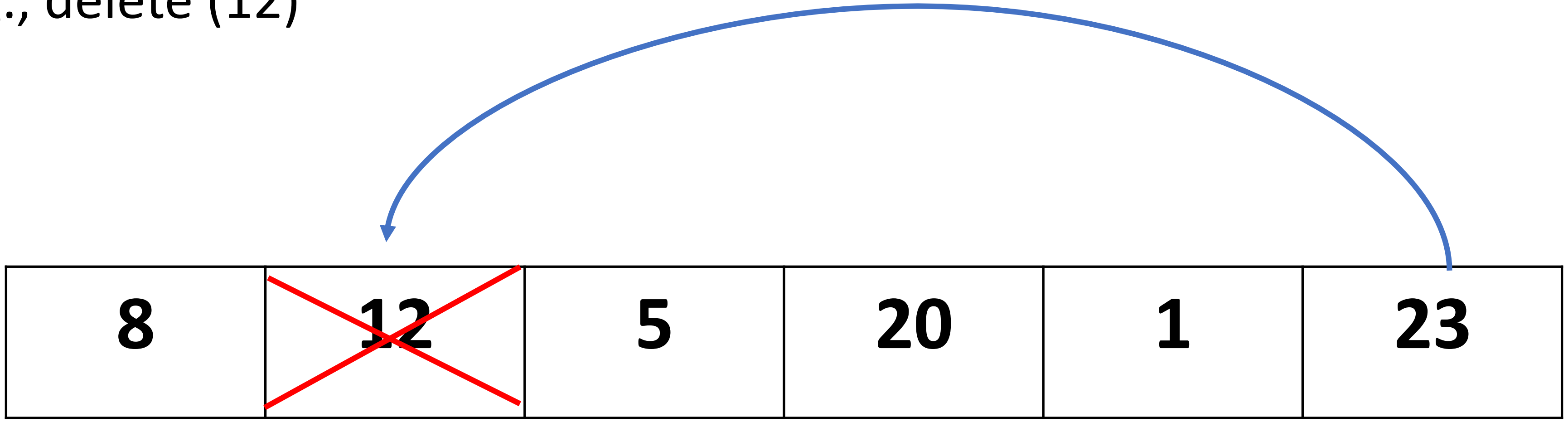
Local Search using Arrays

- RangeSearch: $O(n)$
- NearestNeighbour: $O(n)$
- Insert: $O(1)$
- Delete:
  - Deleting will leave a gap, but we can do it in $O(1)$ by moving the last element into the gap
  - E.g., delete (12)

8	**12**	**5**	**20**	**1**	**23**

Local Search using Arrays

- RangeSearch: $O(n)$
- NearestNeighbour: $O(n)$
- Insert: $O(1)$
- Delete: without find and shifting $O(1)$
  - E.g., delete (12)



Local Search using sorted Arrays

- It allows us to do a binary search
- For a range search:
  - Binary search to find the left end of the range
  - Scan through to find the right end of the range
  - Return everything in the middle

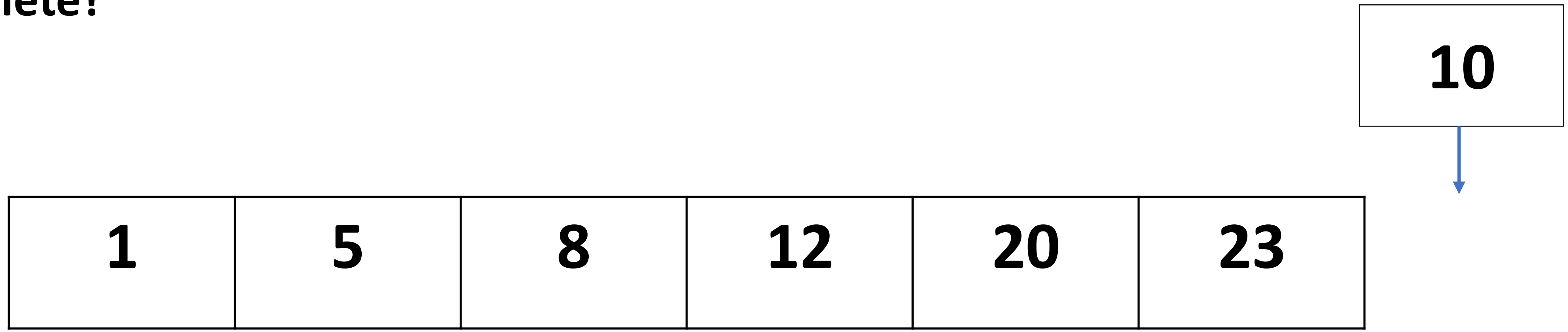
- For nearest neighbor: (in a similar manner)
  - Binary search to find what we want
  - Return the elements on either side of the query

Local Search using sorted Arrays

	Array	Sorted Array
• RangeSearch:	$O(n)$	$O(\log(n))$
• NearestNeighbour:	$O(n)$	$O(\log(n))$

- What about updates in the sorted array?
  - **Insert?** The array still needs to remain sorted! This may destroy the sorted order!
  - **Delete?**

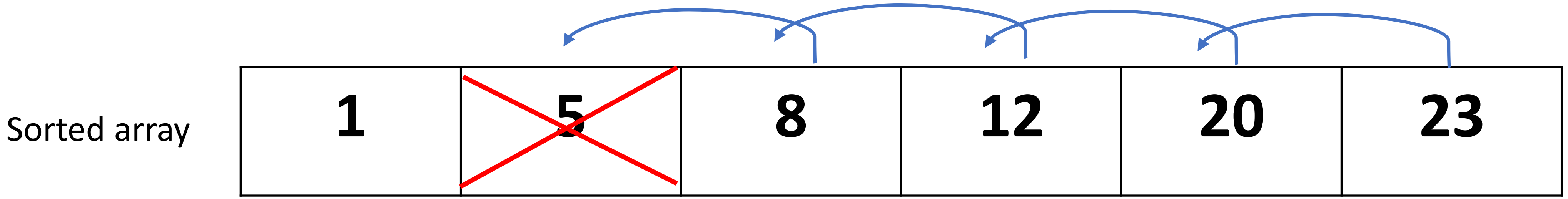
Sorted array



Local Search using sorted Arrays

	Array	Sorted array
• RangeSearch:	$O(\text{n})$	$O(\log(\text{n}))$
• NearestNeighbour:	$O(\text{n})$	$O(\log(\text{n}))$
• Insert:	$O(1)$	$O(\text{n})$

- Delete?
  - It will leave a gap and we need to fill it! We cannot just move the last element into the gap since it will destroy the sorted order!



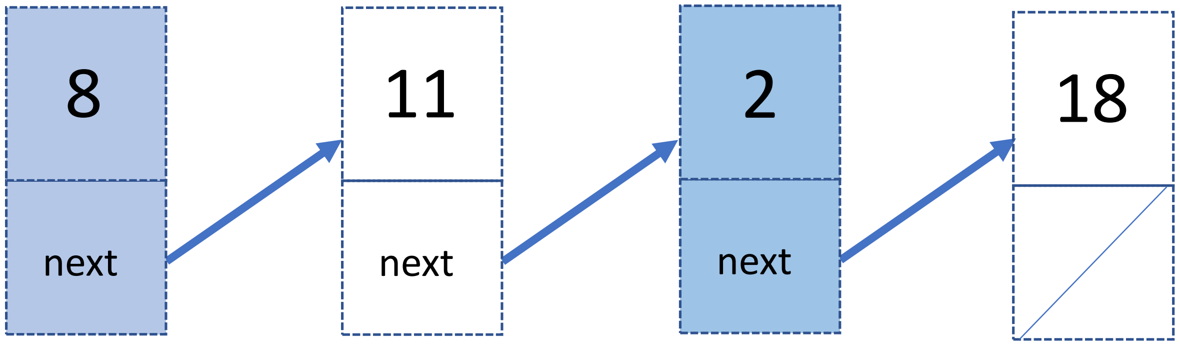
Local Search using sorted Arrays

	Array	Sorted array
• RangeSearch:	$O(n)$	$O(\log(n))$
• NearestNeighbour:	$O(n)$	$O(\log(n))$
• Insert:	$O(1)$	$O(n)$
• Delete:	$O(1)$	$O(n)$

Local Search using Linked List

- RangeSearch: $O(n)$
  - Scan through the list
- NearestNeighbour:
  - Scan through the list
- Insert:
- Delete:

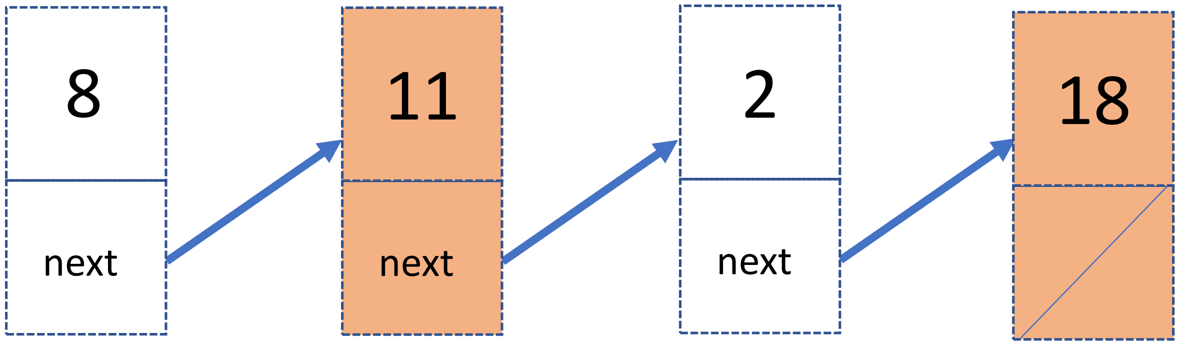
RangeSearch(1,10)



Local Search using Linked List

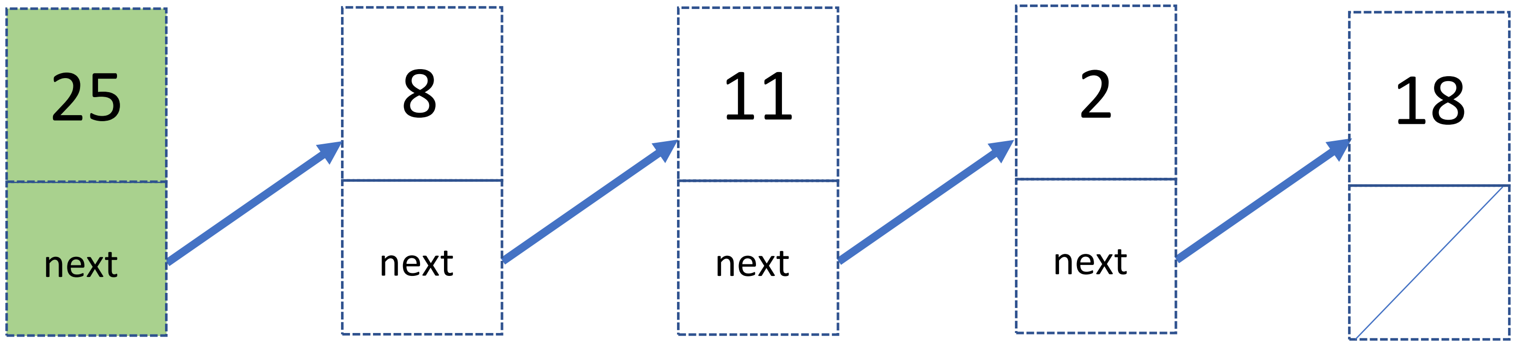
- RangeSearch: $O(n)$
  - Scan through the list
- NearestNeighbour: $O(n)$
  - Scan through the list
- Insert:
- Delete:

NearestNeighbour(16)



Local Search using Linked List

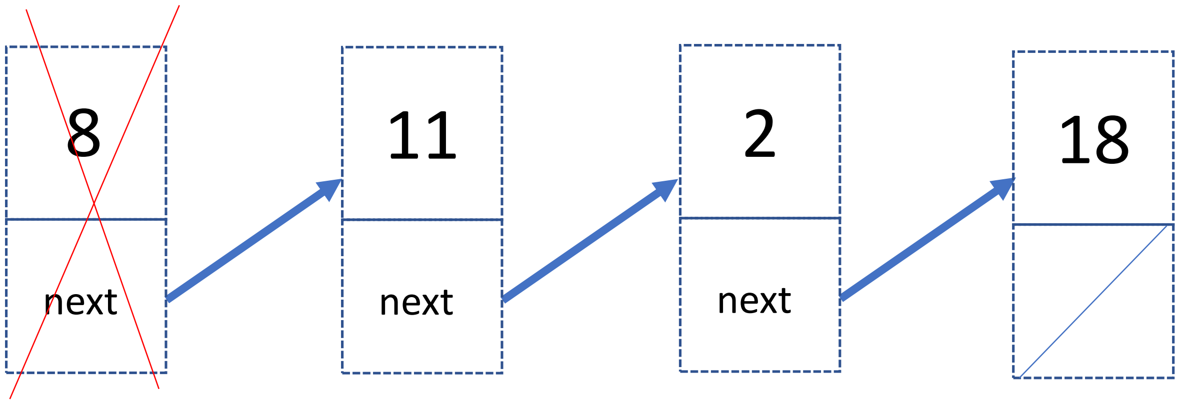
- RangeSearch: $O(n)$
  - Scan through the list $O(n)$
- NearestNeighbour:
  - Scan through the list
- Insert: $O(1)$
- Delete:



Local Search using Linked List

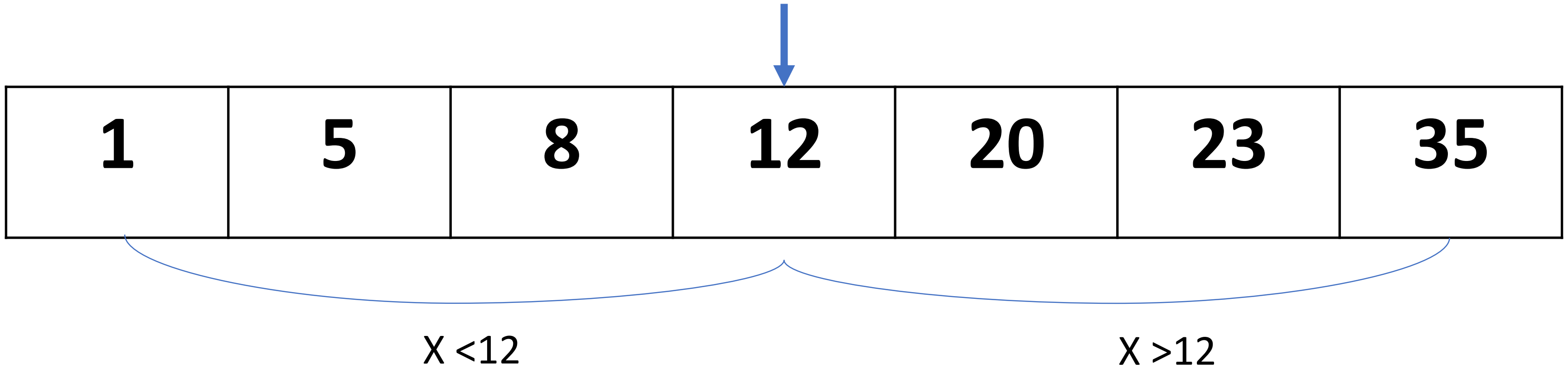
- RangeSearch: $O(n)$
  - Scan through the list $O(n)$
- NearestNeighbour:
  - Scan through the list
- Insert: $O(1)$
- Delete: $O(1)$

Searches are slow and we cannot do binary searches in linked lists!



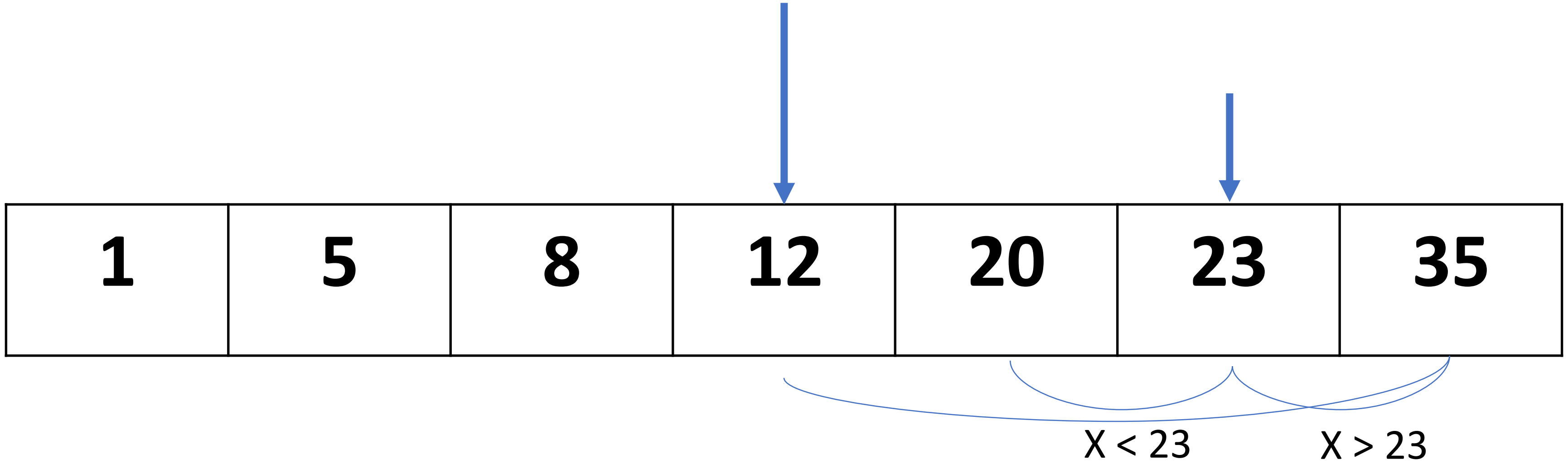
Binary Search

- We want a data structure for a local search problem
- Array and linked list were not suitable
- Search in a sorted array is fast, but updates are not!



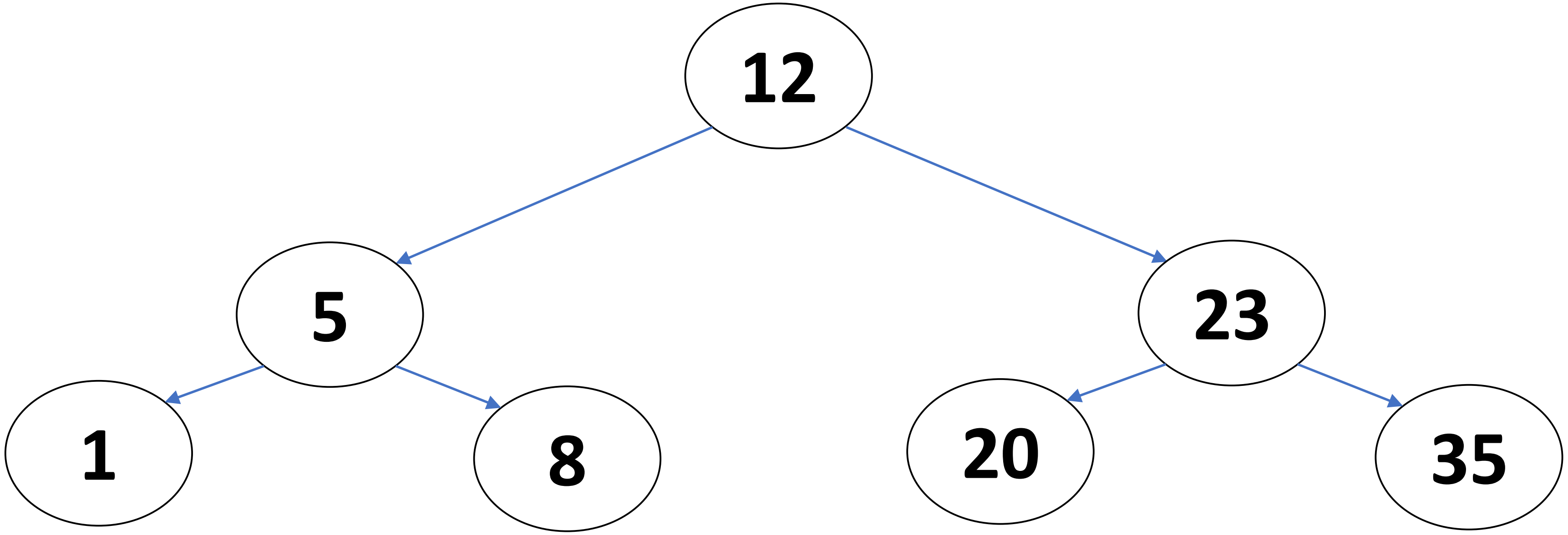
Binary Search

Binary search in a sorted array



Binary Search Tree

Search in a binary search tree is as good as a sorted array, but it is easier to insert into.



1	5	8	12	20	23	35

Non-linear data structure

- The data items are not organized sequentially; Elements could be connected to more than one element to reflect a special relationship among these items.

- It can not be traversed during a single run

Graphs

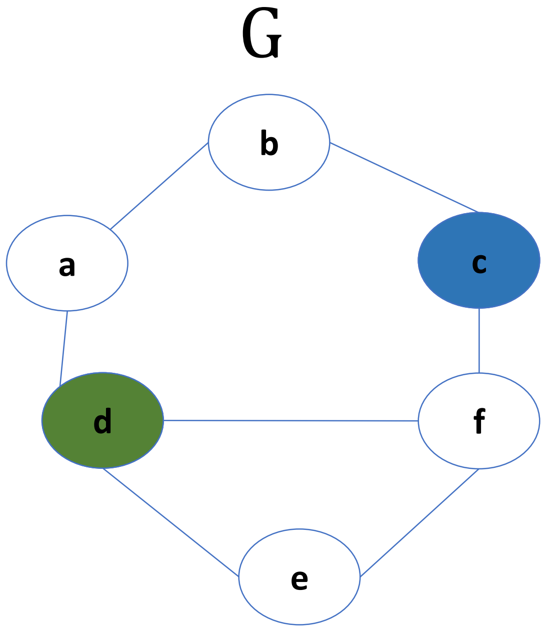
A graph is presented by a pair of objects which are connected by links.

- The interconnected objects are represented by points termed as vertices
- The links that connect the vertices are called edges.

Graph

Graph G(V, E)

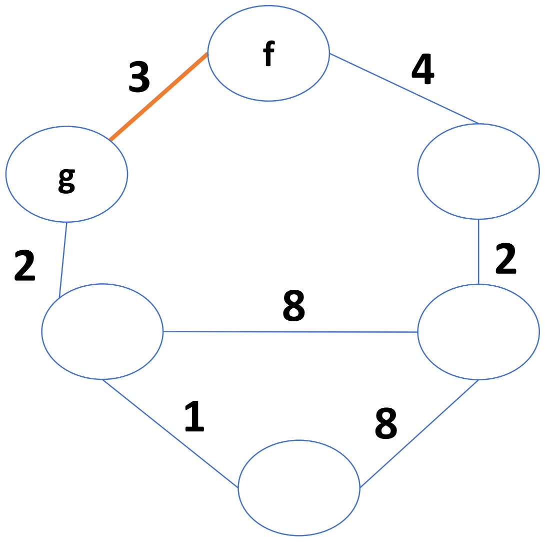
- V: Nodes or vertices
  V={a, b, c, d, e, f}
- E: Edges between pairs of nodes
  E={(a,b),(a,d),(b,c),(c,f),(d,f),(d,e),(e,f)}
- Graph size parameters:
  n= |V|, m=|E|
- Symmetric relationships:
  (a,b) and (b,a) are identical for undirected graphs
- Degree of a node: number of edges connected to the node
  deg(d)=3, deg(c)=2
- Path: A path is a sequence of nodes with the property that each consecutive pair is joined by an edge in G
  Path: b, c, f, e
  Not Path: b, c, f, a



Graph

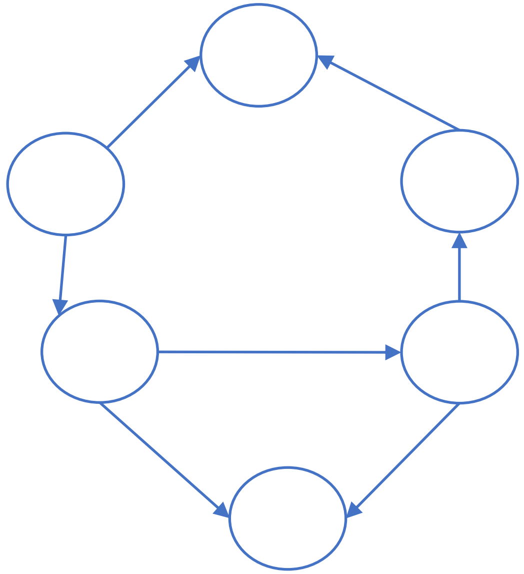
- A node and an edge are incident if the edge contains this node.
- Two vertices(nodes) joined by an edge are called adjacent

Undirected and weighted graph



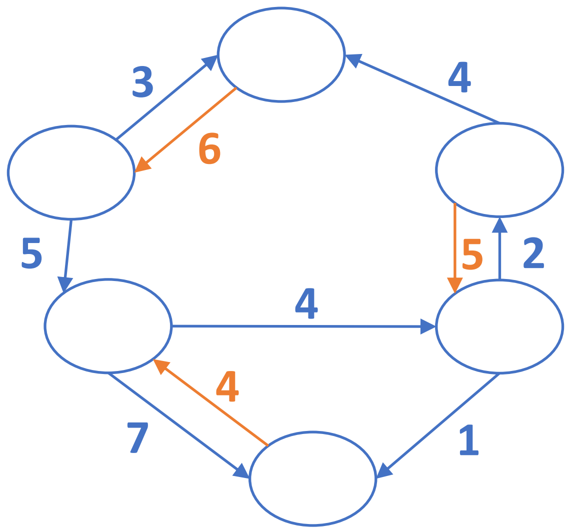
Symmetric relationship between nodes

Directed Graph



Asymmetric relationship between nodes

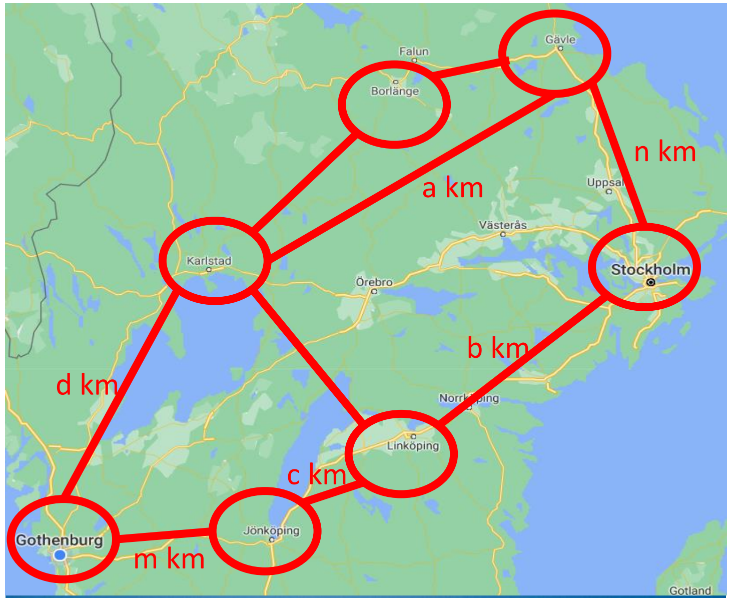
Directed and weighted Graph



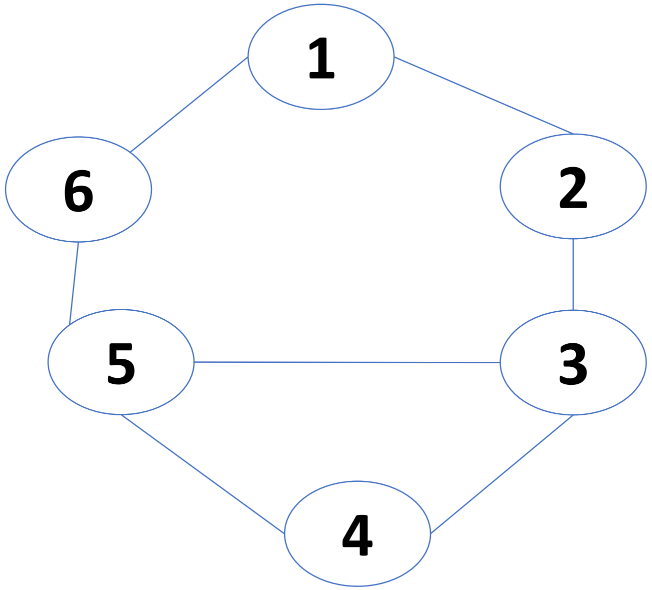
Asymmetric relationship between nodes

Graph - Applications

- Google Maps
- Google search
- Social networks
  - Twitter, Facebook
- Recommendation systems
  - Netflix, YouTube, Facebook
- Logistics of delivering goods
- Driving directions



Graph representation – Adjacency Matrix



$$\begin{array}{c c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency Matrix - Implementation

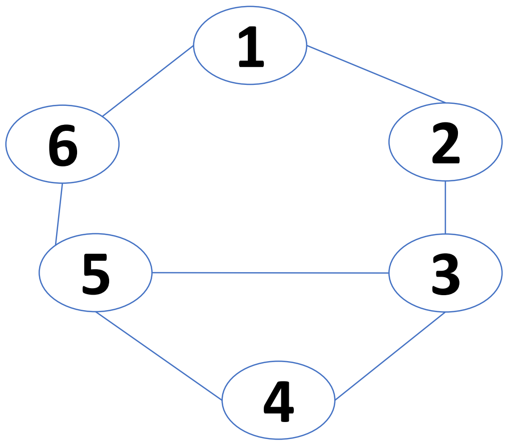
$A=[n][n]$, where n is #nodes

$$a[i][j] = \begin{cases} 1 & \text{if node } i \text{ is connected to node } j \\ 0 & \text{otherwise} \end{cases}$$

- Space proportional to n^2.
- Checking if (u, v) is an edge takes $O(1)$ time.
- Identifying all incident edges of a vertex $O(n)$
- Finding all edges in G requires $O(n^2)$ time

$$\begin{array}{c|c} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Graph representation – Adjacency List



Space Complexity?

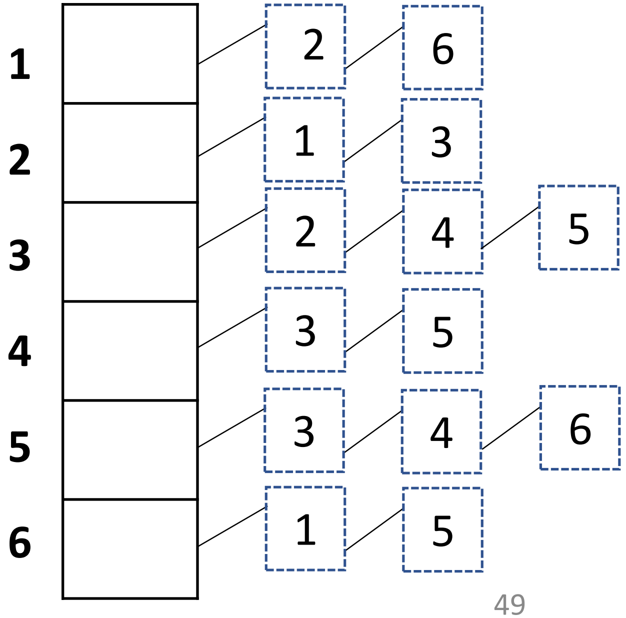
- Each node "u" requires $\mathrm{deg(u)}$ space.
- Exactly two representations of each edge in undirected graphs, thus sum of the size of all the lists is $2\mathrm{m}$.
- Size of the node-indexed array is n.

Thus space is only $O(\mathrm{m}+\mathrm{n})$

Other operations:

- Checking if (u, v) is an edge takes $O(\mathrm{deg(u)})$ time.
- Identifying all edges takes $O(\mathrm{m}+\mathrm{n})$ time.

1 linked list per node



Graph representation

$Note\ |V| = n$

Graph	Adjacency Matrix	Adjacency List
Space complexity	$O(\|V\|^2)$ or $O(n^2)$	$O(\|V\| + \|E\|)$
IsConnected(n_i, n_j)	$O(1)$	$O(\|V\|)$
Add(n_k)	$O(\|V\|^2)$	$O(1)$
GetAdjacent(n_k)	$O(\|V\|)$	$O(\|E\|)$

```
Adjacency Matrix
    a b c d e
  a - - 1 - 1
  b - - - 1 -
  c 1 - - 1 -
  d - 1 1 - 1
  e 1 - - 1 -
```

```
Adjacency List
 a -> 'c', 'e',
 b -> 'd',
 c -> 'a', 'd',
 d -> 'b', 'c', 'e',
 e -> 'a', 'd'
```

Question – Poll

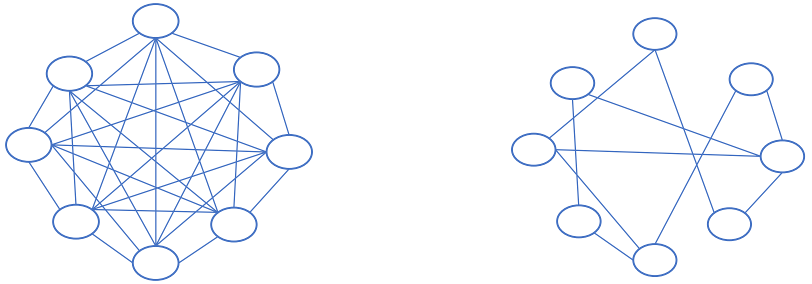
Which of the graph representation is more efficient for implementing dense and sparse graphs, respectively?

1. Adjacency matrix for sparse graphs and adjacency list for dense ones
2. Adjacency matrix for dense graphs and adjacency list for sparse ones
3. Both are good for both types of graphs
4. None of them works for these types of graphs

Graph representation

Dense Graphs

Sparse Graphs



Graph	Adjacency Matrix	Adjacency List
Space complexity	$O(n^2)$	$O(n+m)$

Is it obvious from $O(n^2)$ for Adjacency matrix vs $O(\mathrm{m}+\mathrm{n})$ for Adjacency list?

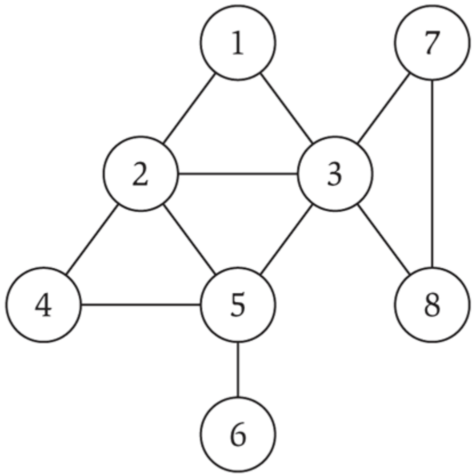
- Max $\deg(\mathrm{u}) = \mathrm{n}\text{-}1$ for any u in G.
- Thus $\mathrm{m} \leq \mathrm{n}(\mathrm{n}\text{-}1)/2$ implies m ≤ n^2
- Therefore $O(\mathrm{m}+\mathrm{n})$ is never worse than $O(\mathrm{n}^2)$
- Much better for sparse graphs, i.e., when $\mathrm{m} \ll \mathrm{n}^2$

General Trees

An undirected graph is a tree if it is connected and does not contain a cycle.

Connected graph: an undirected graph is connected if for every pair of nodes u and v, there is a path from u to v.

Cycle: a cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.

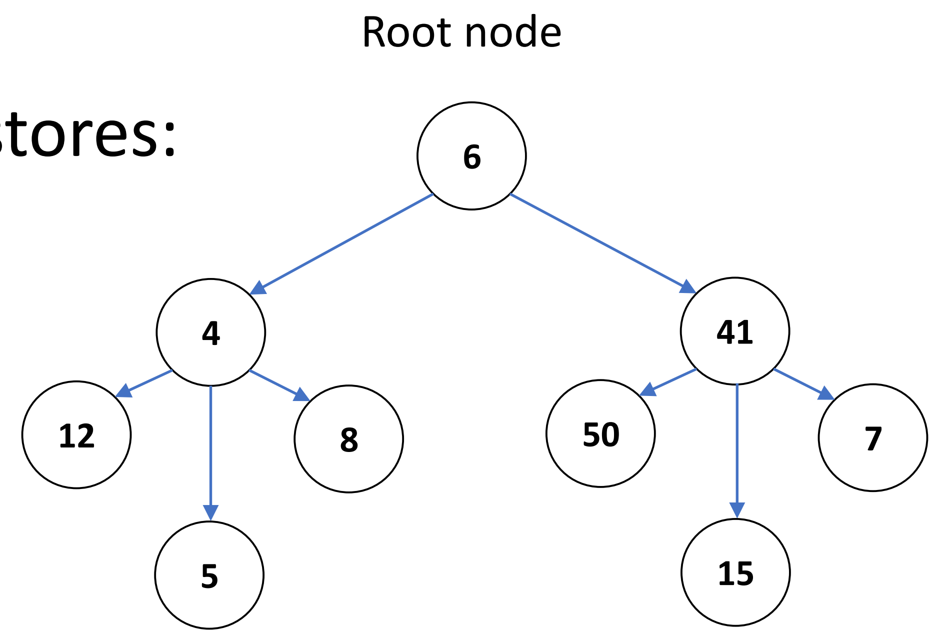


Cycle: 1-2-4-5-3-1
Not a cycle: 1-3-8-7-3-1

Node Data Type In A Tree

In a tree a node is sort of a data type that stores:

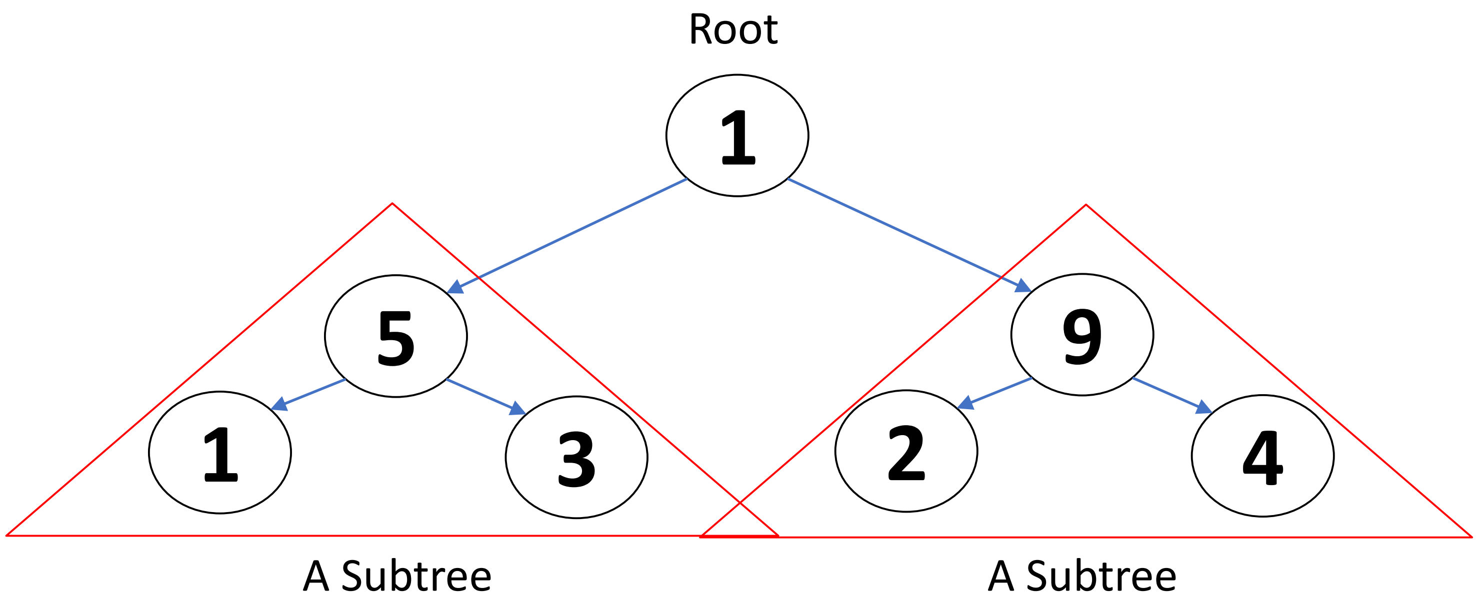
- Key or values to compare to
- Pointers to the children
- Pointers to the parent node (optional)



Binary Tree

Binary tree represents the nodes connected by edges.

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have **at most two child** nodes.



Node Data Type In A Binary Tree

Node is sort of a data type that stores:

- Key or values to compare to
- A pointer to the left child
- A pointer to the right child
- A pointer to a parent node (optional)

```
class Node:


    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Binary Search Tree

Binary Search Tree (BST) are a special type of tree data structure whose *InOrder* traversal gives a sorted list of nodes or vertices.

InOrder traversal:

Walking on the tree with the order

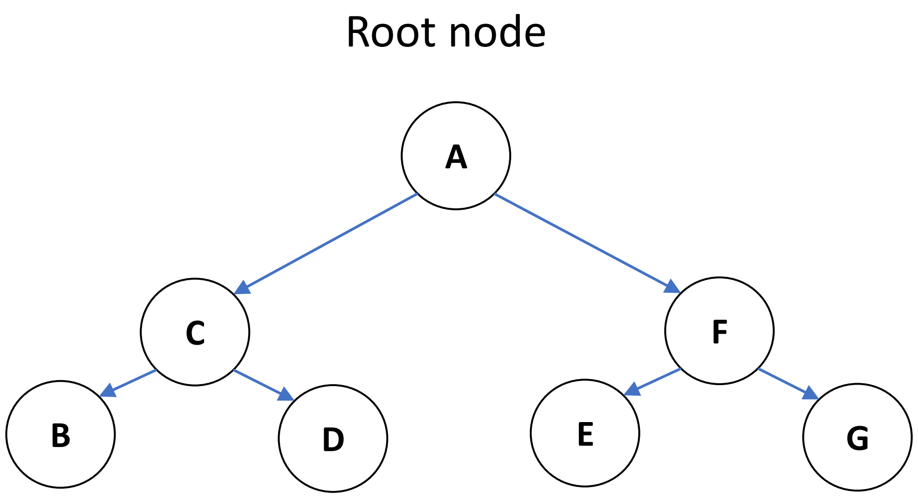
left, root, right

Binary Search Tree

- Root node
- Each node has two child nodes
  - Left: Keys less than or equal to parent’s key
  - Right: Keys larger than parent’s key

Important property of BST

- A’s key is larger than the keys of its left subtree
- A’s key is smaller than the keys of its right subtree

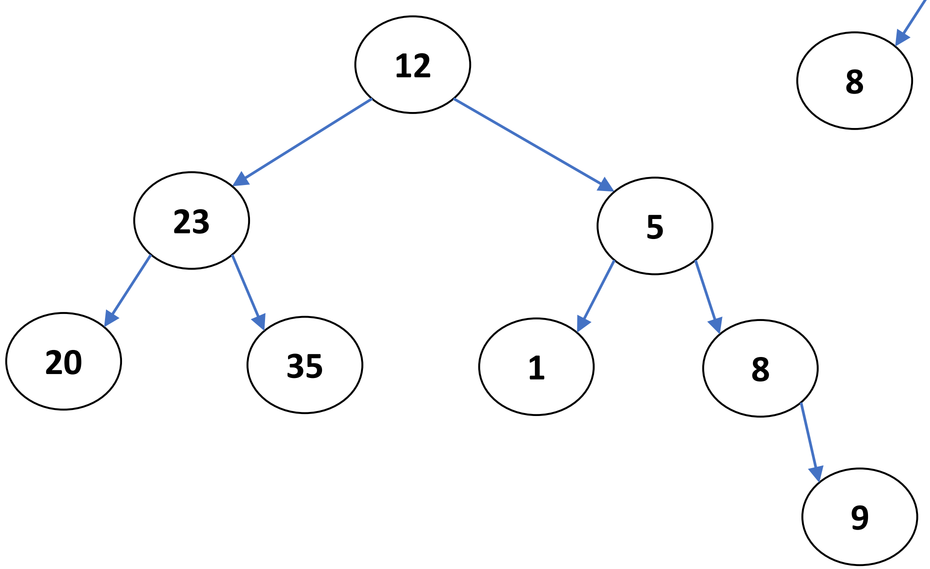



```
left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
```

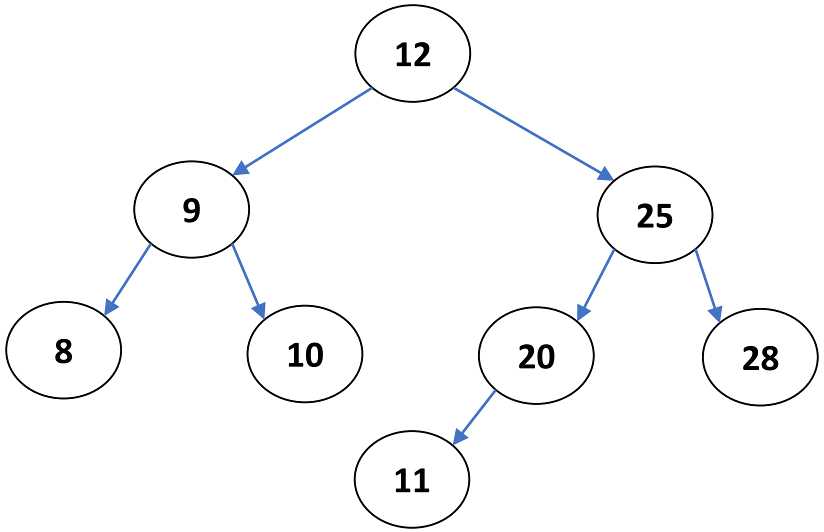
Question - Poll

According to the binary search tree's property, which of the following trees satisfies the properties?

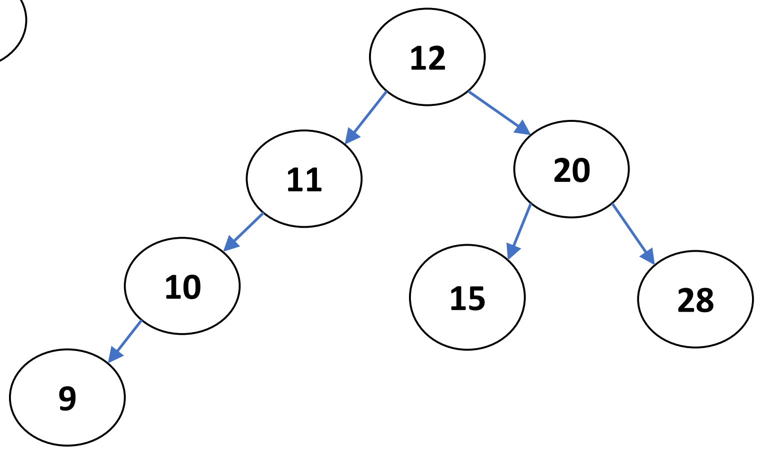
A



B

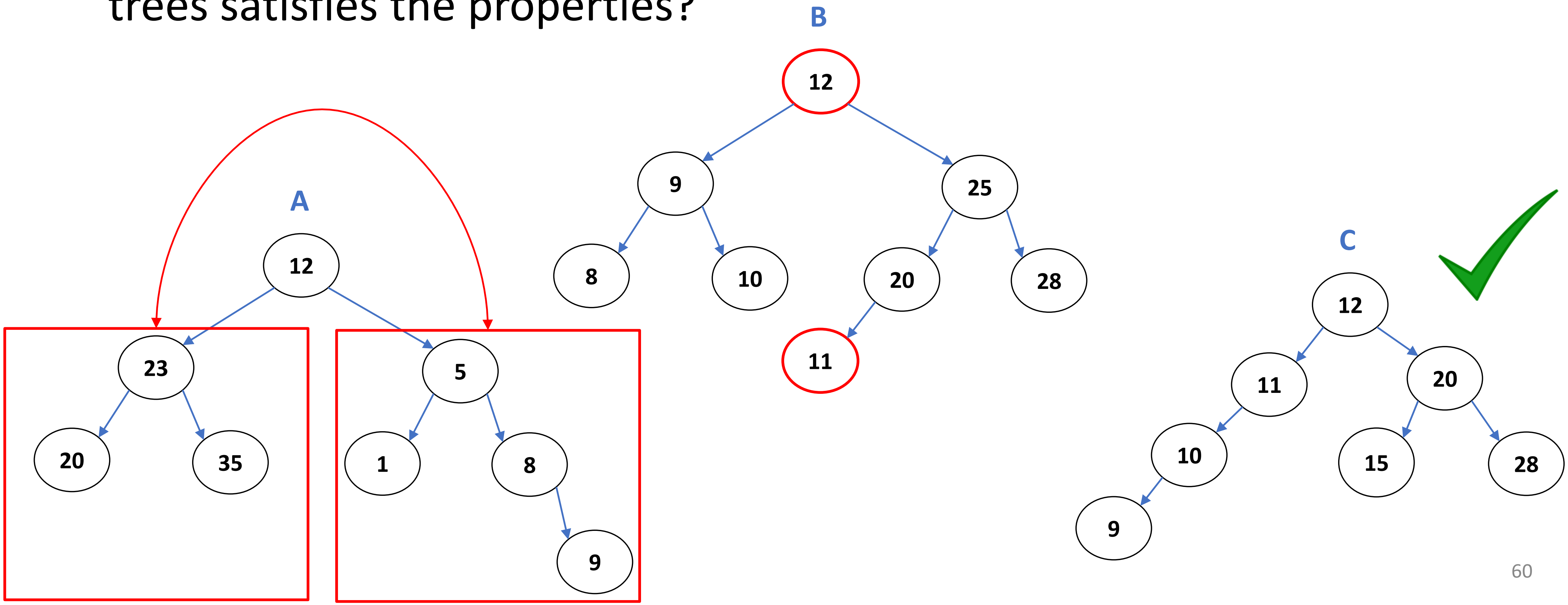


C



Question - Poll

According to the binary search tree's property, which of the following trees satisfies the properties?



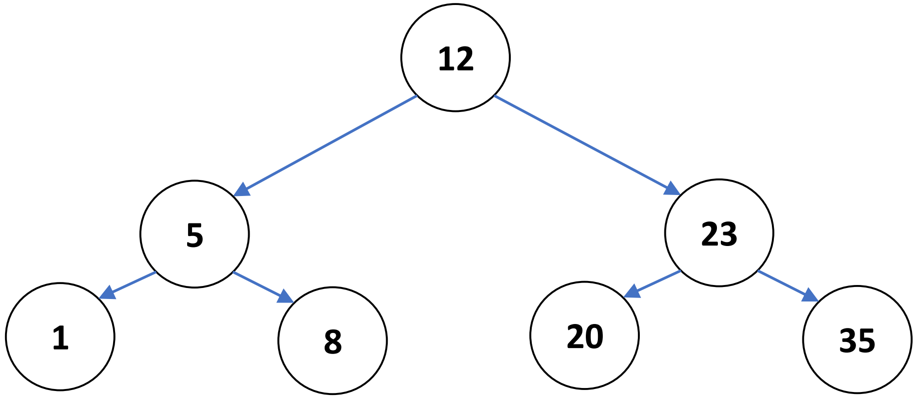
Binary Search Tree – Find() operation

Inputs: A key and root node

Output: the node with the given key

Find(key, Root) -> node

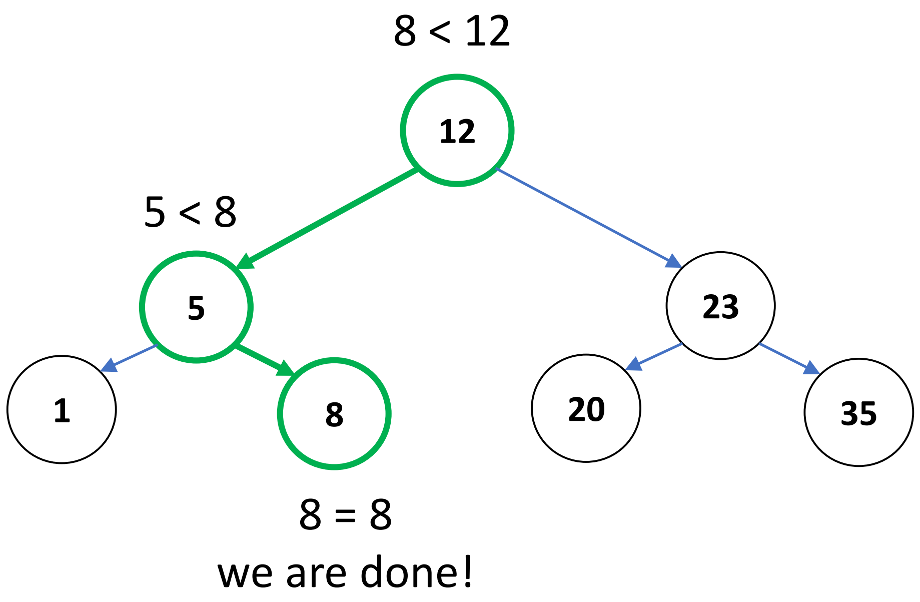
e.g., Find(8)



Binary Search Tree - Find operation

Find(key, Root) -> node

e.g., Find(8)



Binary Search Tree - Find operation

Find(key, Root) -> node

```
Find(k, r)
    if r.key =k:
        return r
    else if r.key > k
        return Find(k, r.left)
    else if r.key < k
    return Find(k, r.right)
```

Binary Search Tree - Operations

Next operation

Input: a node

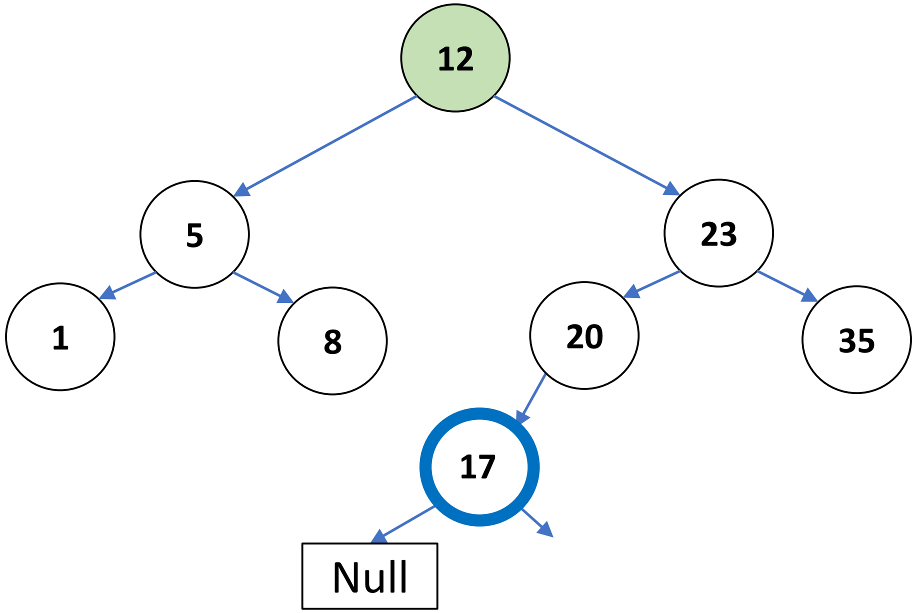
Output: returns the node with the next largest key

Next(node) -> node_NextMax

Binary Search Tree - Operations

Next operation

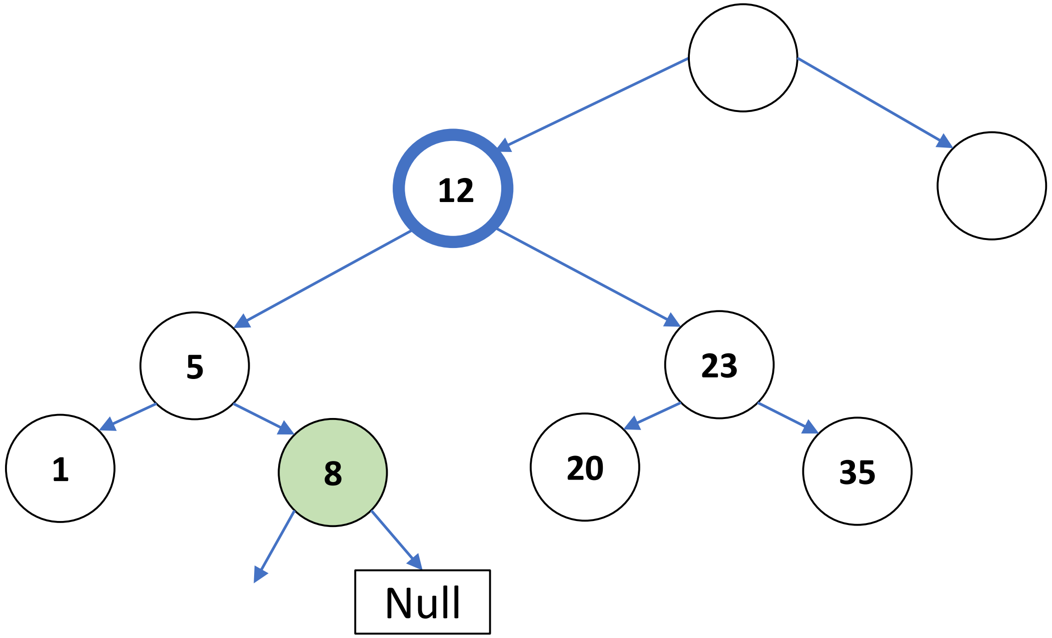
Example1: Next(12)



Binary Search Tree - Operations

Next operation

Example2: Next(8)



Binary Search Tree - Operations

Next operation

Next(node) -> the node with the next largest key

Consider both scenarios

- N has a right child
- N does not have a right child

```
Next(n)
    if n.right ≠ Null:
        return LeftDescendant(n.right)
    else:
        return RightAncestor(n)
```

```
LeftDescendant(n)
    if n.left=Null:
        return n
    else:
        return LeftDescendant(n.left)
```

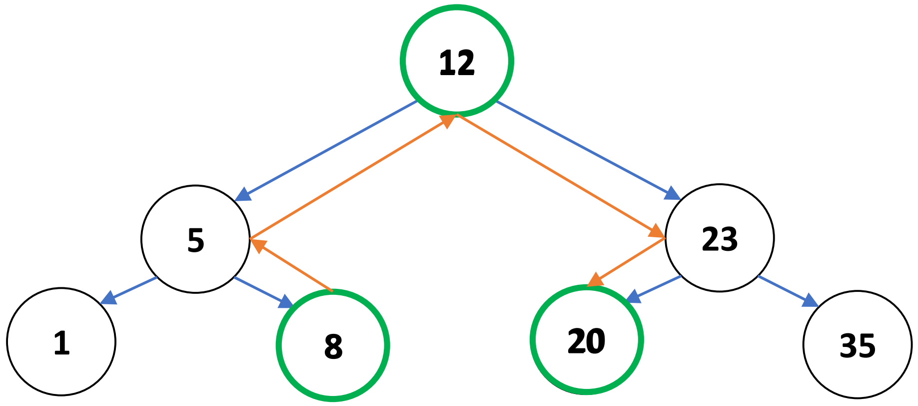
```
RightAncestor(n)
    if n.key < n.parent.key:
        return n.parent
    else:
        return RightAncestor(n.parent)
```

Binary Search Tree - Operations

RangeSearch operation

Example: RangeSearch(7,22)

- Search for the first element in the range
- Find the Next element
- Continue until Next is not valid



Binary Search Tree - Implementation

RangeSearch operation

```
RangeSearch(a,b,r)
    L <- 0
    n <- Find(a,r)
    while n.key <= b:
        if n.key >= a:
            L <- L.append(n)
        n <-Next(n)
    return L
```

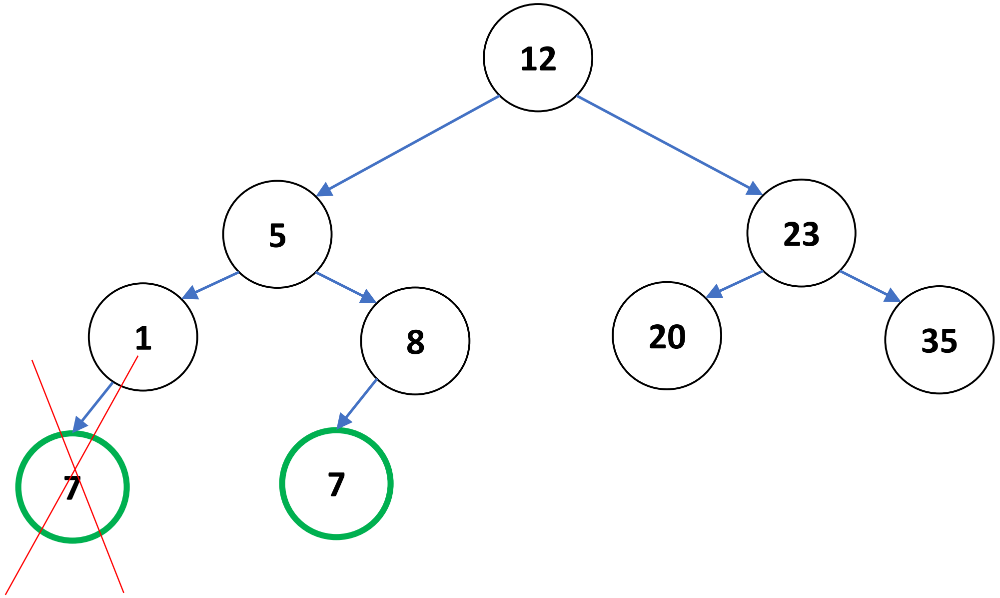
A list that stores everything we find

Binary Search Tree - Operations

Insert operation

insert(k, R) -> adds a node with Key “k” to the tree

e.g., insert (7)



Binary Search Tree - Operations

Insert operation

insert(k, R) -> adds a node with Key “k” to the tree

```
insert(k, R):
  n <- Find(k, R)
  Add the new node with key "k" as a child of n
```

Binary Search Tree - Operations

Delete operation

delete(n): delete the node n

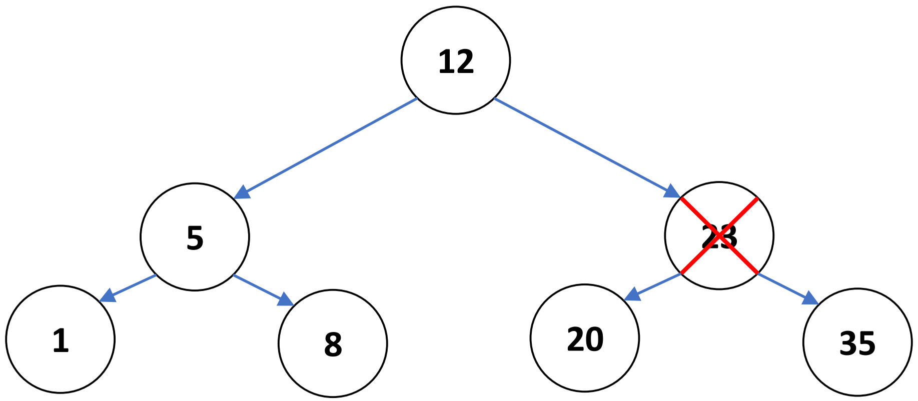
Binary Search Tree - Operations

Delete operation

delete(n)

e.g., delete (23)

Cannot just remove the node since it has child nodes



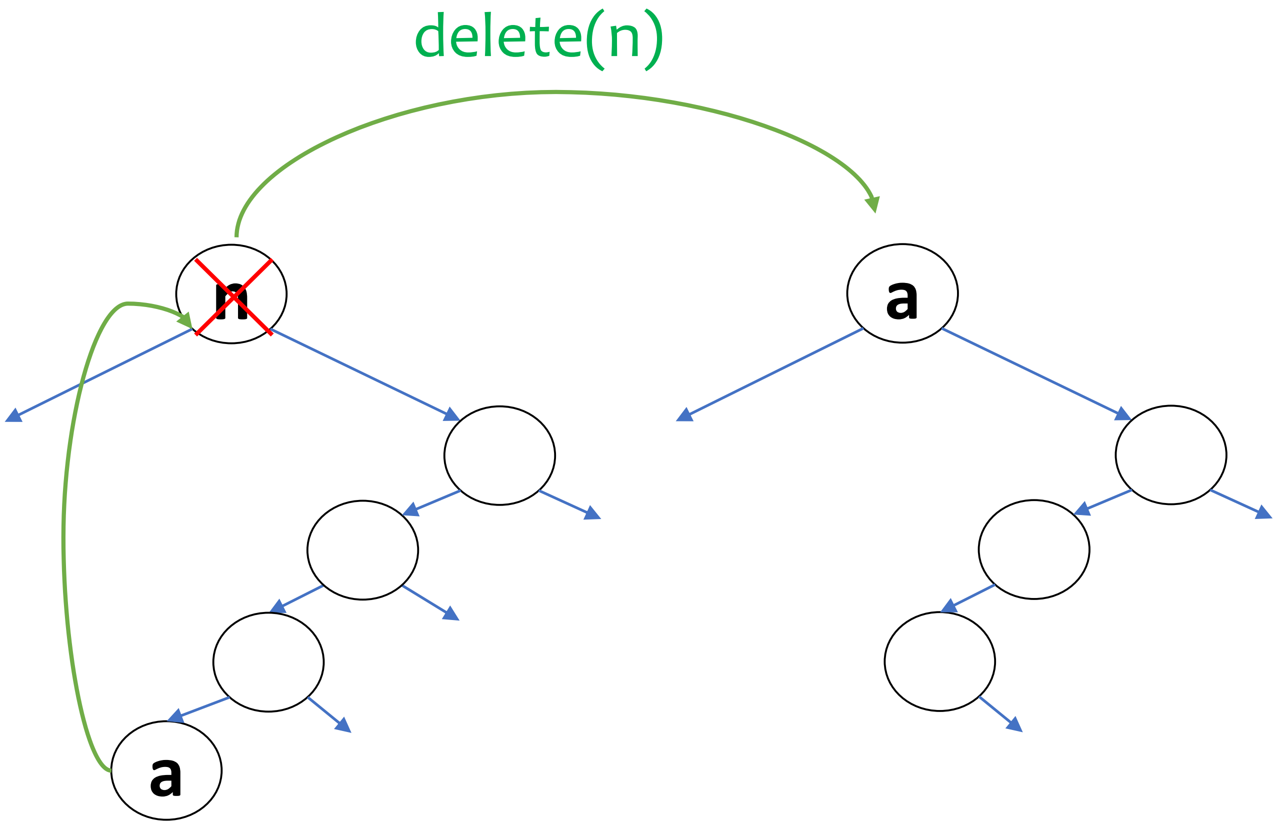
Binary Search Tree - Operations

Delete operation

delete(n)

Case 1:

- Next(n)=a does not have left or right child
    - Replace n with Next(n)=a



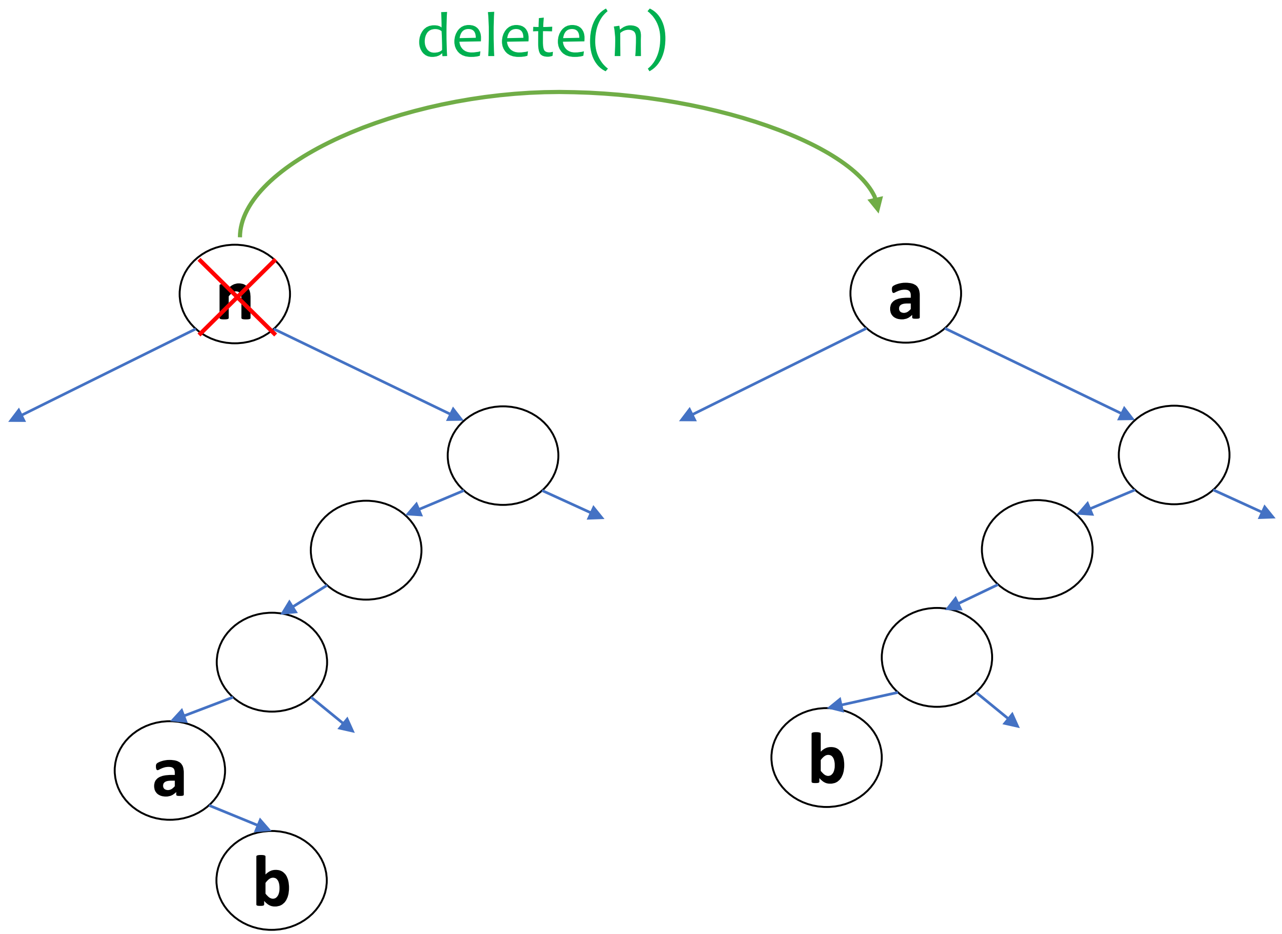
Binary Search Tree - Operations

Delete operation

delete(n)

Case 2:

- Next(n)=a has a right child
  - Replace n with Next(n)=a
  - Promote RightChild(a)=b



Binary Search Tree - Operations

Delete operation

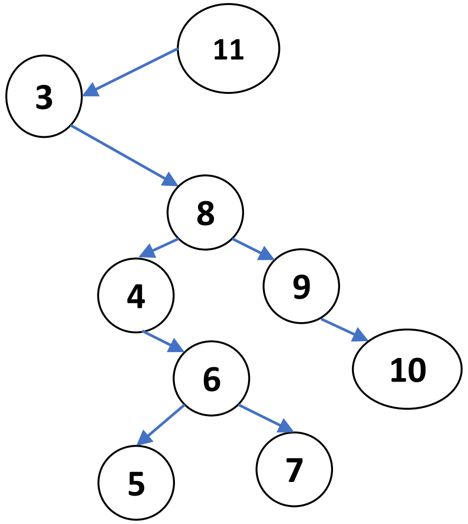
delete(n)

```
delete(n)
if n.right = Null:
     remove n, promote n.left
else:
     a <- Next(n)
     replace n by a, promote a.right
```

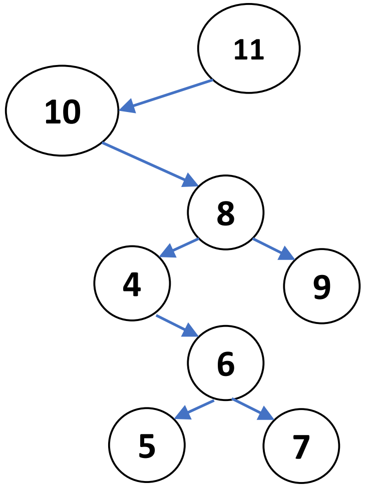
Note “a” does not have any left child!

Question - Poll

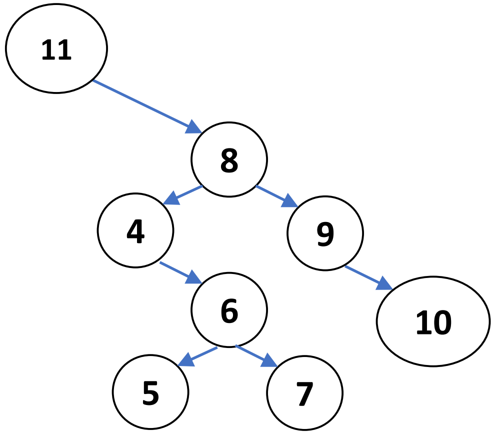
What will be the result tree after query delete(3)?



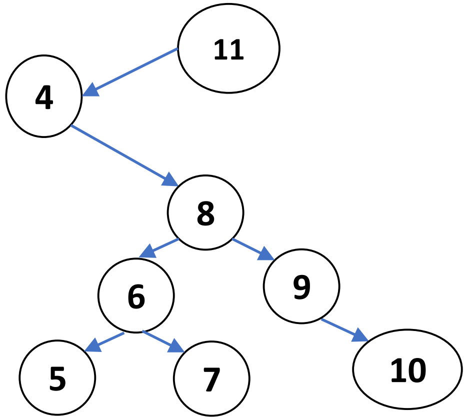
(1)



(2)

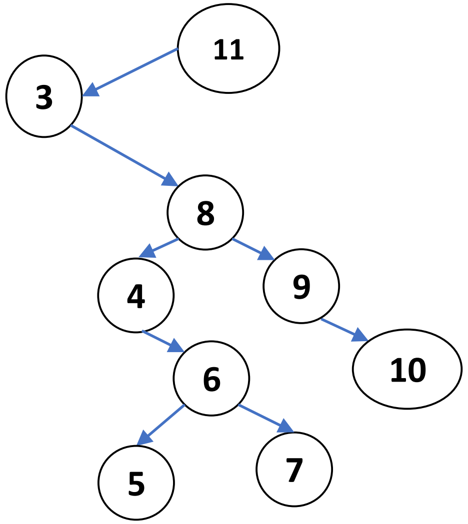


(3)

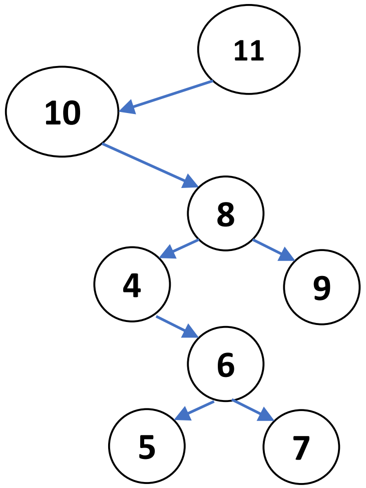


Question - Poll

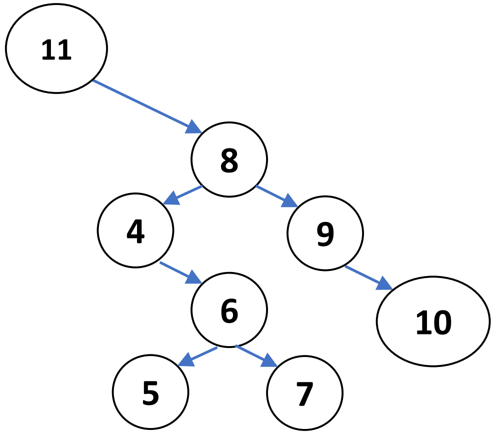
What will be the result tree after query delete(3)?



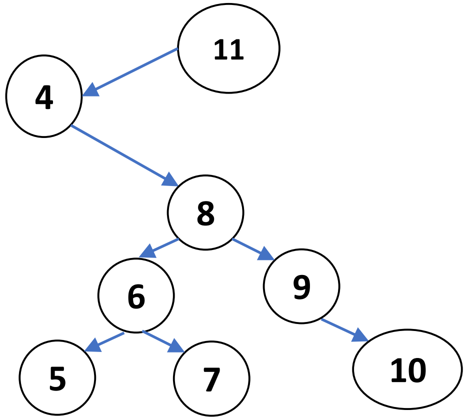
(1)



(2)

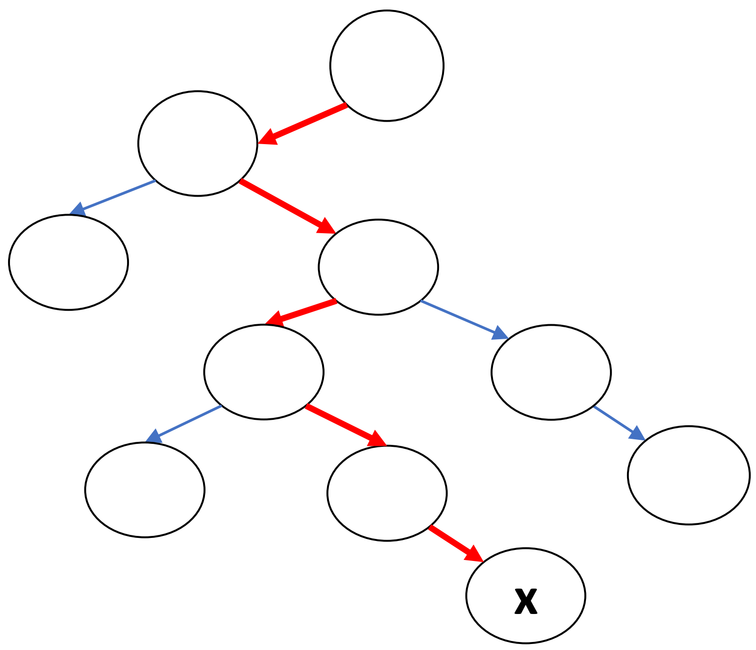


(3)



Binary Search Tree – Runtime

Find(x)

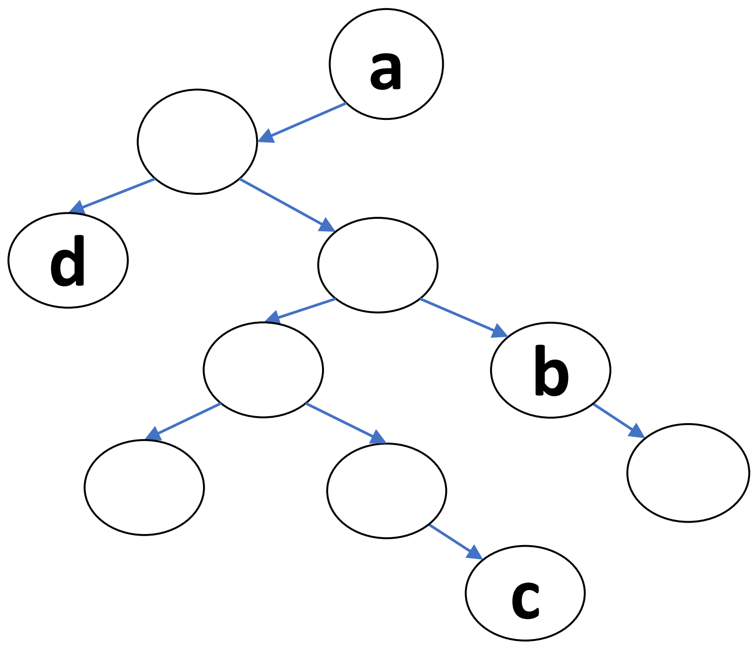


O(Depth of the tree)

Question - Poll

What is the order of fast search in the tree?

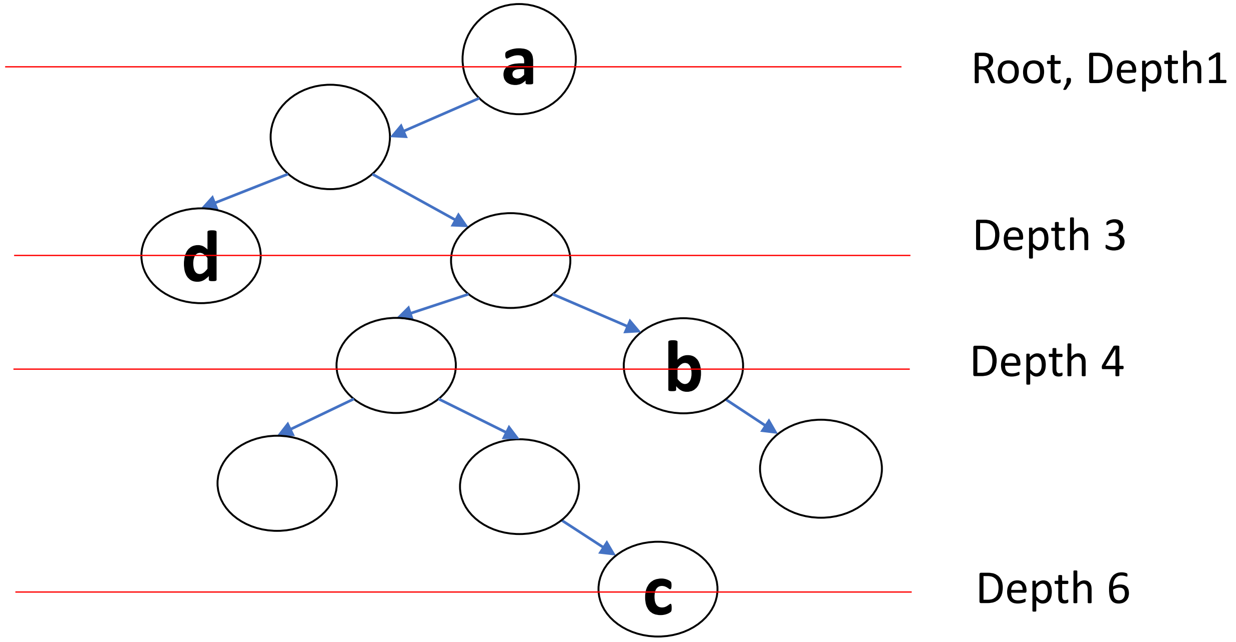
1) a -> b -> c -> d
2) a -> d -> b -> c
3) d -> a -> b -> c
4) c -> b -> d -> a



Question - Poll

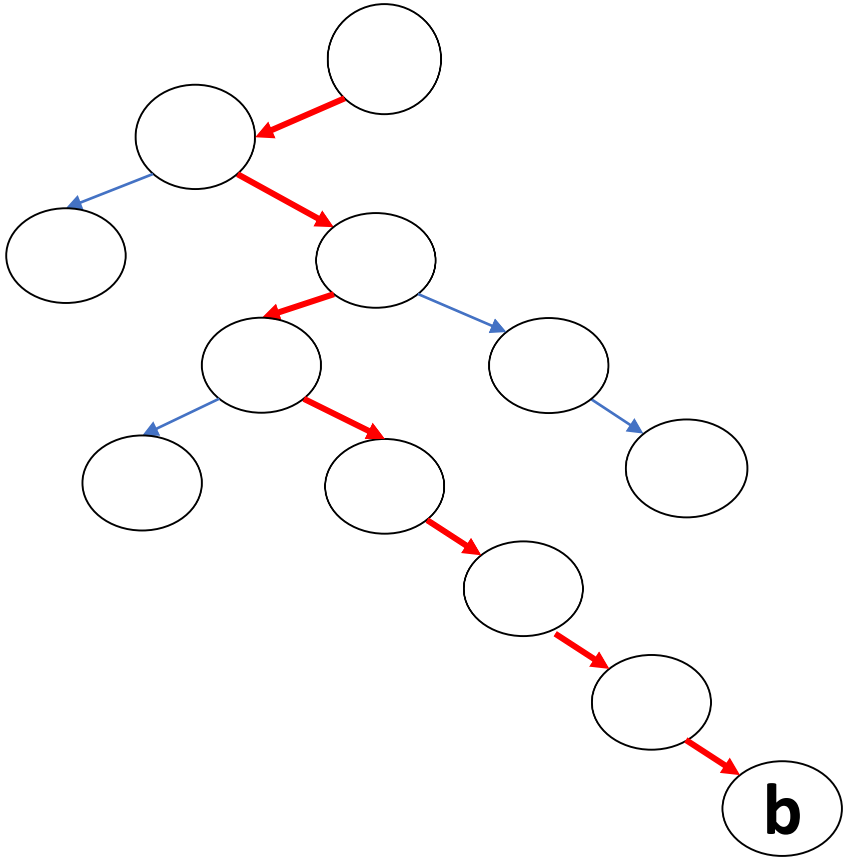
What is the order of fast search in the tree?

1) a -> b -> c -> d
2) a -> d -> b -> c
3) d -> a -> b -> c
4) c -> b -> d -> a



Binary Search Tree

Runtime



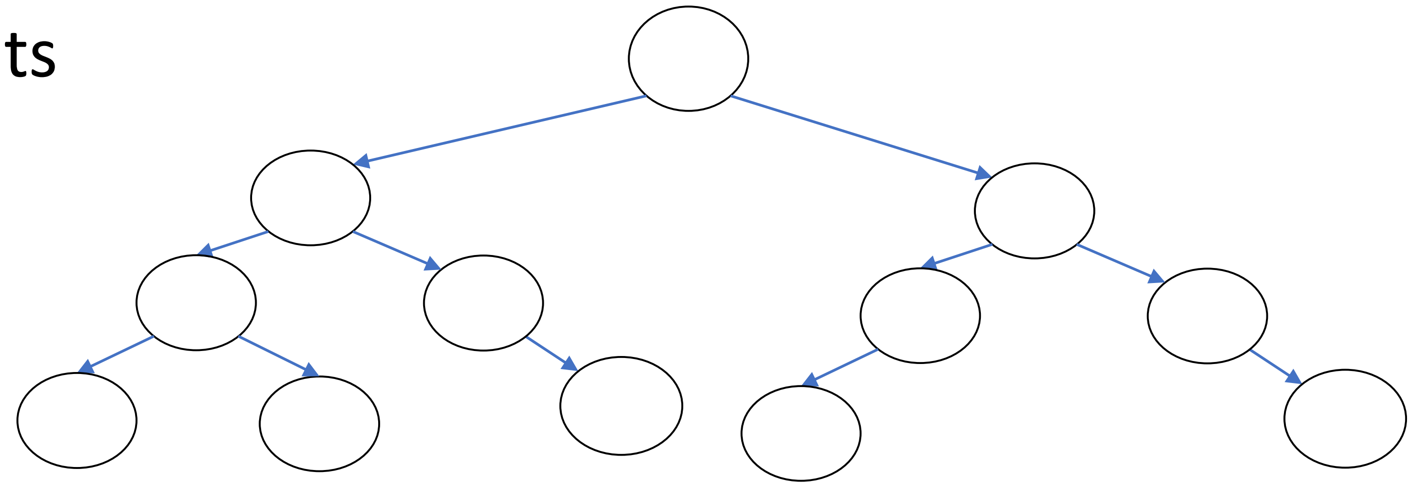
Worse case scenario depth(n)

How to fix this?

Binary Search Tree – Balanced Trees

Desired property:

- Left and right subtrees have almost the same size
- Cut the search space in two
- Subtree has half the size of its parent
- *O*(log(n))



Binary Search Tree – Balanced Trees

How *insert* and *delete* operations would work for a balanced tree?

- They might destroy the balance

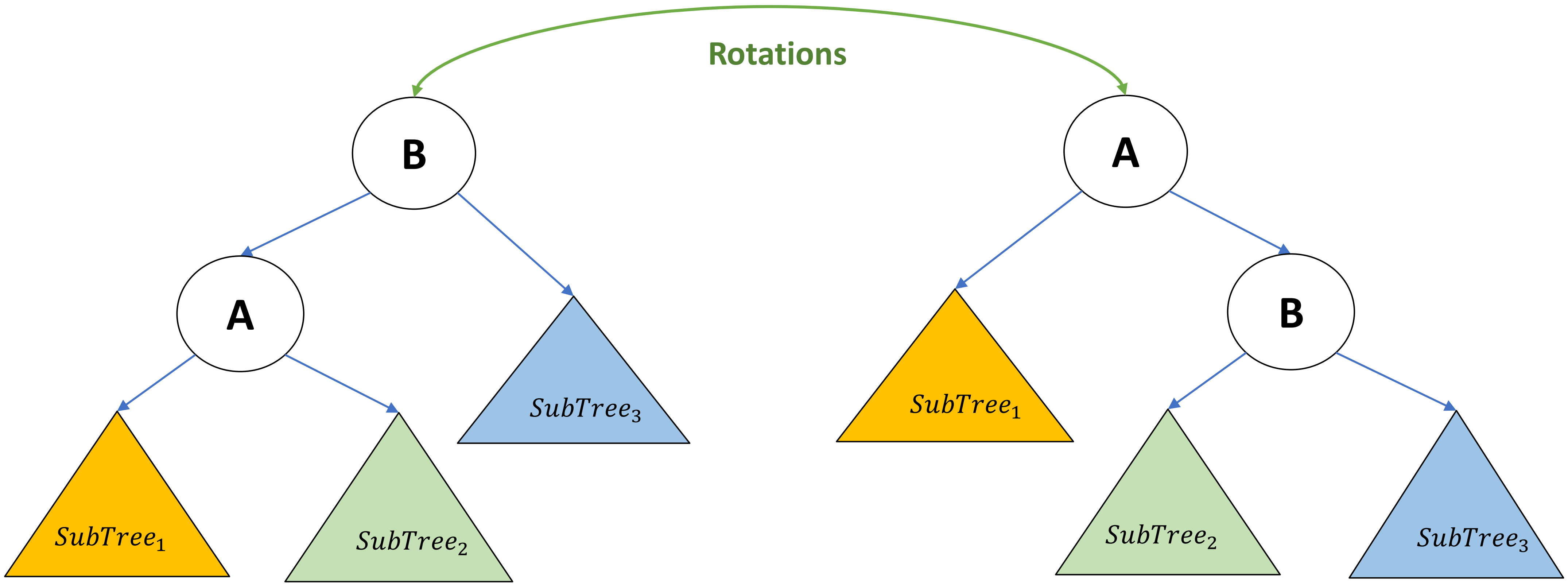
How to solve the issue?

- Rebalance the tree and maintain the balance by rearranging

How to rearrange to maintain the sorting property?

- Use rotations

Binary Search Tree - Rotations



$$Sub_1 < A < Sub_2 < B < Sub_3$$

Tree Traversal

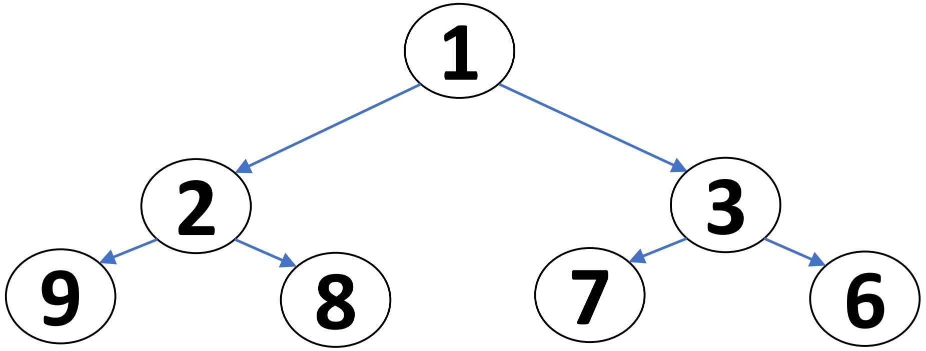
InOrder traversal:

left – root – right

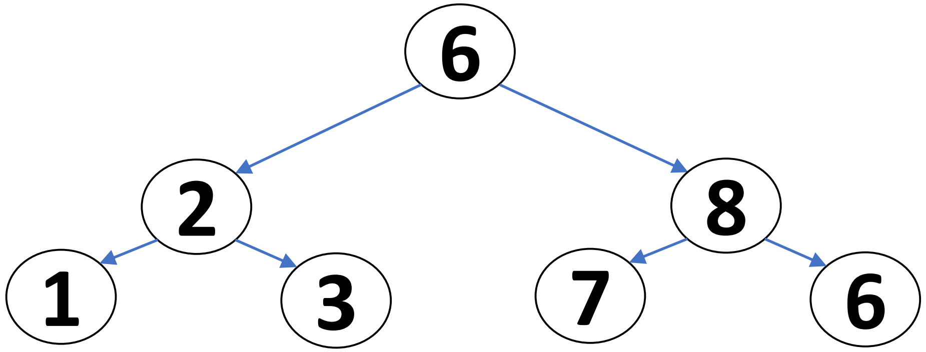
Output: 9, 2, 8, 1, 7, 3, 6

Sort(Output) = 1,2,3,6,7,8,9

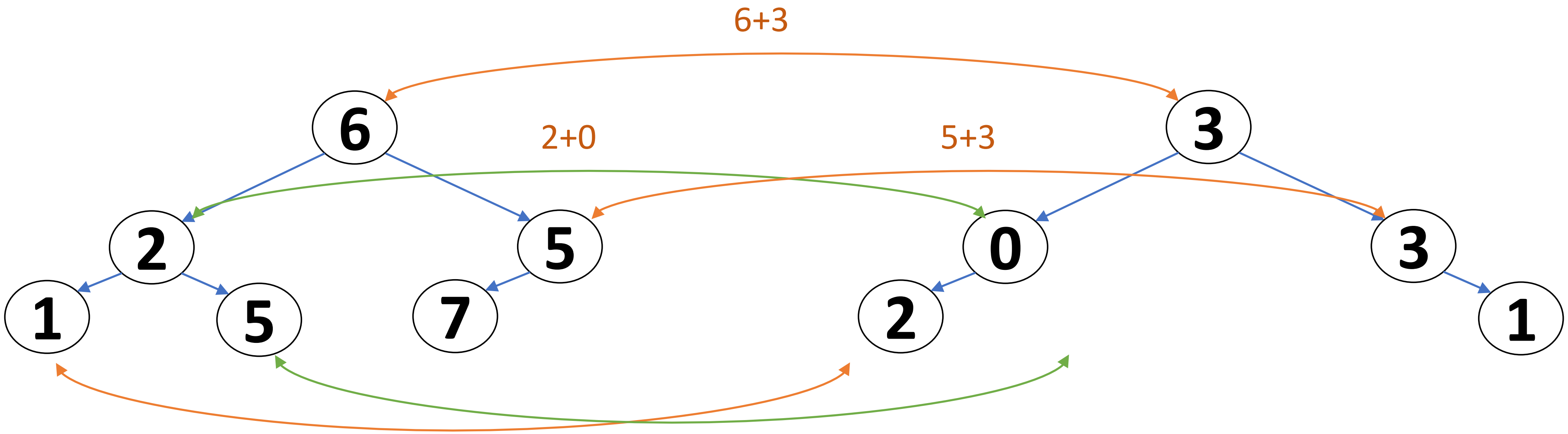
Binary Tree

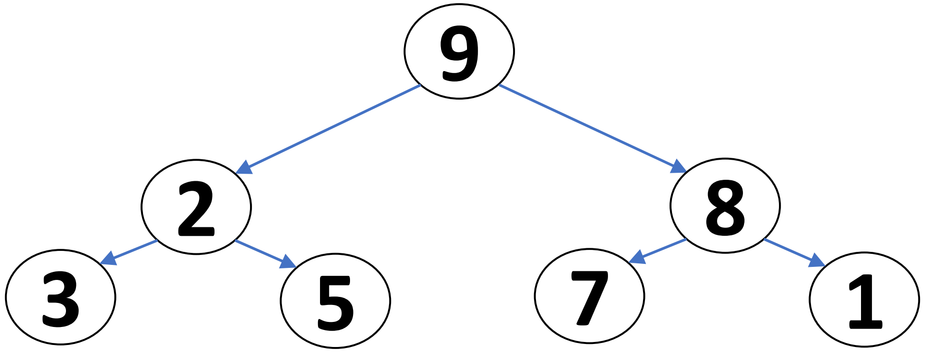


Binary Search Tree



MergeSum





Merged Tree

Advantages of trees and graphs

- Search complexity:
  - Array or linked list: since they are linear structures the time required to search a “linear” list is proportional to the size of the data set.
  - Trees: fast search ($O(\log n)$ comparisons to find a particular node)

- Representation:
  - Linked lists: a node could at most have two pointers (one to its next and one to its previous node)
  - Graphs: a node could have more than two pointers.

Trees applications

- Store hierarchical data, like folder structure, organization structure…

- Allow fast search, insert, delete on a sorted data and finding closest item

- Find shortest path trees which are used in routers and bridges respectively in computer networks

Graphs applications

- Web graph.
  - Node: web page.
  - Edge: hyperlink from one page to another

- Social network graph.
  - Node: people.
  - Edge: relationship between two people