DATA STRUCTURES

Fundamentals and Applications

MARTÍN GONZÁLEZ-RODRÍGUEZ

www.martin-gonzalez.es

Data Structures

Martin Gonzalez-Rodriguez, Ph. D

ISBN 978-1-4467-9147-9

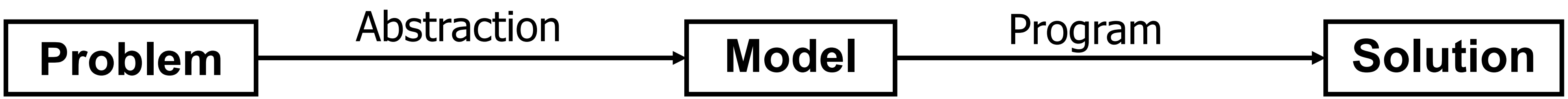
Algorithmic and Design

Martin Gonzalez-Rodriguez, Ph. D.

Problem solving in Engineering

Strategies

- Define the problem (analysis).
- Find a model that represents the problem (abstraction).
- Design an algorithm based on the model to solve the problem.



Programs

The Quote
`Programs = Data Structures + Algorithms`

- Find ways to **store data** and to **design algorithms** able to solve the tasks assigned to the **processes**.

Niclaus Wirth (Wikipedia)

- Term coined by Niclaus Wirth in 1976
  - Turing Award 1984.
    - Designer of the programming languages Euler, Algol, Pascal, Modula, Modula-2 and Oberon.

Data Type

Definition

- Value set that may be assigned to a class property.
  - **PDT** (Predefined Data Type) constitute the **default** data types in a programming language.
    - Integer.
    - Real.
    - Character.
    - Boolean.
    - Reference.

Data Structures

Definition

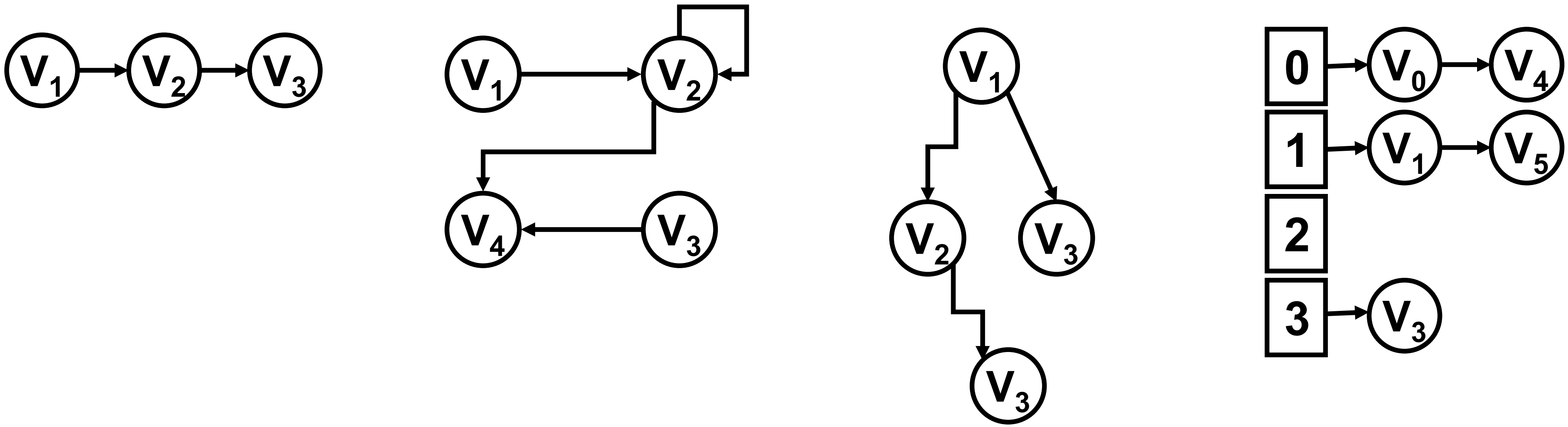
- Data set related to each other in a specify way[1].
  - The **SDT** (Structured Data Types) part of a programming language are collections of data types stored in a sequential order.
    - Arrays.
    - Strings.
    - Classes and objects.
  - There are other *default* data structures, which are usually implemented using classes.
    - Array List.
    - List.
    - Hash Map.
    - Stack.
    - …

[1]*Weiss, Mark Allen; (2000) Estructuras de Datos En Java 2. Addison-Wesley Iberoamericana.*

Data Structures

Classification

- Main data structure families
  - Linear (lists, stacks and queues).
  - Network (graphs).
  - Hierarchical (trees).
  - Dictionaries (hash tables).



- They may be combined to create other structures.

Data Structures

What structure should I use?

- The selection of the right structure for a given scenario depends on…
  1. Adequacy of the structure to the model representation.
  2. Efficiency of the structure.
     - Temporal (speed associated to the algorithms) → $O_T(n)$.
     - Spatial (memory required to implement the structure) → $O_M(n)$.

Algorithmic (essentials)

How many times is *test()* executed?

Algorithm A	$T_A = 3$

```
{
 test();
 test();

 int i=3;
 return (i*test());
}
```

Algorithm B	$T_B = 2$

```
{
 test();
 test();
 if (5%2 == 0) {
  test();
  return (test()%2);
 }
 return (0);
}
```

Algorithmic (essentials)

How many times is *test()* executed?

Algorithm C — $T_C(n) = 4n + 6$

```
{
 test();
 test();
 test();

 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
 }

 test();
 test();
 test();
}
```

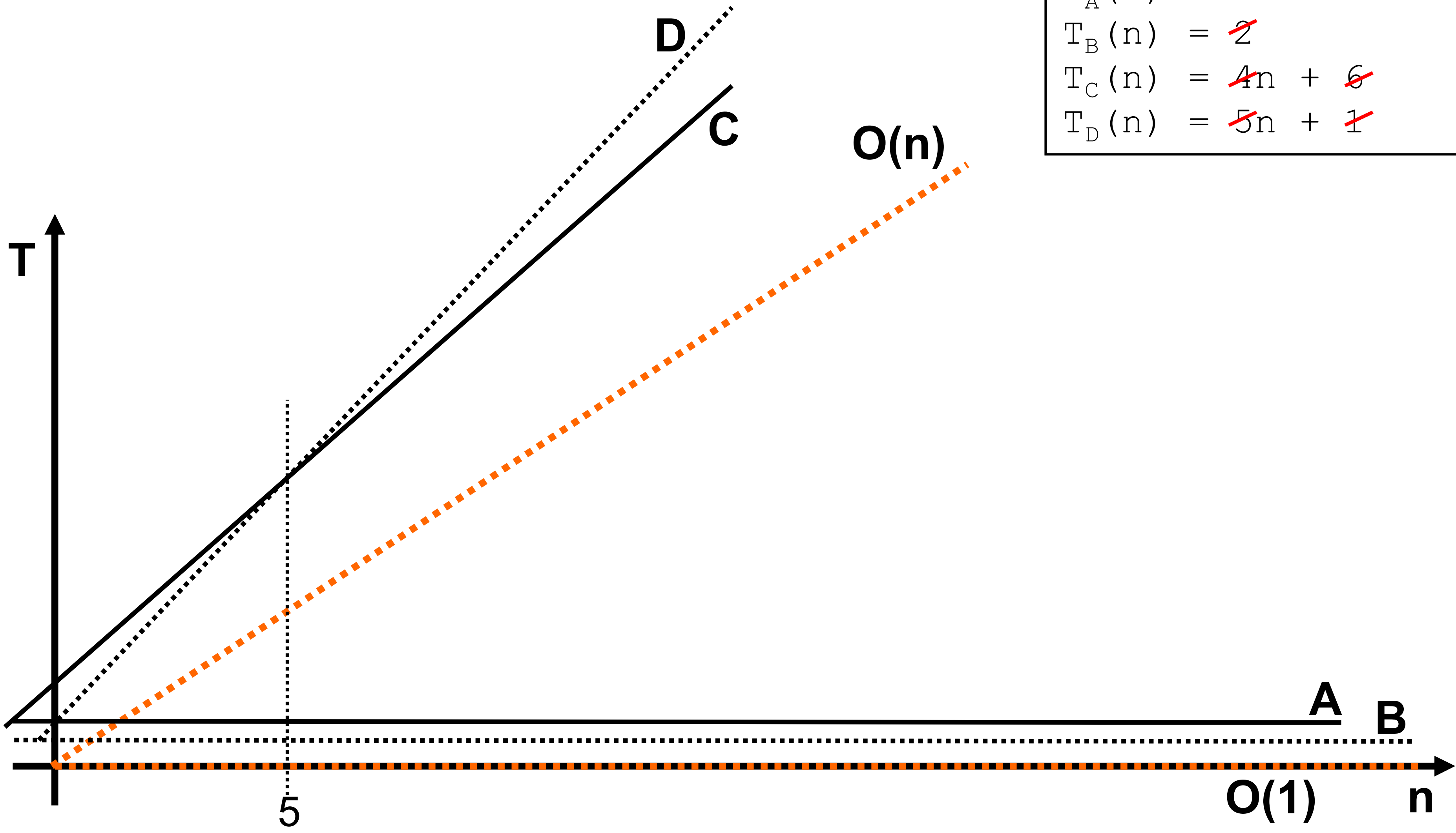
Algorithm D — $T_D(n) = 5n + 1$

```
{
 for (int i=0; i<n; i++) {
  test();
  test();
  test();
  test();
  test();
 }

 test();
}
```

Algorithmic (essentials)

Which is the fastest algorithm?

Execution Times
$T_A(n) = \cancel{3}$
$T_B(n) = \cancel{2}$
$T_C(n) = \cancel{4}n + \cancel{6}$
$T_D(n) = \cancel{5}n + \cancel{1}$



REMEMBER: in algorithmic research, n always assumes an infinite value

Algorithmic (essentials)

Hoy many times is *test()* executed?

Algorithm E $T_E(n) = 2n^2 + 1$

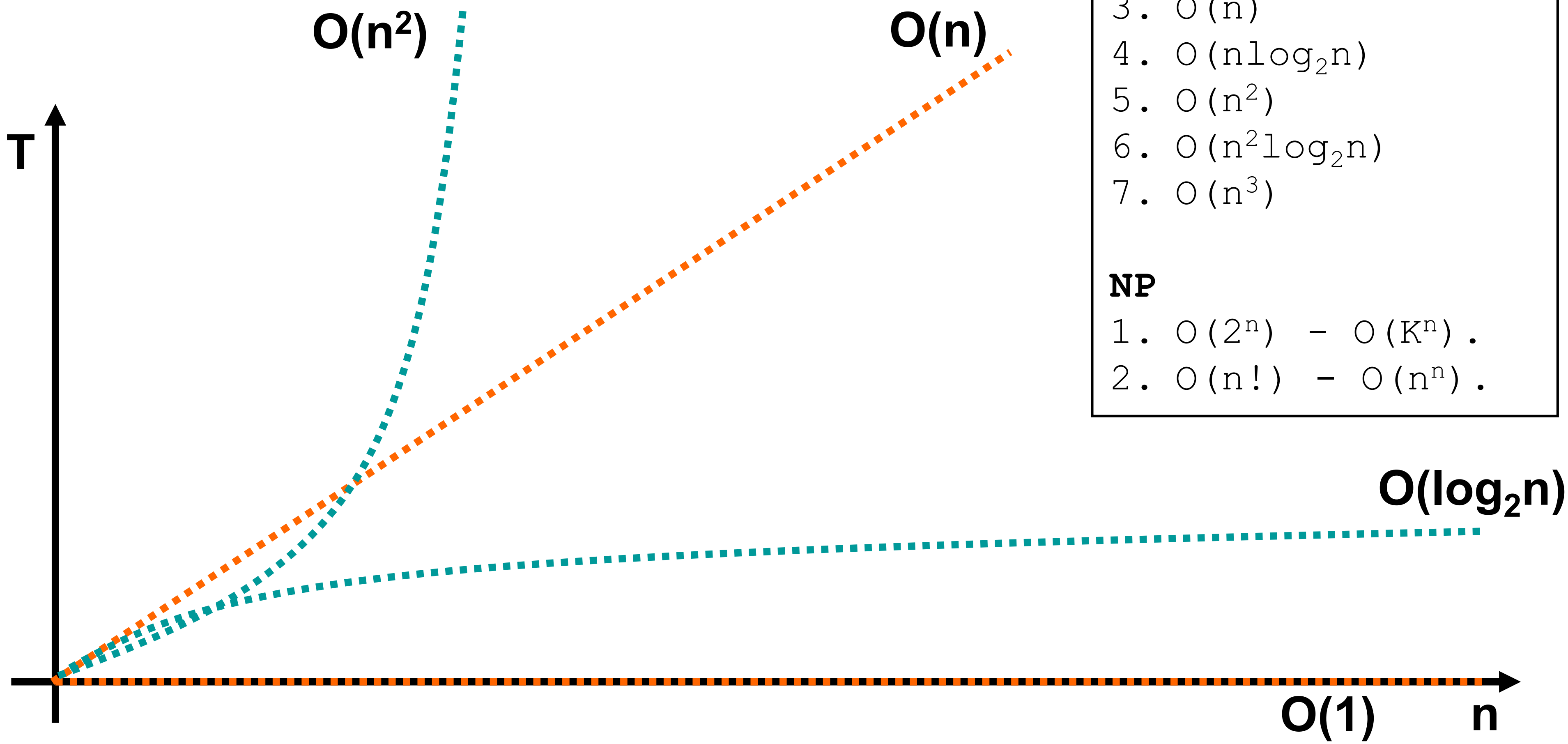
```
{
 for (int i=0; i<n; i++)
  for (int j=0; j<n; j++) {
   test();
   test();
 }
 test();
}
```

Algorithm F $T_F(n) = 2([\log_2 n] + 1) + 1$

```
{
 while (n>0) {
  test();
  test();
  n = n/2;
 }

 test();
}
```

Algorithmic (essentials)

Temporal Complexity



Efficiency Rank

P

1. $O(1)$
2. $O(\log_2 n)$
3. $O(n)$
4. $O(n\log_2 n)$
5. $O(n^2)$
6. $O(n^2\log_2 n)$
7. $O(n^3)$

NP

1. $O(2^n)$ – $O(K^n)$.
2. $O(n!)$ – $O(n^n)$.

Algorithmic (essentials)

Relevance of the Temporal Efficiency

n	$T_A(n) = 2^n$	$T_B(n) = n^3$
10	0.1 seconds	10 seconds
15	3.27 seconds	33.7 seconds
20	1.75 minutes	1.3 minutes
25	0.93 hours	2.5 minutes
30	29.8 hours	4.5 minutes
35	39.7 days	7.14 minutes
40	3.4 years	10.66 minutes
45	1.08 centuries	15.18 minutes

C58 Series

Network Structures

Martin Gonzalez-Rodriguez, Ph D.

Network Data Structures

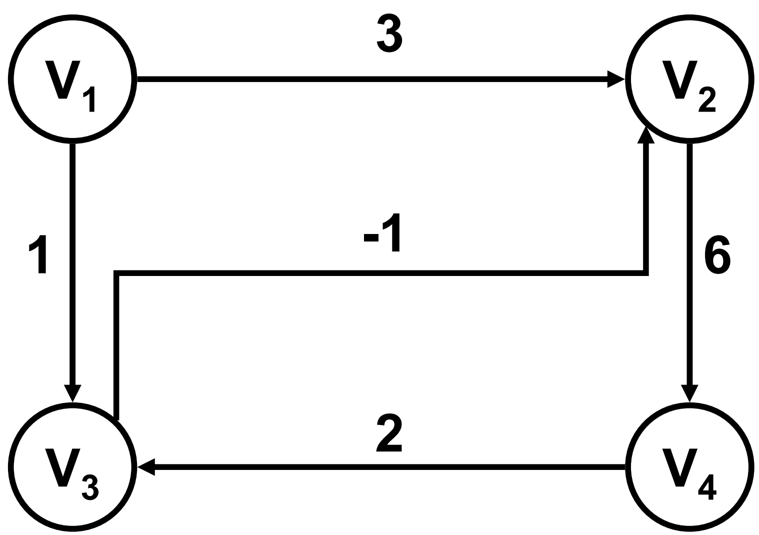
Goal

- Modeling complex conceptual relationships between objects.
  - Transport networks (roads, railways, underground, electricity, gas, oil, etc.).
  - Communication networks (Internet, phone, mail, etc.)
  - Social networks (Facebook, Instagram, debts, etc.).
  - Structures (molecular, neuronal, genetics, etc.).

Definition

What is a Graph?

- A graph is **mathematical model** that represents *arbitrary relationships* between objects.



Definition

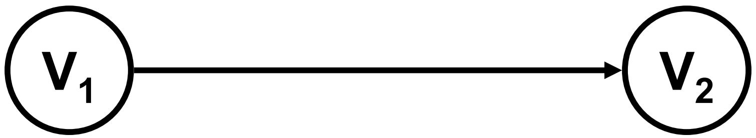
Formal Definition

- A Graph is a pair (V, E) represented by G(V, E) where:
  - **V** is a finite set of **Vertices** (also known as **Nodes)**.

$$\mathbf{V = \{V_1, V_2, ...\}}$$

  - **E** is a set of pairs (v, w) belonging to **V** called **edges**.
    - They represent relationships between the node v and the node w.

$$\mathbf{E = \{(V_1, V_2),...\}}$$

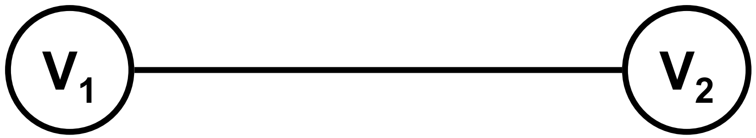


Typologies

Types of Graphs

- If the pairs {v,w} are ordered pairs…
  - They are called **Arcs** and the graph is known as ***directed graph*** *or* ***digraph***.

- If the pairs {v, w} **are not ordered…**
  - They are called **Edges** and the graph is known as ***undirected graph***.



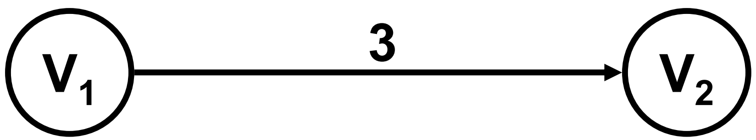
Categories

Types of Graphs

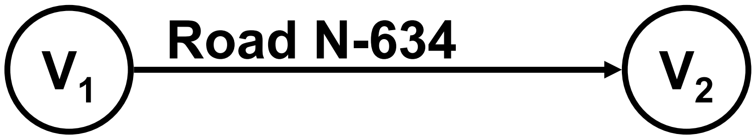
- A **Labeled Graph** is a trio (V, E, W) represented by G(V, E, W) where
  - W is a **finite set** of labels where **each arc or edge** has its own label.

$$\mathbf{W = \{W_1, W_2, \ldots\}}$$

  - The labels can be:
    - **Numbers**. These labels are called **Weights** and may represent costs or benefits.



    - **Characters** or Strings.



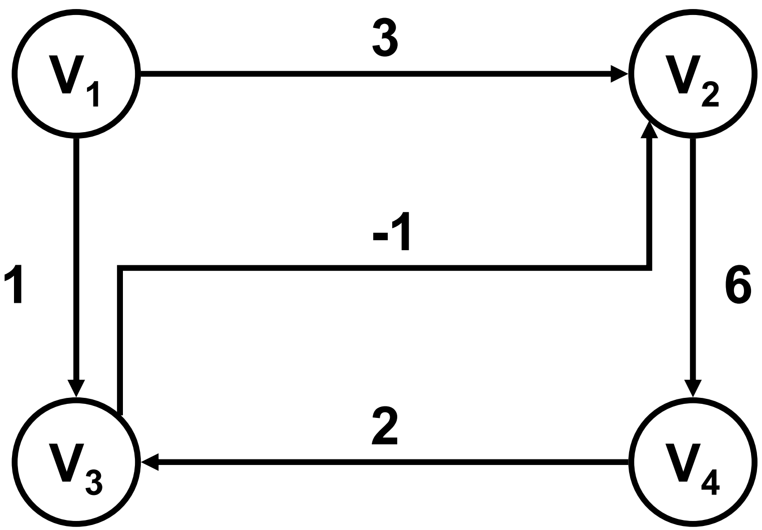
Putting it all Together

Complete formal definition

$$V = \{V_1, V_2, V_3, V_4\}$$

$$E = \{(V_1, V_2), (V_1, V_3), (V_2, V_4), (V_3, V_2), (V_4, V_3)\}$$

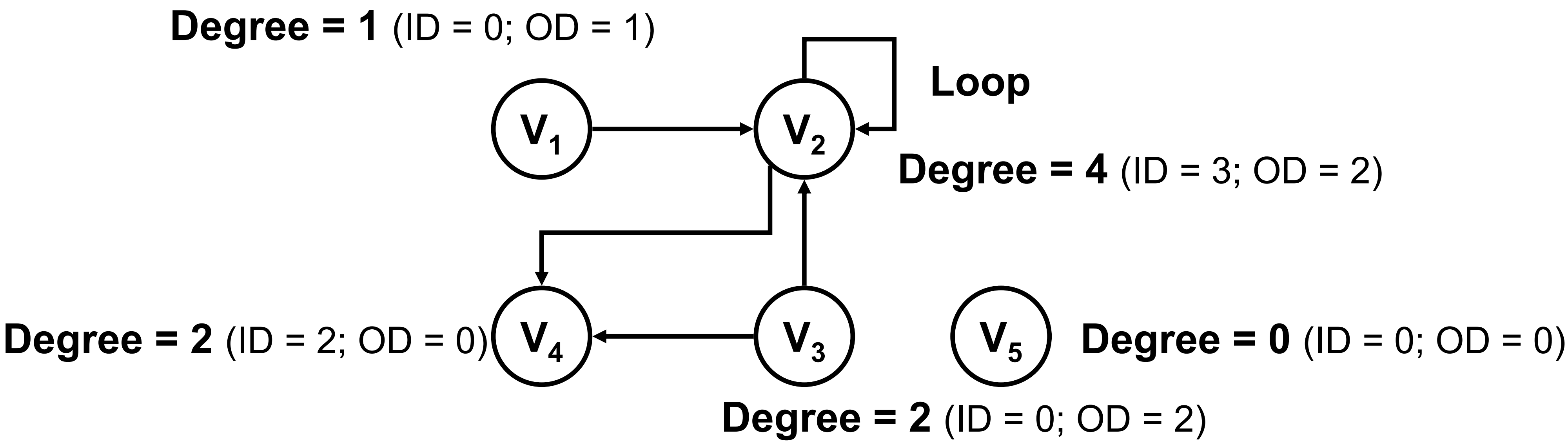
$$W = \{ 3, \quad 1, \quad 6, \quad -1, \quad 2\}$$



Fundamentals

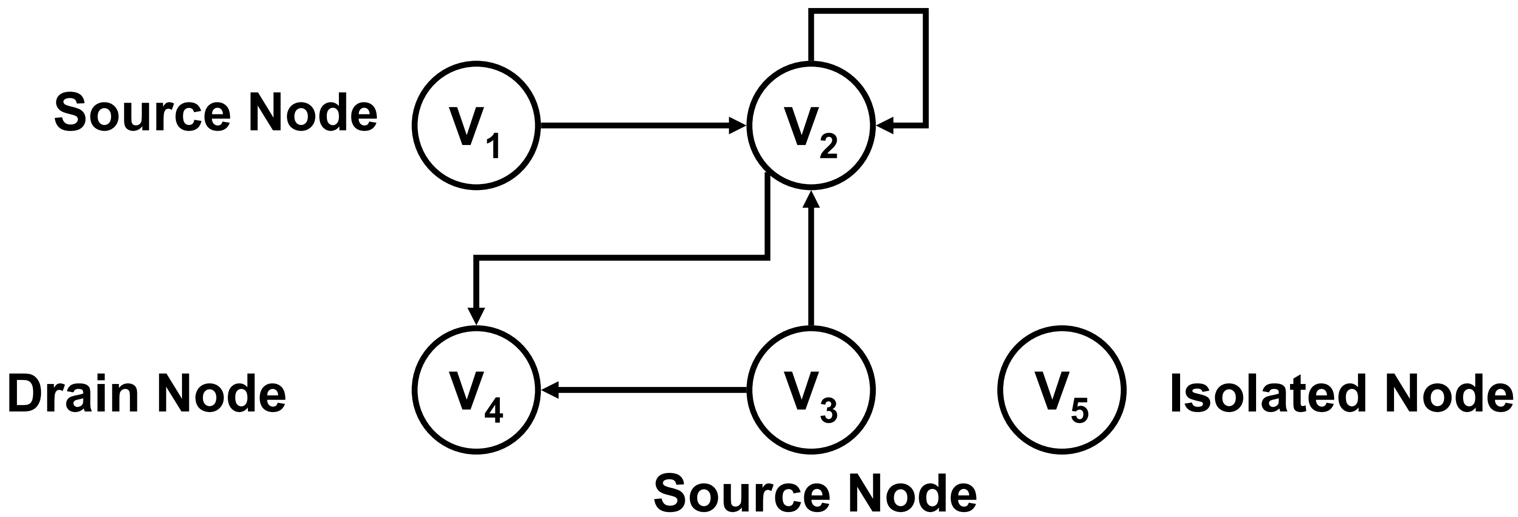
- **Loop**
  - Arc or edge where its departing and arrival node is the same one.

- **Degree** of a node
  - Number of arcs or edges connected to the node.
    - **Input Degree (ID)** of a node:
      - Number of arcs or edges that arrive to the node.
    - **Output Degree (OD)** of a node:
      - Number of arcs or edges that depart from the node.



Fundamentals

- Source node
  - If **OutputDegree > 0** and **InputDegree = 0**.
- Drain Node
  - If **OutputDegree= 0** and **InputDegree> 0**.
- Isolated Node
  - If **OutputDegree= 0** and **InputDegree= 0**.



Capacity of a Node

n = number of nodes in a graph

- n = Cardinality of the V set.

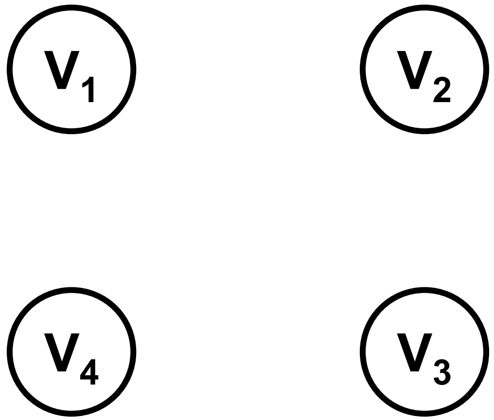
$$V = \{V_1, V_2, \ldots, V_{n-1}, V_n\}$$

- The value of n is used as a parameter to calculate the performance level of the graph's methods.

Capacity of a Node

Estimation of the number of arcs based on n

- $A_{min}(n)$: **Minimum** number of arcs

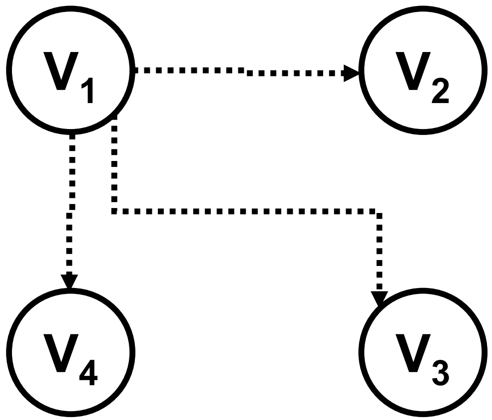


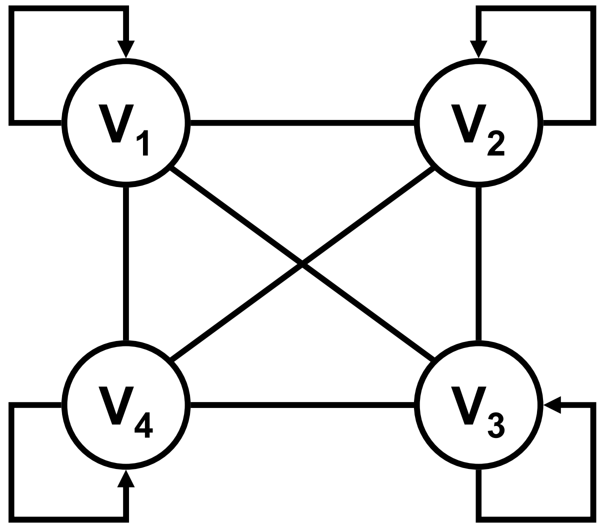
$\mathbf{A_{min}(n) = 0}$

Capacity of a Node

Estimation of the number of arcs based on n

- $A_{max}(n)$: **Maximum** number of arcs (**Complete Graph**)





$$\mathbf{A_{max}(n) = n(n - 1) = n^2 - n \text{ (without loops)}}$$

$$\mathbf{A_{max}(n) = n^2 - n + n = n^2 \text{ (including loops)}}$$

Memory Storage

Graph density

- **Heavy Graphs**: $A(n) \rightarrow n^2$.
  - Number or arcs close to the number of arcs in a complete graph
  - Maximum efficiency is reached when the graph is implemented on static memory (matrix, arrays).

- **Light Graphs**: $A(n) \rightarrow n$.
  - An average of one arc per node.
  - Maximum efficiency is reached when the graph is implemented on dynamic memory (lists) as it requires very few links.

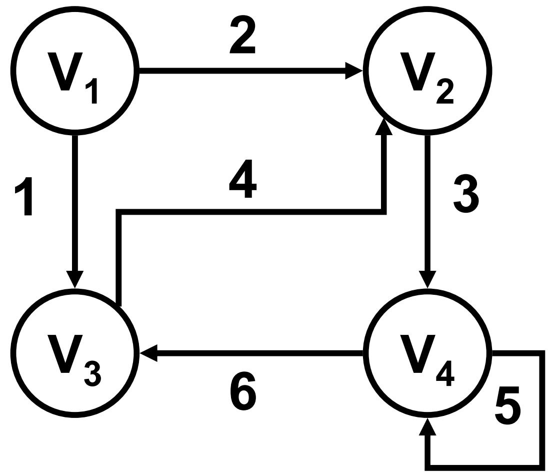
Graph Class – Matrix

Adjacency Matrix

```
ArrayList<GraphNode<T>> nodes;
private boolean[][] edges;
private double[][] weight;
int size; // number of nodes stored in the structure (nodes.size)
```

- **nodes**: stores objects of the node class.
- The cell **edges[i,j]** contains *true* **only when** there is an edge that departs from i and arrives to j.
- The cell **weight[i, j]** stores the weight of the edge that departs from i and arrives to j.
  - Weights *can be* null (0,0).
  - If this arc does not exist, its value is null (0,0).

Graph Class – Matrix



nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

edges

	0	1	2	3	4
0	F	T	T	F	
1	F	F	F	T	
2	F	T	F	F	
3	F	F	T	T	
4					

size = 4

weight

	0	1	2	3	4
0	0	2	1	0	
1	0	0	0	3	
2	0	4	0	0	
3	0	0	6	5	
4					

size = 4

Efficiency Analysis

Performance of Adjacency Matrixes

- Advantages
  - Random access to the information contained in any matrix cell.
    - Access O(1).

- Disadvantages
  - It is difficult to determine a efficient size for the matrix.
    - It should be the closed possible value to n.
  - Wastage of memory when used with light graphs (empty matrix).
    - Memory required: $O_M(n^2)$.

- Best scenario of application
  - Heavy graphs.

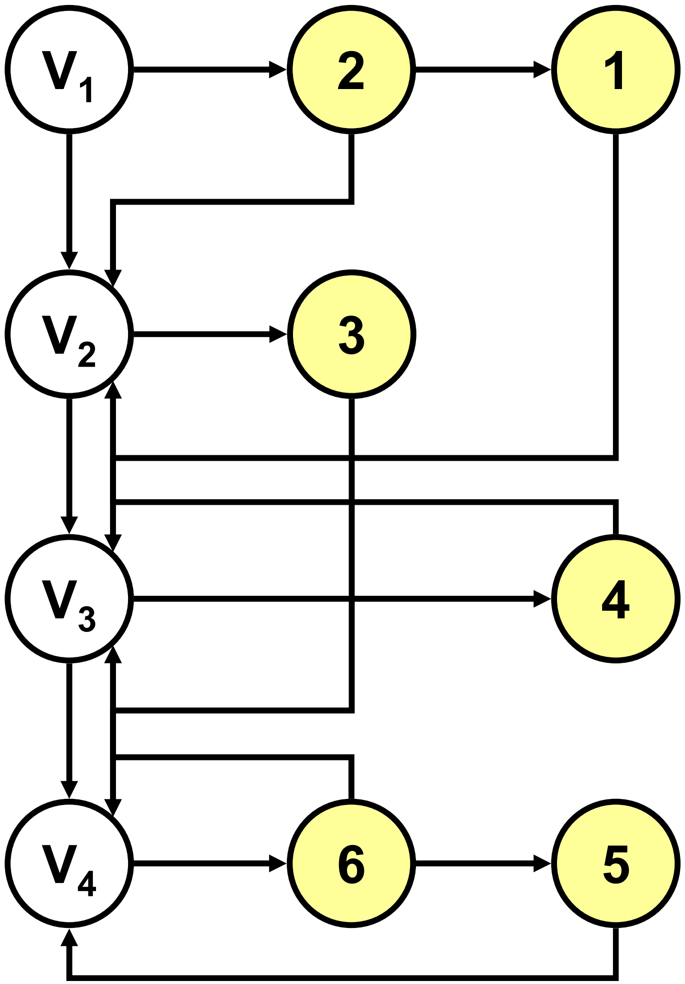
Graph Class – List

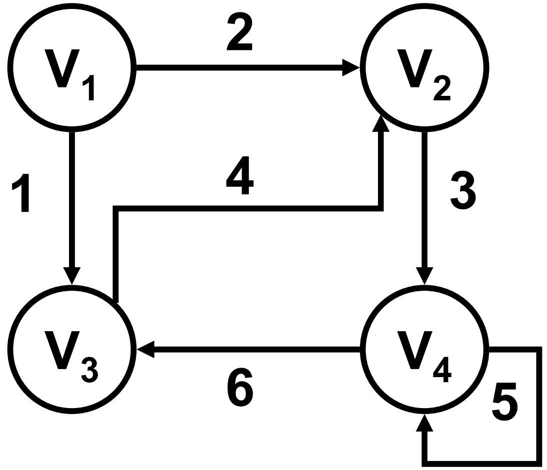
Adjacency List

```
class Edge{
        private double weight;
        private Node target;
}

class Node <T> {
        private T node;
        private LinkedList<Edge> edges;
}

private LinkedList<Node> nodes;
```

- Lists containing lists
  - The main list (*nodes*) contains a collection V of nodes.
  - Each list in this node contains a list including information regarding to its adjacent nodes (the *edges* collection).

Graph Class – List





Efficiency Analysis

Performance of the Adjacency Lists

- **Advantages**
  - The required memory depends on the actual number of nodes and the number of edges.
    - Storage required: $O_M(K_1 n + K_2 a)$, where K_1 = #bytes per node and K_2 = #bytes per arc.

- **Disadvantages**
  - It is required to make complex sequential searches over the lists.
    - Access O(n).
  - If the graph is heavy, there is a high memory wastage level related to the references (pointers) required to link the list nodes.
    - The highest level o memory wastage is produced in **complete graphs**.

- **Best scenario of application**
  - Light graphs.

Graph Class – Basic Methods

Adjacency Matrix

Method	**Complexity**
graph (constructor)	O(1)
getNode	O(n)
addNode	O(n)
removeNode	O(n)
existEdge?	O(n)
addEdge	O(n)
removeEdge	O(n)
print	O(n^2)

Graph Class – Basic Methods

graph (fragment)	**O(1)**
`size = 0;`	

nodes

0	1	2	3	4

size = 0

Graph Class – Basic Methods

getNode (Pseudo code) **O(n)**

```
public int getNode (T node)
{
 for (int i=0; i<size; i++)
  if (nodes[i].equals(node))
   return (i); // returns the node’s position

 return (-1); // search fails, node does not exist
}
```

nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

DISCUSSION: What is the temporal complexity of this algorithm?

Graph Class – Basic Methods

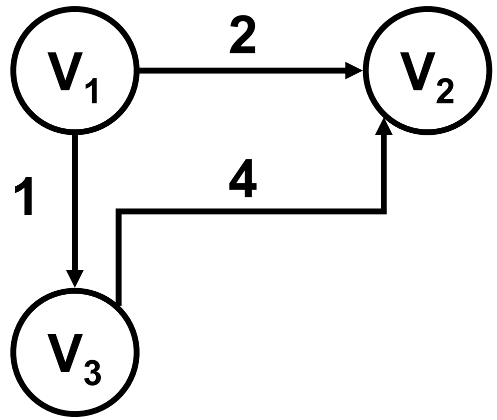
addNode (Pseudo code) **O(n)**

```
public void addNode (T node)
{
 // precondition: node does not exits and there is
 // available space for the node.

 if (getNode(node)== -1 && size<nodes.length)
 {
  nodes[size] = node;

  //inserts void edges
  for (int i=0; i<=size; i++)
  {
   edges[size][i]=false;
   edges[i][size]=false;
   weight[size][i]=0.0;
   weight[i][size]=0.0;
  }
  ++size;
 }
}
```

DISCUSSION: What is the temporal complexity of this algorithm?

Graph Class – Basic Methods

Before inserting V_4



nodes

0	1	2	3	4
V_1	V_2	V_3		

size = 3

edges

	0	1	2	3	4
0	F	T	T		
1	F	F	F		
2	F	T	F		
3					
4					

size = 3

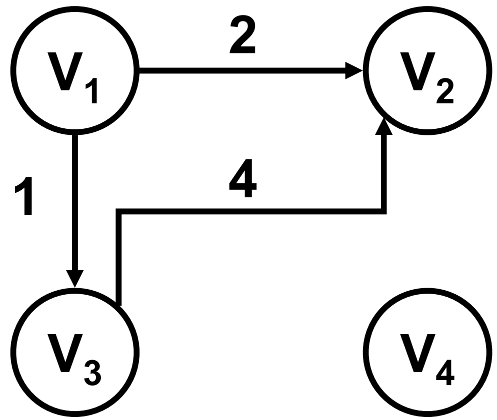
weight

	0	1	2	3	4
0	0	2	1		
1	0	0	0		
2	0	4	0		
3					
4					

size = 3

Graph Class – Basic Methods

After inserting V_4



nodes

0	1	2	3	4
V_1	V_2	V_3	V_4	

size = 4

edges

	0	1	2	3	4
0	F	T	T	F	
1	F	F	F	F	
2	F	T	F	F	
3	F	F	F	F	
4					

size = 4

weight

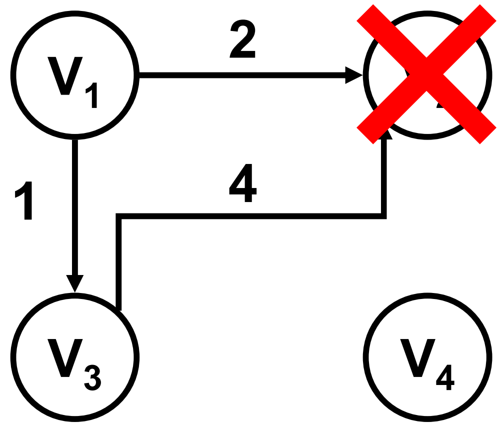
	0	1	2	3	4
0	0	2	1	0	
1	0	0	0	0	
2	0	4	0	0	
3	0	0	0	0	
4					

size = 4

DISCUSSION: How to delete V_2?

Graph Class – Basic Methods

After deleting V_2



nodes

0	1	2	3	4
V_1	V_4	V_3	V_4	

size = 3

edges

	0	1	2	3	4
0	F	F	T	F	
1	F	F	F	F	
2	F	F	F	F	
3	F	F	F	F	
4					

size = 3

weight

	0	1	2	3	4
0	0	0	1	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	0	0	0	
4					

size = 3

Graph Class – Basic Methods

removeNode (Pseudo code) **O(n)**

```
public void removeNode (T node){
   int i = getNode(node);

   if (i>=0) {
    --size;
    if (i != size+1) { // it is not the last node
     nodes[i] = nodes[size]; //replaces by the last node

     //replace elements in the vectors edges and weights
     for (int j=0; j<=size; j++) {
      edges[j][i]=edges[j][size];
      edges[i][j]=edges[size][j];
      weight[i][j]=weight[size][j];
      weight[j][i]=weight[j][size];
      }
      // loop (diagonal)
      edges[i][i] = edges[size][size];
      weight[i][i] = weight[size][size];
    }
}
```

DISCUSSION: How to implement the *existsEdge* method?

Graph Class – Basic Methods

existsEdge (Pseudo code) **O(n)**

```
public boolean existsEdge (T origin, T destination)
{
 int i=getNode(origin);
 int j=getNode(destination);

 // precondition: both nodes must exist.
 // if don't… should we throw an exception?

 if (i>=0 && j>=0)
  return(edges[i][j]);
 else
  return (false);
}
```

DISCUSSION: How to implement the *addEdge* method?

Graph Class – Basic Methods

addEdge (Pseudo code) **O(n)**

```
public void addEdge (T origin, T destination, double
edgeWeight)
{

 // precondition: the edge must not already exist.
 if (!existEdge(origin, destination))
 {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=true;
  weight[i][j]=edgeWeight;
 }
 else
  ; // what about throwing an exception here?
}
```

DISCUSSION: How to implement the *removeEdge* method?

Graph Class – Basic Methods

removeEdge (Pseudo code)	**O(n)**

```
public void removeEdge (T origin, T destination){

 // precondition: the edge must exist.
 if (existsEdge(origin, destination)) {
  int i=getNode(origin);
  int j=getNode(destination);

  edges[i][j]=false;
  weight[i][j]=0.0;
 }
 else
  ; // what about throwing an exception?
}
```

DISCUSSION: How to implement the *print* method?

Graph Class – Basic Methods

print (Pseudo code) **$O(n^2)$**

```
public void print(){

 for (int k=0; k<size; k++)
  nodes[k].print();

 for (int i=0; i<size; i++) {
  for (int j=0; j<size; j++) {
   System.out.print(edges[i][j] + "(");
   System.out.print(weight[i][j] + ") ");
  }
  System.out.println();
}
```

DISCUSSION: What is the temporal complexity of this algorithm?

Graph Class – Advanced Methods

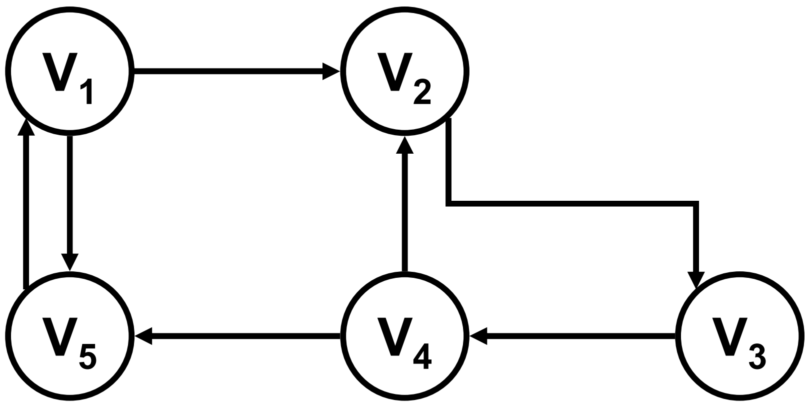
Adjacency Matrix

Method	**Complexity**
Dijkstra	$O(n^2)$
Floyd	$O(n^3)$
Depth-first search	$O(n^2)$
Prim / Warshall	$O(n^2)$

More Graph Fundamentals

- **Pathway** between two nodes V_i, V_j ($V_i \neq V_j$)
  - Sequence of nodes (and their related edges) that allow to access node V_j from node V_i.
    - Pathway between V_1 and V_5
      - » $C_A = V_1, V_5$.
      - » $C_B = V_1, V_2, V_3, V_4, V_5$.
      - » $C_C = V_1, V_2, V_3, V_4, V_2, V_3, V_4, V_5$.
      - » …

- **Length** of a path between two nodes V_i, V_j ($V_i \neq V_j$)
  - Numbers of edges required to reach V_j.
  - It is the number of nodes in the pathway **minus one**.
    - Longitude of pathways between V_1 and V_5
      - » $L(C_A) = 1$.
      - » $L(C_B) = 4$.
      - » $L(C_C) = 7$.

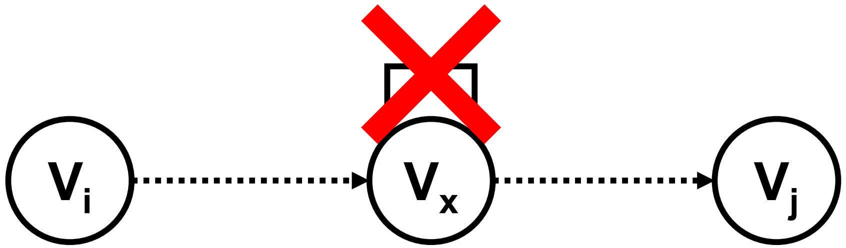


PROBLEM: pathways of infinite length

More Graph Fundamentals

- **Simple pathway** between two nodes V_i, V_j ($V_i \neq V_j$)
  - Is a pathway that does not contain any node more than one time.

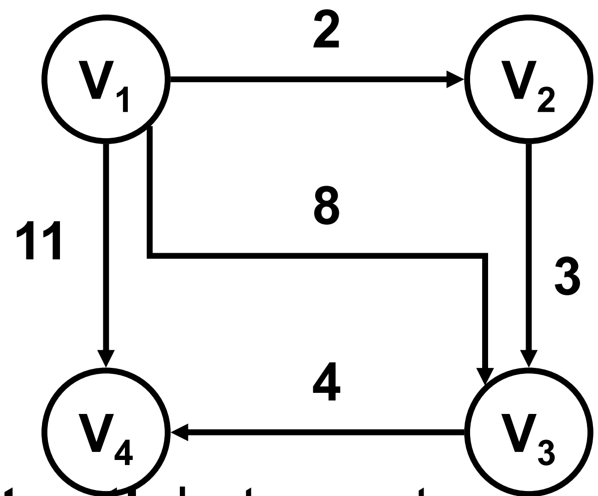
Simple pathway theorem
If there is a pathway between a the nodes V_i (origin) and V_j (destiny), the **there is at least a simple pathway** between V_i and V_j.



- It is **possible** to eliminate loops and cycles along the way to convert it into a **simple pathway**.

More Graph Fundamentals

- **Minimum Length pathway** between two nodes V_i, V_j ($V_i \neq V_j$)
  - It is the path that uses the minimum number of arcs.
    - The Minimum longitude pathway **is simple**.
    - Minimum longitude pathway between V_1 and V_4
      - » $C_A = V_1, V_4$ (Longitude 1).

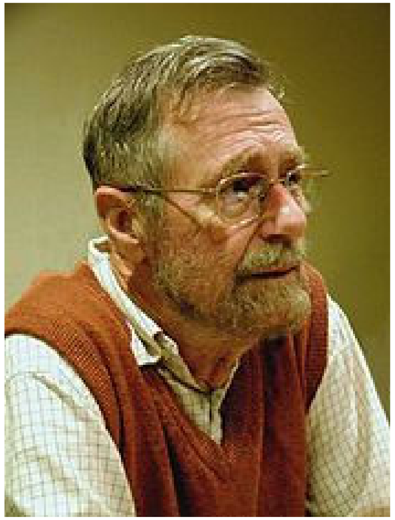


- **Minimum cost path** between two nodes V_i, V_j ($V_i \neq V_j$)
  - It is the path that uses those arcs whose sum of weights is the minimum possible.
    - Minimum cost path between V_1 and $V_{4.}$ $C_A = V_1, V_2, V_3, V_4$ (Cost 9).

Dijkstra Algorithm

Problem to solve

- Which is the minimum cost path to reach every node in a graph departing from a specified node ***v***?
  - Which is the cheapest route for going to Barcelona **from Oviedo**?
  - Which is the shortest path to reach Madrid **from Oviedo**?
    - And the route to Valencia? And the pathway to Seville? And to Bilbao?... **From Oviedo**.



Edger Dijkstra (Wikipedia)

- Developed by the Dutchman researcher Edger Dijkstra in 1956
  - Turing Award 1972.

Dijkstra Algorithm

Products obtained

- **Vector D** (one-dimensional) AKA Minimum Costs
  - Stores the minimum cost **value** for going from ***v*** to every other node in the graph.

- **Vector P** (one-dimensional) AKA Minimum Cost Paths
  - Stores the minimum cost **path** for going from ***v*** to every other node in the graph.

Vector D

V_2	V_3	V_4	V_5	V_6
1	5	3	6	∞

Minimum cost for going from V_1 to all the other nodes

Vector P

2	3	4	5	6
1	4	1	3	-

V_3 is reached via V_4

Accessing V_5 requires visiting V_3 first

Dijkstra Algorithm

Initialization

- Initial values for the **Set S**
  - Nodes whose minimum access cost from ***v*** is already known.
  - Started with node ***v***. It is the only one whose minimum access cost is already known (cost from ***v*** to ***v*** is 0).
    - S = {v}.

- Initial values for the **Vector D** of Minimum Cost
  - Copy the row related to node ***v*** from a modified **weight** vector…
    - **…replacing** the values containing a cost equal to 0 by ∞.
    - The **cost** to move from one node to another one **using a** (direct) **way that does not exist is infinite**.
    - During the first iteration, only the costs of moving from ***v*** to any other node **using a direct path** (size = 1) are already known.

Dijkstra Algorithm

Example

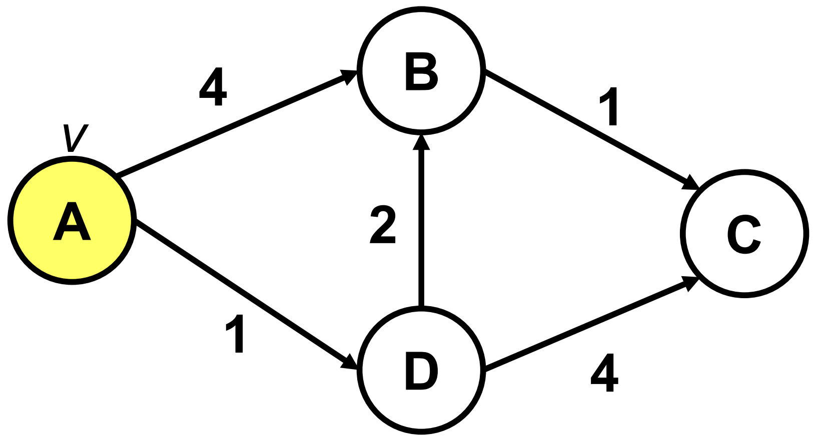
S = {A}

Vector D

B	C	D
4	∞	1

Vector P

B	C	D
-	-	-



Dijkstra Algorithm

Example

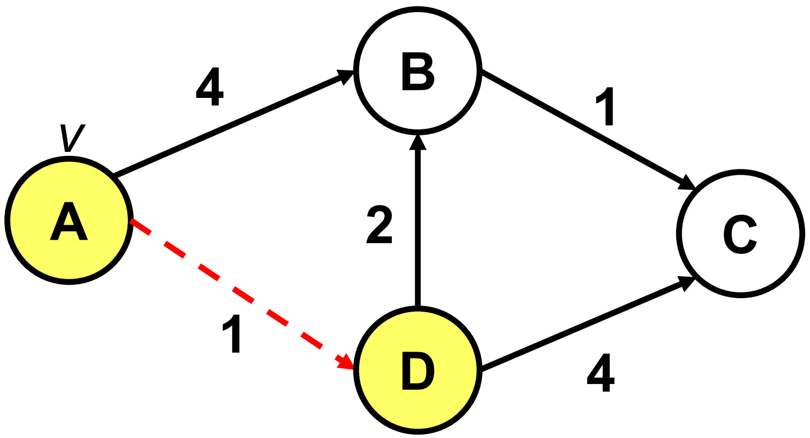
S = {A, D}

Vector D

B	C	D
3	5	1

Vector P

B	C	D
D	D	-



Dijkstra Algorithm

Example

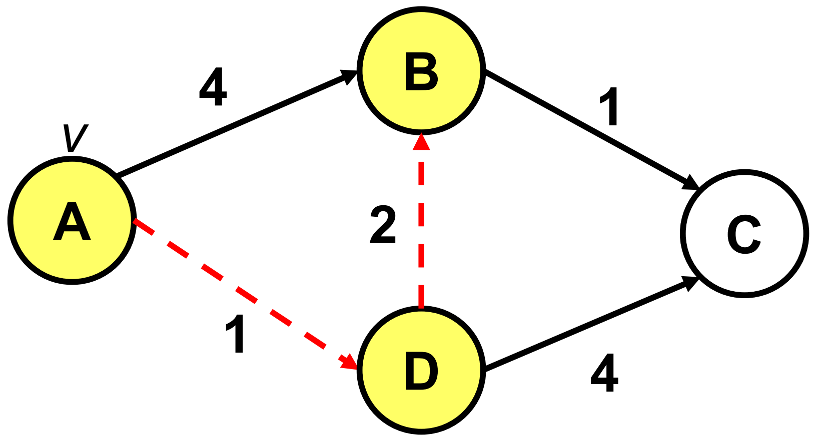
S = {A, D, B}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Dijkstra Algorithm

Example

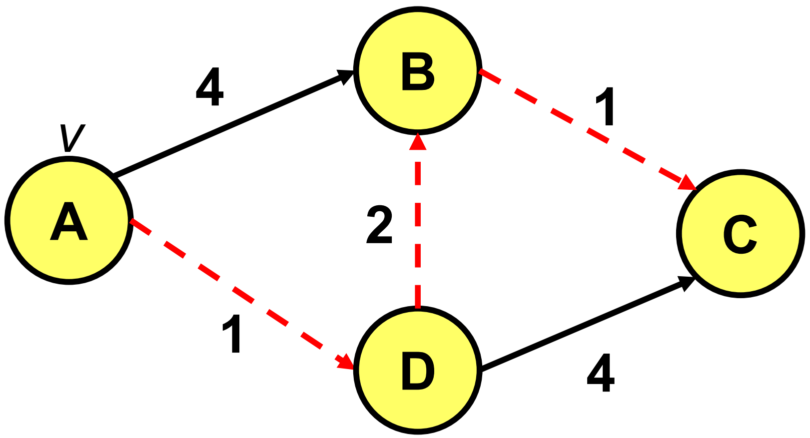
S = {A, D, B, C}

Vector D

B	C	D
3	4	1

Vector P

B	C	D
D	B	-



Dijkstra Algorithm

The Algorithm

For each iteration…

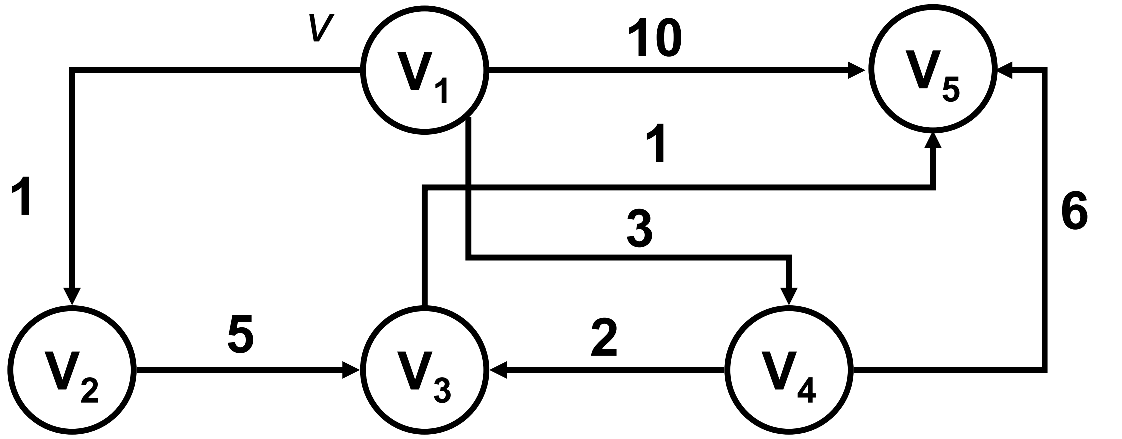
```
1. Evaluate the cost of every arc {k, w} where k belongs
   to the S set and w belongs to the V-S set.
2. Select the arc of minimum cost, adding w to the S set.
    a. w is the node with the lowest cost in D!
3. For each node m in V-S update costs:

    if (D[w] + weight[w][m] < D[m]) {
     D[m] = D[w] + weight[w][m];
     P[m] = w;
    }
```

- **Stopping condition**
  - **Set S == Set V** (all the nodes in the graph have been evaluated).
    - **n – 1 iterations** done.

Dijkstra Algorithm

Exercise

- Cost from V_1.

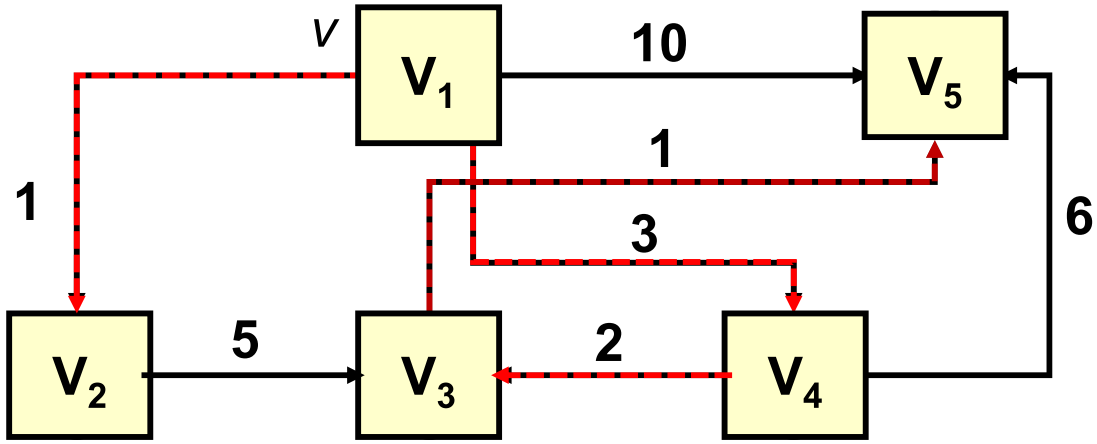


it	S	w	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		1	∞	3	10	--	1	-	1	1	-

Dijkstra Algorithm

Exercise

- Cost from V_1.



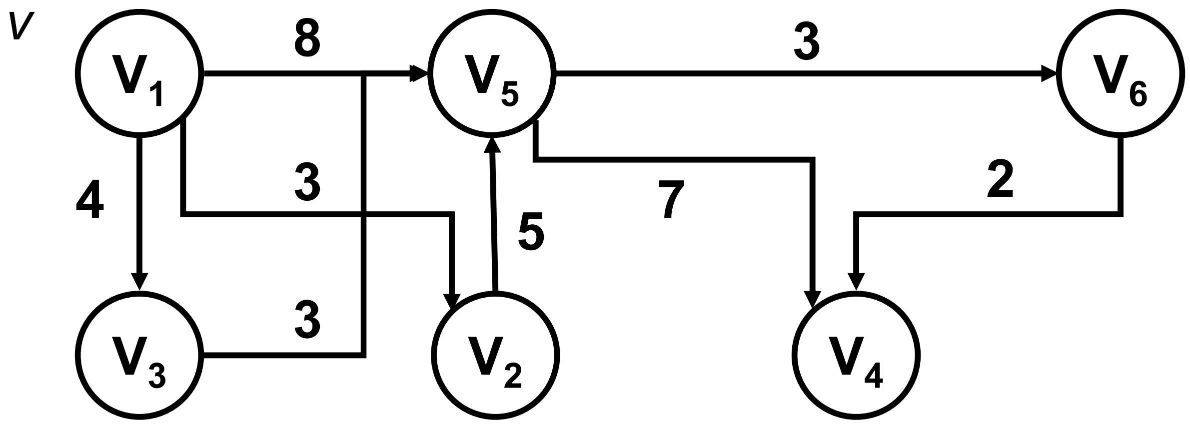
it	S	w	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		1	∞	3	10	--	1	-	1	1	-
2	1, 2	2	1	6	3	10	--	1	2	1	1	-
3	1, 2, 4	4	1	5	3	9	--	1	4	1	4	-
4	1, 2, 3, 4	3	1	5	3	6	--	1	4	1	3	-
5	1, 2, 3, 4, 5	5	1	5	3	6	--	1	4	1	3	-

It is your turn!

Dijkstra Algorithm

Exercise

- Cost from V_1.

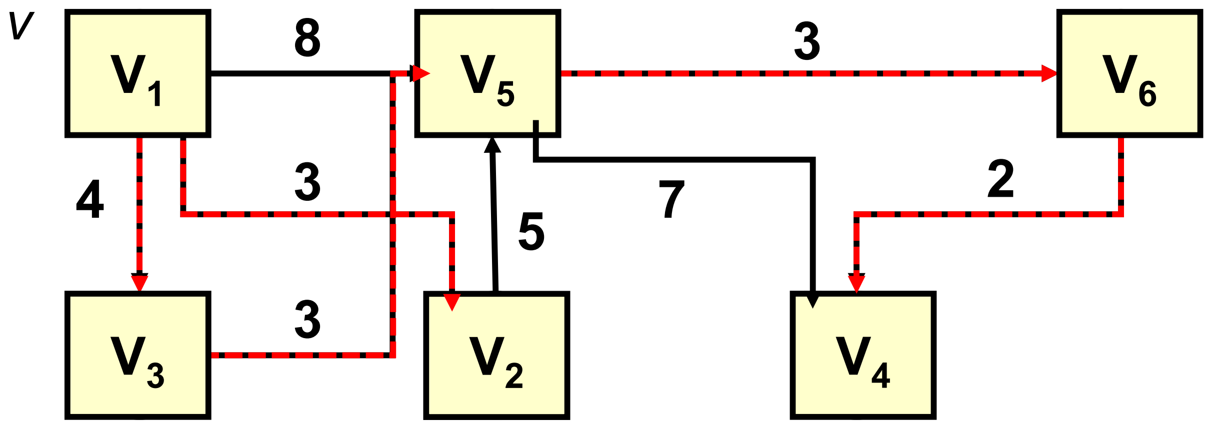


it	S	W	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-

Dijkstra Algorithm

Exercise

- ❖ Cost from V_1.



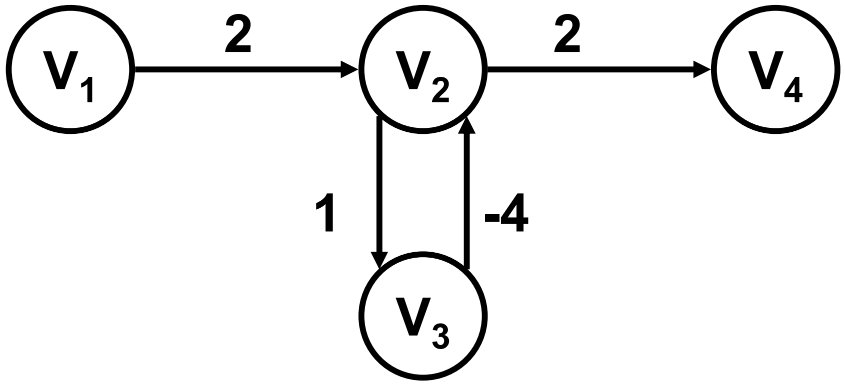
it	S	W	Vector D: V_2	V_3	V_4	V_5	V_6	Vector P: 2	3	4	5	6
1	1		3	4	∞	8	∞	1	1	-	1	-
2	1, 2	2	3	4	∞	8	∞	1	1	-	1	-
3	1, 2, 3	3	3	4	∞	7	∞	1	1	-	3	-
4	1, 2, 3, 5	5	3	4	14	7	10	1	1	5	3	5
5	1, 2, 3, 5, 6	6	3	4	12	7	10	1	1	6	3	5
6	1, 2, 3, 4, 5, 6	4	3	4	12	7	10	1	1	6	3	5

Dijkstra Algorithm

Conclusions

- Dijkstra assumes costs of going from one node to itself as 0
  - Therefore, D[v] is not calculated.

- The algorithm does not work with negative costs (bonuses)
  - The minimum cost path may not be a simple one!



The minimum cost path between V_1 and V_4 includes a infinite loop between V_2 and V_3

- It can calculate the **Minimum Length Path** for a graph too
  - **Substitute cost for 1 in *weight*!**

DISCUSSION: What is the temporal complexity of this algorithm?

Dijkstra Algorithm

Temporal Complexity

For each iteraction… n – 1 iteractions

```
1. Evaluate the cost of every arc {k, w} where k is owned
   by the S set and w by the V-S set.
2. Select the arc of minimum cost, adding w to the S set.
    a. w is the node with the lowest cost in D!
```
O(n)

```
3. For each node m in V-S do:

    if (D[w] + weight[w][m] < D[m]) {
     D[m] = D[w] + weight[w][m];
     P[m] = w;
    }
```
O(n)

$O(n^2)$

Floyd-Warshall Algorithm

Problem to solve

- Calculates minimum costs between ***any*** pair of nodes
  - What is the cheapest way to get to Barcelona from Oviedo, Seville or Burgos?
    - Should we run Dijkstra n times? (one time per departing node?).

Robert Floyd (Wikipedia)

Stephen Warshall (Wikipedia

- Developed by American researchers Robert Floyd and Stephen Warshall in 1962

Floyd-Warshall Algorithm

Obtained Products (1/2)

- **Vector A** AKA Minimum Cost Vector
  - Stores the minimum cost for going **from any node to every one else in the graph**.

Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
V_6	∞	∞	∞	2	∞	0

Floyd-Warshall Algorithm

Obtained Products(2/2)

- **Vector P** AKA Minimum Cost Paths
  - Stores the sequence of nodes part of **all the paths of minimum cost**.

printPath (fragment)

```
private void printPath(int i, int j)
{
 int k = P[i][j];
 if (k>0) {
  printPath (i, k);
  System.out.print ('-' + k);
  printPath (k, j);
 }
}

System.out.print (departure);
printPath (departure, arrival);
System.out.println ('-' + arrival);
```

Matrix P	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

TEST: Print the path between V_1 and V_6

Floyd-Warshall Algorithm

Starting

- Initial values for the **Vector A** (minimum cost values)
  - Copy the values of a modified **weight** vector in the same way as Dijkstra's algorithm does
    - **Change** the values of cost 0 by ∞.
    - **But… include values of 0 in the main diagonal** (costs of going from a node to itself are considered null).

Floyd-Warshall Algorithm

The Algorithm

Floyd (fragment) **O(n³)**

```
for (int k=0; k<size; k++)                O(n)
 for (int i=0; i<size; i++)               O(n)
  for (int j=0; j<size; j++)              O(n)
    if (A[i][k] + A[k][j] < A[i][j])
    {
        A[i][j] = A[i][k] + A[k][j];
        P[i][j] = k;
    }
```

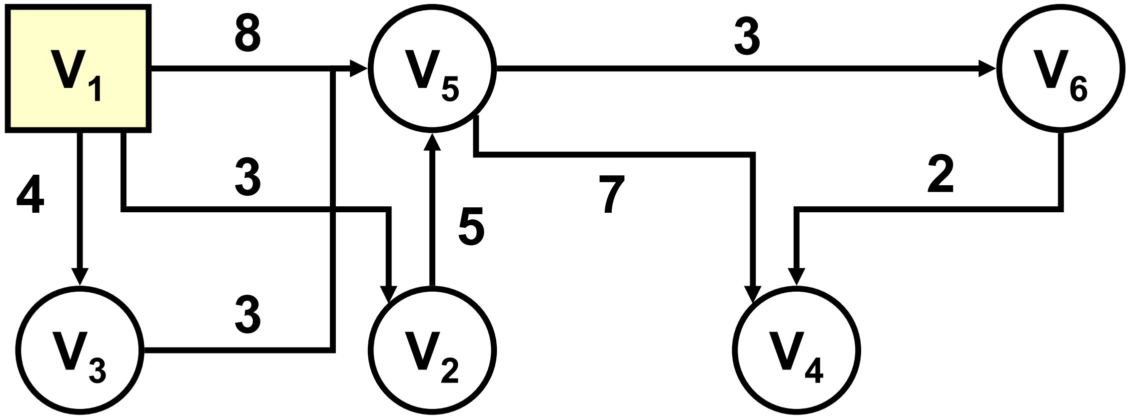
- For each iteration, the node k is evaluated (all paths must go through that node)
  - There are **n iterations**
    - Equivalent to adding nodes into the S set in Dijkstra.
  - Every iteration calculates the cost of going from **any node i** to any **other node j** through the **node k**.
    - **If the cost of using k is lower** than the recorded so far in vector A, the value of A[i,j] and P[i,j] must be updated indicating that the minimum cost path uses k.

DISCUSSION: What is the temporal complexity of this algorithm?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_0 (V_1)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

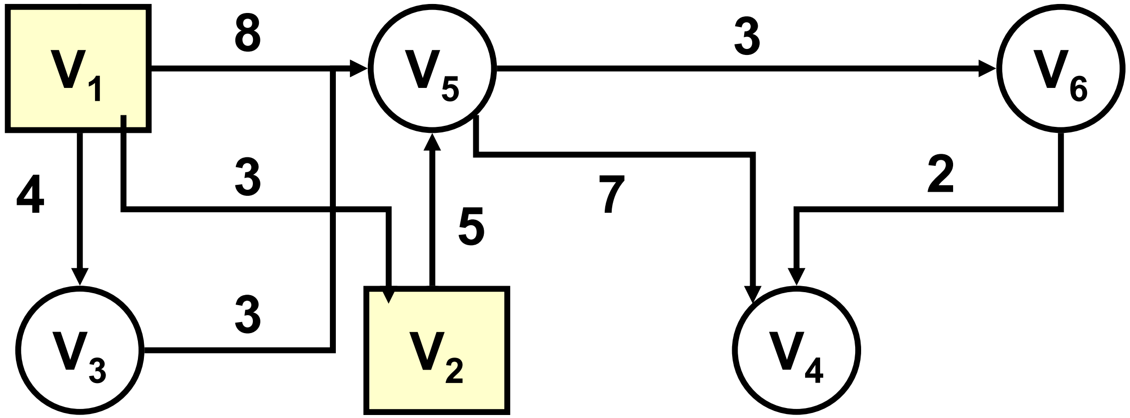
Vector P	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

Going from V_2 to V_3 via V_1 (cost ∞ + 4 = ∞) is cheaper that going with cost ∞?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_1 (V_2)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	8	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

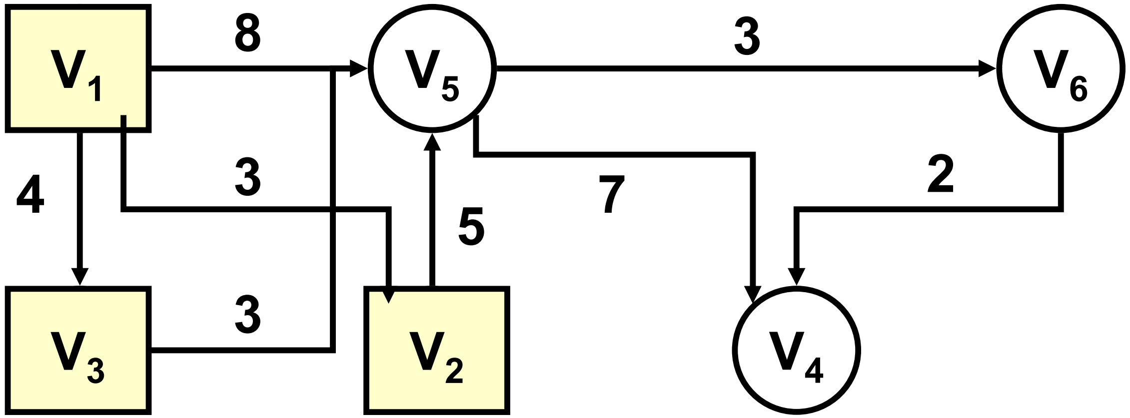
Vector P	1	2	3	4	5	6
V_1	-	-	-	-	-	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

Going from V_1 to V_5 via V_2 (cost 3 + 5 = 8) is cheaper than going with cost A_0 8?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_2 (V_3)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

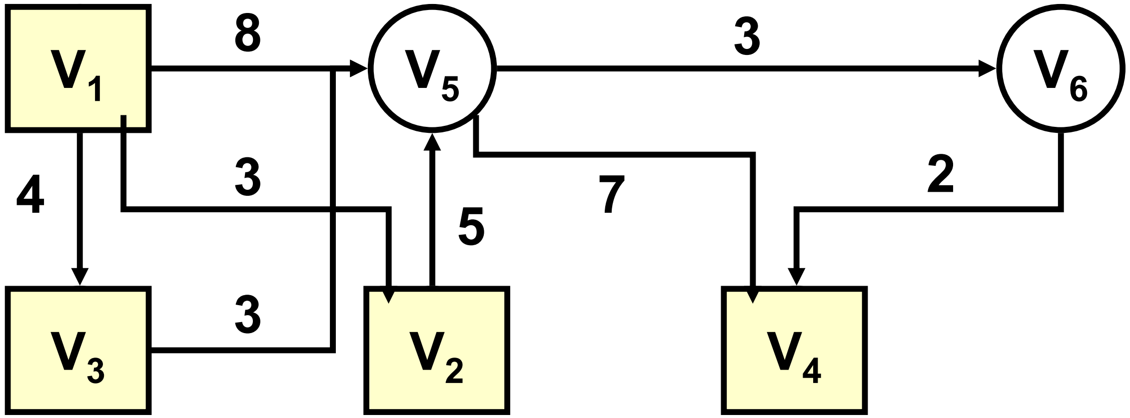
Vector P	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

Going from V_1 to V_5 via V_3 (cost 4 + 3 = 7) is cheaper than going with cost A_1 (8)?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_3 (V_4)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	∞	7	∞
V_2	∞	0	∞	∞	5	∞
V_3	∞	∞	0	∞	3	∞
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

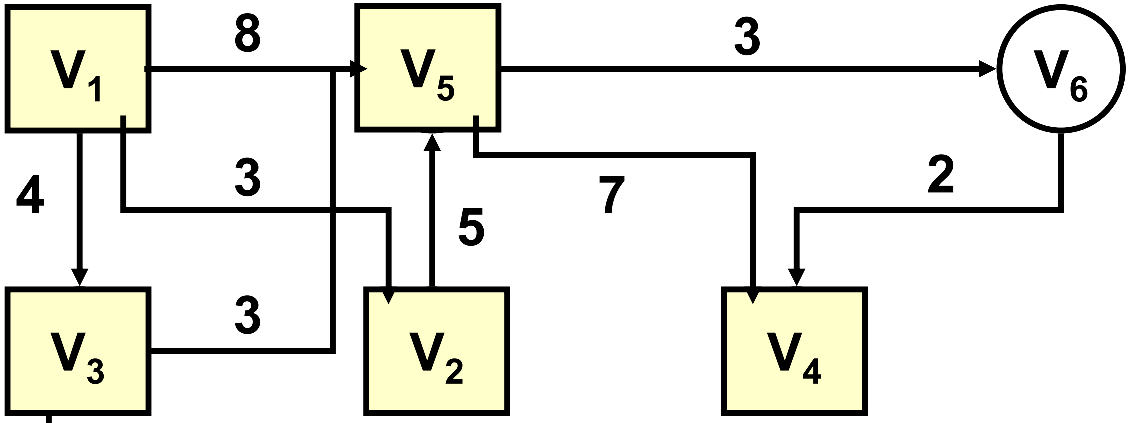
Vector P	1	2	3	4	5	6
V_1	-	-	-	-	3	-
V_2	-	-	-	-	-	-
V_3	-	-	-	-	-	-
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

Going from V_5 to V_6 via V_4 (cost 7 + ∞ = ∞) is cheaper than going with cost A_2 (3)?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_4 (V_5)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	14	7	10
V_2	∞	0	∞	12	5	8
V_3	∞	∞	0	10	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	7	0	3
V_6	∞	∞	∞	2	∞	0

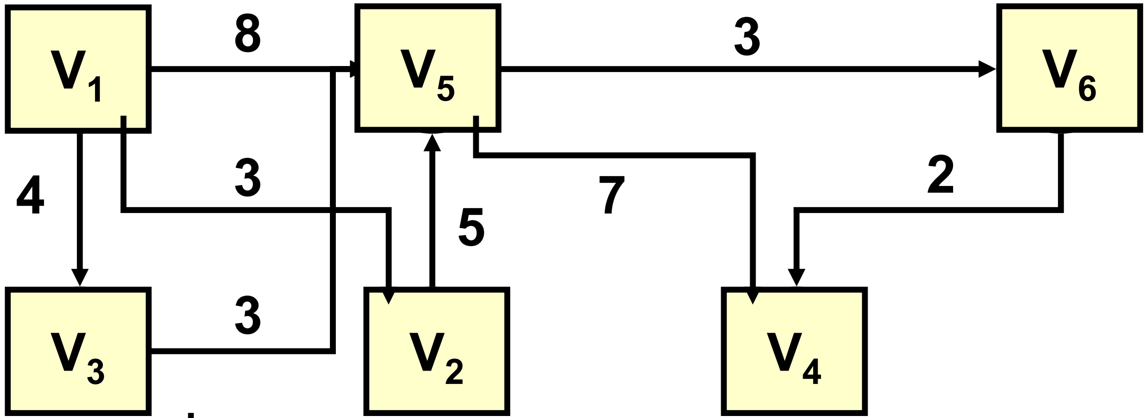
Vector P	1	2	3	4	5	6
V_1	-	-	-	5	3	5
V_2	-	-	-	5	-	5
V_3	-	-	-	5	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	-	-	-
V_6	-	-	-	-	-	-

Going from V_1 to V_4 via V_5 (cost 7 + 7 = 14) is cheaper than going with cost A_3 (∞)?

Floyd-Warshall Algorithm

Exercise 1

- Vector A_5 (V_6)



Vector A	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	3	4	12	7	10
V_2	∞	0	∞	10	5	8
V_3	∞	∞	0	8	3	6
V_4	∞	∞	∞	0	∞	∞
V_5	∞	∞	∞	5	0	3
V_6	∞	∞	∞	2	∞	0

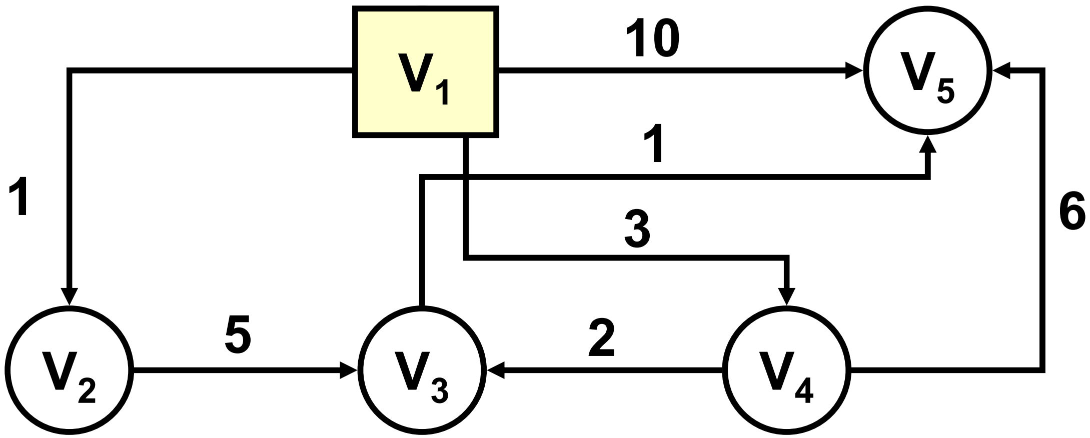
Vector P	1	2	3	4	5	6
V_1	-	-	-	6	3	5
V_2	-	-	-	6	-	5
V_3	-	-	-	6	-	5
V_4	-	-	-	-	-	-
V_5	-	-	-	6	-	-
V_6	-	-	-	-	-	-

Going from V_1 to V_4 va V_6 (cost 10 + 2 = 10) is cheaper than going with cost A_4 (14)?

Floyd-Warshall Algorithm

Exercise 2

- Vector A_0 (V_1)



Vector A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

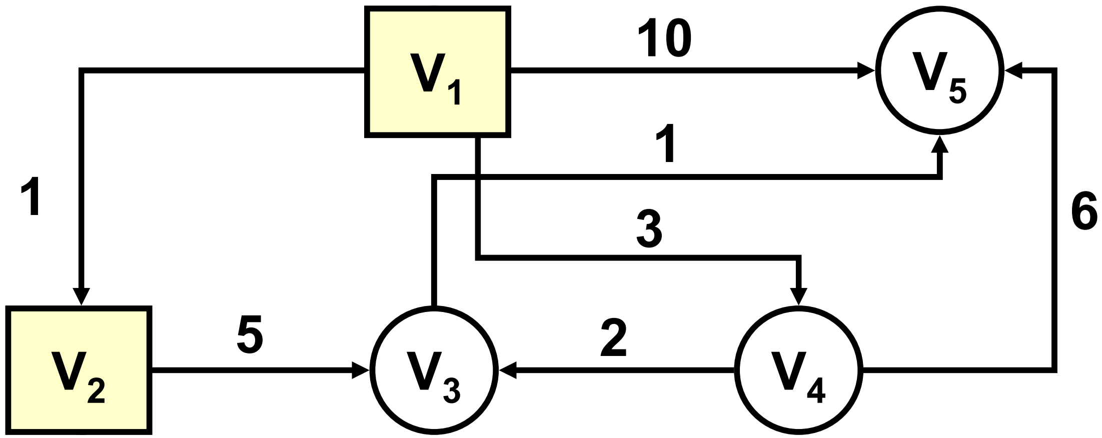
Vector P	1	2	3	4	5
V_1	-	-	-	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

Going from V_4 to V_5 via V_1 (cost ∞ + 10 = ∞) is cheaper than going with cost 6?

Floyd-Warshall Algorithm

Exercise 2

- Vector A_1 (V_2)



Vector A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	10
V_2	∞	0	5	∞	∞
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	6
V_5	∞	∞	∞	∞	0

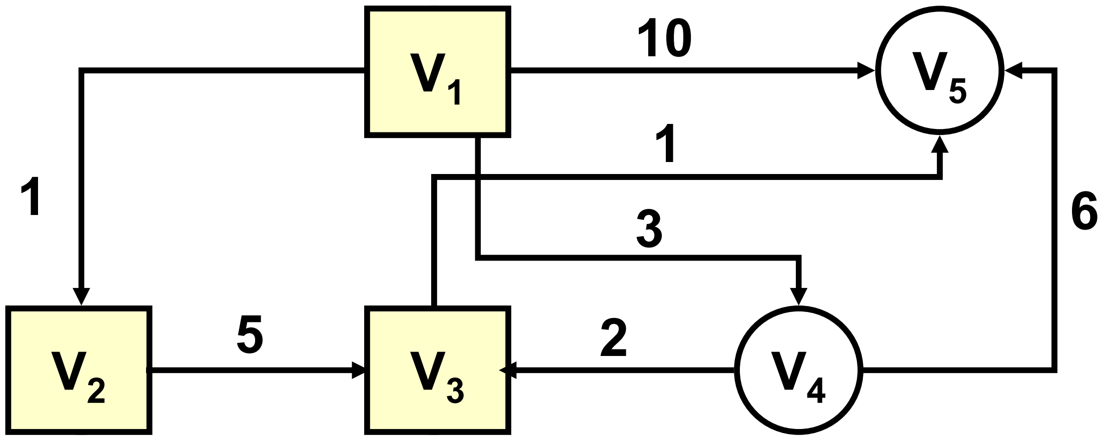
Vector P	1	2	3	4	5
V_1	-	-	2	-	-
V_2	-	-	-	-	-
V_3	-	-	-	-	-
V_4	-	-	-	-	-
V_5	-	-	-	-	-

Going from V_1 to V_3 via V_2 (cost 1 + 5 = 6) is cheaper than going with cost A_0 (∞)?

Floyd-Warshall Algorithm

Exercise 2

- Vector A_2 (V_3)



Vector A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	6	3	7
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

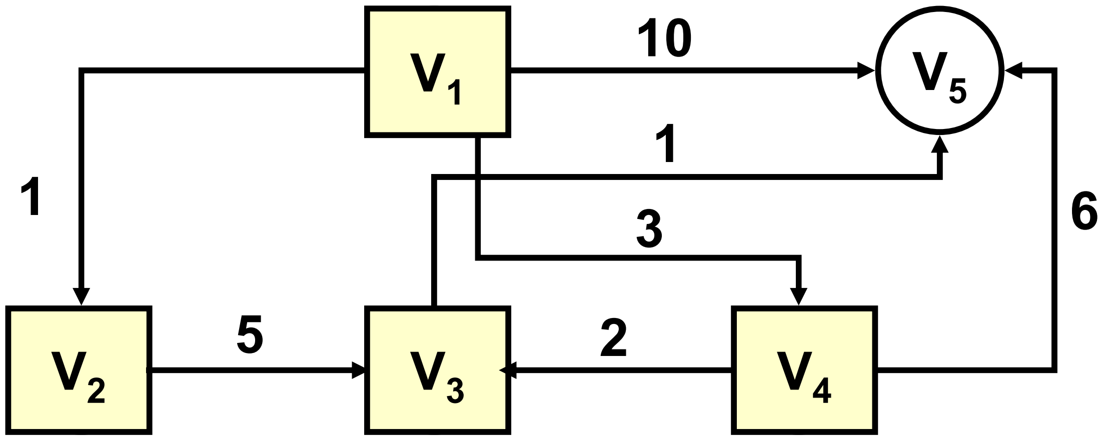
Vector P	1	2	3	4	5
V_1	-	-	2	-	3
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

Going from V_1 to V_5 via V_3 (cost 6 + 1 = 7) is cheaper than going with cost A_1 (10)?

Floyd-Warshall Algorithm

Exercise 2

- Vector A_3 (V_4)



Vector A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

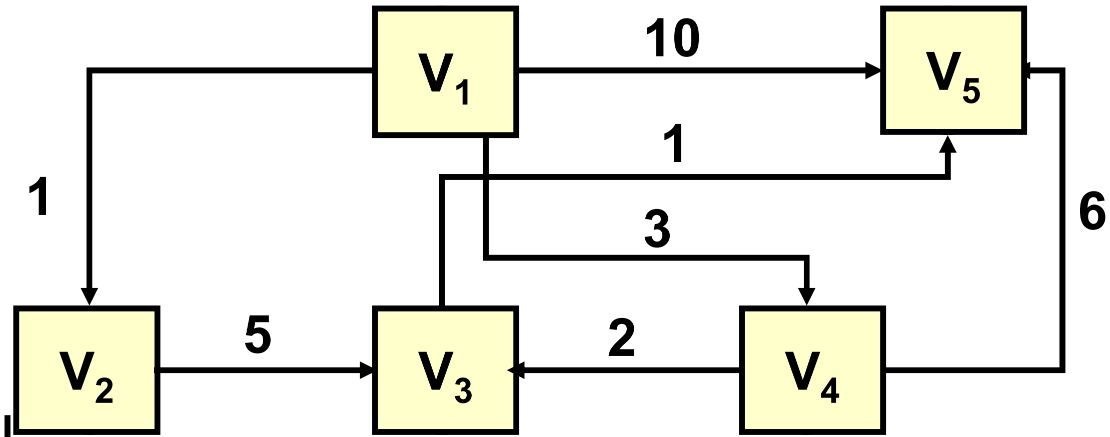
Vector P	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

Going from V_1 to V_3 via V_4 (cost 3 + 3 = 5) is cheaper than going with cost A_2 (6)?

Floyd-Warshall Algorithm

Exercise 2

- Matriz A_4 (V_5)



Vector A	V_1	V_2	V_3	V_4	V_5
V_1	0	1	5	3	6
V_2	∞	0	5	∞	6
V_3	∞	∞	0	∞	1
V_4	∞	∞	2	0	3
V_5	∞	∞	∞	∞	0

Vector P	1	2	3	4	5
V_1	-	-	4	-	4
V_2	-	-	-	-	3
V_3	-	-	-	-	-
V_4	-	-	-	-	3
V_5	-	-	-	-	-

Going from V_1 to V_2 via V_5 (cost 6 + ∞ = ∞) is cheaper than going with cost A_3 (1)?

Floyd-Warshall Algorithm

Floyd for special routes

- It is possible to modify the algorithm to calculate paths going through a **specific set of nodes L**.

Floyd (fragment)

```
    for (int k=0; k<size; k++)
     if (k in L)
      for (int i=0; i<size; i++)
       for (int j=0; j<size; j++)
        if (A[i][k] + A[k][j] < A[i][j])
        {
            A[i][j] = A[i][k] + A[k][j];
            P[i][j] = k;
        }
```

Floyd-Warshall Algorithm

Center of a Directed Graph

- The center of a graph is the node ***v*** **closest to the farthest node**.
  - Where should be placed the distribution center for a region?
  - Where should be placed the central railway or main hospital in a city?

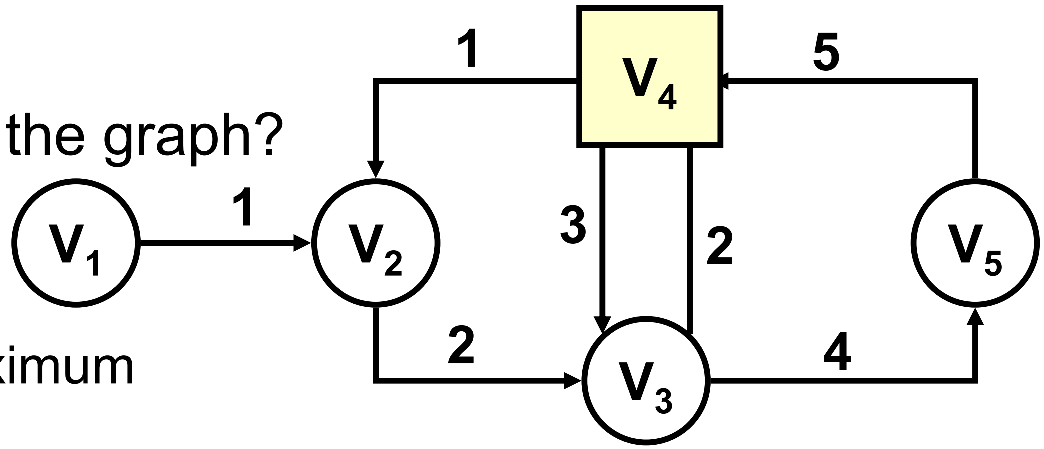
- Eccentricity
  - The eccentricity of a node ***v*** is the **maximum of the costs** of all the paths of minimum costs with destination ***v***.
  - The center of graph is located in the node with the **minimum** eccentricity.

Algorithm to obtain the center of graph
1. Run Floyd to obtain the **vector of minimum cost A.** 2. Search for **the maximum cost in each column** (eccentricity for each destination node). 3. Select the node **with the minimum eccentricity** as the center of the graph.

Floyd-Warshall Algorithm

Exercise

- What node is the center of the graph?



Get the minimum of the maximum

Vector A Original

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	∞	∞	∞
V_2	∞	0	2	∞	∞
V_3	∞	∞	0	2	4
V_4	∞	1	3	0	∞
V_5	∞	∞	∞	5	0

Vector A Final

	V_1	V_2	V_3	V_4	V_5
V_1	0	1	3	5	7
V_2	∞	0	2	4	6
V_3	∞	3	0	2	4
V_4	∞	1	3	0	7
V_5	∞	6	8	5	0

Pick up the maximum in each column

Depth-First Search (DFPrint)

Problem to Solve

- Visit all the nodes in a graph from an initial node. Follow the path pointed by its edges.
  - Based on the strategy of visiting the children nodes first *(depth-first)*.
  - It is necessary to verify the visited nodes somehow.

resetVisited **O(n)**

```
public void resetVisited ()
{
 for (int i=0; i<size; i++)
  nodes[i].setVisited(false);
}
```

Depth-First Search (DFPrint)

Problem to Solve

- Visit all the nodes in a graph from an initial one. Follow the path pointed by its edges.
  - Based on the strategy of visiting the children nodes first *(depth-first)*.
  - It is necessary to verify the visited nodes somehow.

Deep-first print (pseudo code)

```
public void DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  for each node w accessible from v do
   if (!nodes[w].getVisited())
    DFPrint(w);
}
```

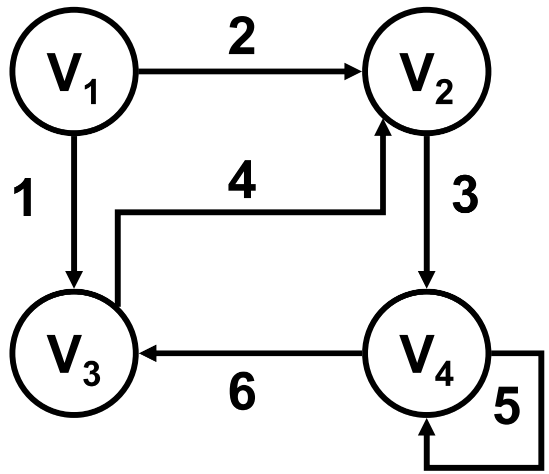
Depth-First Search (DFPrint)

Exercise *DFPrint* (V_1)

- Before visiting V_1

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
F	F	F	F	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



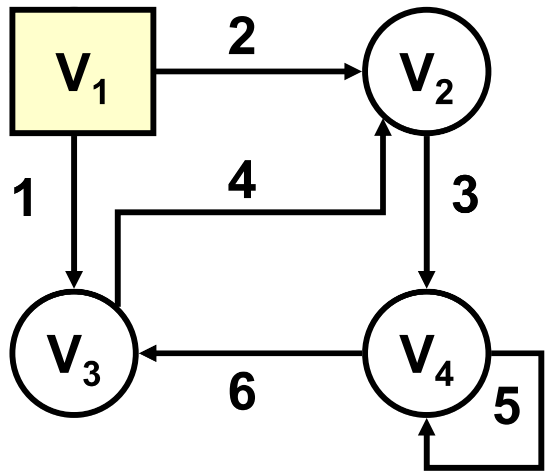
Depth-First Search (DFPrint)

Exercise *DFPrint (V_1)*

- Visit V_1

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	F	F	F	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



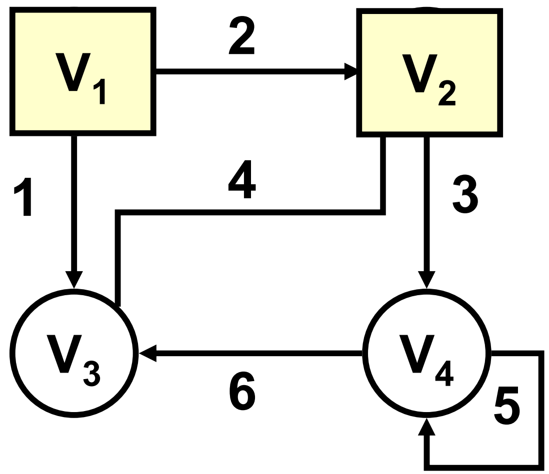
Depth-First Search (DFPrint)

Exercise *DFPrint* (V_1)

- Visit V_2

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	T	F	F	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



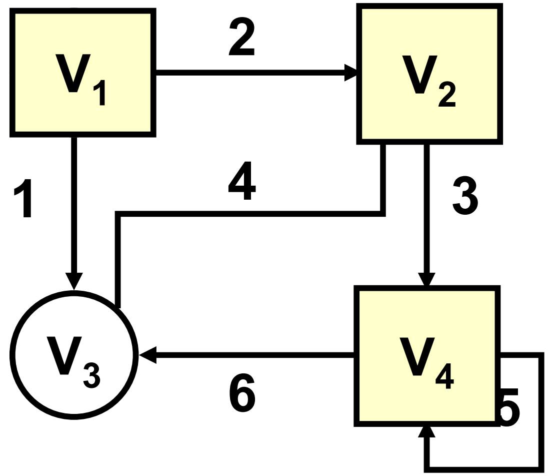
Depth-First Search (DFPrint)

Exercise *DFPrint* (V_1)

- Visit V_4

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	T	F	T	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	

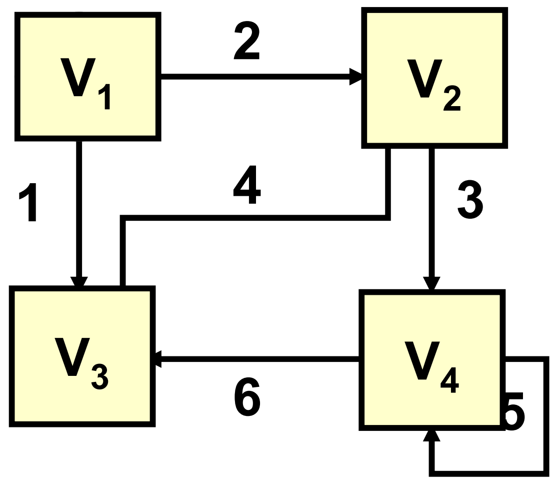


Depth-First Search (DFPrint)

Exercise *DFPrint* (V_1)

- Visit V_3

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	T	T	T	



	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	

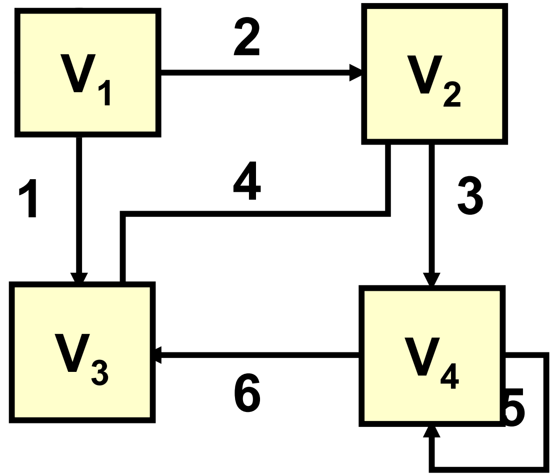
Depth-First Search (DFPrint)

Exercise *DFPrint (V_1)*

- Continue visit in V_4

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	T	T	T	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	

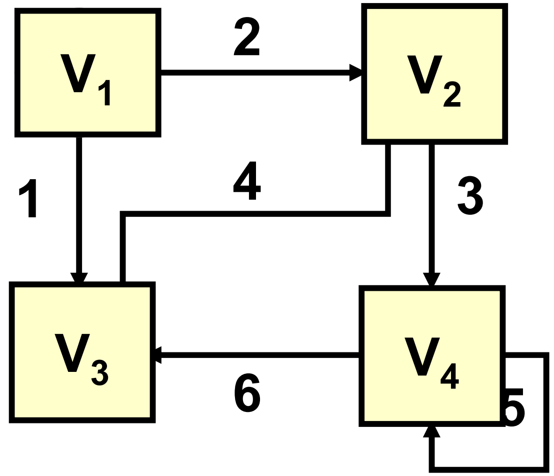


Depth-First Search (DFPrint)

Exercise *DFPrint (V_1)*

- Continue visit in V_1

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
T	T	T	T	



	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	n
V_3	F	T	F	F	
V_4	F	F	T	T	
		n			

$O(n^2)$

DISCUSSION: What is the temporal complexity of this algorithm?

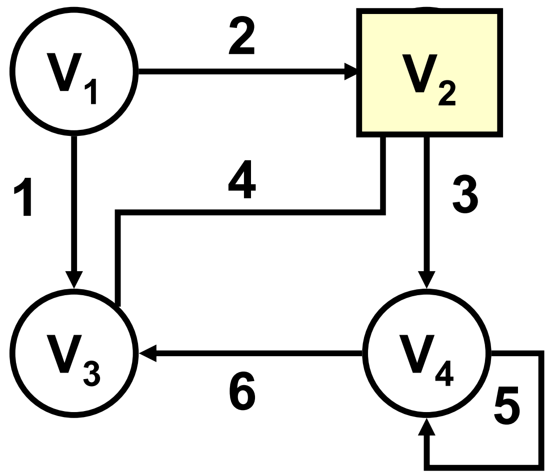
Depth-First Search (DFPrint)

Exercise *DFPrint* (V_2)

- Visit V_2

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
F	T	F	F	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



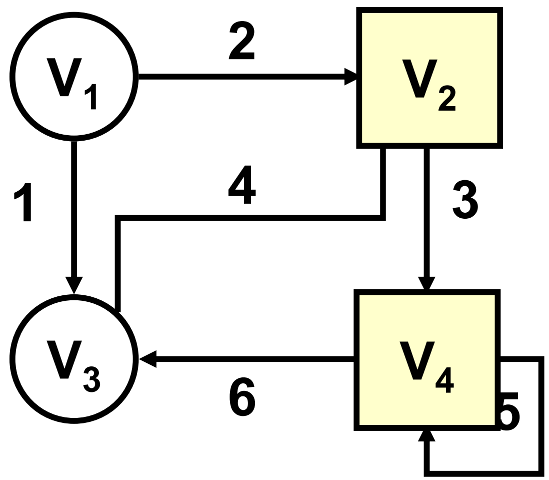
Depth-First Search (DFPrint)

Exercise *DFPrint (V_2)*

- Visit V_4

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
F	T	F	T	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



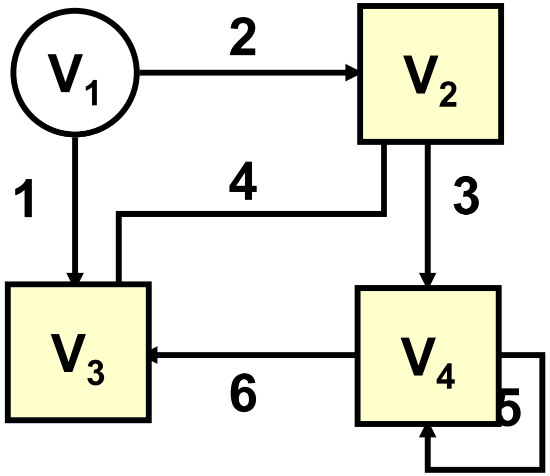
Depth-First Search (DFPrint)

Exercise *DFPrint (V_2)*

- Visit V_3

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



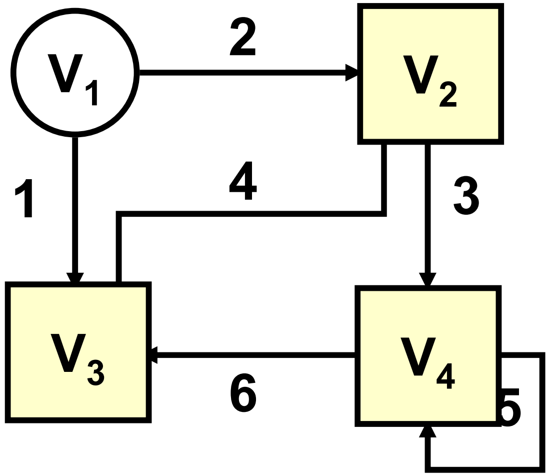
Depth-First Search (DFPrint)

Exercise *DFPrint (V_2)*

- Continue visit in V_4

0	1	2	3	nodes
V_1	V_2	V_3	V_4	
F	T	T	T	

	1	2	3	4	arcs
V_1	F	T	T	F	
V_2	F	F	F	T	
V_3	F	T	F	F	
V_4	F	F	T	T	



Depth-First Search (DFPrint)

Making sure to visit all the nodes along the graph

Special call to *DFPrint*

```
resetVisited();

For (int i=0; i<size; i++)
 if (!nodes[i].getVisited())
   DFPrint (i);
```

Depth-First Search (DFPrint)

Depth **first** search

- Improvement in the *DFPrint* algorithm to stop its execution once a condition is verified true in a specific node.

DFSearch (pseudocode)

```
public boolean DFPrint(int v){
  nodes[v].setVisited(true);
  nodes[v].print();

  if (boolean_condition(v))
   return (true);

   for each node w accessible from v do
    if (!nodes[w].getVisited())
     DFPrint(w);

  return (false);
}
```

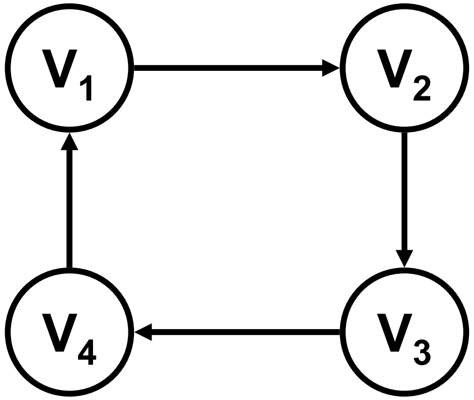
More Fundamentals

❖ **Strongly connected node**

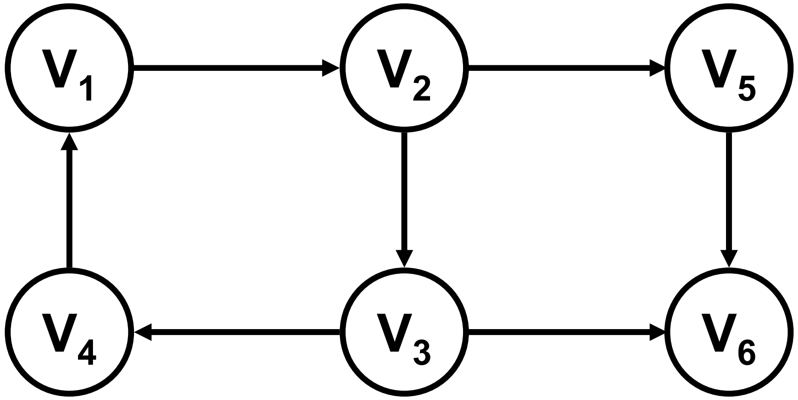
- When there is a direct path from every node to anyone else **and** vice versa.

❖ **Strongly connected graph**

- If **all the nodes** in the graph are strongly connected.
  - If there is a strongly connected node in the graph, everyone else will be strongly connected as well, and therefore the graph itself.



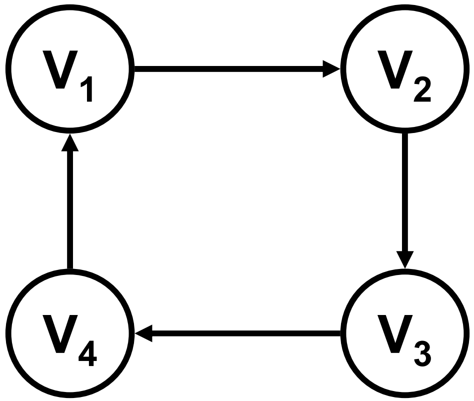
Strongly connected graph



Weakly connected graph (see V_6)

More Fundamentals

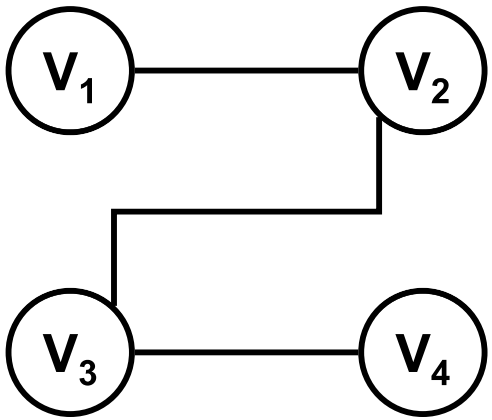
- **Cycle over a node**
  - Path from one node to itself.
    - Cycle for V_1
      - C = V_1, V_2, V_3, V_4 (longitude 4).



More Fundamentals

❖ Trees

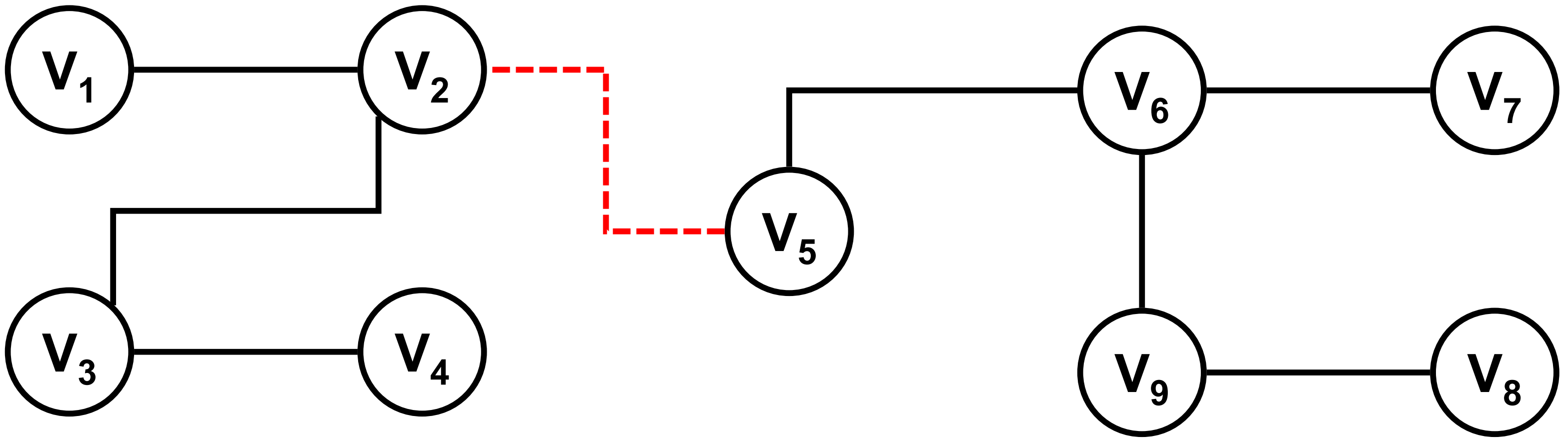
- Connected graph without cycles
  - Any tree with n > 0 nodes, has n - 1 edges.
  - If we add an extra edge it will become part of a cycle (the graph would not be a tree anymore!).
  - For any pair of nodes, there would be only one simple path connecting them them.



More Fundamentals

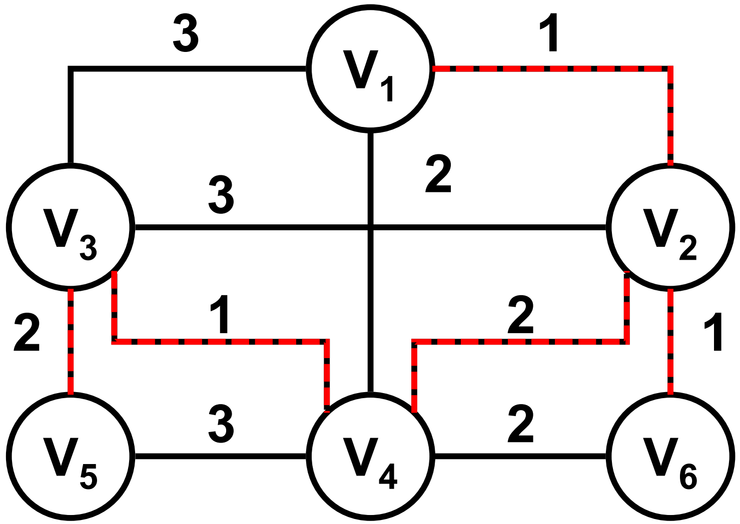
❖ Spanning Trees

- It is a tree that connects **all the nodes** in a given graph.



More Fundamentals

- **Minimum Spanning Tree**
  - It a tree where the sum of the weights of its edges reaches the minimum possible.
    - Allows to connect all the components in a network in the cheapest possible way.



Prim's Algorithm

Problem to Solve

- Obtains the minimum spanning tree
  - Which roads should be built to connect all the European cities in the cheapest way?
  - How to connect all the computers in a city with the minimum amount of cable?

Robert C. Prim (Wikipedia)

- Developed by the American researcher Robert C. Prim in 1957.

Prim's Algorithm

Initialization

- **T Set** (empty)
  - Stores the edges part of the Minimum Spanning Tree.

- **U set** (starts with any node in the graph)
  - Similar to the S set in the Dijkstra's algorithm. It stores the nodes evaluated in each iteration.

For each iteration (while U != V)

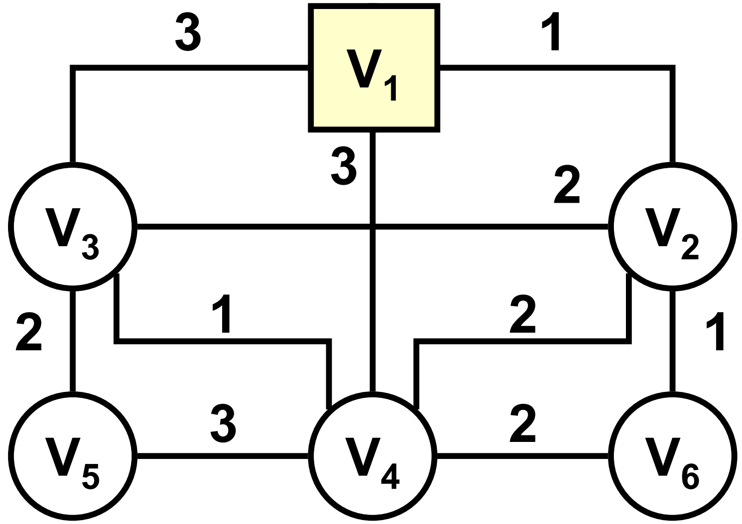
`1. Evaluate all the edges {u, v} where u is part of U and v is part of V - U selecting the edge with the lowest cost`
`2. T = T + {u, v}`
`3. U = U + {v}`

- **Stopping Condition**
  - **U Set == V Set** (all nodes in the graph have been explored).
    - **n – 1 iterations.**

Prim's Algorithm

Exercise 1

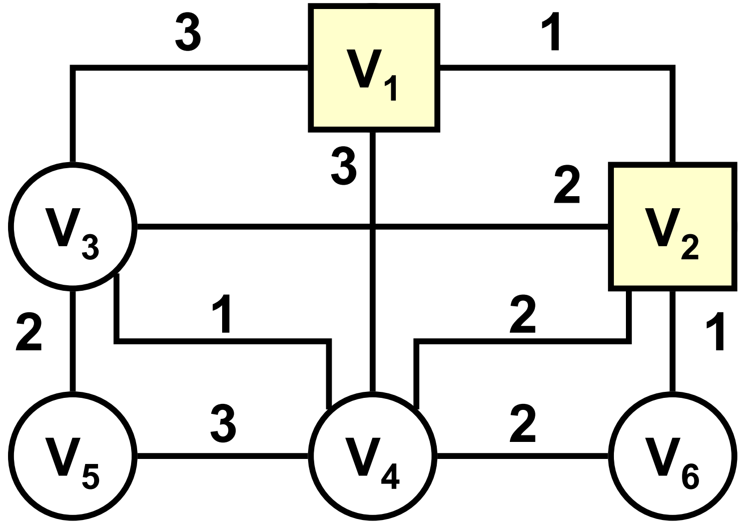
- Starting with V_1.



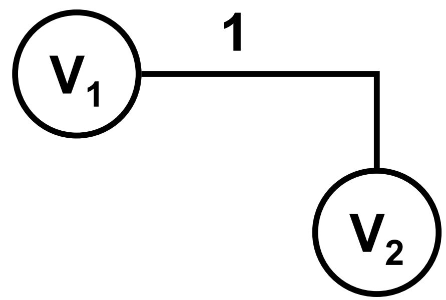
it	U	w
1	1	

Prim's Algorithm

Exercise 1

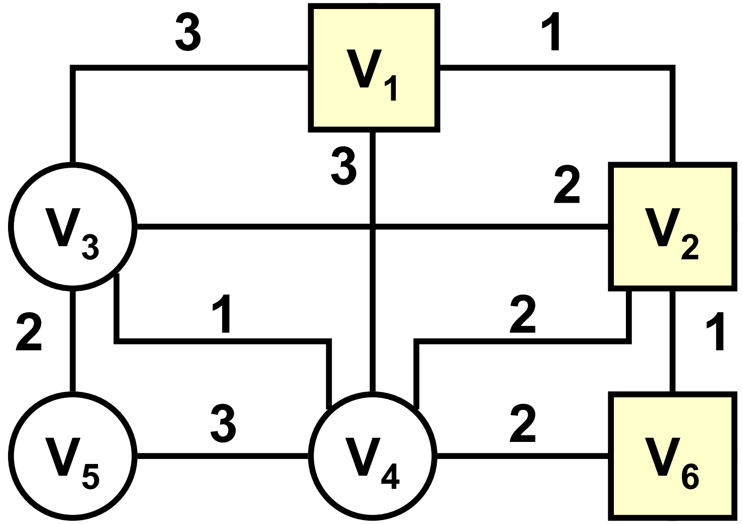


it	U	w
1	1	
2	1, 2	2

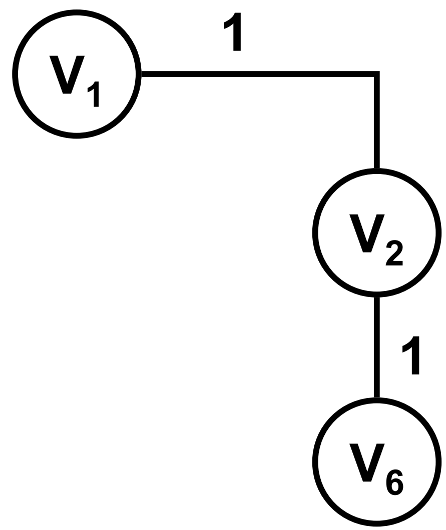


Prim's Algorithm

Exercise 1



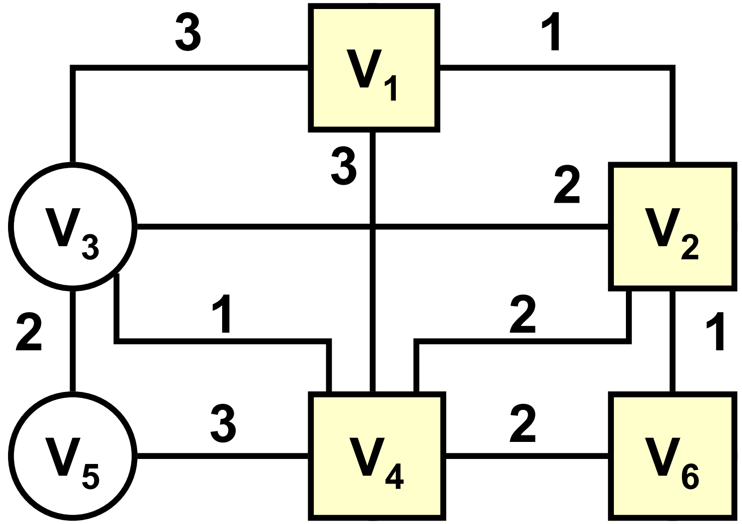
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6



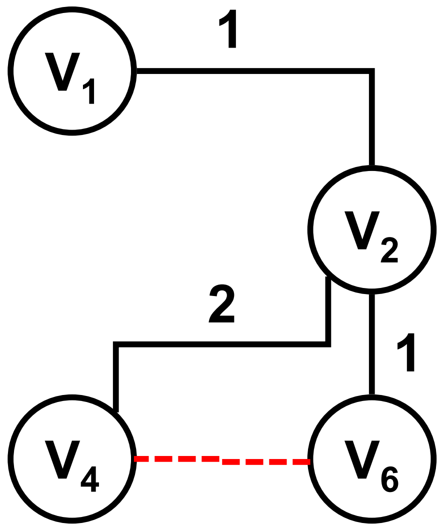
Prim's Algorithm

Exercise 1

- We could select V_3 too

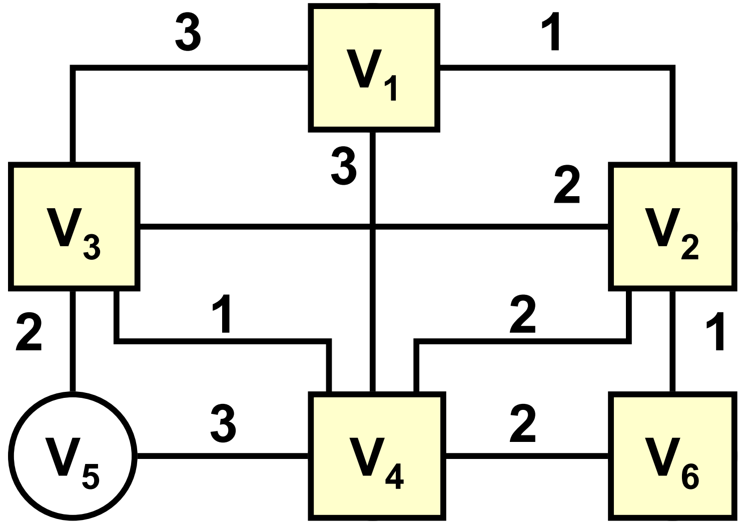


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4

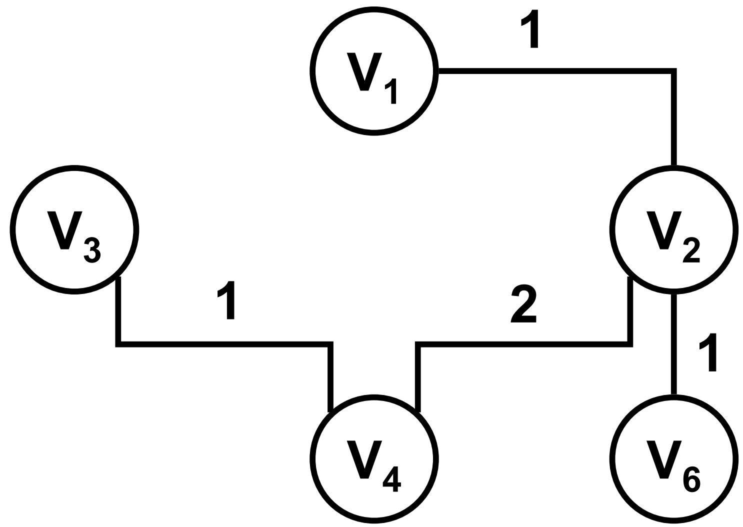


Prim's Algorithm

Exercise 1

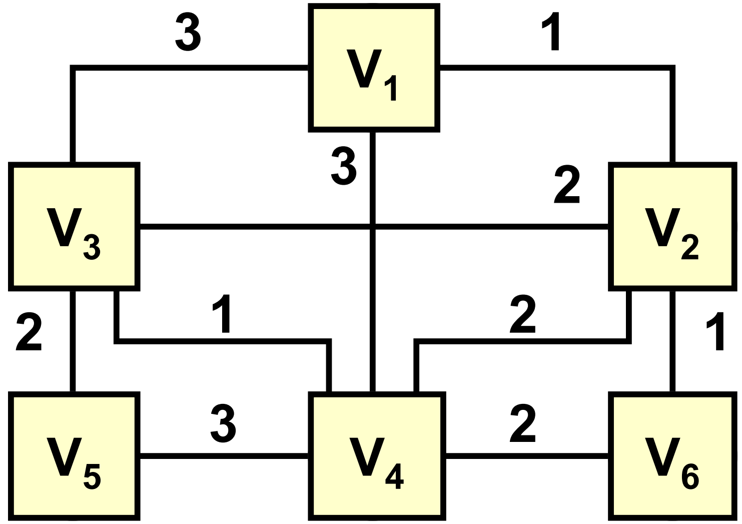


it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3

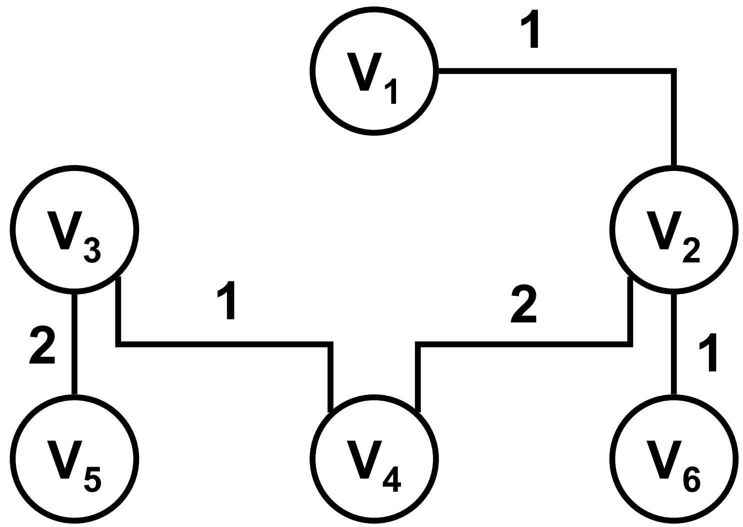


Prim's Algorithm

Exercise 1



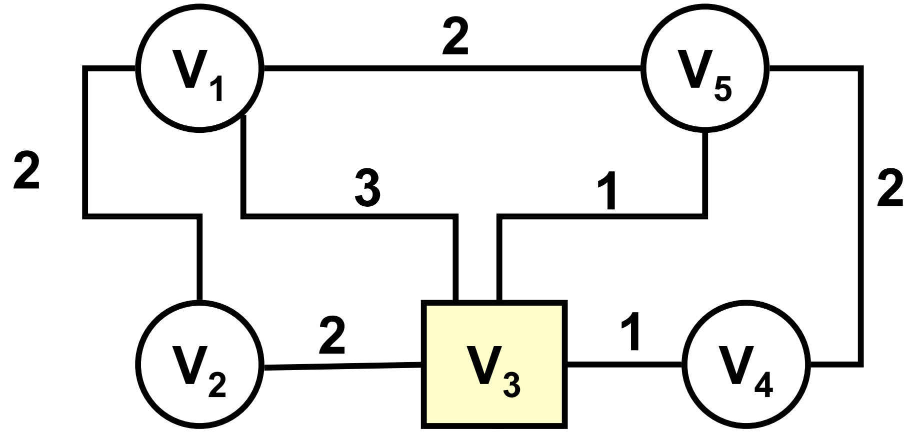
it	U	w
1	1	
2	1, 2	2
3	1, 2, 6	6
4	1, 2, 4, 6	4
5	1, 2, 3, 4, 6	3
6	1, 2, 3, 4, 5, 6	5



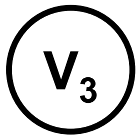
Prim's Algorithm

Exercise 2

- Starting with V_3.



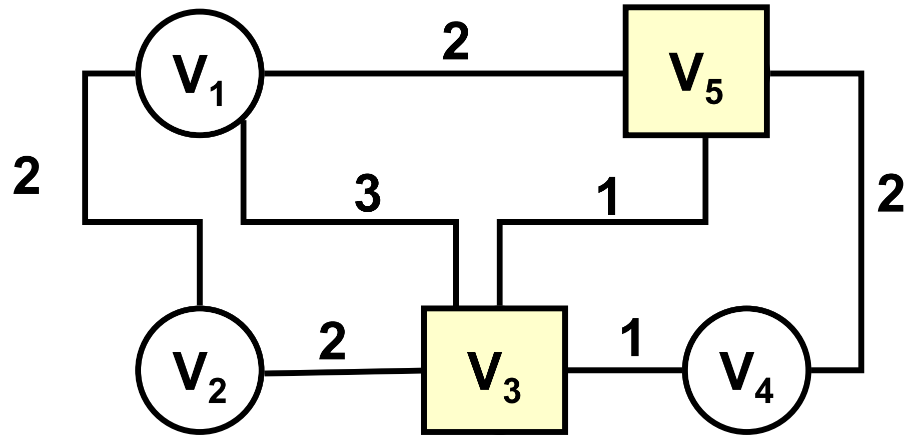
it	U	w
1	3	
2		
3		
4		
5		



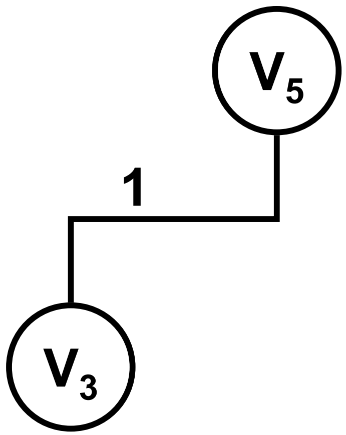
Prim's Algorithm

Exercise 2

- We could select V_4 too

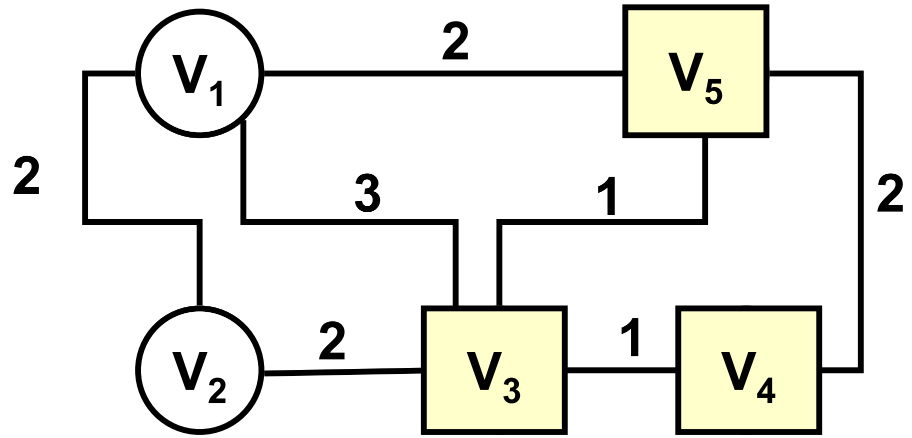


it	U	w
1	3	
2	3, 5	5
3		
4		
5		

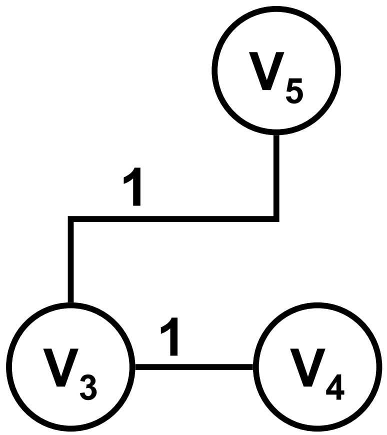


Prim's Algorithm

Exercise 2



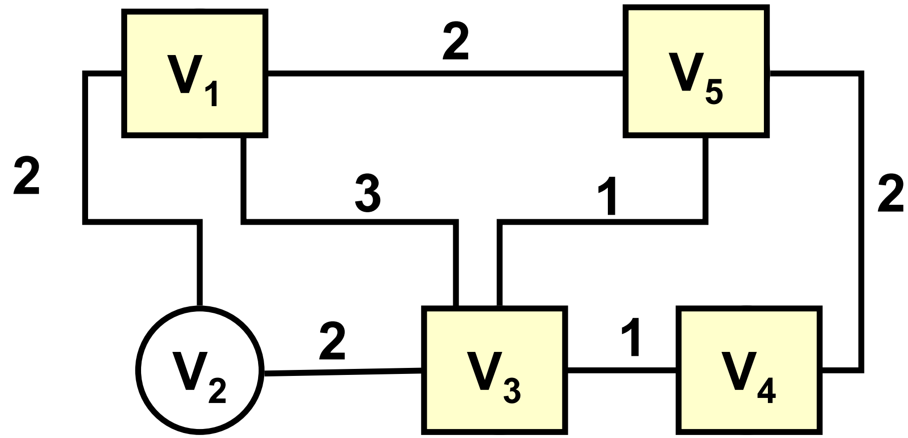
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4		
5		



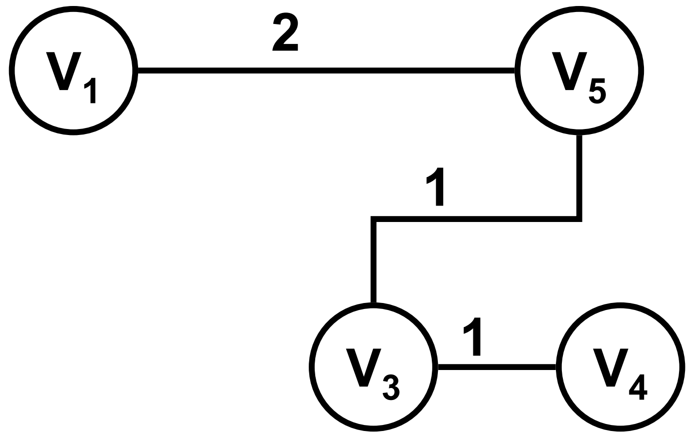
Prim's Algorithm

Exercise 2

- We could select V_2 too



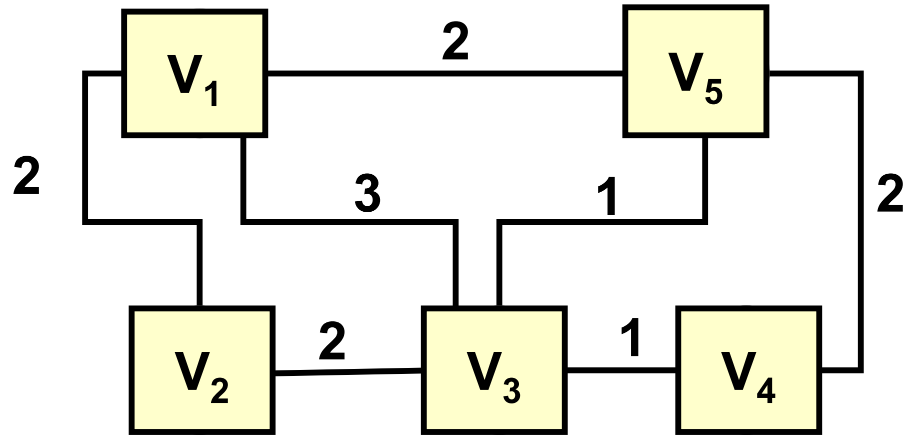
it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5		



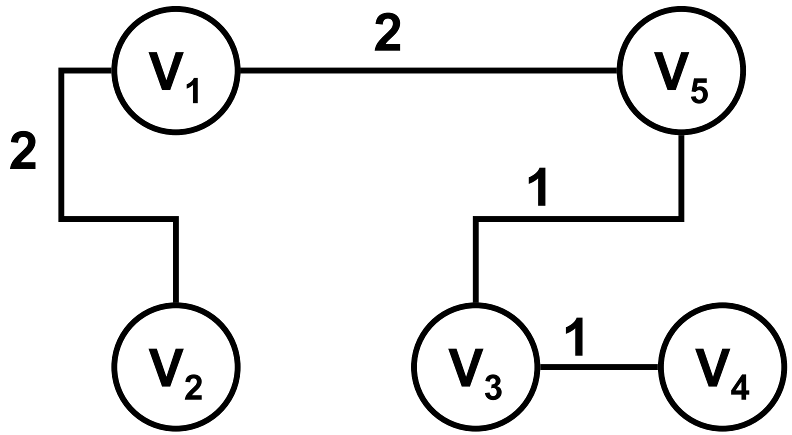
Prim's Algorithm

Exercise 2

- Alternative option: $\{V_2, V_3\}$



it	U	w
1	3	
2	3, 5	5
3	3, 4, 5	4
4	1, 3, 4, 5	1
5	1, 2, 3, 4, 5	2



Prim's Algorithm

Conclusions

- The resulting tree depends upon…
  - Starting node.
  - Selection of the edge of minimum cost in each iteration.
    - There can be more than one edge of minimum cost.

For each iteration (while U != V)	**n**

`1. Evaluate all the edges {u, v} where u is part of U and`
`   v is part of V - U selecting the one of the lowest cost`
`2. T = T + {u, v}`
`3. U = U + {v}`
	n^2

$O(n^3)$

DISCUSSION: What is the temporal complexity of this algorithm?

Prim's Algorithm

- Optimization
  - Using auxiliary sorted vectors to select the edge of minimum cost, reducing the complexity to O(n).
    - More speed obtained thanks to an increase in the use of memory.

For each iteration (while U != V)	**n**	

`1. Evaluate all the edges {u, v} where u is part of U and v is part of V - U selecting the one of the lowest cost`
`2. T = T + {u, v}`
`3. U = U + {v}`
		n

$O(n^2)$

C59 Series

Hierarchical Structures

Dr. Martin Gonzalez-Rodriguez

Hierarchical Structures

Goal

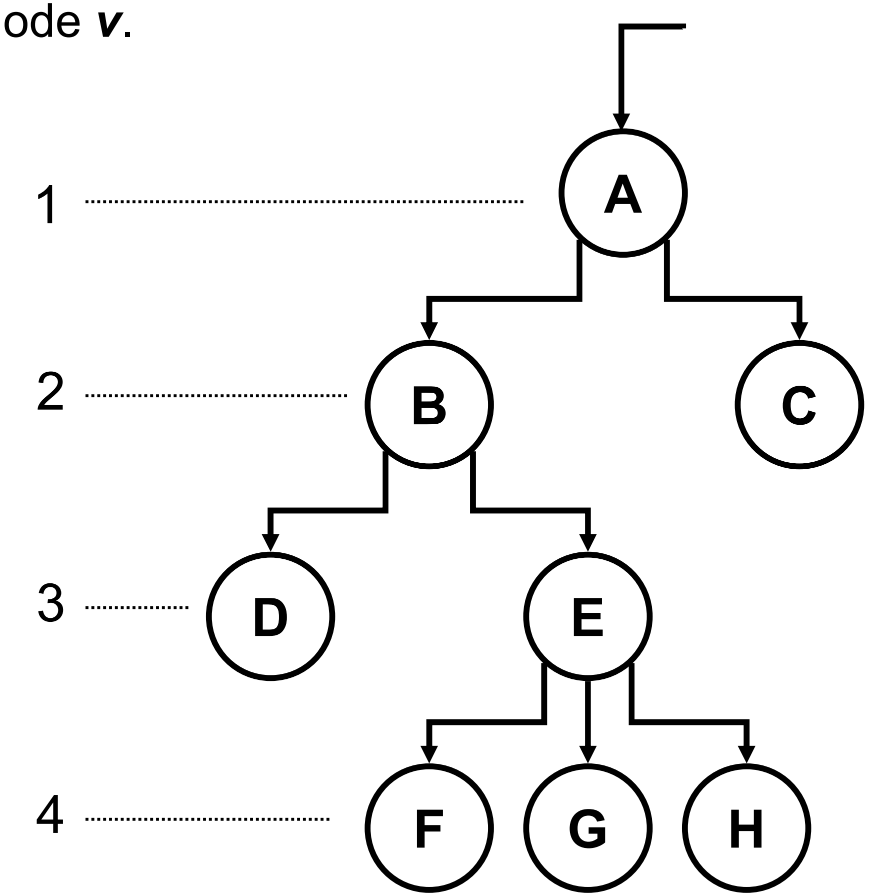
- Modeling order relationships between elements.
  - Social hierarchies (the army, the structure of a company, etc.).
  - Grammar modeling (lexical trees, syntactical trees, etc.)
  - Computer Science models (class hierarchy, file systems, etc.).
  - Classification systems (taxonomic ranks, phylogenetic trees, genealogical, sports, etc.).

Fundamentals

What is a Tree?

- In Computer Science[1] a tree is a connected graph without cycles including a root node.
  - Given a node called **root** and any other node ***v***, there only exists one directed path from the root to that node ***v***.

Basic Elements
1. Root. 2. Children(direct descendant). 3. Father (direct ascendant). 4. Leaf (terminal node). 5. Inner node. 6. Node's degree. 7. Tree's degree. 8. Node's level. 9. Height(depth).



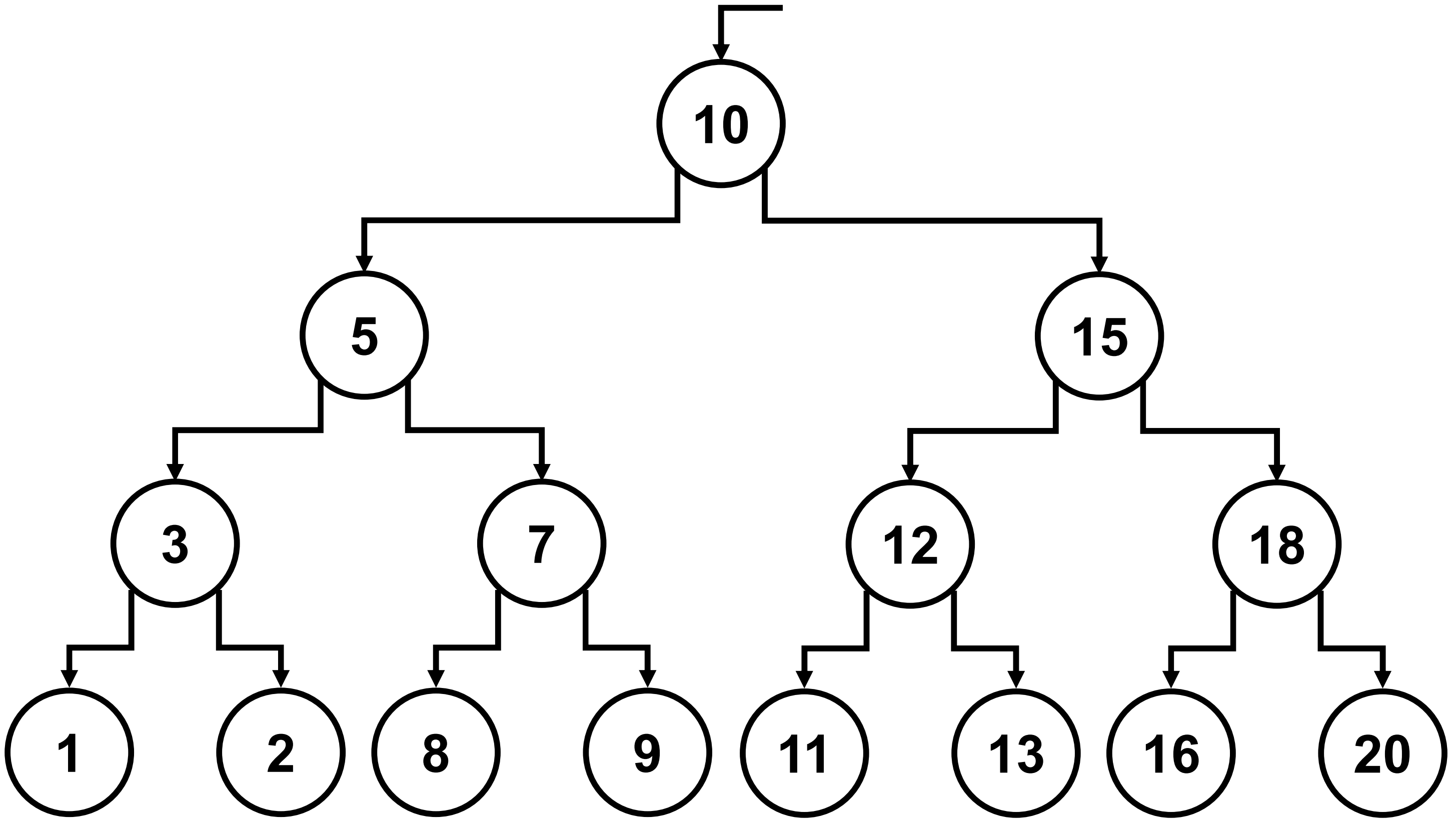
[1]*In mathematics this concept is referred to as 'arborescence'.*

REMEMBER: A tree is a special type of graph

Fundamentals

Complete Tree

- Tree containing the maximum number of nodes for its height **h** and **degree g**.
  - Is a tree with all of its levels full of nodes.
  - Maximum performance when **searching from the root**.



$n = 2^h - 1$

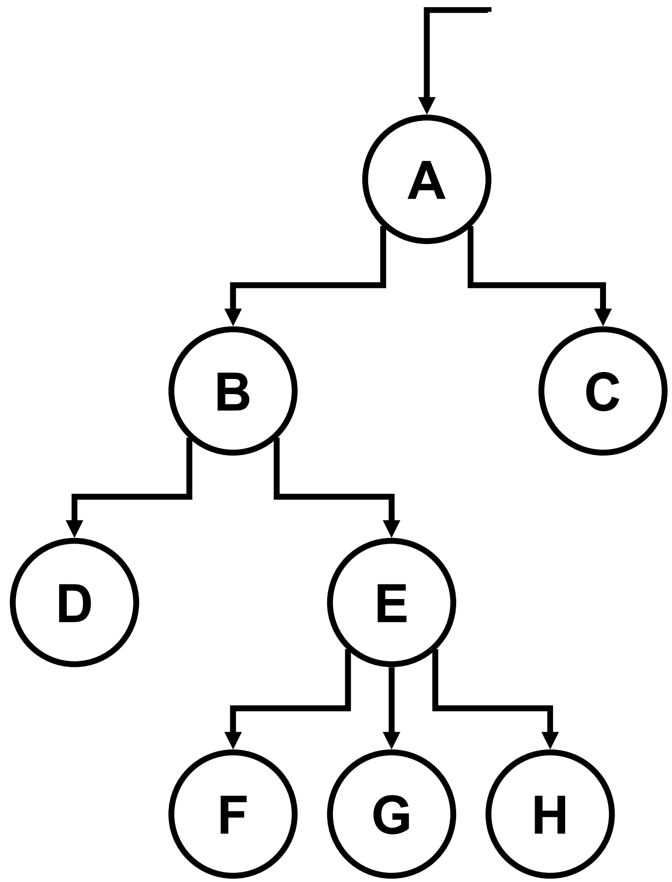
$\log_2(n + 1) = h$

DISCUSSION: What is the maximum longitude for a search path (given n)?

Metrics

Search Paths (Average Length)

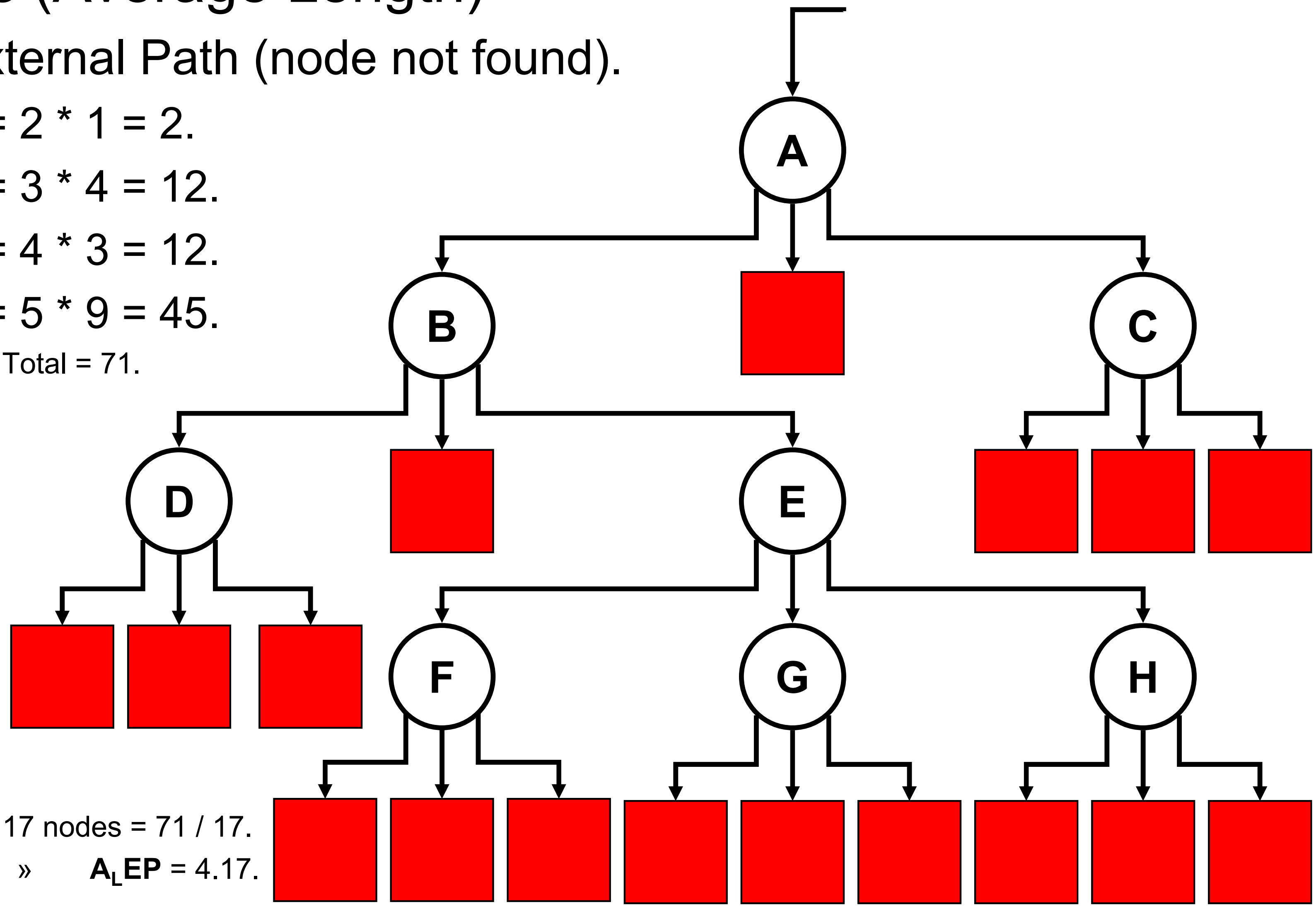
- IP: Internal Path (node found)
  - Searching A = 1.
  - Searching B and C = 2 p/u = 4.
  - Searching D and E = 3 p/u = 6.
  - Searching F, G and H = 4 p/u = 12.
    - Total = 23.
      - » For 8 nodes = 23 / 8 = **A_LIP = 2.87**



Metrics

Search Paths (Average Length)

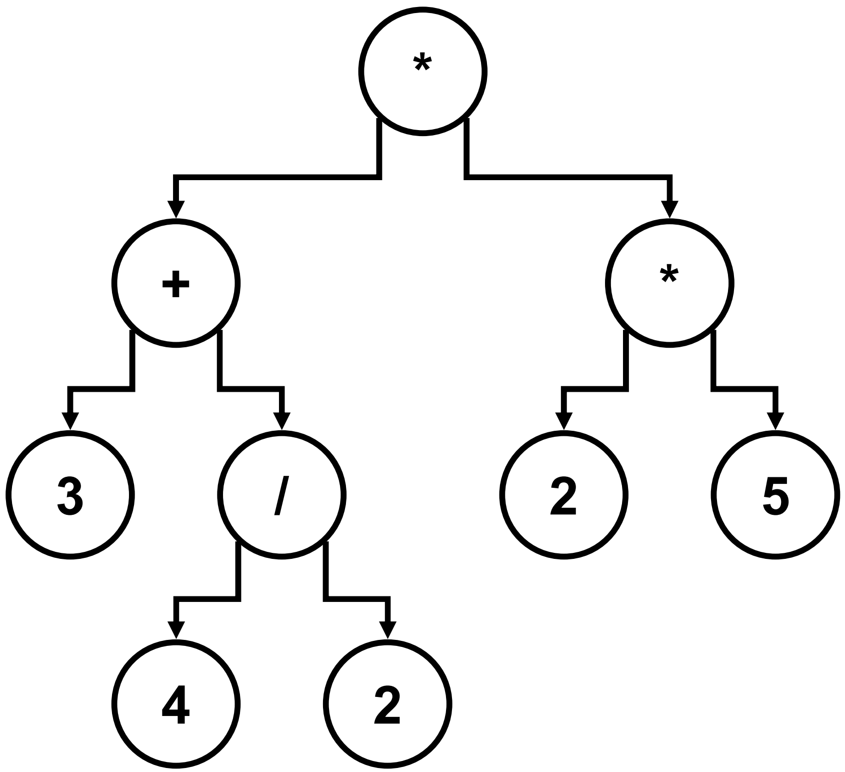
- EP: External Path (node not found).
  - $L_2 = 2 * 1 = 2$.
  - $L_3 = 3 * 4 = 12$.
  - $L_4 = 4 * 3 = 12$.
  - $L_5 = 5 * 9 = 45$.
    - Total = 71.
    - 17 nodes = 71 / 17.
      - **A_LEP** = 4.17.



Binary Tree

Degree 2 Tree

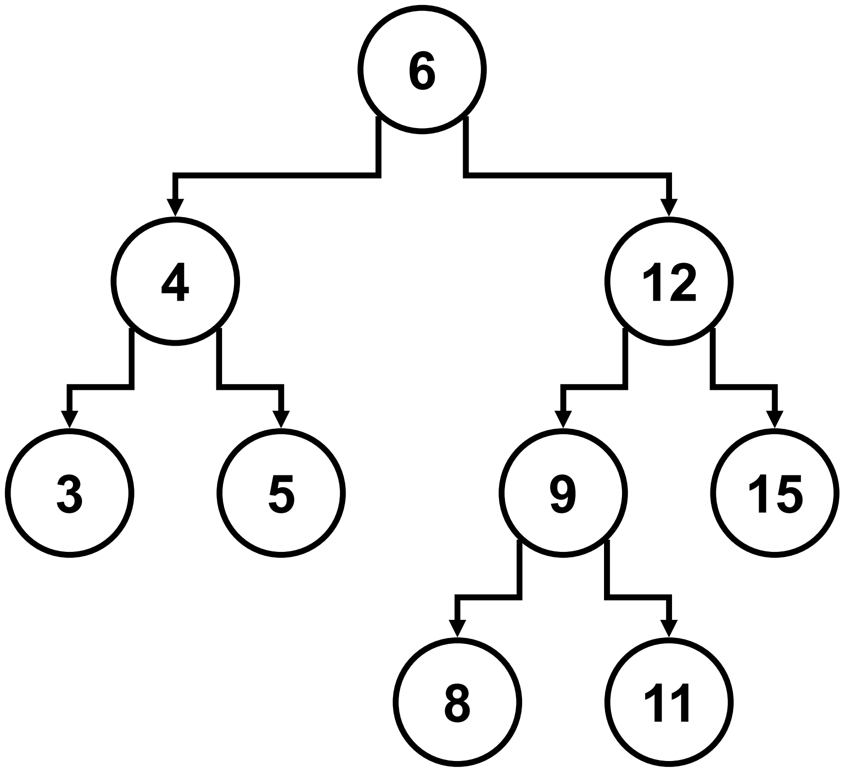
- Models hierarchical relationships between pairs of elements related to a node located in an upper level.
  - Genealogical Trees.
  - Cup competitions.
  - Binary operators.



Binary Search Tree (BST)

Binary Tree designed to make search efficient operations

- The following applies **to each node**…
  - **Left sub tree**: contains elements whose keys **are smaller** than the parent node's key.
  - **Right sub tree**: contains elements whose keys **are greater** than the parent node's key.



Binary Search Tree (BST)

Structure and essential methods

Class BSTNode

```
public class BSTNode <T extends Comparable <T>>
{
 private T element;
 private BSTNode<T> left;
 private BSTNode<T> right;
}
```

- **Essential Methods**
  - Add.
  - Search.
  - Remove.
  - toString.

Binary Search Tree (BST)

Insert

❖ Recursive Procedure

- General Case 1:
  - If the key of the node to be inserted is **smaller** than the current node's key, **insert the node to the left**.
- General Case 2:
  - If the key of the node to be inserted is **greater** than the current node's key, **insert the node to the right**.
- Stop condition 1:
  - If the key of the node to be inserted is **the same** as the current node’s key, **the node exists**! Error: **repeated keys are not allowed**.
- Stop condition 2:
  - If the current **node equals null** a leaf has been reached. Create a new node and insert it there.

Binary Search Tree (BST)

add

```
private BSTNode<T> add (BSTNode<T> theRoot, T element){
 if (theRoot == null)
  return new BSTNode<T>(element);

 if (element.compareTo(theRoot.getElement()) == 0)
  throw new RuntimeException("element already exists!");

 if (element.compareTo(theRoot.getElement()) < 0)
  theRoot.setLeft (add(theRoot.getLeft(), element));

 if (element.compareTo(theRoot.getElement()) > 0)
  theRoot.setRight (add(theRoot.getRight(), element));
}
```

CLASSWORK

PLAYGROUND

- **Exercise BST 1.** start with an empty Binary Search Tree...
  a) Add the following sequence of elements: 5, 7, 9, 3, 1, 2, 6.
     – Analyze the temporal complexity of each insertion.

- **Exercise BST 2**. start with an empty Binary Search Tree…
  a) Add the following sequence of elements: 7, 6, 5, 4, 3, 2, 1.
     – Analyze the temporal complexity of each insertion.
  b) Add node 8.
     – Analyze the temporal complexity of adding this element.

Best case complexity: O(1)

Worst case complexity: O(n)

DISCUSSION: What is the temporal complexity of this algorithm?

Binary Search Tree (BST)

Search

```
private boolean search (BSTNode<T> theRoot, T element)
{
 if (theRoot == null)
  return false;
 else
  if (element.compareTo(theRoot.getElement()) == 0)
   return true;
  else
   if (element.compareTo(theRoot.getElement()) < 0)
    return search(theRoot.getLeft(), element);
   else
    if (element.compareTo(theRoot.getElement()) > 0)
     return search (theRoot.getRight(), element);
}
```

Best case complexity: O(1)

Worst case complexity: O(n)

DISCUSSION: What is the temporal complexity of this algorithm?

Binary Search Tree (BST)

Remove

```
private BSTNode<T> remove (BSTNode<T> theRoot, T element)
{
  if (theRoot == null)
   throw new RuntimeException("element does not exist!");
  else
   if (element.compareTo(theRoot.getElement()) < 0)
    theRoot.setLeft(remove (theRoot.getLeft(), element));
   else
    if (element.compareTo(theRoot.getElement()) > 0)
     theRoot.setRight(remove (theRoot.getRight(), element));
    else {
     // node found
     // How to delete it?
        }
   return theRoot;
}
```

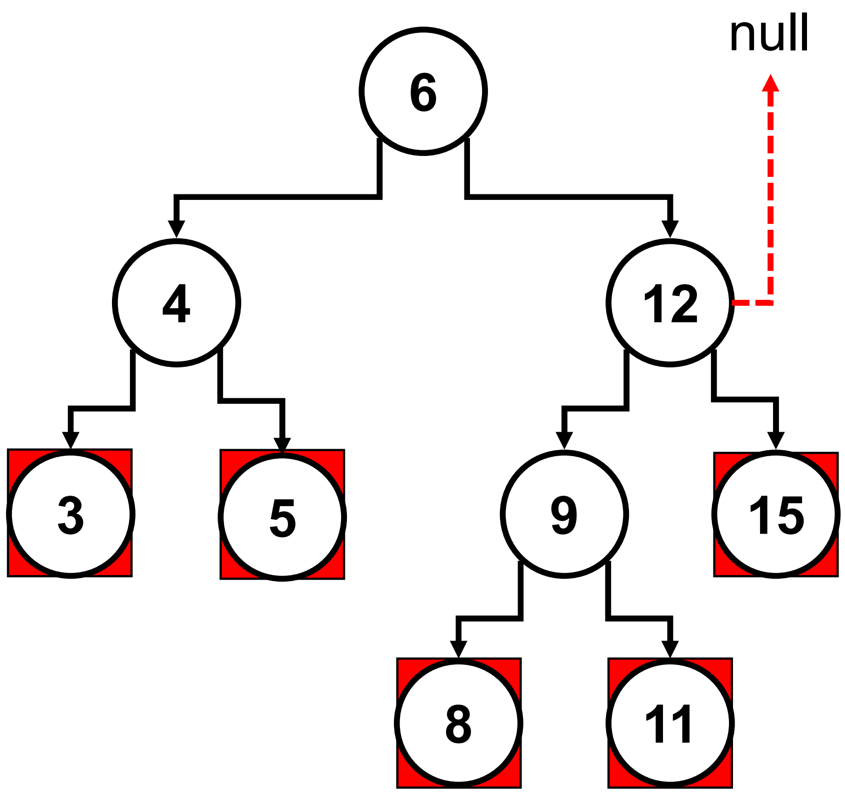
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery I: Deleting an element without children (leaves).
  - A *null* value is assigned to the reference.

ZOOM IN

```
else {
  if (theRoot.getLeft() == null &&
   theRoot.getRightDer() == null)
    return(null);



}
```



TASK: delete element 15

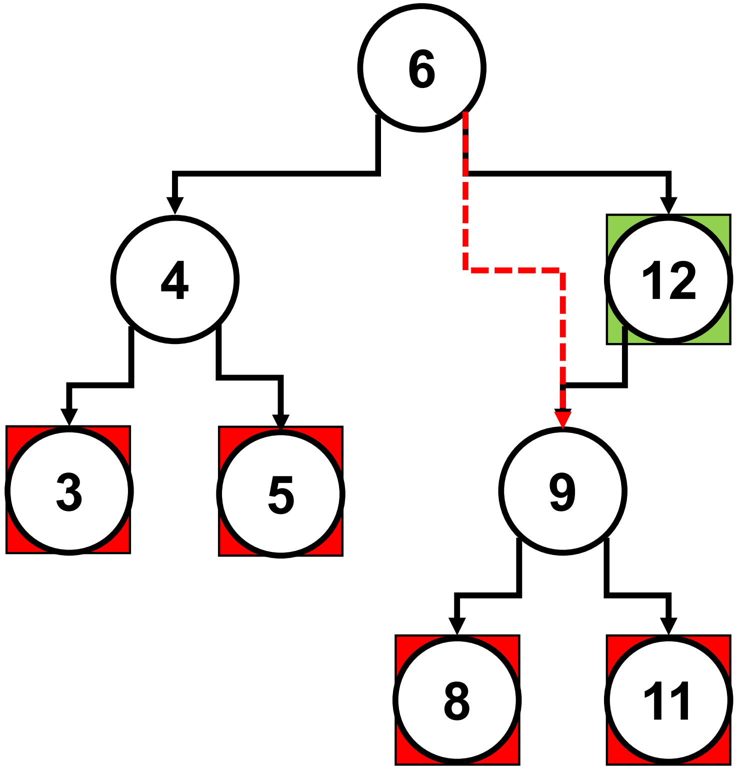
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery II: Deleting an element with only one child.
  - The reference is reassigned to this only child.

ZOOM IN

```
else {
 if (theRoot.getLeft() == null)
    return theRoot.getRight();
  else
   if (theRoot.getRight() ==
 null) return theRoot.getLeft();

}
```



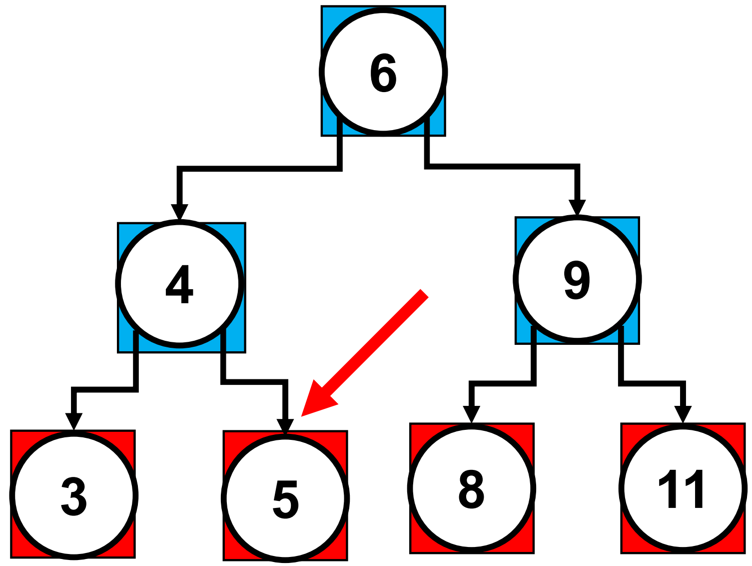
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery III: Deleting an element with two children.
  - Substitute the content of this node by the greatest node (pivot) in its left sub tree.
    - Proceed to delete the pivot (we might face scenery I or II but never scenery III).

ZOOM IN

```
else {
 if (theRoot.getLeft() == null)
  return theRoot.getRight();
 else
  if (theRoot.getRight() == null)
   return theRoot.getLeft();
  else {
theRoot.setElement(getMax(theRoot.get
Left()));


  }
}
```



TASK: Delete element 6

Binary Search Tree (BST)

getMax

```
public T getMax(BSTNode<T> theRoot)
{
 if (theRoot == null)
  return null;
 else
  return getMaxRec(theRoot);
}


private T getMaxRec(BSTNode<T> theRoot)
{
  if (theRoot.getRight () == null)
   return theRoot.getElement();
  else
   return getMaxRec(theRoot.getRight());
}
```

EXERCISE: Create a recursive method *getMax*

Binary Search Tree (BST)

getMax

```
private T getMax(BSTNode<T> theRoot)
{
 while (theRoot.getRight() != null)
  theRoot = theRoot.getRight();

 return theRoot.getElement();
}
```

EXERCISE: Create an iterative method *getMax*

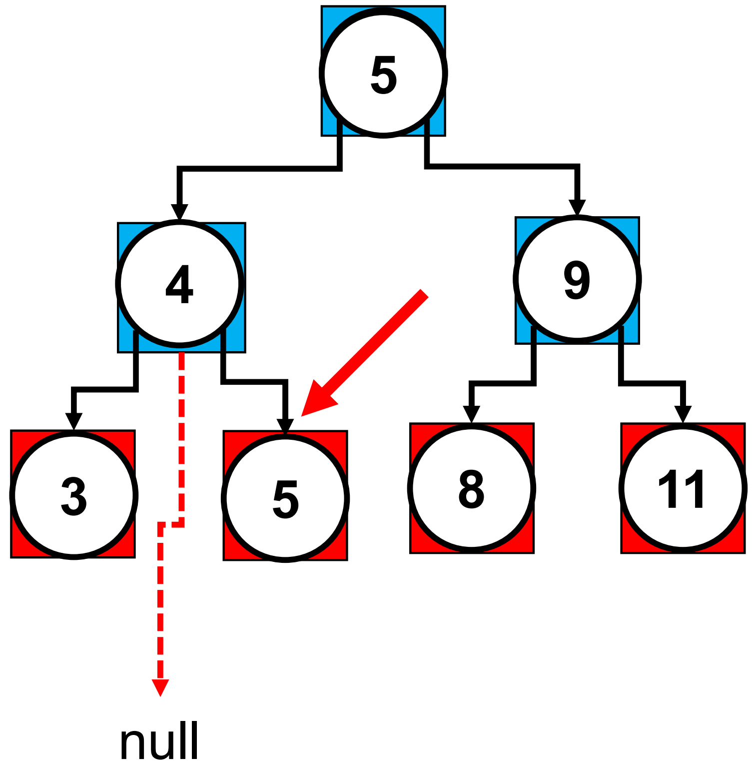
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery III: Deleting an element with two children.
  - Substitute the content of this node by the greatest node (pivot) in its left sub tree.
    - Proceed to delete the pivot (we will face scenery I or II but never scenery III).

ZOOM IN

```
else {
 if (theRoot.getLeft() == null)
  return theRoot.getRight();
 else
  if (theRoot.getRight() == null)
   return theRoot.getLeft();
  else {
theRoot.setElement(getMax(theRoot.getLeft(
)));
 theRoot.setLeft(remove (theRoot.getLeft(),
 theRoot.getElement()));
  }
}
```



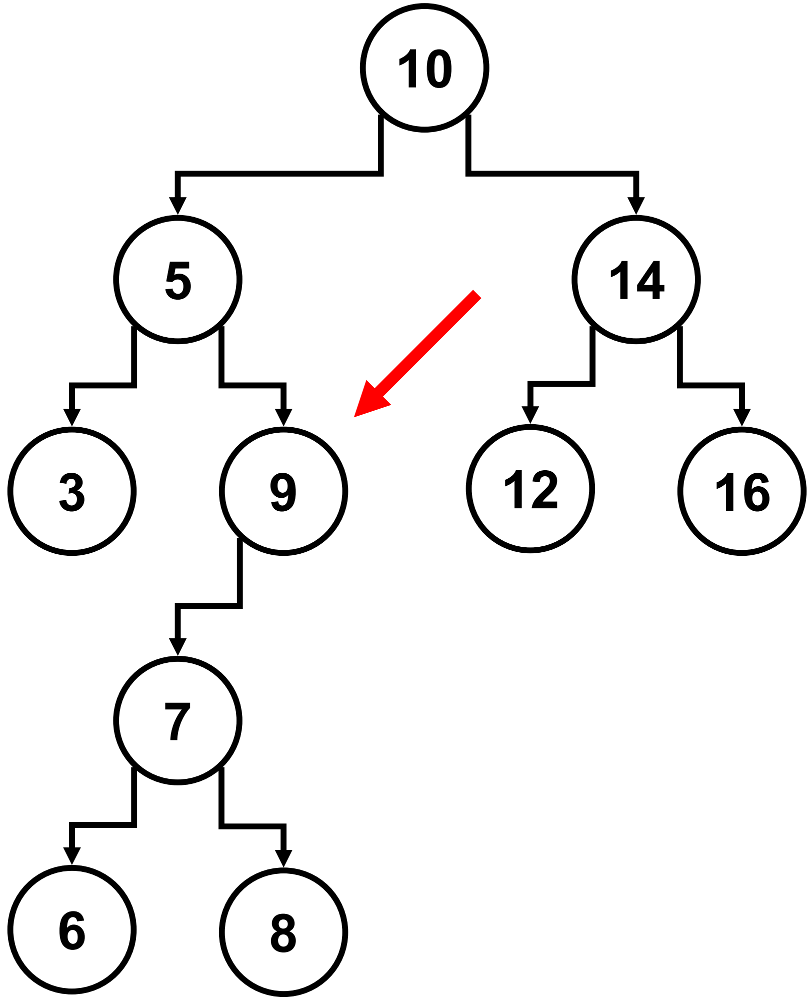
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery III: Deleting an element with two children.

ZOOM IN

```
else {
 if (theRoot.getLeft() == null) return
theRoot.getRight();
 else
  if (theRoot.getRight() == null)
   return theRoot.getLeft();
  else {
theRoot.setElement(getMax(theRoot.getLeft
()));
 theRoot.setLeft(remove
(theRoot.getLeft(),theRoot.getElement()))
;}}
```



EVENT: Deleting element 10

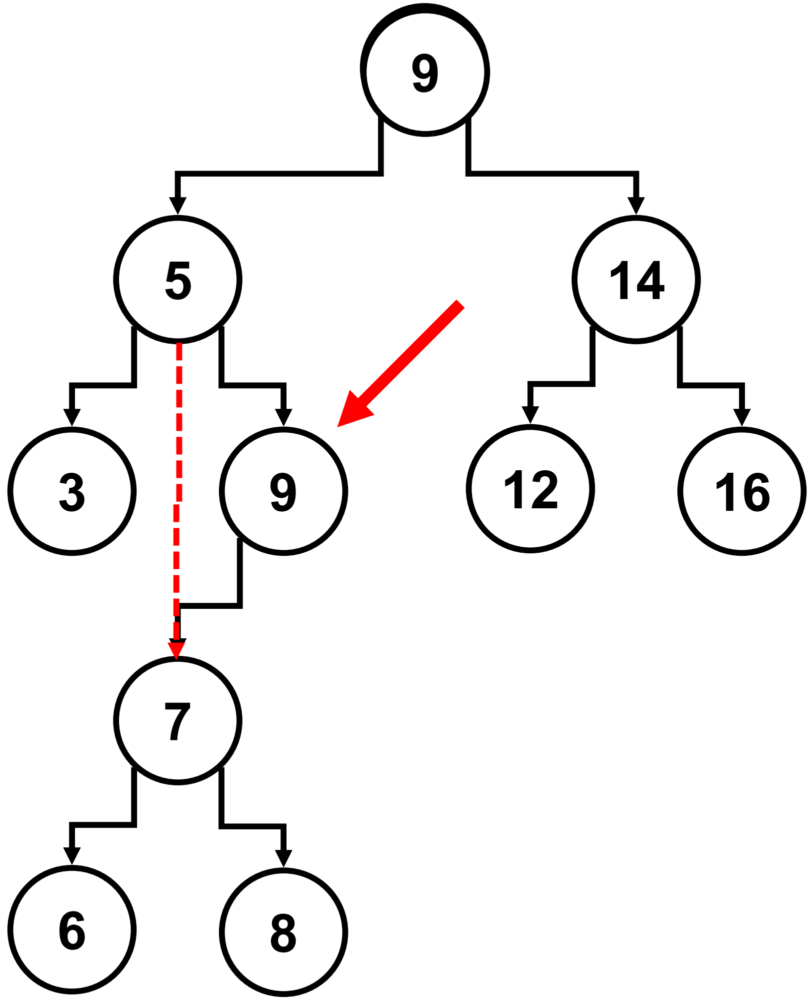
Binary Search Tree (BST)

Special sceneries of deletion

- Scenery III: Deleting an element with two children.

ZOOM IN

```
else {
 if (theRoot.getLeft() == null) return
theRoot.getRight();
 else
  if (theRoot.getRight() == null)
   return theRoot.getLeft();
  else {
theRoot.setElement(getMax(theRoot.getLeft
()));
 theRoot.setLeft(remove
(theRoot.getLeft(),theRoot.getElement()))
;}}
```



Binary Search Tree (BST)

Remove

```
private BSTNode<T> remove (BSTNode<T> theRoot, T element){
  if (theRoot == null)
   throw new RuntimeException("element does not exist!");
  else
   if (element.compareTo(theRoot.getElement()) < 0)
    theRoot.setLeft(remove (theRoot.getLeft(), element));
   else
    if (element.compareTo(theRoot.getElement()) > 0)
     theRoot.setRight(remove (theRoot.getRight(), element));
    else {
     if (theRoot.getLeft() == null) return theRoot.getRight();
      else
      if (theRoot.getRight() == null) return theRoot.getLeft();
      else {
        theRoot.setElement(getMax(theRoot.getLeft()));
 theRoot.setLeft(remove (theRoot.getLeft(), heRoot.getElement()));
}}
   return theRoot;     }
```

Best case complexity: O(1)

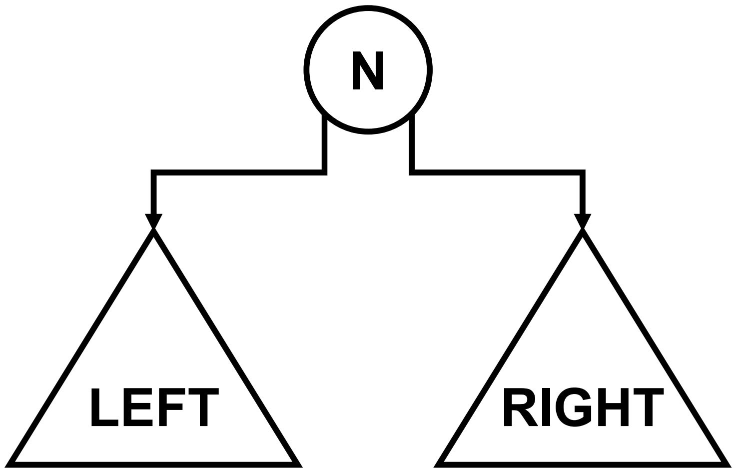
Worst case complexity: O(n)

DISCUSSION: What is the temporal complexity of this algorithm?

Binary Search Tree (BST)

Traversing a Binary Tree

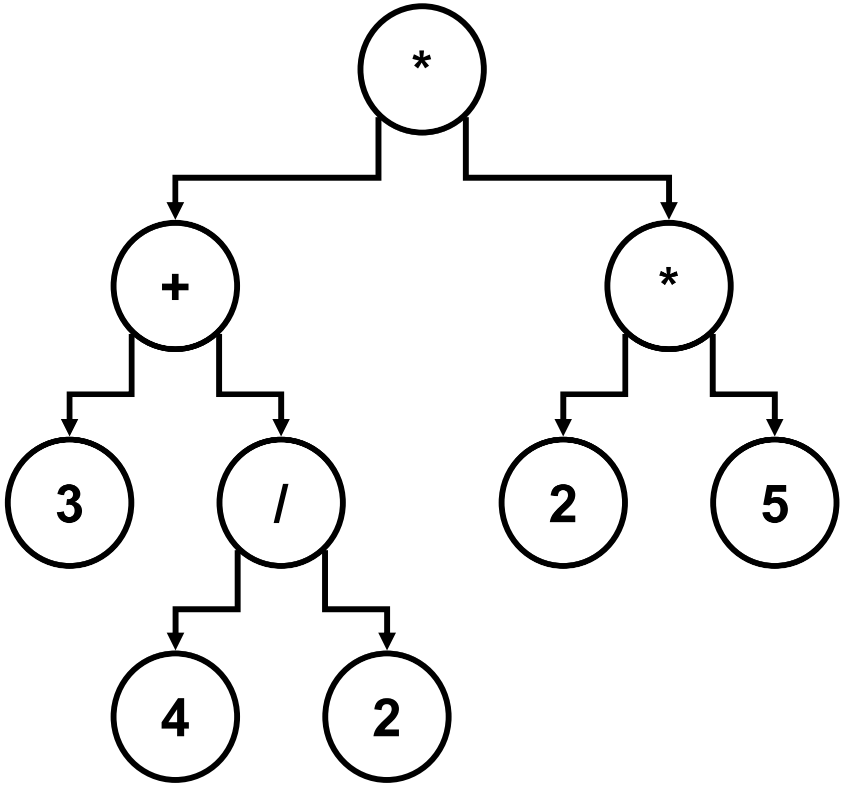
- **pre order**
  - The node **is analyzed first**, followed by the sub trees.
    - N-LEFT-RIGHT or N-RIGHT-LEFT.
- **in order**
  - The node is **analyzed between** the two sub trees.
    - LEFT-N-RIGHT or RIGHT-N-LEFT.
- **post order**
  - The node is analyzed **after both sub trees**.
    - LEFT-RIGHT-N or RIGHT-LEFT-N.



Binary Search Tree (BST)

Exercise

❖ Traverse the Tree

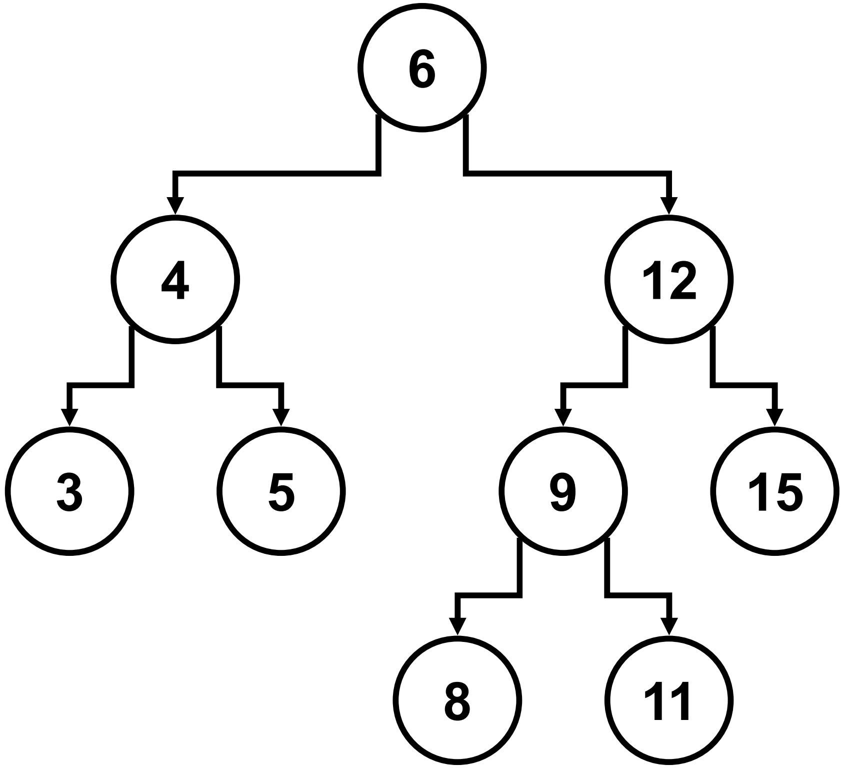


- **preorder**: * + 3 / 4 2 * 2 5 (prefix).
- **inorder**: 3 + 4 / 2 * 2 * 5 (infix).
- **postorder**: 3 4 2 / + 2 5 * * (reverse polish notation).

Binary Search Tree (BST)

PLAYGROUND

- Traverse the Tree



- **preorder**: 6, 4, 3, 5, 12, 9, 8, 11, 15. (prefix).
- **inorder**: 3, 4, 5, 6, 8, 9, 11, 12, 15. (infix).
- **postorder**: 3, 5, 4, 8, 11, 9, 15, 12, 6 (postfix).

Binary Search Tree (BST)

toString (Preorder traverse)

```
private String toString (BSTNode<T> theRoot)
{
 if (theRoot != null)
  return (theRoot.toString()
          + toString(theRoot.getLeft())
           + toString(theRoot.getRight()));
 else
  return ("-");
 }
```

Best case complexity: O(n) **Worst case complexity: O(n)**

DISCUSSION: What is the temporal complexity of this algorithm?

Binary Search Tree (BST)

Performance

- Performance in such kind of trees depends on their height
  - H range: [$\log_2 n$, n]

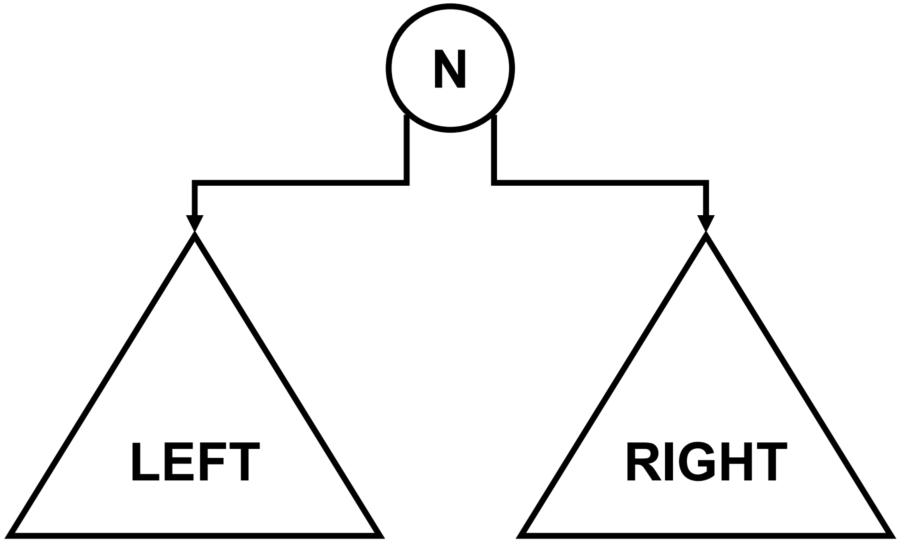
Method	Best case complexity	Worst case complexity
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)
Traverse	O(n)	O(n)

- Goal
  - Minimize the tree height, avoiding the creation of degenerated trees.

PBT

Perfectly Balanced Trees (PBT)

- Ensures the minimum height condition for a binary tree
  - **Condition**: For every node n, $|\#_{lzq} - \#_{der}| <= 1$.
    - $\#_{left}$ = number of nodes in the left sub tree.
    - $\#_{right}$ = numbers of nodes in the right sub tree.



- All PBTs are minimum height trees but…
  - All minimum height trees are PBTs?

BST vs PBT

Insertion and deletion have poor performance in PBTs

- These operations require **destroying and rebuilding** the tree again after their execution.

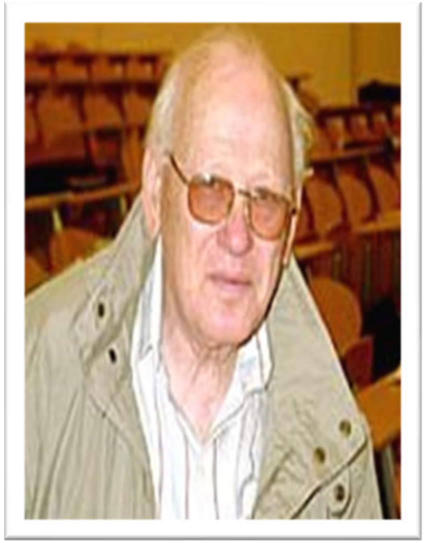
Method	**BST(worst case)**	**PBT (any case)**
Insert	O(n)	O(n)
Search	O(n)	$O(\log_2 n)$
Deletion	O(n)	O(n)

- PBTs make sense **only when** the number of searches is massively **higher than** the use of other operations.

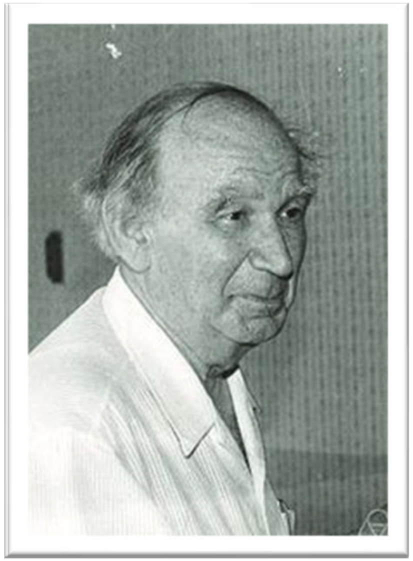
AVL Trees

Problem to solve

- Designing a tree providing a $\log_2(n)$ temporal complexity in the worst case for the three basic operations
  - Insert, Search, Deletion.



Georgii Adelson-Velskii (Wikipedia)



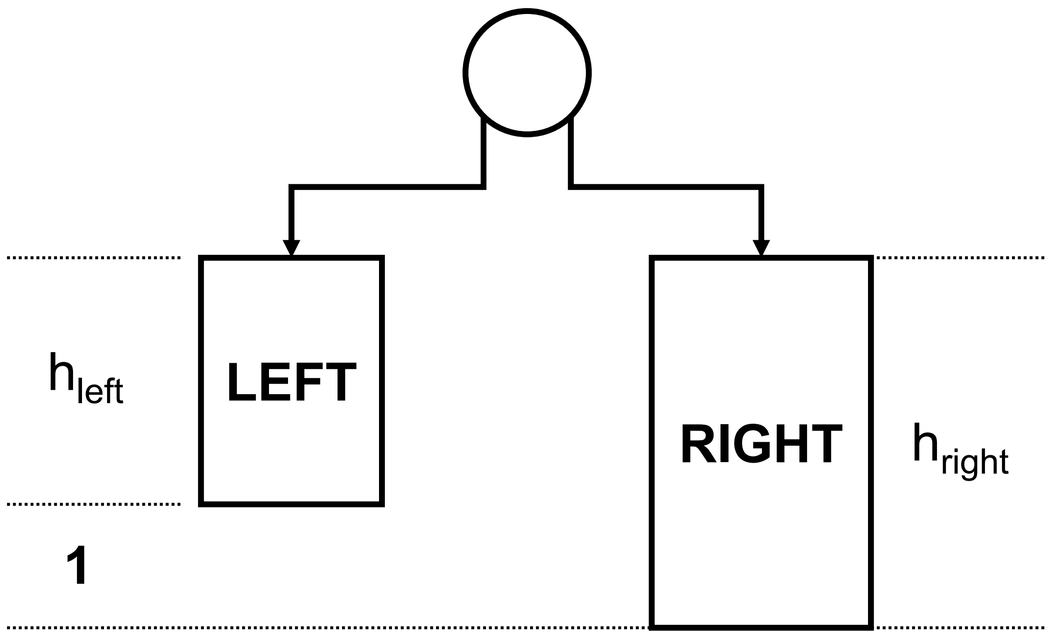
Yevgeni Landis (Wikipedia)

- Developed by the Soviet researchers Georgii Adelson-Velskii and Yevgeniy Landis 1962.

AVL Trees

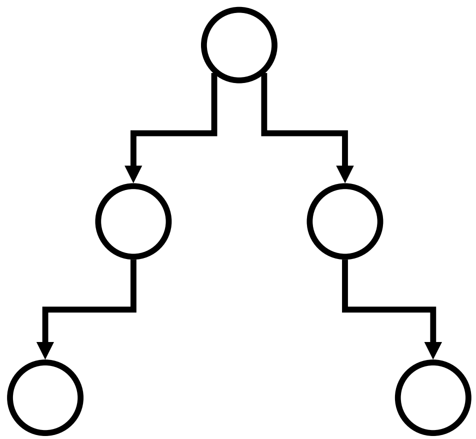
Adelson-Velskii and Landis Trees

- AKA Weakly Balanced Trees
  - **Condition**: every node n must verify: $|h_{left} - h_{right}| <= 1$.
    - h_{left} = height of the left sub tree.
    - h_{right}= height of the right sub tree.

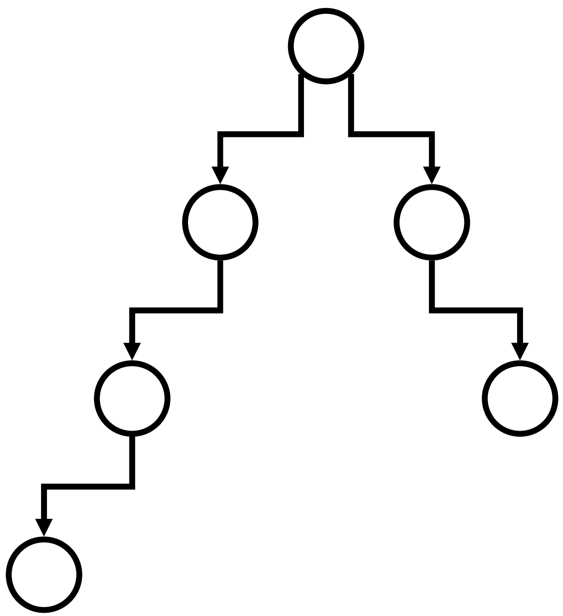


AVL Trees

Examples



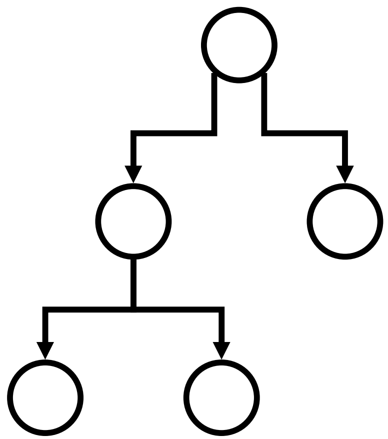
AVL? .... Yes
PBT? .... Yes
Minimum Height? ...... Yes



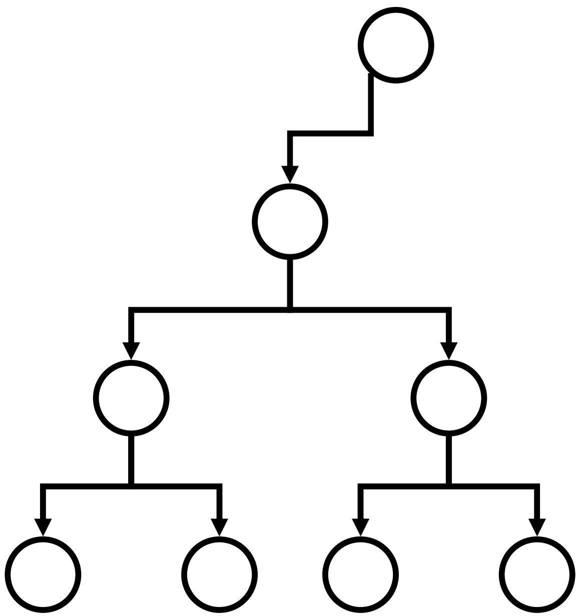
AVL? .... No
PBT? .... No
Minimum Height? ...... No

AVL Trees

Examples



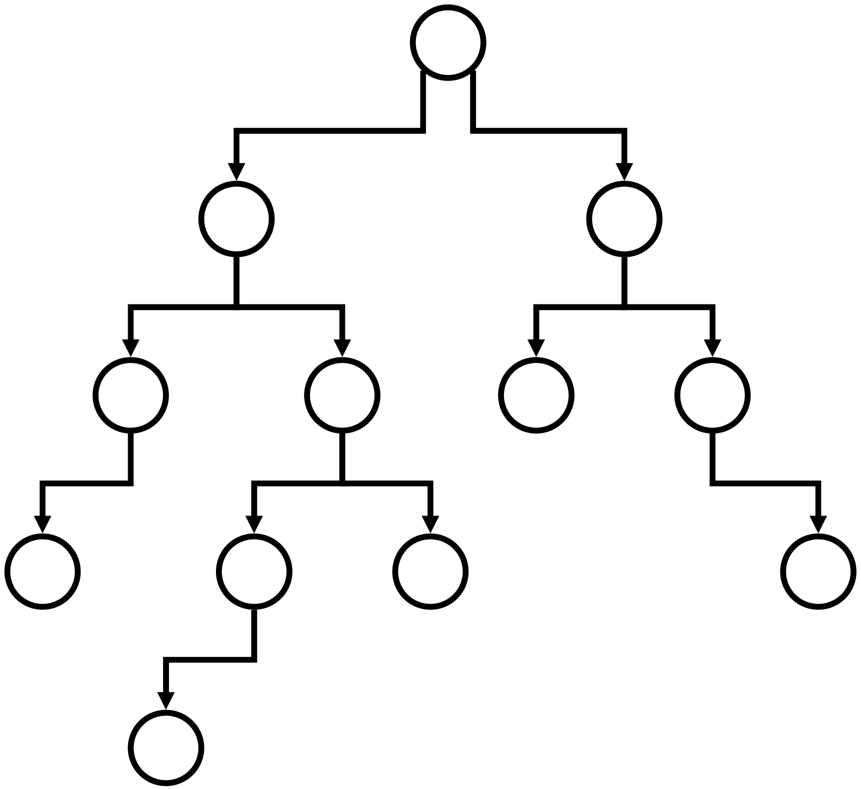
AVL? .... Yes
PBT? .... No
Minimum Height? ...... Yes



AVL? .... No
PBT? .... No
Minimum Height? ...... Yes

AVL Trees

Examples



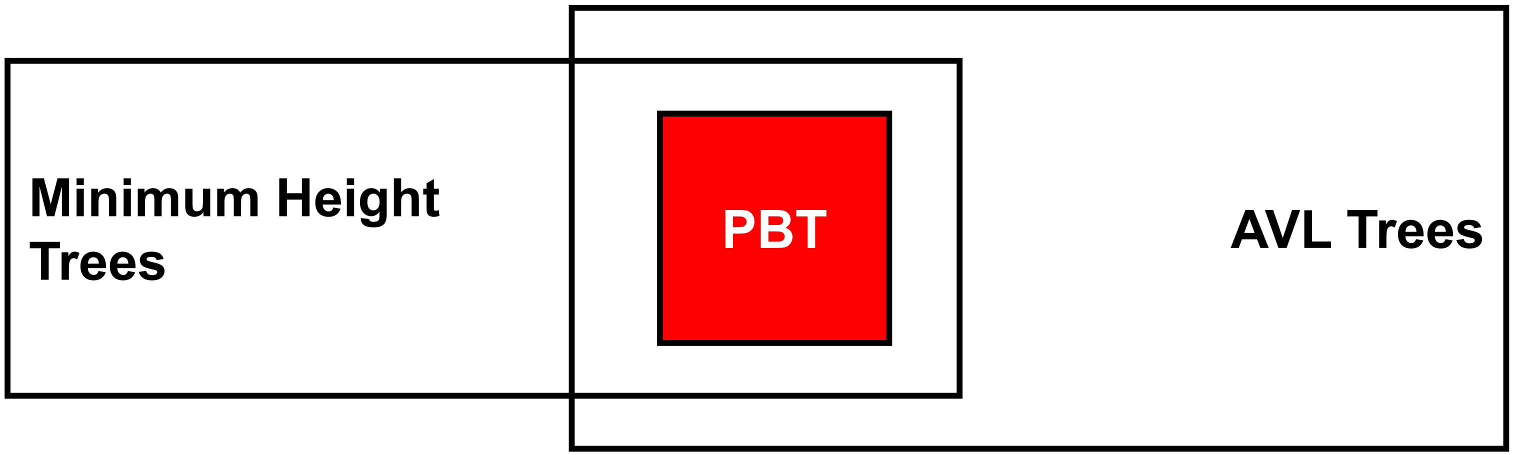
AVL? .... **Yes**
PBT? .... **No**
Minimum Height? ...... **No**

AVL Trees

Properties

- Every PBT is an AVL
  - Not every AVL is a PBT.
  - Not every AVL is a Minimum Height Tree.

- Not any Minimum Height Tree is an AVL
  - As seen in the examples.



AVL Trees

Ok, AVL are not Minimum Height Trees but…

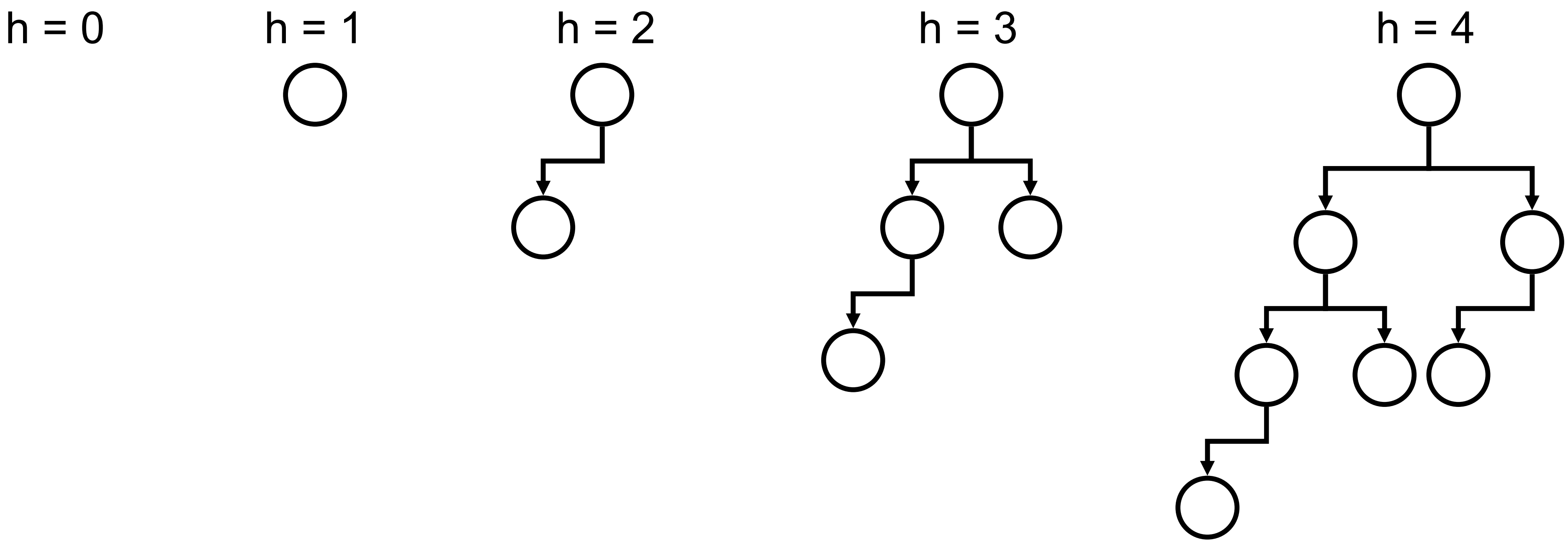
- What is their maximum height?
  - Is it **lower enough** to provide high performance in the basic operations?
  - How far is this maximum height from the minimum height ($\log_2 n$)?

- Adelson-Velskii and Landis built a series of AVL tree with the highest possible height for measuring the difference statistically
  - They used Fibonacci trees.
    - The AVL trees are built in the worst possible way to reach the maximum height.

AVL Trees

Fibonacci Trees

- The height (h) is determined in advance.
  - For **h = 0**, use an empty tree(T_0).
  - For **h = 1**, Use (T_1), or a single node tree.
  - For **h > 1**, Use $T_h = (T_{h-1}, x, T_{h-2})$.



AVL Trees

Adelson-Velskii and Landis demonstrated…

Limit for the maximum height in a Fibonacci tree
$h_{MaxFib}(n) <= 1.44Log_2n$

Height range in an AVL tree
$h_{PBT}(n) <= h_{AVL}(n) <= h_{MaxFib}(n)$ $Log_2n <= h_{AVL}(n) <= 1.44Log_2n$

- In the worst case, the height of an AVL exceeds the height of an PBT in a 44%

Worst case temporal complexity in the three basic operations
$O(Log_2n) <= O(h_{AVL}(n)) <= O(1.44Log_2n)$ $O(1.44Log_2n)$ $\mathbf{O(Log_2n)}$

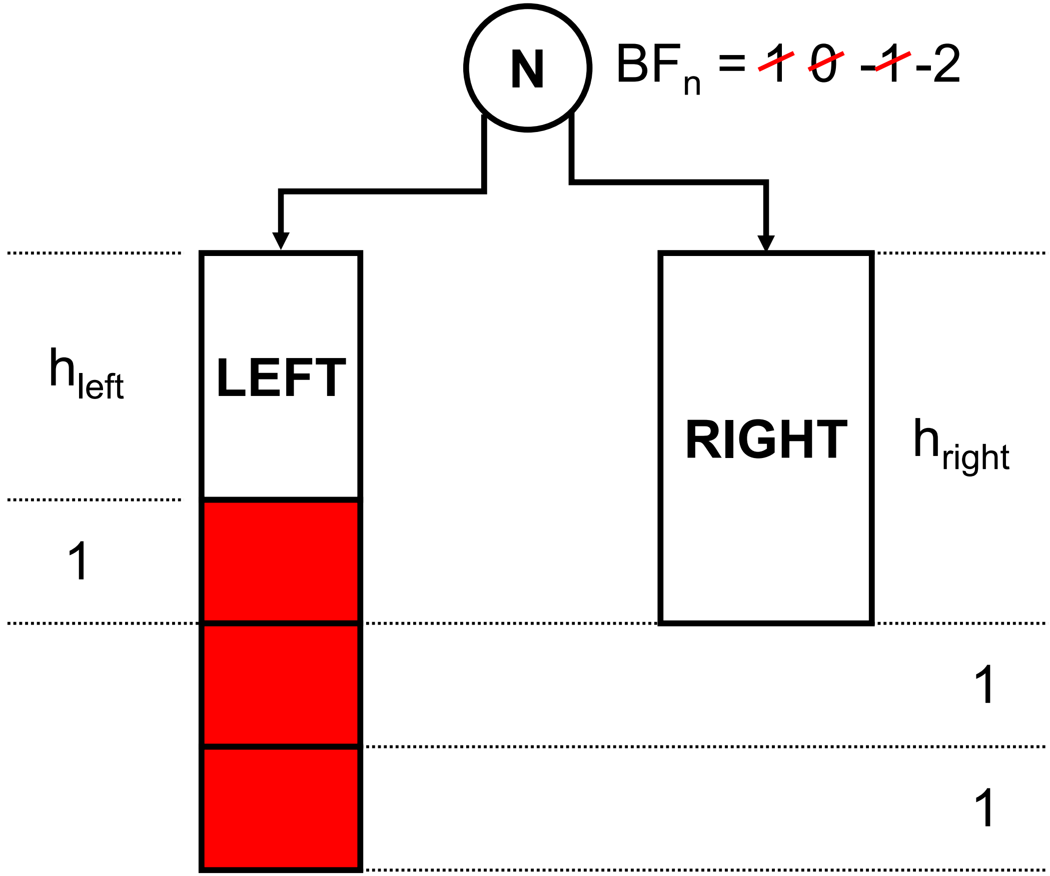
AVL Trees

Balance Factor (BF)

- $BF_n = h_{right} - h_{left}$

- Possible Scenarios:
  - $h_{left} > h_{right}$ ($BF_n = -1$).
  - $h_{left} = h_{right}$ ($BF_n = 0$).
  - $h_{left} < h_{right}$ ($BF_n = 1$).

- Unbalanced when
  - $|BF_n| > 1$.



Insertion

- Insert the node using the standard procedure. **If the height changes proceed** to…
  - Recalculate the BF when coming back from the recursive calls (updating the BF of the nodes being part of the search path).
  - If $|BF_n| > 1$ for any ***n*** rebalance the nodes (two possible scenarios).

Class AVLTreeNode

```
public class AVLNode <T extends Comparable <T>>{
        private T element;
        public  AVLNode<T> left;
        private AVLNode<T> right;
        int BF; // int height;
}
```

AVL Trees

Add (Pseudo code)

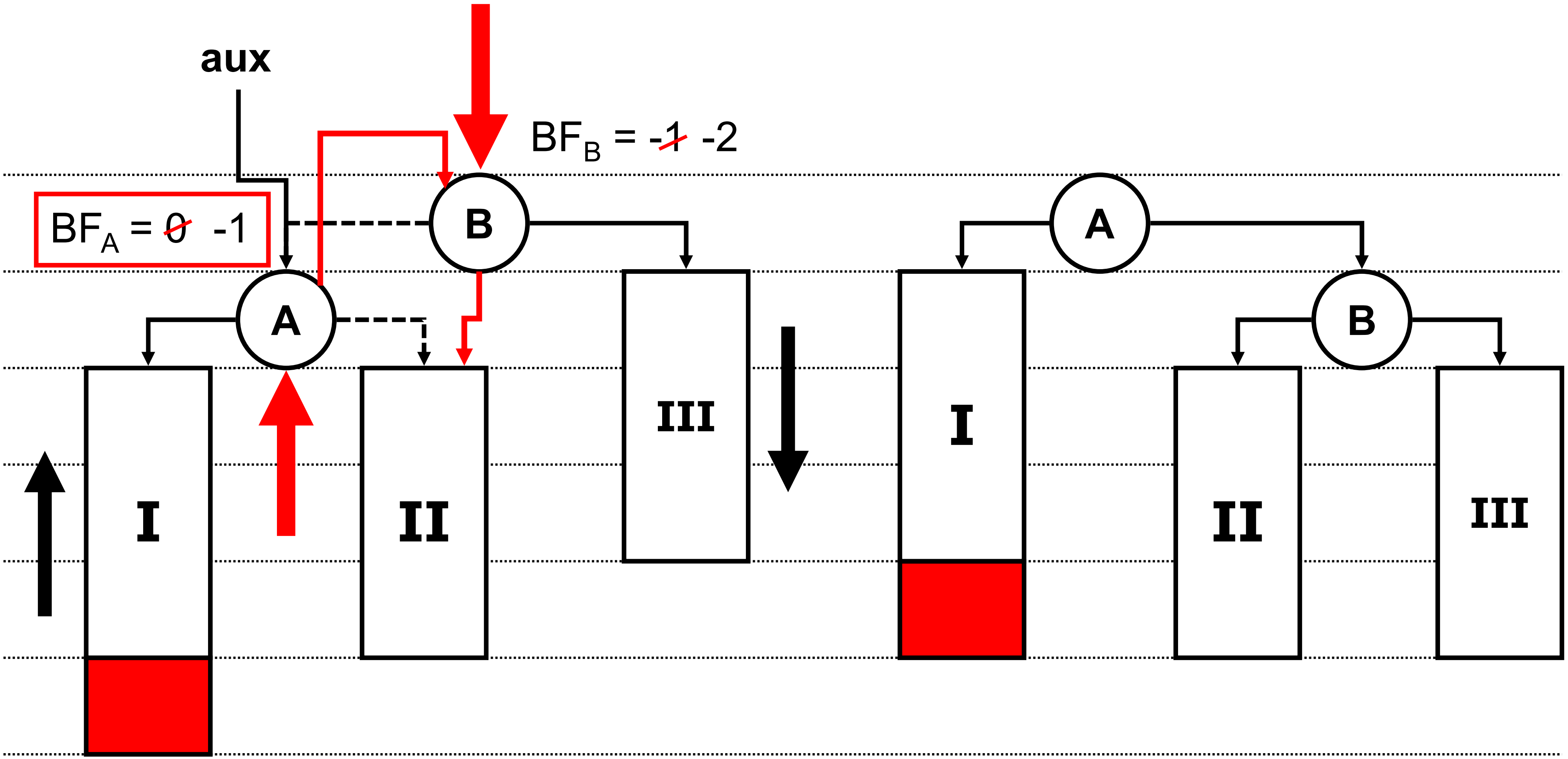
```
private AVLNode<T> add (AVLNode<T> theRoot, T element)
{
 if (theRoot == null)
  return new AVLNode<T>(element);

 if (element.compareTo(theRoot.getElement()) == 0)
  throw new RuntimeException("the element already
exist!");

 if (element.compareTo(theRoot.getElement()) < 0)
  theRoot.setLeft(add(theRoot.getLeft(), element));
 else
  theRoot.setRight(add(theRoot.getRight(), element));

 return(updateBF (theRoot));
}
```

UpdateBF (Pseudo code)

```
private AVLNode<T> updateBF (AVLNode<T> theRoot){

 if (theRoot.getBF() == -2)
 {
  if (theRoot.getLeft().getBF() <=0)
   theRoot = singleLeftRotation (theRoot);
  else
   theRoot = doubleLeftRotation (theRoot);
 }
 else if (theRoot.getBF() == 2)
 {
  if (theRoot.getRight().getBF() >= 0)
   theRoot = (singleRightRotation (theRoot));
  else
   theRoot = (doubleRightRotation (theRoot));
  }

  theRoot.updateHeight();
  return (theRoot);
}
```

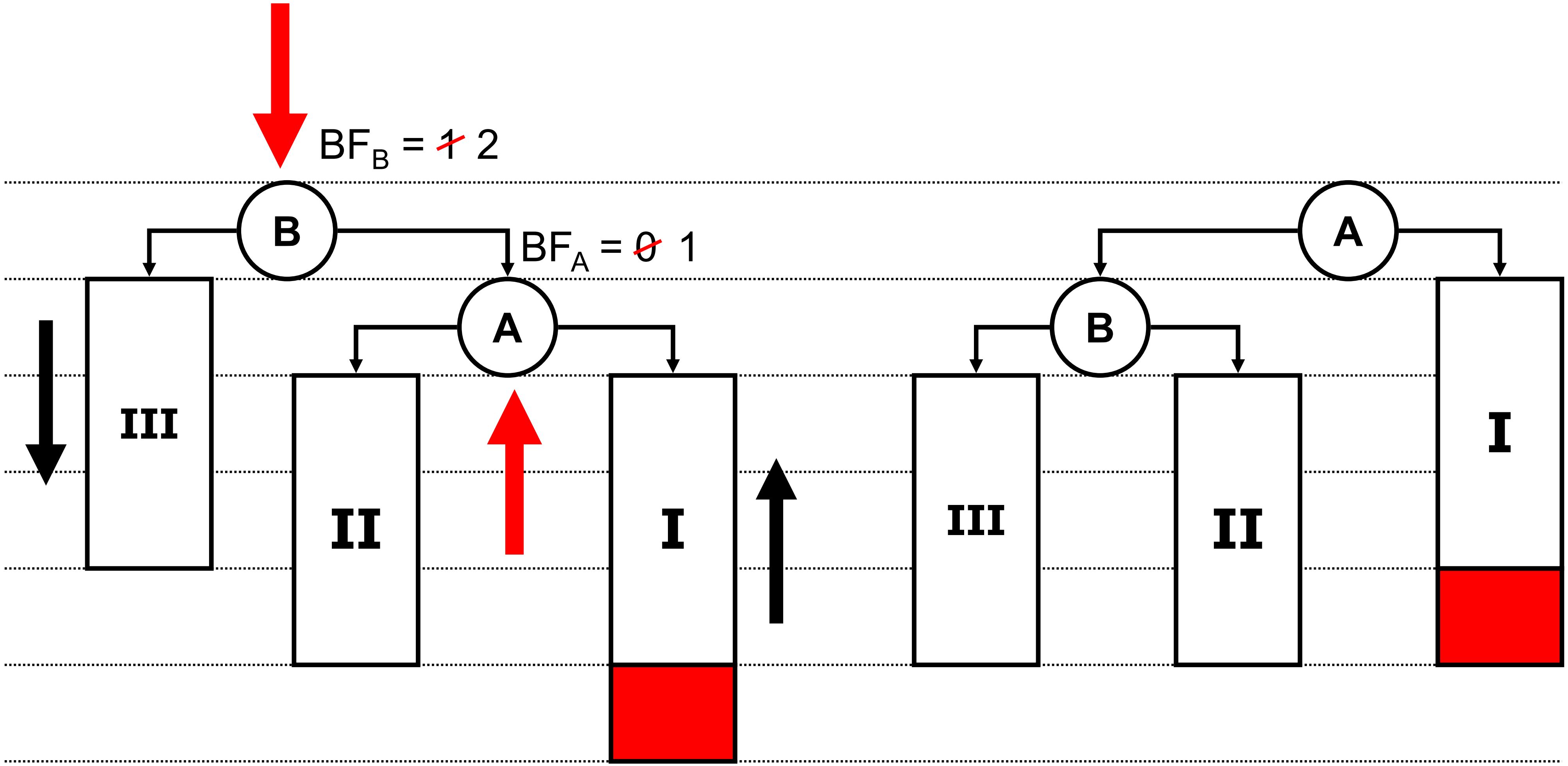
Case Ia

- **Simple balance** (left)



Case Ib

- **Simple Balance** (right)



CLASSWORK

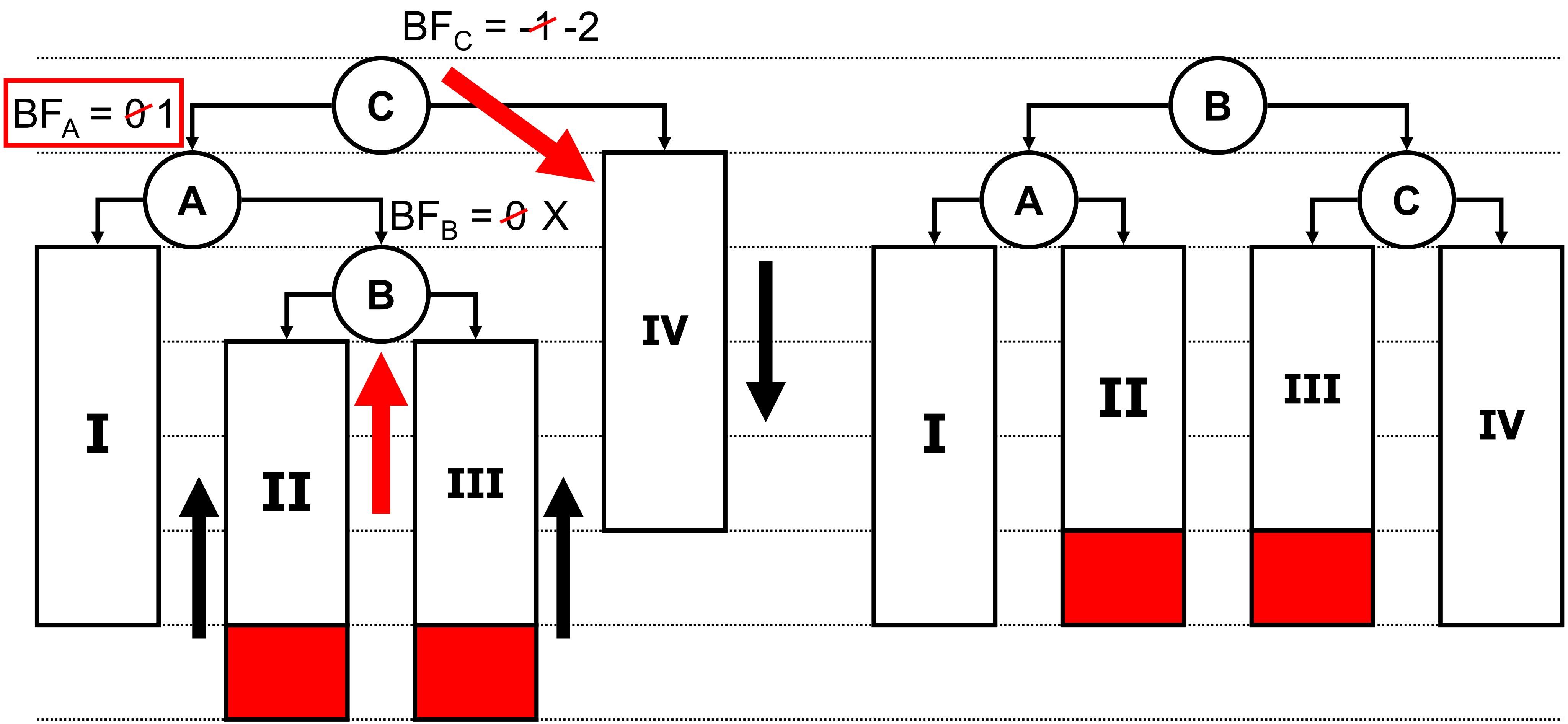
PLAYGROUND

- **Exercise AVL 1**. start with an empty AVL tree…
  a) Insert the elements sequence 7, 6, 5, 4, 3, 2, 1.
     - Analyze the temporal complexity of every insertion.

  b) Insert the elements sequence 8, 9, 10.

Case IIa

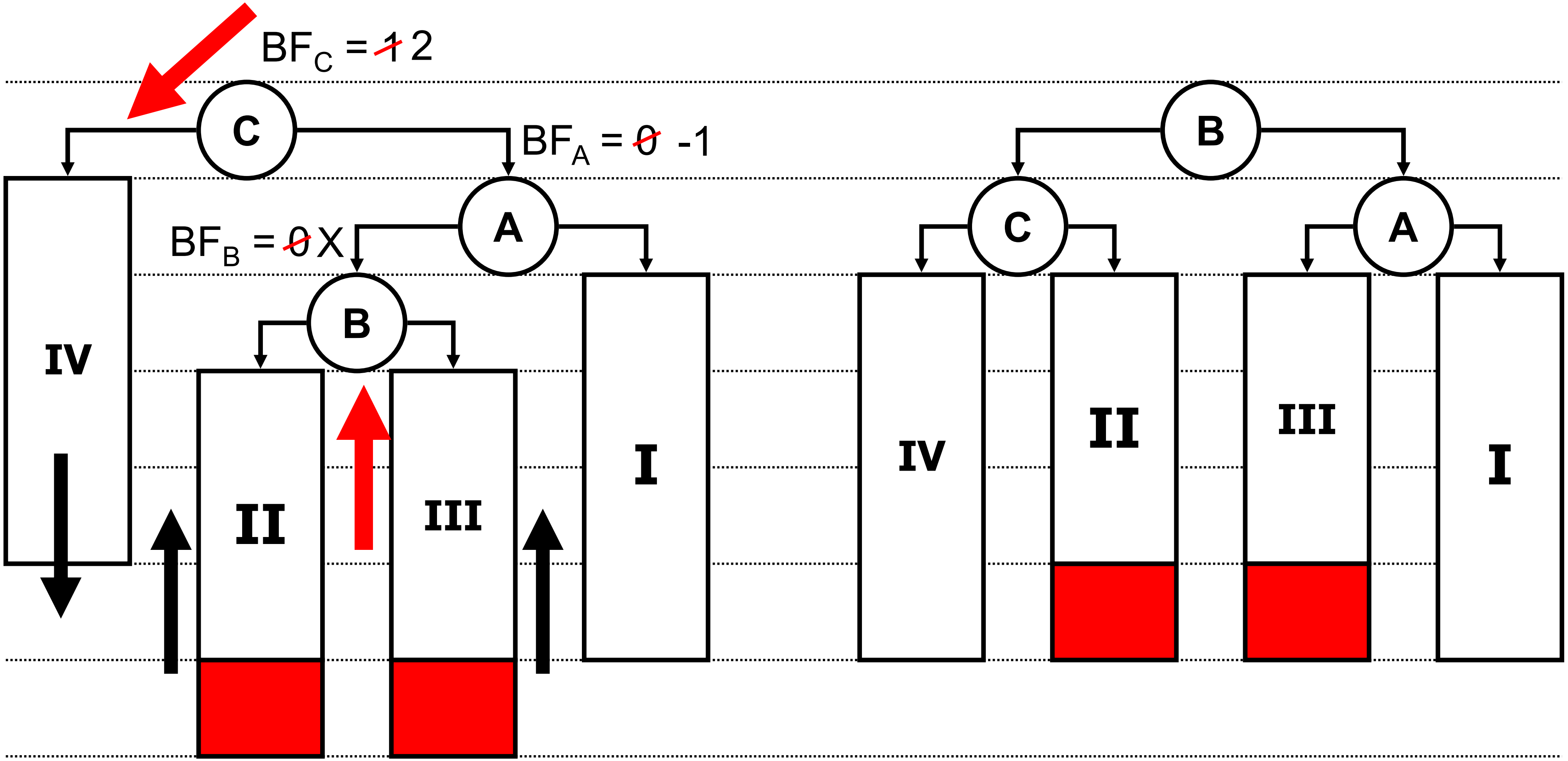
❖ **Double Balance** (left)



AVL Trees

Case IIb

- **Double Balance** (right)



PLAYGROUND

- **Exercise AVL 2**. start with an empty AVL tree…
  - Insert the elements sequence 1, 2, 3, 4, 5, 6, 10, 11, 8, 7.
    - Analyze the temporal complexity of every insertion.

- **Exercise AVL 3**. start with an empty AVL tree…
  - Insert the elements sequence 5, 2, 10, 15, 12, 9, 7, 8, 6.

Deletion

- Delete as usual… **if the tree's height changes…**
  - Recalculate the BFs coming back from the recursion calls (update the BF in every node of the search path).
    - In terms of height change, the deletion of a node in the left sub tree is equivalent to insert a node in the right sub tree.
  - If $|BF_n| > 1$ rebalance must be done.

- Balance must be applied to the whole search path!
  - Rebalancing of a subtrees **does not ensure a full balance** in the whole tree.
    - Unlike insertion, deletion rebalancing must be done all the way long until reaching the root.

AVL Trees

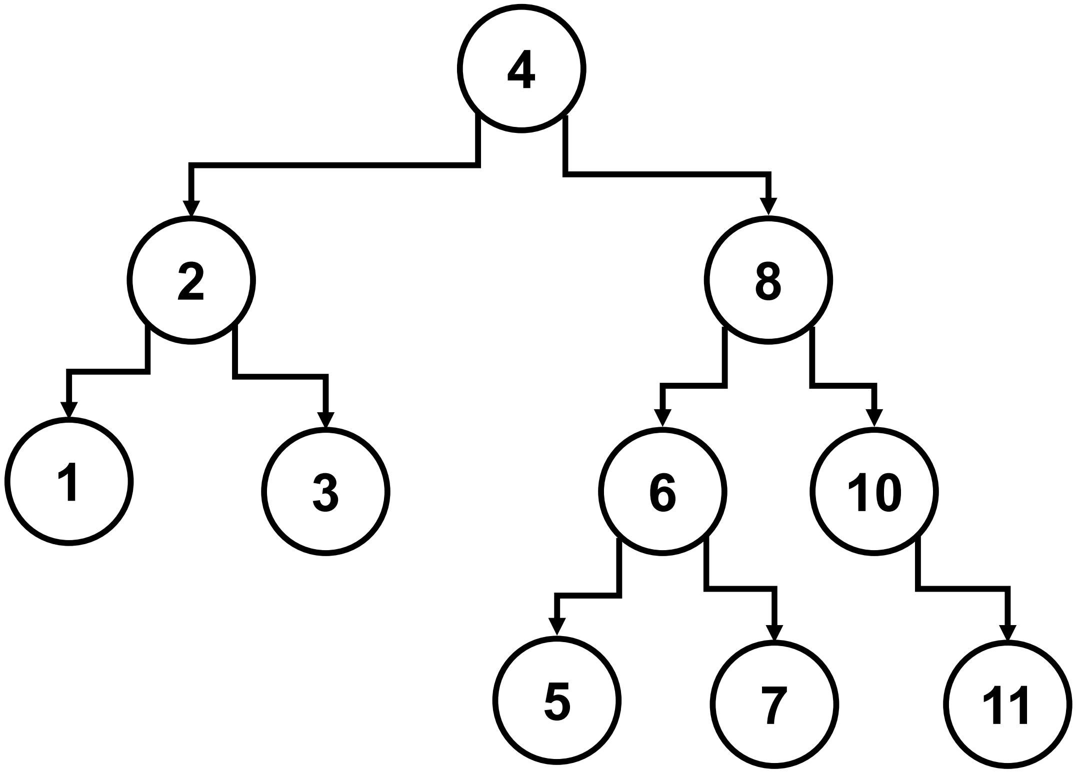
Remove (Pseudo code)

```
private AVLNode<T> remove (AVLNode<T> theRoot, T element)
{
  if (theRoot == null)  throw new RuntimeException("element does
not exist!");
  else
  if (element.compareTo(theRoot.getElement()) < 0)
   theRoot.setLeft(remove (theRoot.getLeft(), element));
  else
   if (element.compareTo(theRoot.getElement()) > 0)
    theRoot.setRight(remove (theRoot.getRight(), element));
  else {
    if (theRoot.getLeft() == null) return theRoot.getRight();
    else {
     if (theRoot.getRight() == null) return theRoot.getLeft();
     else // copies the max value from the left subtree...
      theRoot.setElement(getMax(theRoot.getLeft()));
theRoot.setLeft(remove (theRoot.getLeft(), theRoot.getElement()));
    }}
  return (updateBF (theRoot));
}
```

CLASSWORK

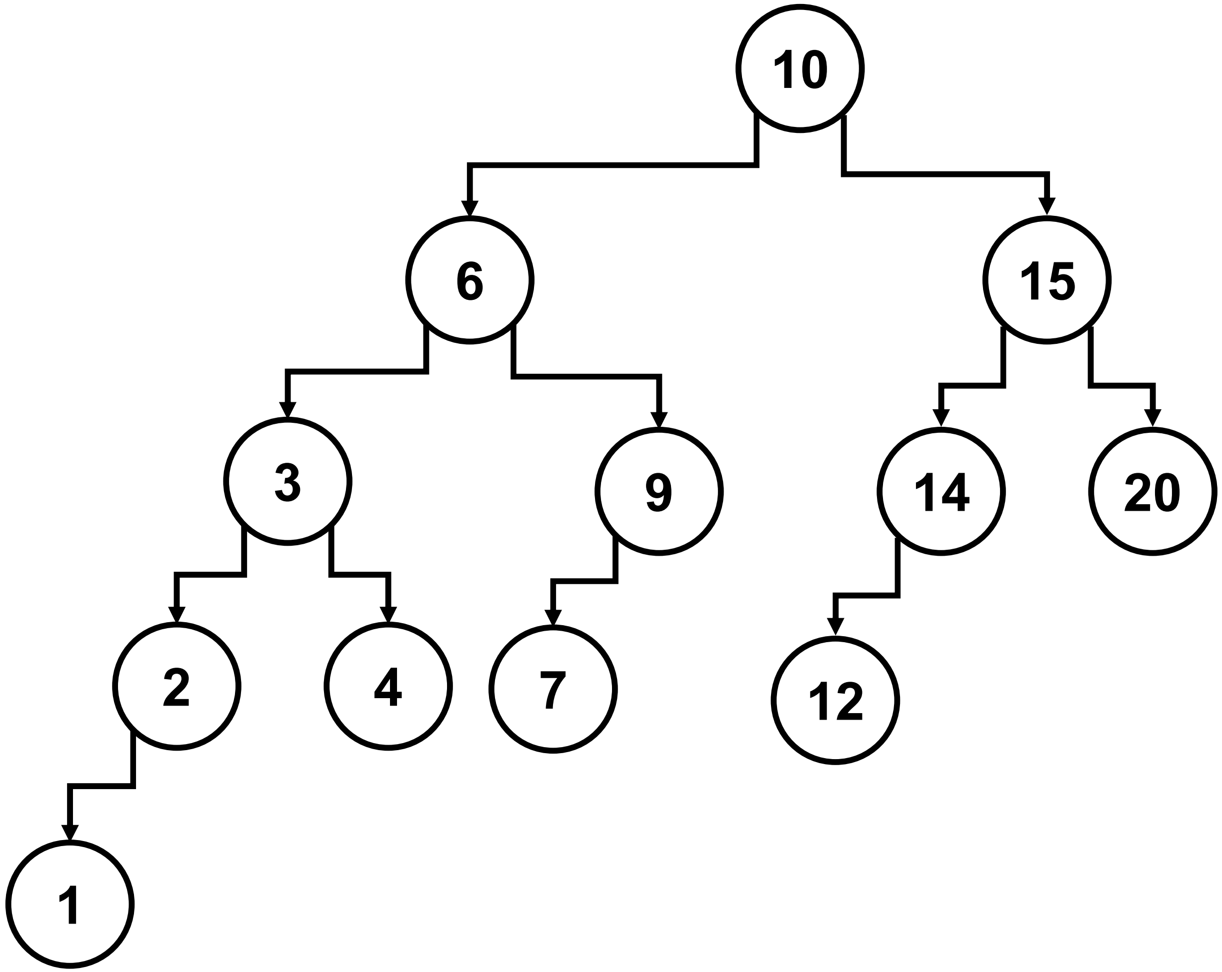
PLAYGROUND

- **Exercise AVL 4**. Starting from the resulting AVL tree of the exercise AVL 2…
  - Delete the sequence of elements: 1, 3, 4, 7, 11, 10.
    - Analyze the temporal complexity of every insertion.



PLAYGROUND

- **Exercise AVL 5**. Starting from this AVL tree…
  - Delete the sequence of elements 20, 4, 10, 9, 6, 3.



AVL Trees

Performance

- Worst case
  - Rebalancing an AVL affects the search path only
    - Its longitude is $O(\log_2 n)$

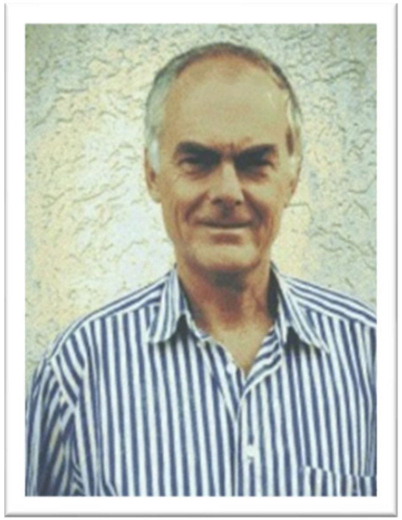
$$\text{Log}_2 n <= \text{Search path longitude} <= 1.44\text{Log}_2 n$$

Method	**PBT**	**AVL**
Insert	$O(n)$	$O(\log_2 n)$
Search	$O(\log_2 n)$	$O(\log_2 n)$
Deletion	$O(n)$	$O(\log_2 n)$

B Trees (Bayer & McCreight)

Problem to Solve

- Building trees on secondary memory (disk) storing **massive amounts of elements supporting a logarithmic access**
  - **Reduce the tree's height** distributing multiple elements on each level.



Rudolf Bayer (Wikipedia)

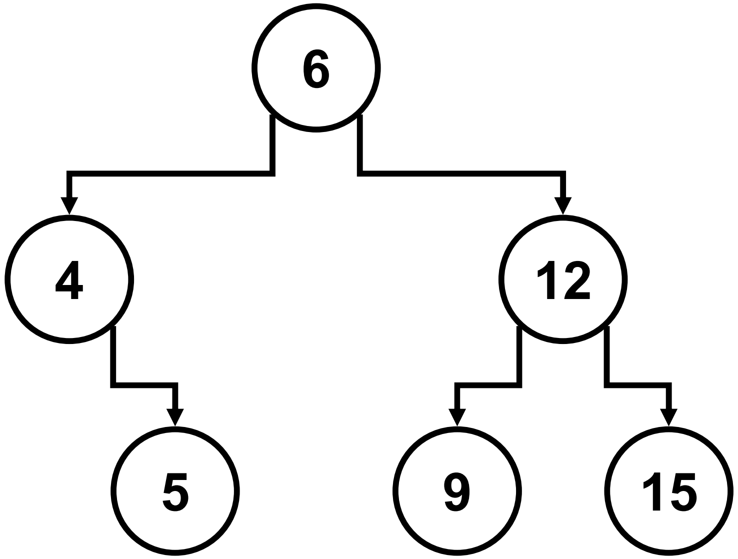
Edward M. McCreight (Wikipedia)

- Developed in 1972 by the German researcher Rudolf Bayer and the Swiss researcher Edward M. McCreight.

B Trees (Bayer & McCreight)

Storing trees on disk

- It is more efficient to process multiple elements in RAM rather to access them one by one in the hard disk.



	data	left	right
0	6	2	1
1	12	3	4
2	4	-1	5
3	9	-1	-1
4	15	-1	-1
5	5	-1	-1

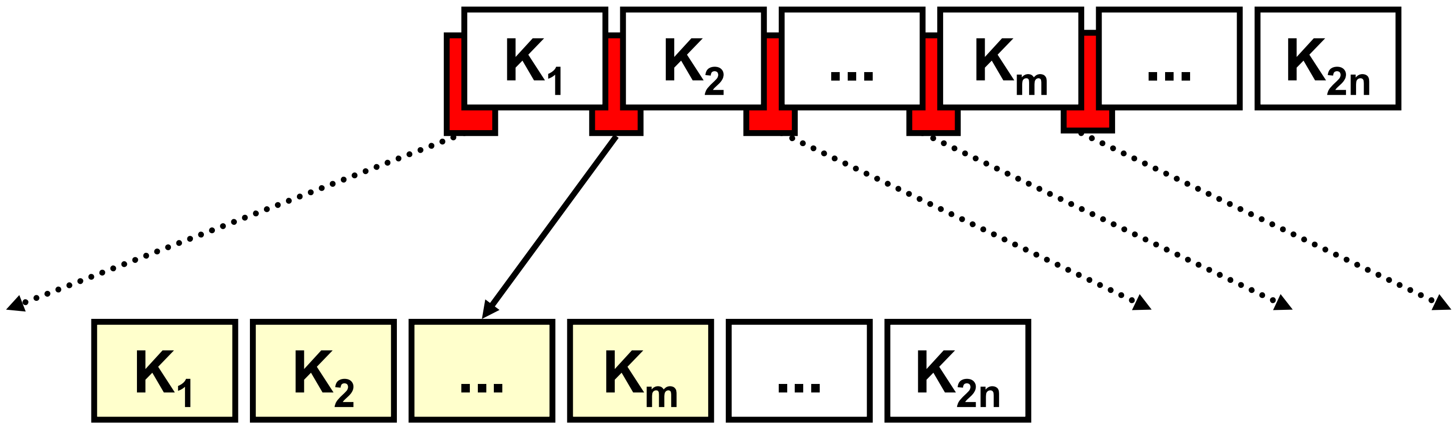
Detecting that an element does not exist in a AVL tree of 1.000.000 elements requires…

… between 20 and 28 disk accesses

B Trees (Bayer & McCreight)

Definition

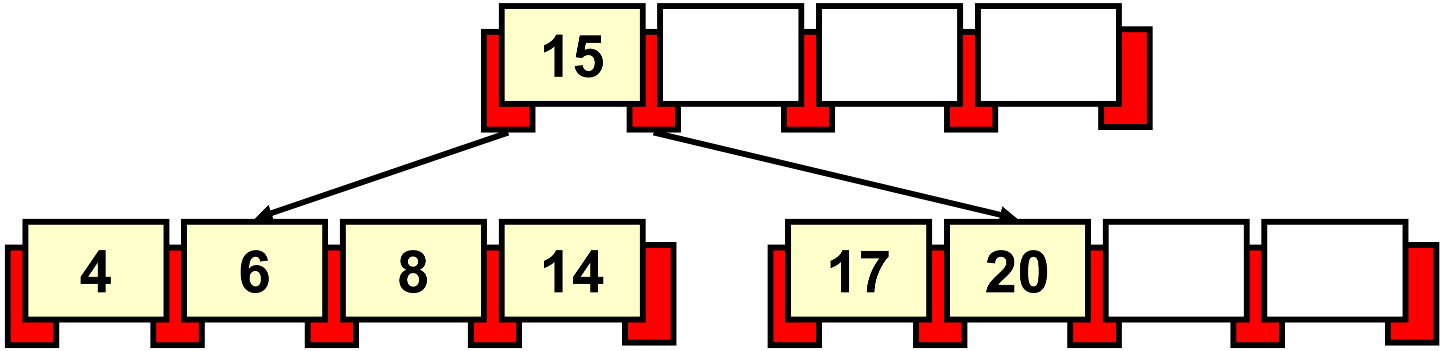
- An B Tree of **order** (B-n) is a tree where…
  - All the leaves are located in the same level.
  - Every node (usually called ***page***) contains m elements (keys) **stored in a sorted way**.
    - The root page contains 1 <= m <= 2n keys.
    - Any non root page contains n <= m <= 2n keys.
  - Every **non leave** page has m + 1 children pages.

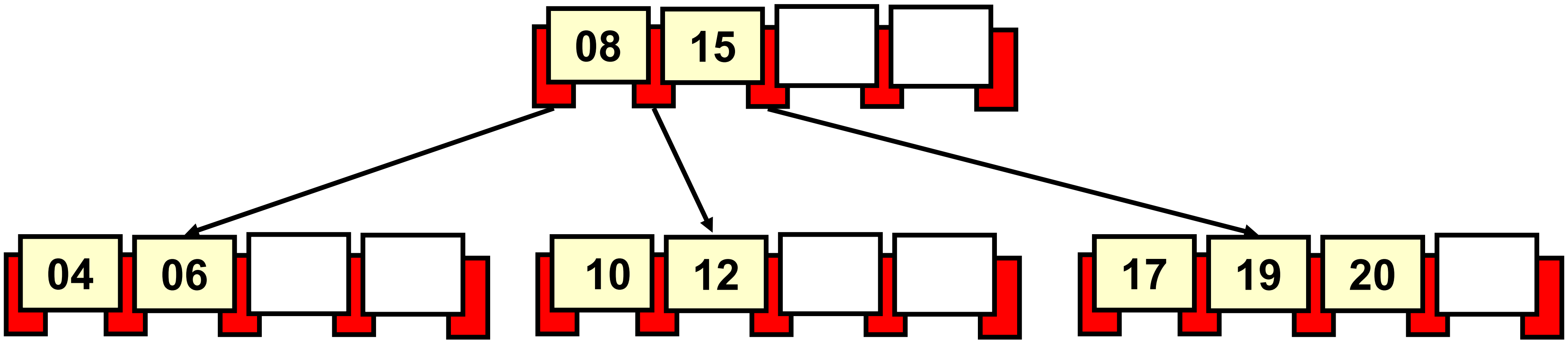


B Trees (Bayer & McCreight)

Examples

- B-2 trees





B Trees (Bayer & McCreight)

Bnode (Pseudo code)

```
class BPage <T extends Comparable <T>> {
 private final static int n= …;
 private final static int 2n = 2*n;

 T elements[1..2n];
 BPage<T> links [0..2n];
 int m;
}

Or…

class BPage <T extends Comparable <T>> {
 private final static int n= …;
 private final static int 2n = 2*n;

 LinkedList<T> elements;
 LinkedList<BPage> links;
 int m; // can be substituted by elements.size();
}
```

B Trees (Bayer & McCreight)

Capacity of a B Tree of order n

- Given a B-n of height **h**, the **minimum** number of keys (N_{Min}) that it can store is…
  - The capacity of a degenerated B-n tree (maximum height).

Level	Pag. per Level	Minimum m value	Total
1	1	1	1
2	2	n	2 n
3	2(n + 1)	n	2 n * (n + 1)
4	$2(n + 1)^2$	n	$2 n * (n + 1)^2$
...			
h	$2(n + 1)^{h-2}$	n	$2 n * (n + 1)^{h-2}$

$$N_{Min} = 1 + 2n * \sum_{i=2}^{h} (n + 1)^{i-2}$$

B Trees (Bayer & McCreight)

Maximum height of a B-n tree

- h_{max} is defined as
  - $N = 1 + 2\,n * \sum_{i=2}^{h} (n + 1)^{i-2}$
    - N is the number of keys in the tree.

- $h_{max} \approx 1 + \text{Log}_{n+1}(N+1)/2$
  - If the constant **n** is greater enough, h_{max} may be estimated as:
    - $h_{max} \approx \text{Log}_n N$.

Range for the height of a B-n tree
$h <\approx 1 + \text{Log}_{n+1}(N+1)/2$ $O(h) <\approx O(\text{Log}_n N)$

- The higher the order of the tree (n) the lower its height.

B Trees (Bayer & McCreight)

Capacity of a B Tree of order n

- Given a B-n of height **h**, the **maximum** number of keys (N_{Max}) that it can store is…
  - The capacity of a complete (compact) B-n tree (minimum height).

Level	Pag. Per Level	Maximum m value	Total
1	1	2n	2n
2	(2n + 1)	2n	2 n * (2n + 1)
3	$(2n + 1)^2$	2n	$2n * (2n + 1)^2$
4	$(2n + 1)^3$	2n	$2n * (2n + 1)^3$
…			
h	$(2n + 1)^{h-1}$	2n	$2n * (2n + 1)^{h-1}$

$$N_{Max} = 2n * \sum_{i=1}^{h} (2n + 1)^{i-1}$$

B Trees (Bayer & McCreight)

Minimum height of a B-n

- h_{Min} is calculated as…
  - $N = 2n * \sum_{i=1}^{h} (2n + 1)^{i-1}$.
    - N is the number of keys in the tree.

- $h_{min} \approx Log_{2n+1}(N+1)$.
  - If the constant **n** is great enough, h_{min} may be estimated as:
    - $h_{min} \approx Log_{2n}N$.

Range for the height of a B-n tree
$Log_{2n+1}(N+1) \lesssim h \lesssim 1 + Log_{n+1}(N+1)/2$
$O(Log_{2n}N) \le O(h) \le O(Log_{n}N)$

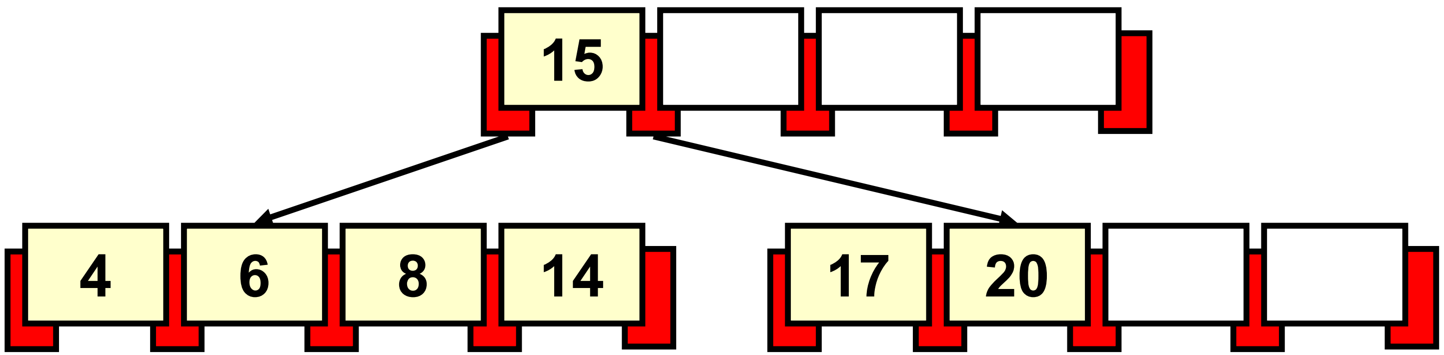
- The higher the order of the tree (n) the lower its height.

B Trees (Bayer & McCreight)

Searching

- Look for the element X among the *elements* in the page
  - Sequential search.
  - Binary search.

- If the search fails, the algorithm stops in the position j (elements[j]) of the page such that 0 <= j <=m
  - Load the page *links[j]* and repeat the search over again.
  - This recursive process is repeated over and over again until finding X or reaching a null link (determining that the element does not exist).



DISCUSSION: What is the temporal complexity of this algorithm?

B Trees (Bayer & McCreight)

Temporal Complexity

Best Case

- The element is found in the root
  - O(m) = O(1).
  - As 1 <= m <= 2n, **m** it can be considered as constant value.

Worst Case

- The search is performed in a degenerated tree and the element does not exist
  - O(h) * O(m).
  - $O(\log_n N) * O(1) = O(\log_n N)$.

Detecting that an element does not exist in a B-10 tree of 1.000.000 elements requires…

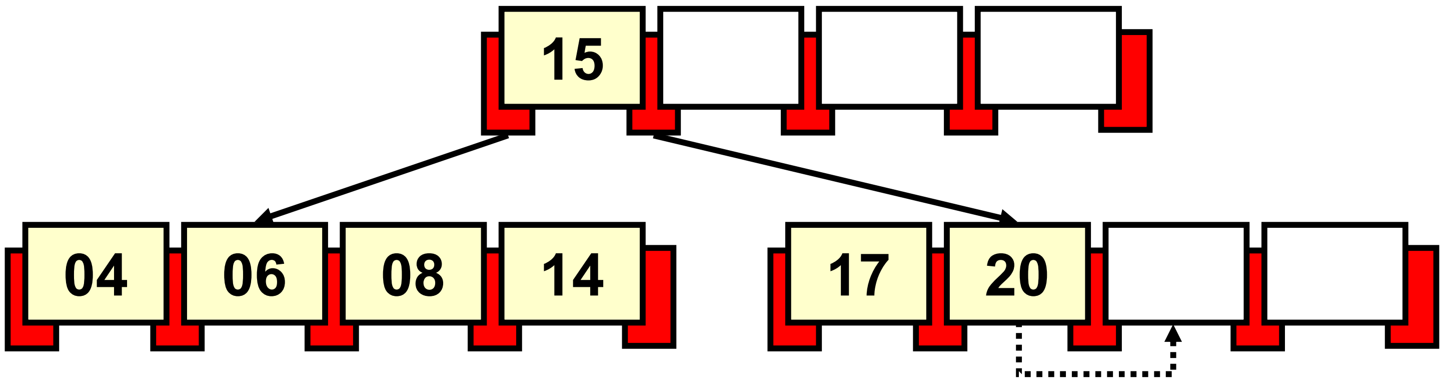
… between 5 and 6 disk accesses

… an AVL tree would require between 20 and 28 accesses

B Trees (Bayer & McCreight)

Insertion

- **Case I:** Leaf page has m < 2*n keys.
  - Move all the elements with a key greater than that of the object to be inserted one slot to the right in order to create a new empty slot.

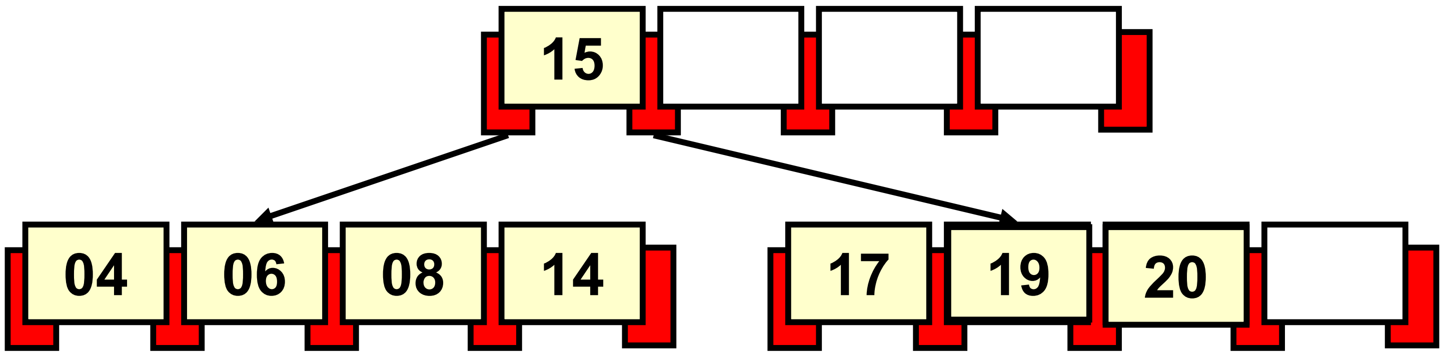


- Insertion is **always** done in the leaves and it is produced **only** as a result of an unsuccessful search.

B Trees (Bayer & McCreight)

Insertion

- **Case I:** Leaf page has m < 2*n keys.
  - Move all the elements with a key greater than that of the object to be inserted one slot to the right to create a new empty slot.

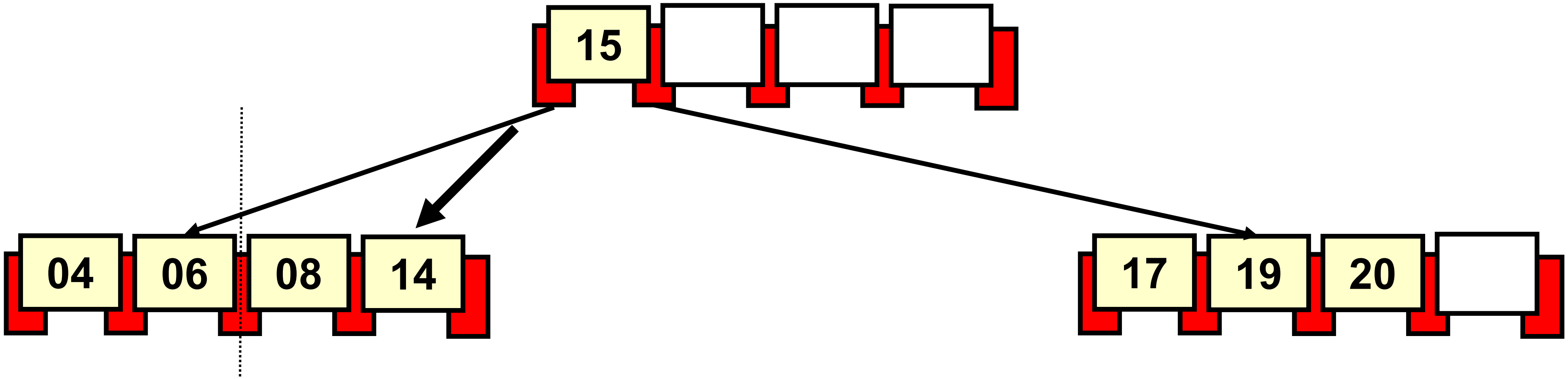


- Insertion is **always** done in the leaves and it is produced **only** as a result of an unsuccessful search.

B Trees (Bayer & McCreight)

Insertion

- **Case 2**: Leaf page has m = 2*n claves (**Overflow**).
  - Split the leaf in two and distribute the keys among them
    - Last (m+1)/2 keys in a new leaf.
    - First (m+1)/2 keys remain in the original leaf.
    - Central element (median) is inserted in the upper page to become a new index.
  - If the upper page is full, the process is executed again. It can be repeated over and over again **until reaching the root**.
    - Splitting the root in two is the only way that a B tree can increase its height.

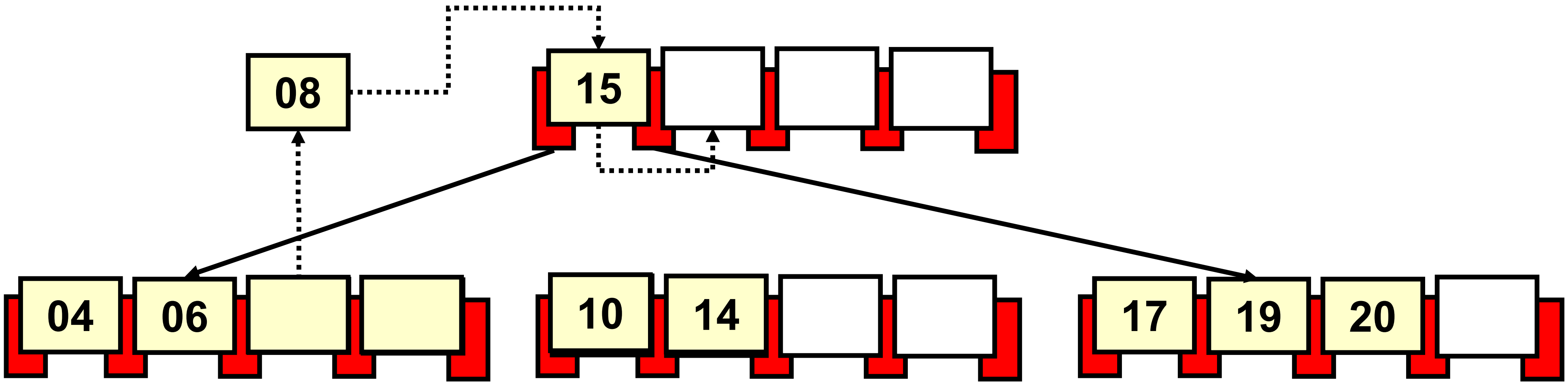


EVENT: Inserting element 10

B Trees (Bayer & McCreight)

Insertion

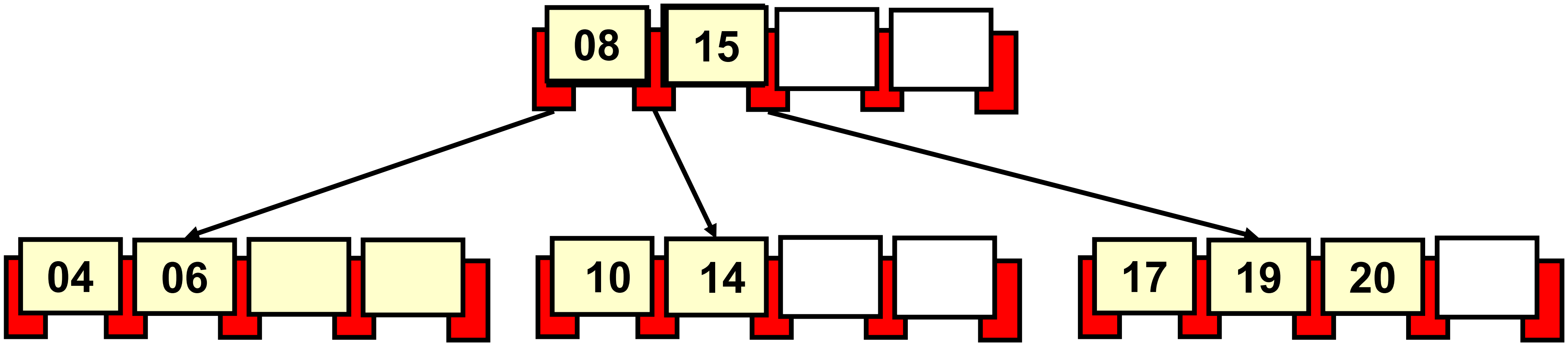
- **Case 2**: Leaf page has m = 2*n claves (**Overflow**).
  - Split the leaf in two and distribute the keys among them
    - Last (m+1)/2 keys in a new leaf.
    - First (m+1)/2 keys remain in the original leaf.
    - Central element (median) is inserted in the upper page to become a new index.
  - If the upper page is full, the process is executed again. It can be repeated over and over again **until reaching the root**.
    - Splitting the root in two is the only way that a B tree can increase its height.



B Trees (Bayer & McCreight)

Insertion

- **Case 2**: Leaf page has m = 2*n claves (**Overflow**).
  - Split the leaf in two and distribute the keys among them
    - Last (m+1)/2 keys in a new leaf.
    - First (m+1)/2 keys remain in the original leaf.
    - Central element (median) is inserted in the upper page to become a new index.
  - If the upper page is full, the process is executed again. It can be repeated over and over again **until reaching the root**.
    - Splitting the root in two is the only way that a B tree can increase its height.



B Trees (Bayer & McCreight)

Temporal Complexity for the Insertion operation

- **Best Case**
  - Element is inserted in the leaf of a minimum height tree with slots enough to avoid splitting.
    - $O(\log_{2n}(N)) + O(m) = O(\log_{2n}(N))$.

- **Worst Case**
  - Element is inserted in a maximum height tree and the insertion requires the splitting of all the pages along the search path.
    - $O(\log_{n}(N)) * O(n) = O(\log_{n}(N))$.

CLASSWORK

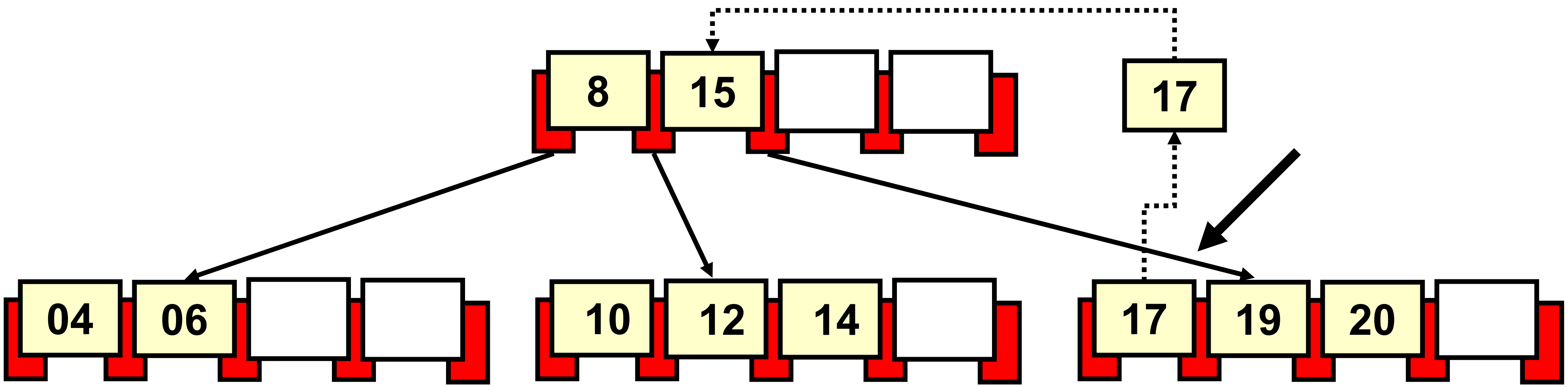
PLAYGROUND

- **Exercise B Tree (Insertion)**. Starting from an empty B-2 tree…
  a) Insert the key sequence 6, 11, 5, 4, 8, 9, 12.
  b) Insert the key 21.
  c) Insert the key sequence 14, 10, 19, 28.
  d) Insert the key sequence 3, 17, 32, 15, 16.
  e) Insert the key sequence 26, 27.

B Trees (Bayer & McCreight)

Deletion

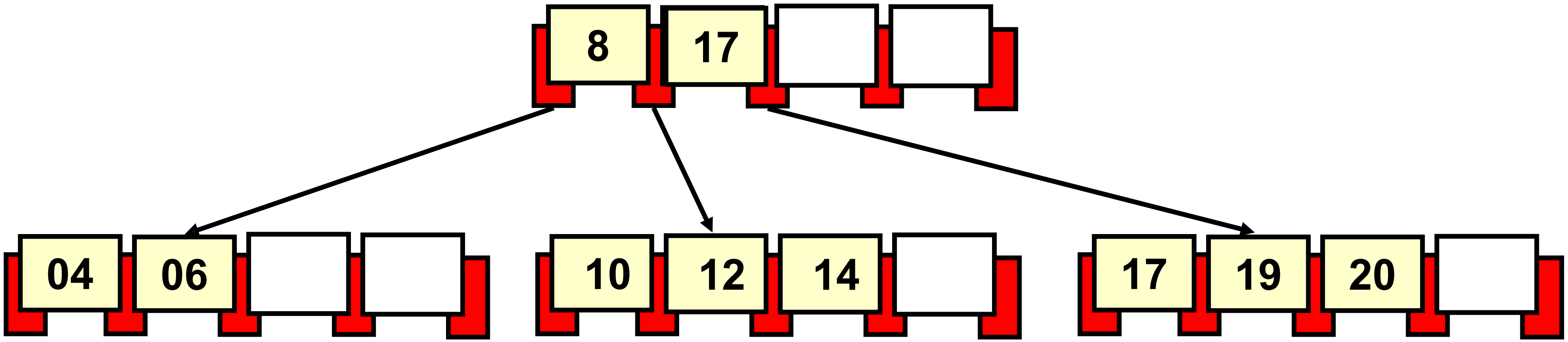
- **Deleting an inner element**
  - Substitute the element by its successor
    - The successor is found in the first slot of the leftist leaf on the right sub tree.
  - Delete the element in the **source page**.



B Trees (Bayer & McCreight)

Deletion

- **Deleting an inner element**
  - Substitute the element by its successor
    - The successor is found in the first slot of the leftist leaf on the right sub tree.
  - Delete the element from the **source page**.

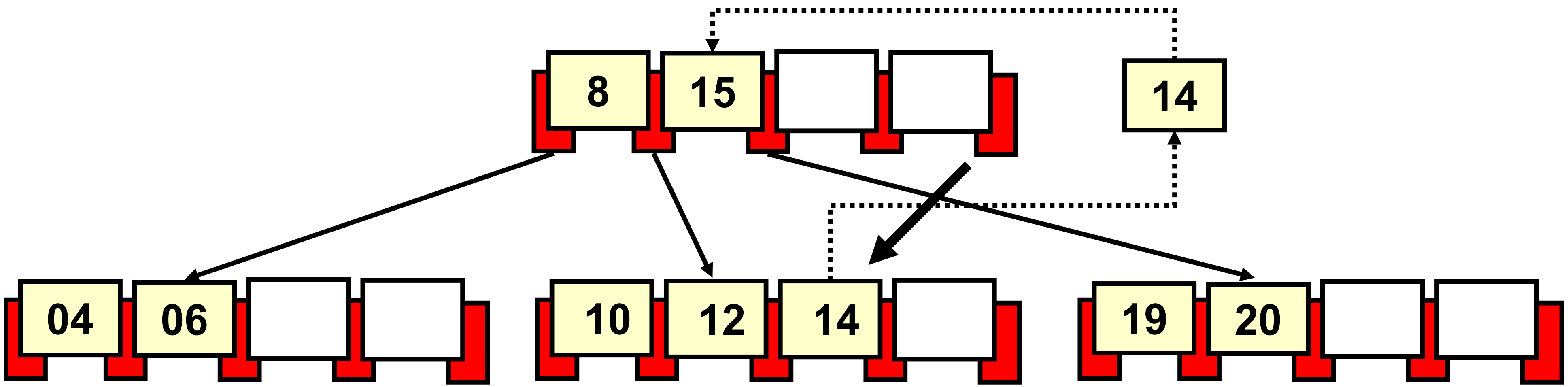


B Trees (Bayer & McCreight)

Deletion

❖ **Deleting an inner element**

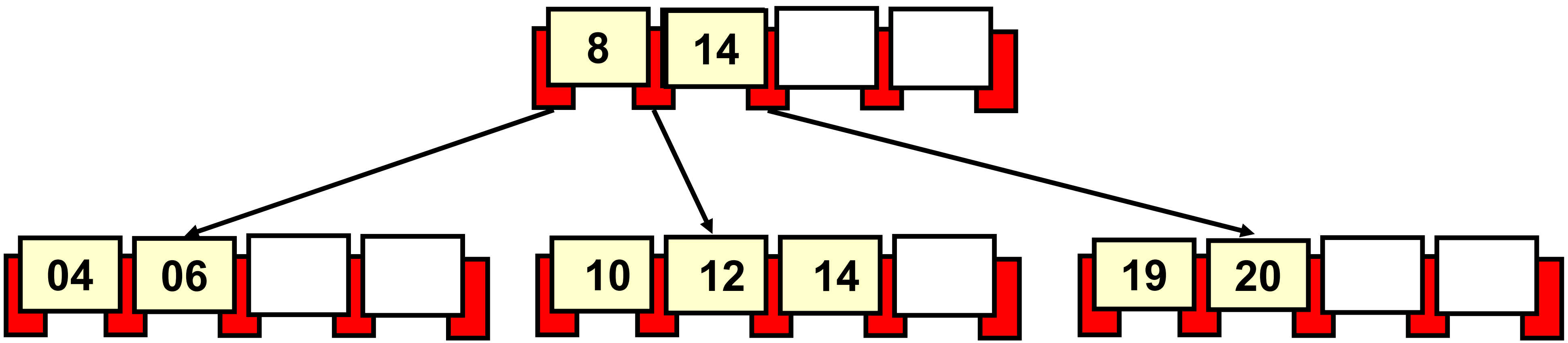
- Substitute the element by its successor
  - The successor is found in the first slot of the leftist leaf on the right sub tree.
- If the **source page is under a critical situation**…
  - Try to substitute the element by its predecessor (located in the last slot of the rightist leaf of the left sub tree).
  - The page is under a critical situation if m=n before the substitution.
- Delete the element from the **source page**.



B Trees (Bayer & McCreight)

Deletion

- **Deleting an inner element**
  - Substitute the element by its successor
    - The successor is found in the first slot of the leftist leaf on the right sub tree.
  - If the **source page is under a critical situation**…
    - Try to substitute the element by its predecessor (located in the last slot of the rightist leaf on the left sub tree).
    - The page is under a critical situation if m=n before the substitution.
  - Delete the element from the **source page**.

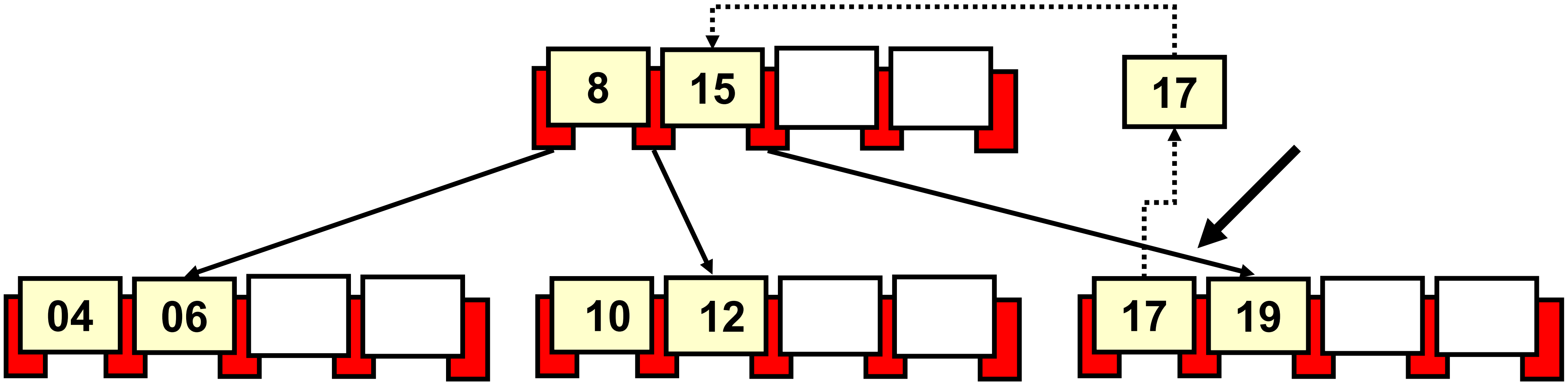


B Trees (Bayer & McCreight)

Deletion

❖ Deleting an inner element

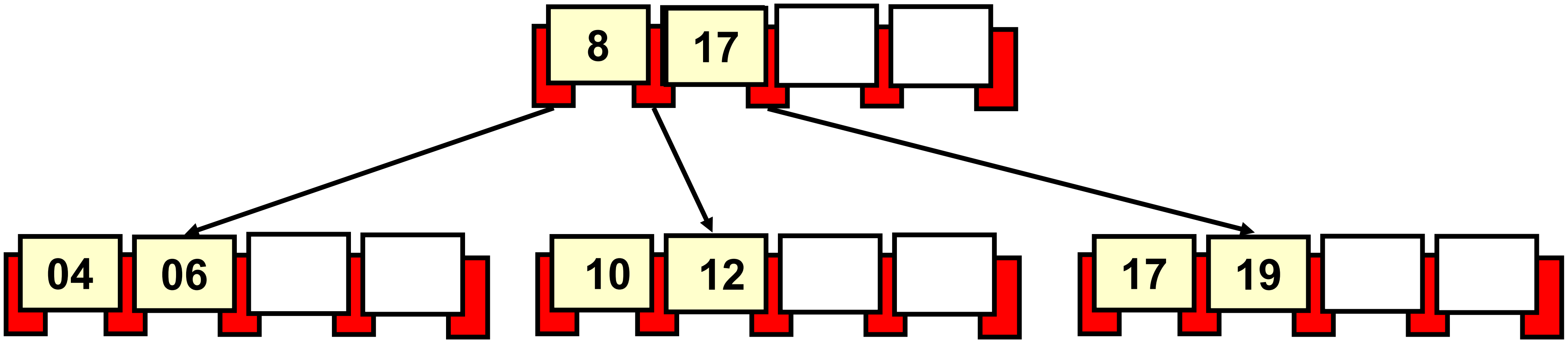
- Substitute the element by its successor
  - The successor is found in the first slot of the leftist leaf on the right sub tree.
- If the **source page is under a critical situation**…
  - Try to substitute the element by its predecessor (located in the last slot of the rightist leaf on the left sub tree).
  - The page is under a critical situation if m=n before the substitution.
- If the **source page** is under a critical situation**, substitute the element by its successor**.
- Delete the element from the **source page**.



B Trees (Bayer & McCreight)

Deletion

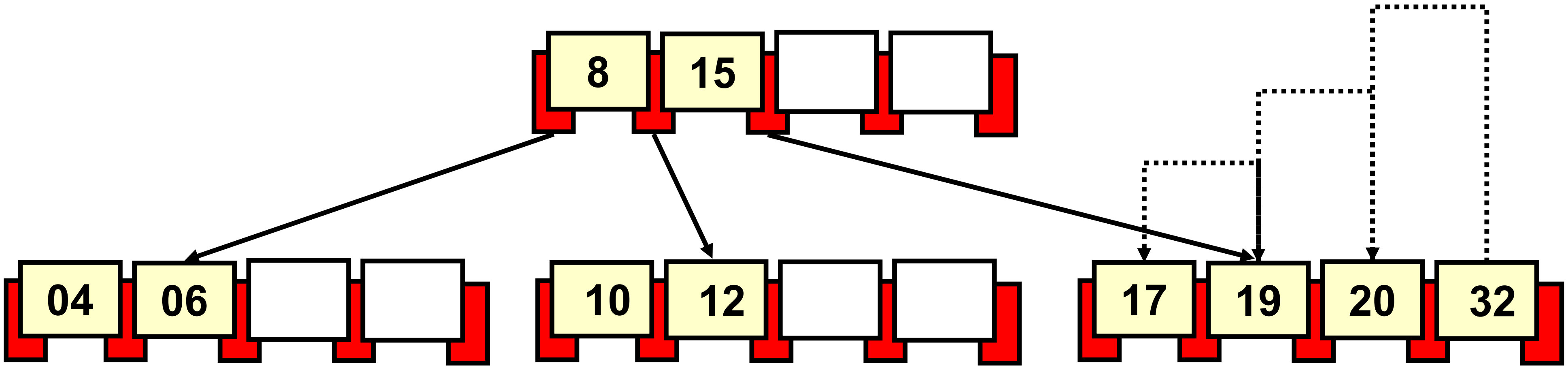
- **Deleting an inner element**
  - Substitute the element by its successor
    - The successor is found in the first slot of the leftist leaf on the right sub tree.
  - If the **source page is under a critical situation**…
    - Try to substitute the element by its predecessor (located in the last slot of the rightist leaf on the left sub tree).
    - The page is under a critical situation if m=n before the substitution.
  - If the **source page** is under a critical situation**, substitute the element by its successor**.
  - Delete the element from the **source page**.



B Trees (Bayer & McCreight)

Deletion

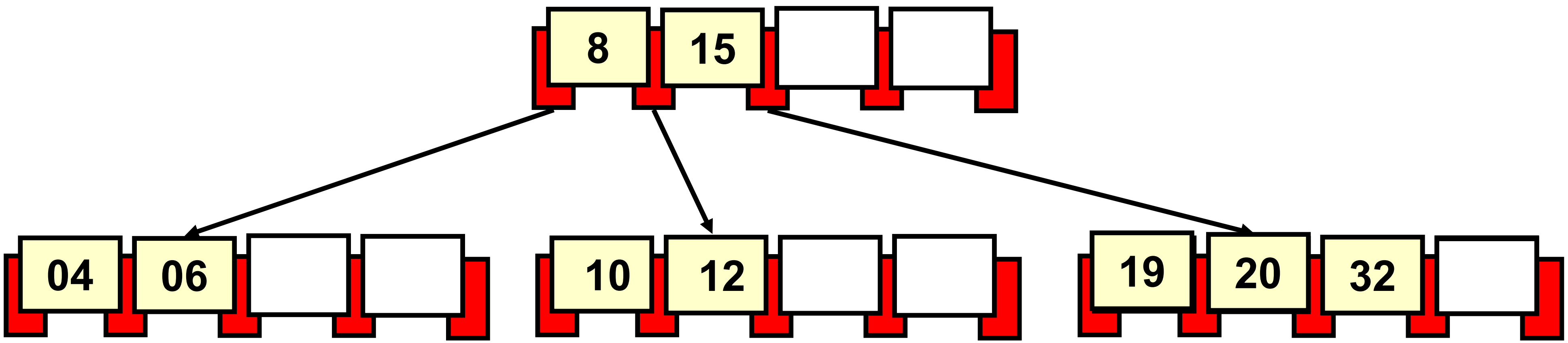
- **Deleting an element from a leaf page**
  - **Case 1**: the page has n < m keys.
    - Elements to right of the element are moved one position to the left (hiding the now empty slot).



B Trees (Bayer & McCreight)

Deletion

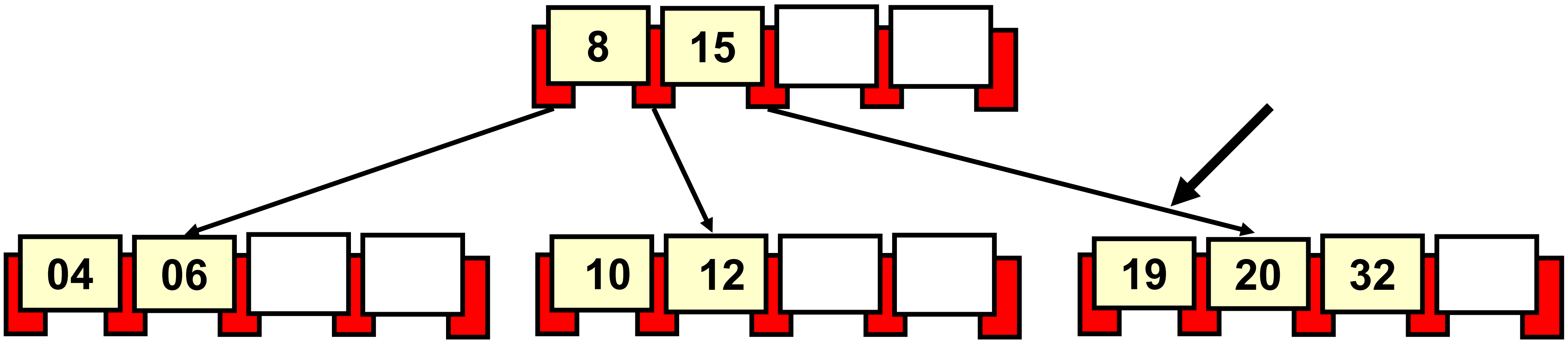
- **Deleting an element from a leaf page**
  - **Case 1**: the page has n < m keys.
    - Elements to right of the element are moved one position to the left (hiding the now empty slot).



B Trees (Bayer & McCreight)

Deletion

- **Deleting an element from a leaf page**
  - **Case 2**: the page has n = m keys (*underflow*).
    - Search among the adjacent leaves in order to get someone with n < m to borrow a key.
      - The page **to the right is verified first** (if it exists). If it can not provide any key, the search process **is attempted again on the page to the left**.
      - The leaf can not provide keys when it its under a critical situation (n = m).



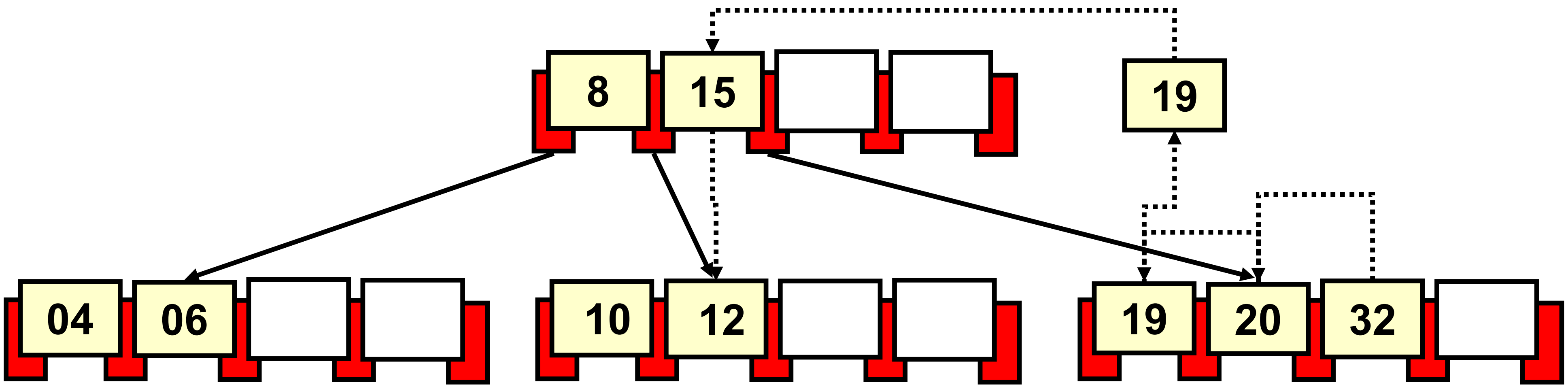
EVENT: Deleting element 12

B Trees (Bayer & McCreight)

Deletion

❖ **Deleting an element from a leaf page**

- **Case 2**: the page has n = m keys (*underflow*).
  - Borrowing is done through the upper page.
  - The borrowed element is sent to the upper page to replace the index element. The former index **is sent down** to the page requiring the extra element where **it replaces the deleted element.**

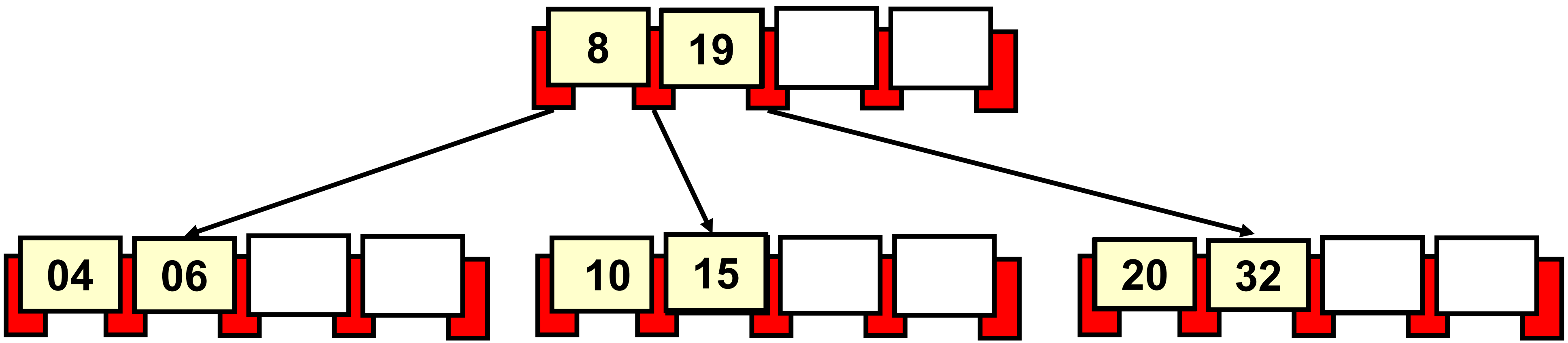


B Trees (Bayer & McCreight)

Deletion

❖ Deleting an element from a leaf page

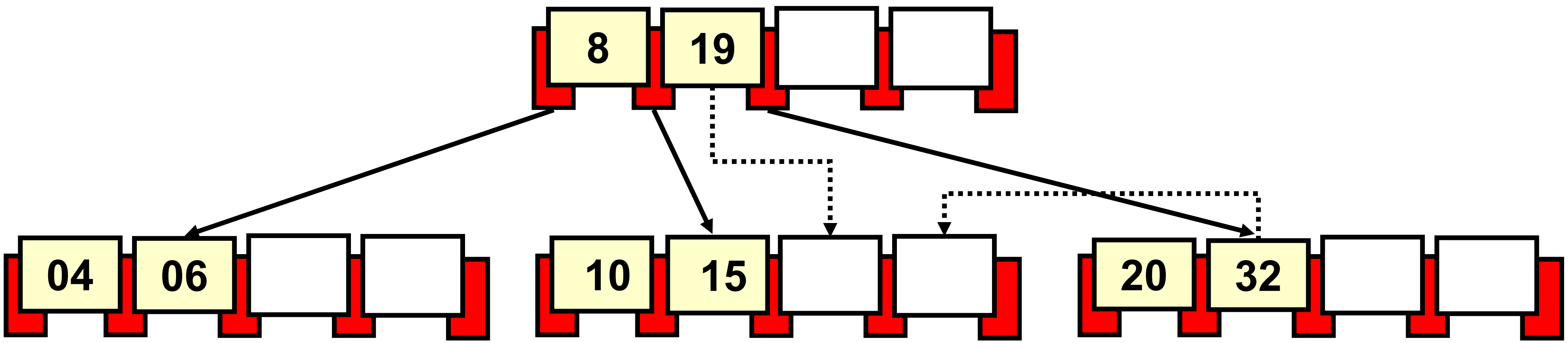
- **Case 2**: the page has n = m keys (*underflow*).
  - Borrowing is done through the upper page.
  - The borrowed element is sent to the upper page to replace the index element. The former index **is sent down** to the page requiring the extra element where **it replaces the deleted element.**



B Trees (Bayer & McCreight)

Deletion

- **Deleting an element from a leaf page**
  - **Case 2b**: the page has n = m keys (*underflow*) and no page can provide elements.
    - Both adjacent pages (left and right) are under a critical situation.
    - The page merges with the page on the right (if it does not exist, the page is merged with the one on the left).
      - The resulting page includes the elements of both pages plus the index element that **must be deleted from the upper page**.
      - The deletion of the index in the upper page **may conduct to a recursive deletion in all the pages of the search path**.
        - If this process reaches the root, **it will reduce the height of the tree**.

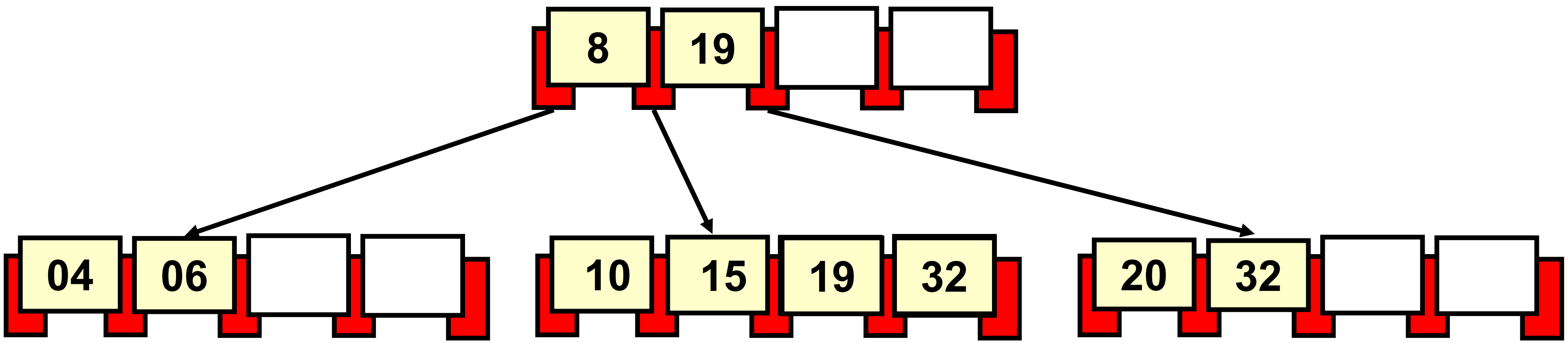


B Trees (Bayer & McCreight)

Deletion

❖ Deleting an element from a leaf page

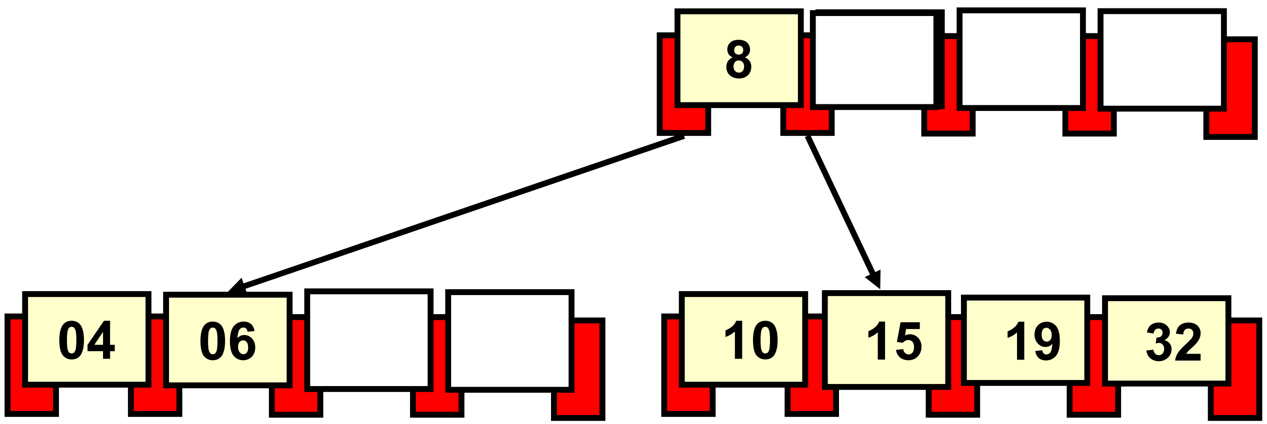
- **Case 2b**: the page has n = m keys (*underflow*) and no page can provide elements.
  - Both adjacent pages (left and right) are under a critical situation.
  - The page merges with the page on the right (if it does not exist, the page is merged with the one on the left).
    - » The resulting page includes the elements of both pages plus the index element that **must be deleted from the upper page**.
    - » The deletion of the index in the upper page **may conduct to a recursive deletion in all the pages of the search path**.
      - » If this process reaches the root, **it will reduce the height of the tree**.



B Trees (Bayer & McCreight)

Deletion

- **Deleting an element from a leaf page**
  - **Case 2b**: the page has n = m keys (*underflow*) and no page can provide elements.
    - Both adjacent pages (left and right) are under a critical situation.
    - The page merges with the page on the right (if it does not exist, the page is merged with the one on the left).
      - » The resulting page includes the elements of both pages plus the index element that **must be deleted from the upper page**.
      - » The deletion of the index in the upper page **may conduct to a recursive deletion in all the pages of the search path**.
        - » If this process reaches the root, **it will reduce the height of the tree**.



B Trees (Bayer & McCreight)

Temporal Complexity Deletion

- **Best Case**
  - Case 1 on a Minimum Height B tree: $O(\log_{2n}(N)) + O(m) = O(\log_{2n}(N))$.

- **Worst Case**
  - Element deleted from a Maximum Height B Tree applying case 2b triggering a page merging process from the leaves to the root
    - $O(\log_{n}(N)) * O(n) = O(\log_{n}(N))$.

CLASSWORK

PLAYGROUND

- **Exercise B Tree (deletion)**. Starting from the B-2 Tree used in the last exercise…
  a) Delete key 11.
  b) Delete key 15.
  c) Delete key 6.
  d) Delete key 16.
  e) Delete key 10.
  f) Delete key 12.
  g) Delete key 28.
  h) Delete key 27.

Priority Queues

Goal

- Model linear structures where their items are managed according to an **associated priority**.
  - Printing queues.
  - Management of Air Traffic Control systems (ATC).
  - Process management in CPUs.
  - Emergency and contingency plans.
  - Waiting queues in Hospitals.

Priority Queues

Problem to Solve

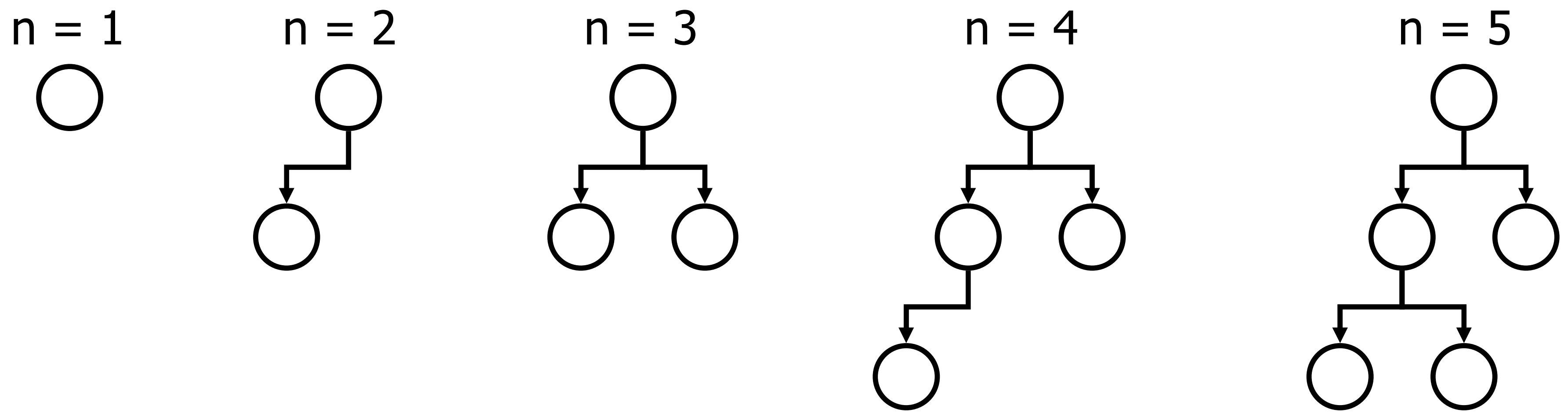
- Optimizing two crucial operations…
  1. Insert item (labeled with a priority level).
  2. Remove the element with the highest priority level.

- Priority queues are frequently implemented using **Binary Heaps**
  - Provides a **$O(\log_2(n))$ complexity** for both operations.
  - Can be implemented using **vectors** (avoiding the use of dynamic memory).

Binary Heaps

What is a Binary Heap?

- It is a complete binary tree (except for the lowest level, which may not be complete).
  - The last level is filled from left to right.



Range for the height of a Binary Heap

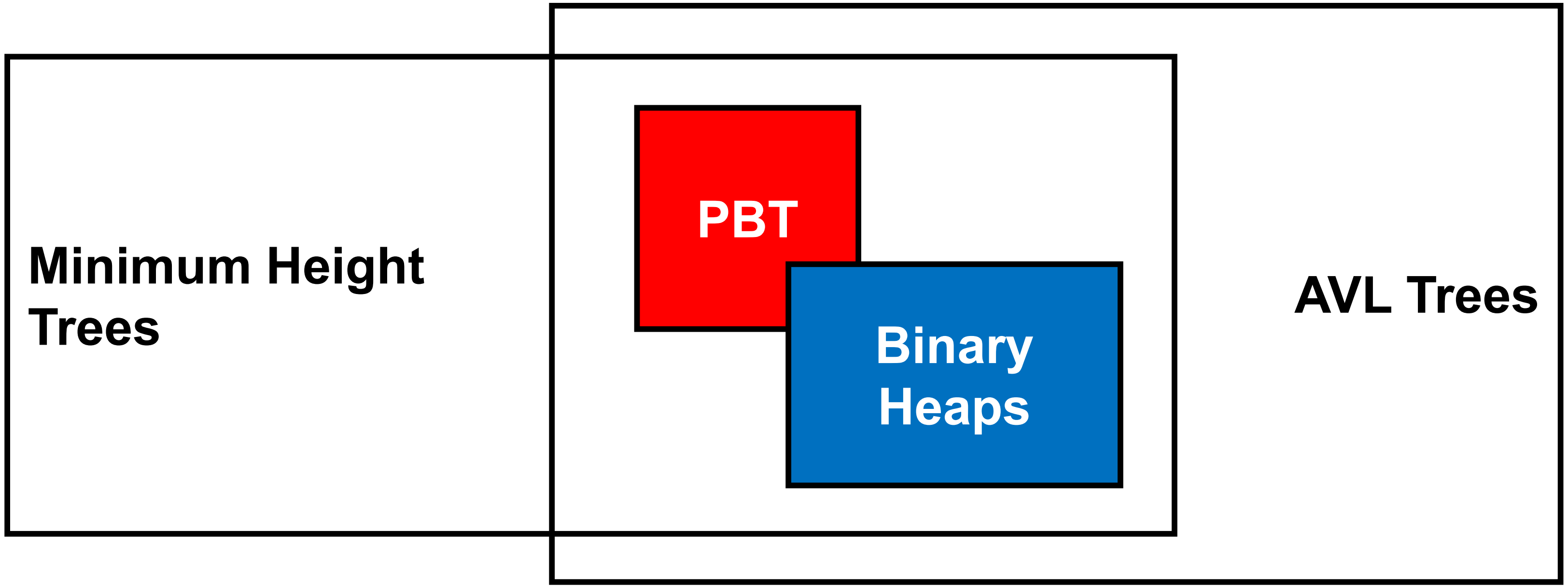
$h = E[\log_2 n] + 1$

$O(h) <= O(\mathrm{Log}_2 n)$

Binary Heaps

Properties

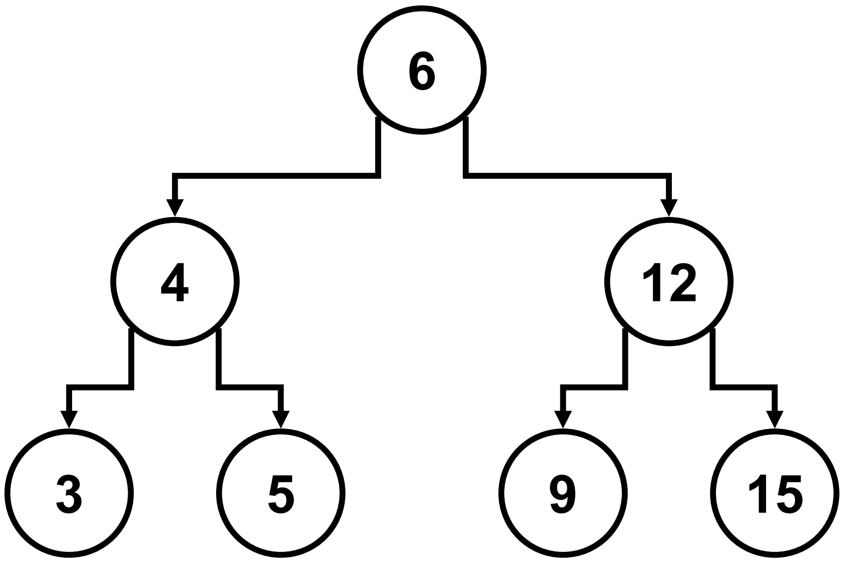
- Any Binary Heap is also a Minimum Height Tree

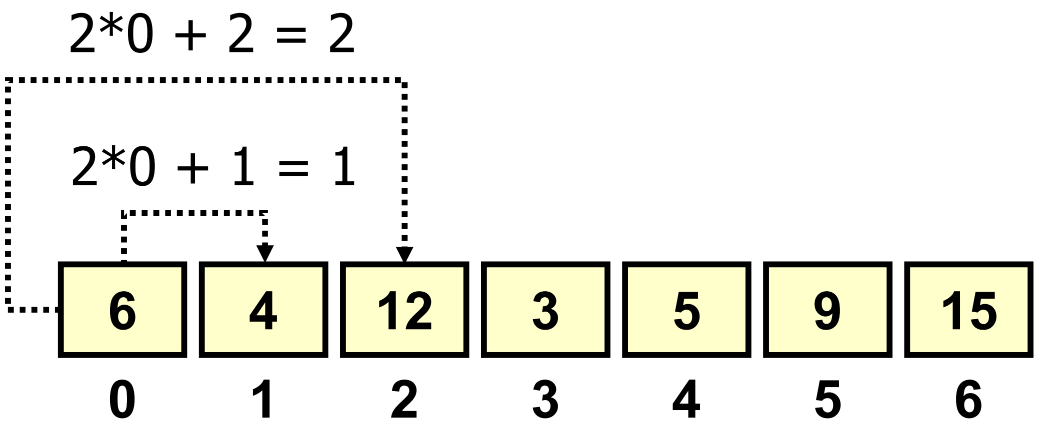


Binary Heaps

Properties

- Due to these constraints, Binary Heaps can be implemented using vectors (does not need dynamic memory)
  - The tree's root is saved in the first slot of the vector.
  - Given a node placed in the ***i*** slot of the vector:
    - Its **left** children will be stored in the slot 2i +1.
    - Its **right** children will be stored in the slot 2i + 2.





Binary Heaps

Heaps are sorted and can not have duplicated items

- **Minimum Heap**
  - Every node has a key **lower** than that of its children.
  - The **item with the lowest key** is placed in the heap's root (slot 0 in the vector).
    - Optimizes the operations ***Add*** *and* ***getMin***.

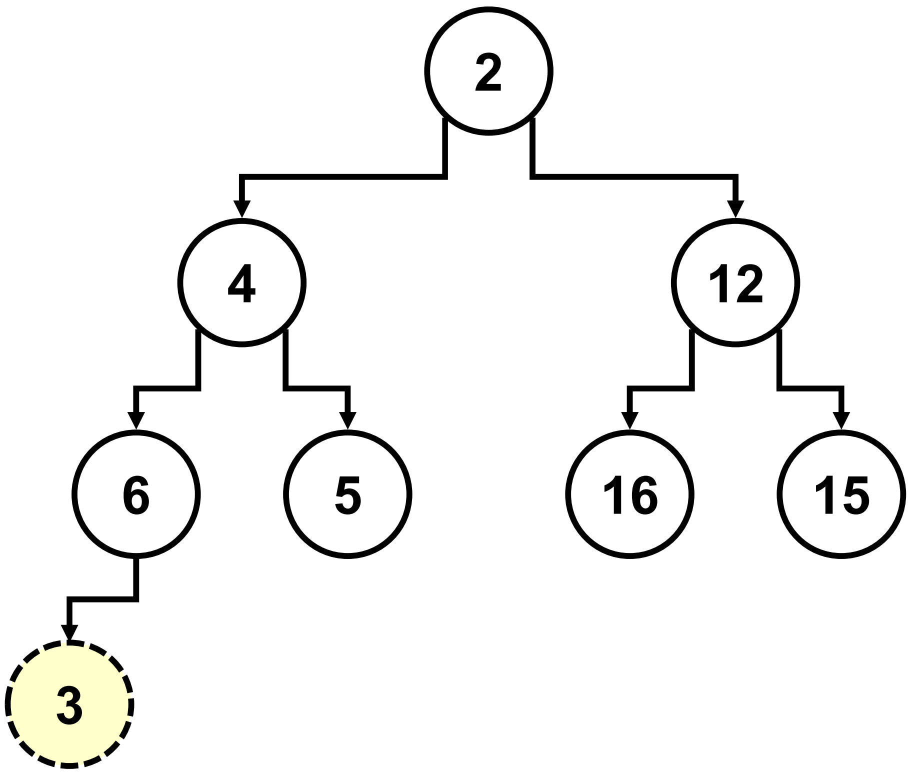
- **Maximum Heap**
  - Every node has a key **greater** than that of its children.
  - The **item with the greater key** is placed in the heap's root (slot 0 in the vector).
    - Optmizes the operations ***Add*** and ***getMax***.

NOTICE: From now on we will work on Minimum Heaps.

Binary Heaps

Insertion (Ascending Filtering)

1. Place the element to be inserted in the last slot of the vector.
2. Repeat until the element reaches the root (slot 0 in the vector) or its key is greater than that of its father.
   - If the item's key is lower than its father's key (placed in the slot E[(i-1)/2]) interchange their positions.



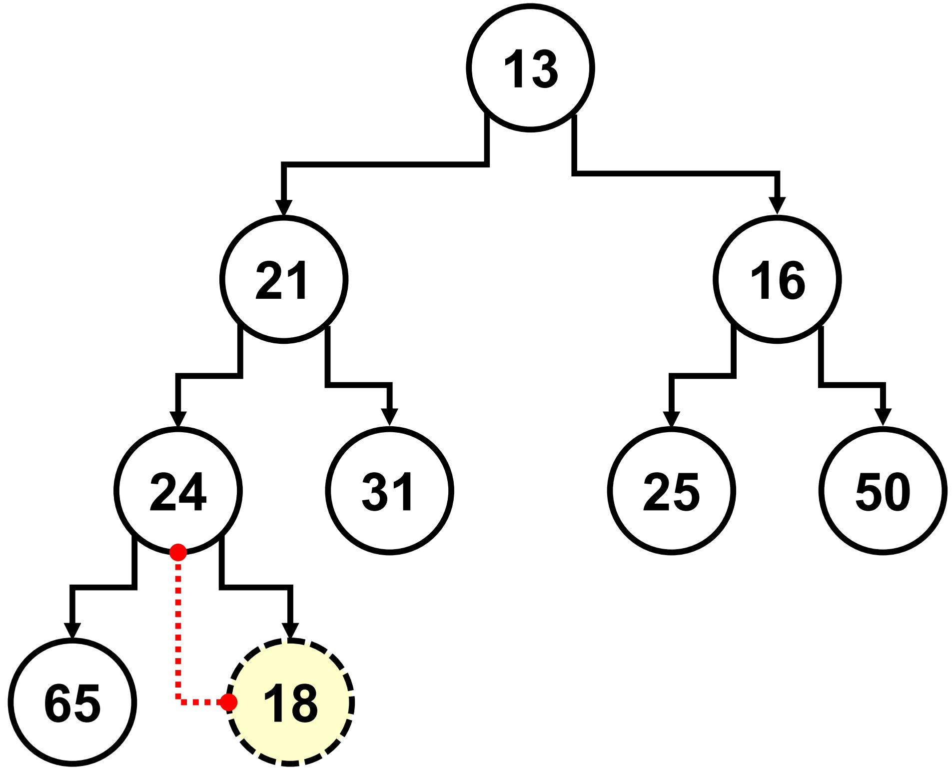
Best case Complexity : O(1)

Worst case Complexity: $O(\log_2 n)$

DISCUSSION: What is the temporal complexity of this algorithm?

Binary Heaps

Exercise



Compare Slot E[(i-1)/2]

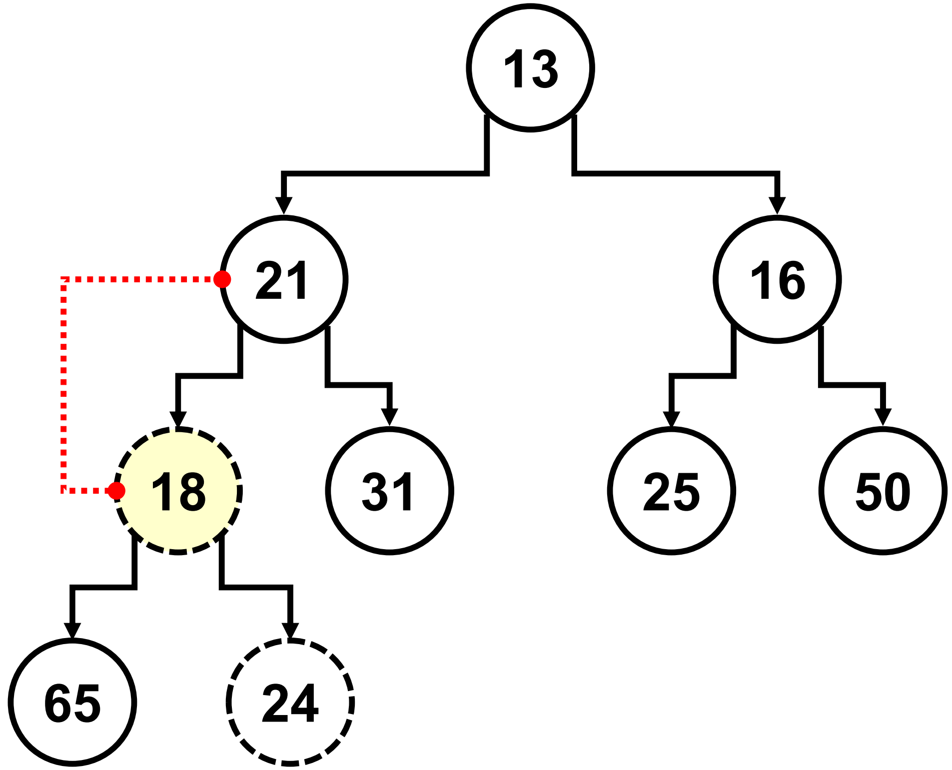
Compare Slot E[(8-1)/2] = E[7/2] = 3

it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18

EVENT: Insert item 18

Binary Heaps

Exercise



Compare Slot E[(i-1)/2]

Compare Slot E[(3-1)/2] = E[2/2] = 1

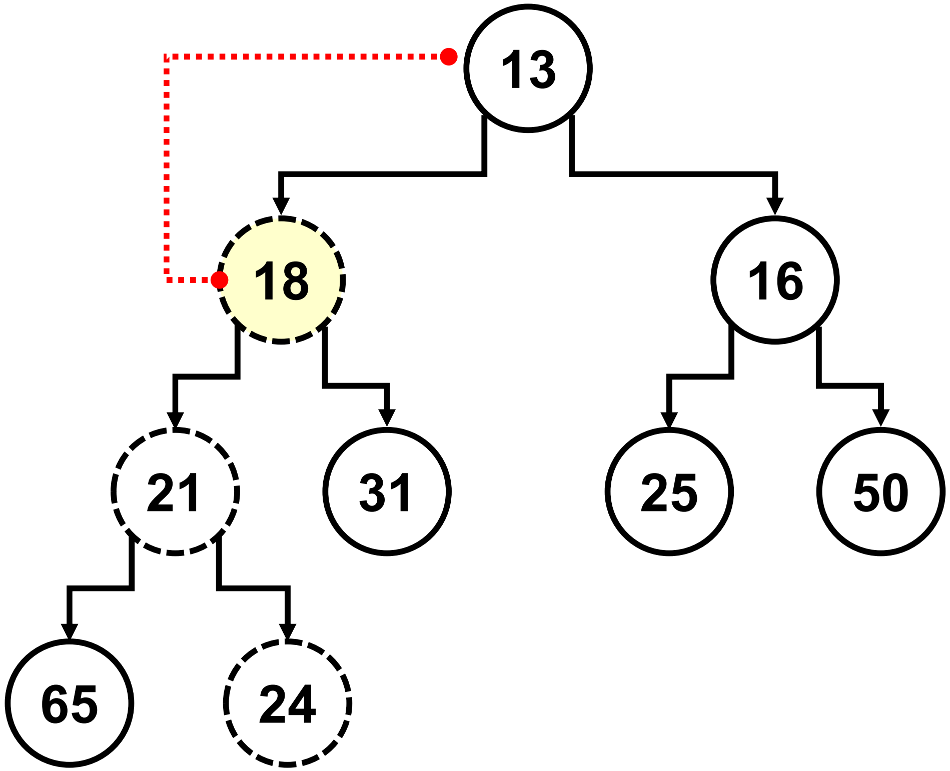
it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18
2	13	21	16	18	31	25	50	65	24

Binary Heaps

Exercise

Compare Slot E[(i-1)/2]

Compare Slot E[(1-1)/2] = E[0/2] = 0



it	0	1	2	3	4	5	6	7	8
1	13	21	16	24	31	25	50	65	18
2	13	21	16	18	31	25	50	65	24
3	13	18	16	21	31	25	50	65	24

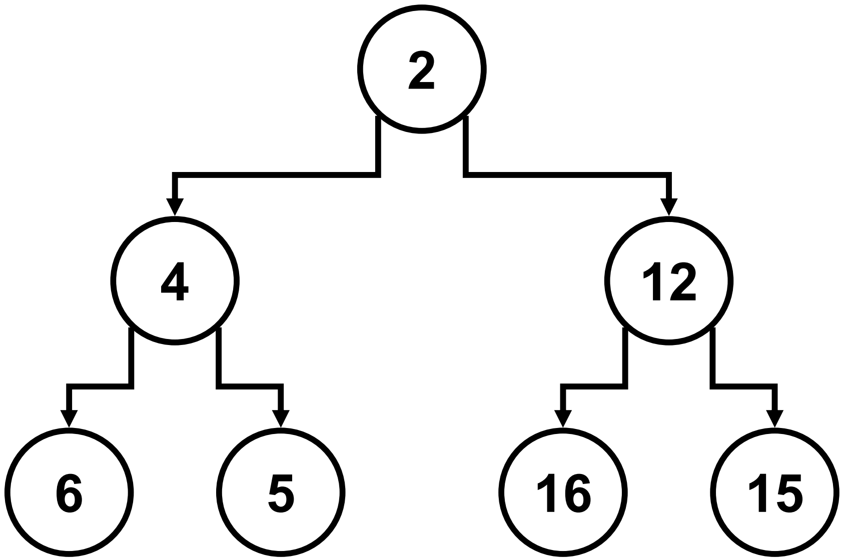
Binary Heaps

Remove (Descending Filtering)

```
1. Return the item placed in the root (minimum key).
2. Place the last item of the vector in the root's
   position applying descending filtering.
3. Repeat until the pivot reaches a leaf or its key
   is lower than that of both of its children.
   •  Interchange the position of the pivot and the
      children owning the lowest key.
```

O(1)

O($\log_2 n$)

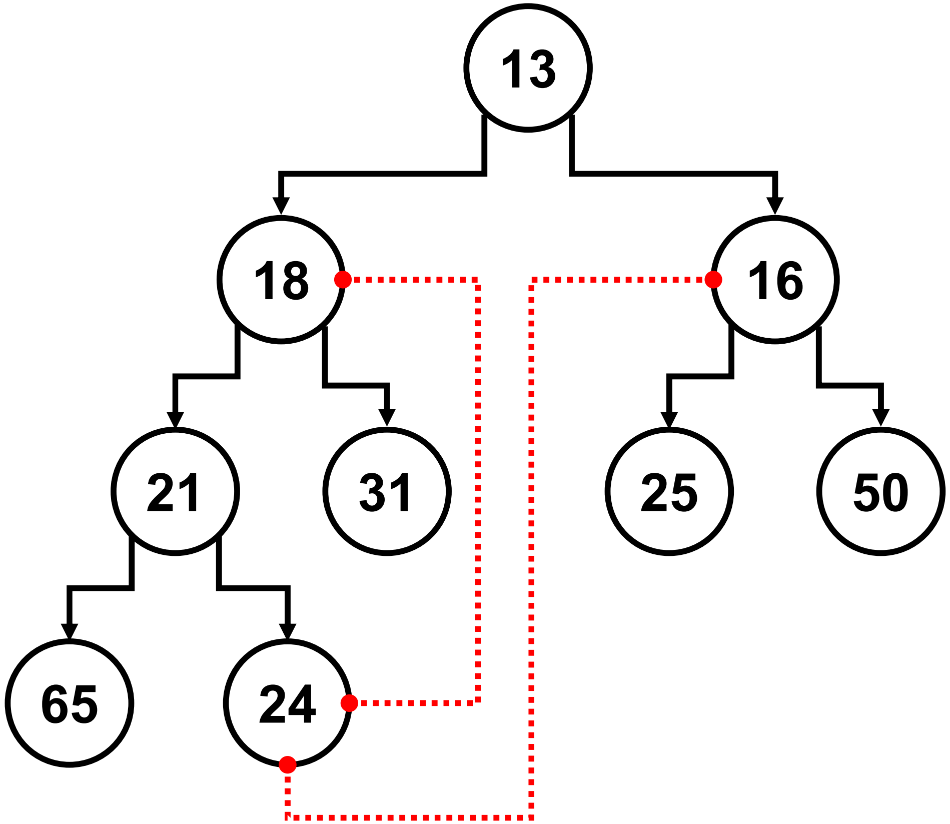


DISCUSSION: What is the temporal complexity of this algorithm?

Binary Heaps

Exercise 1

Compare with Slots 2(0)+1 and 2(0)+2

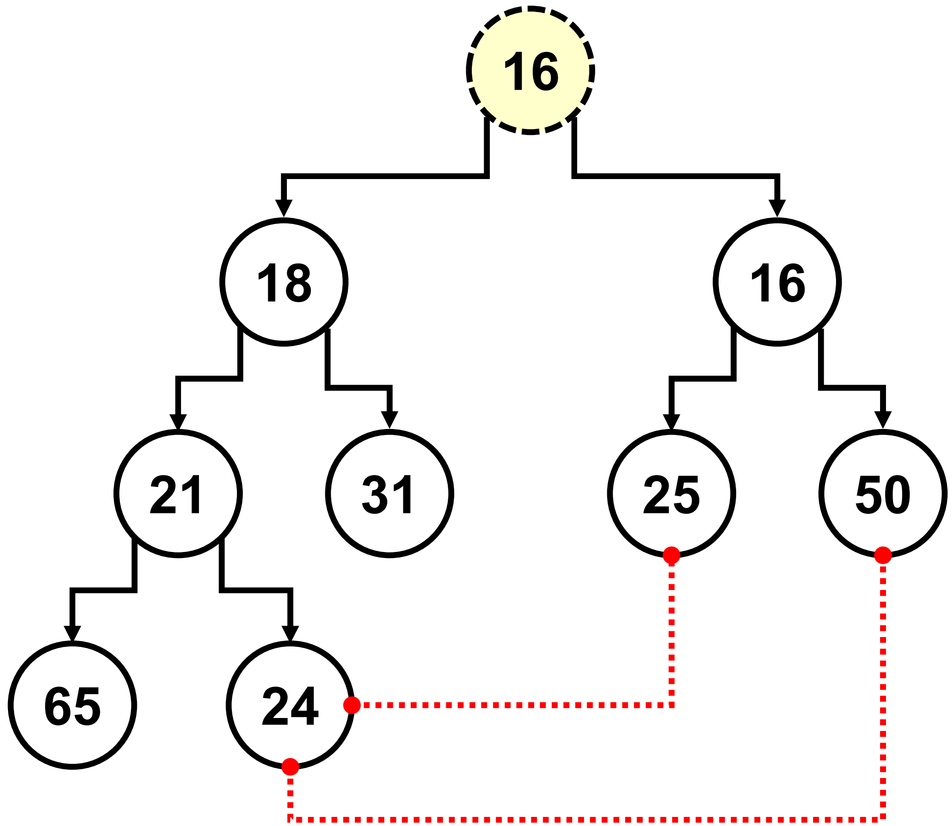


it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24

Binary Heaps

Exercise 1

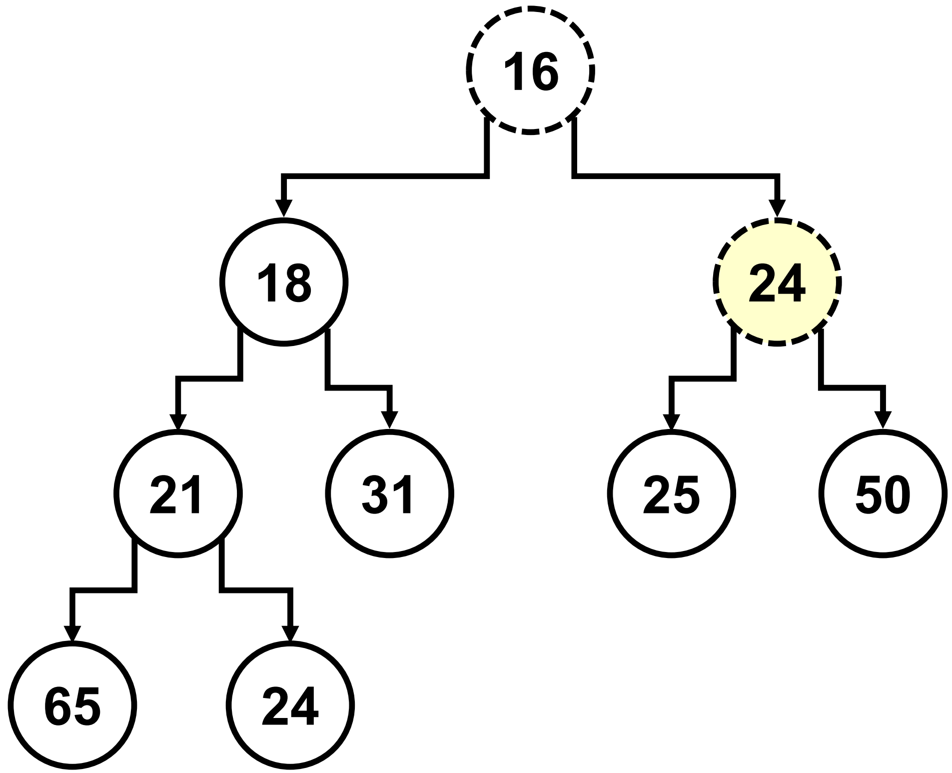
Compare with Slots 2(2)+1 and 2(2)+2



it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24
2	16	18	16	21	31	25	50	55	24

Binary Heaps

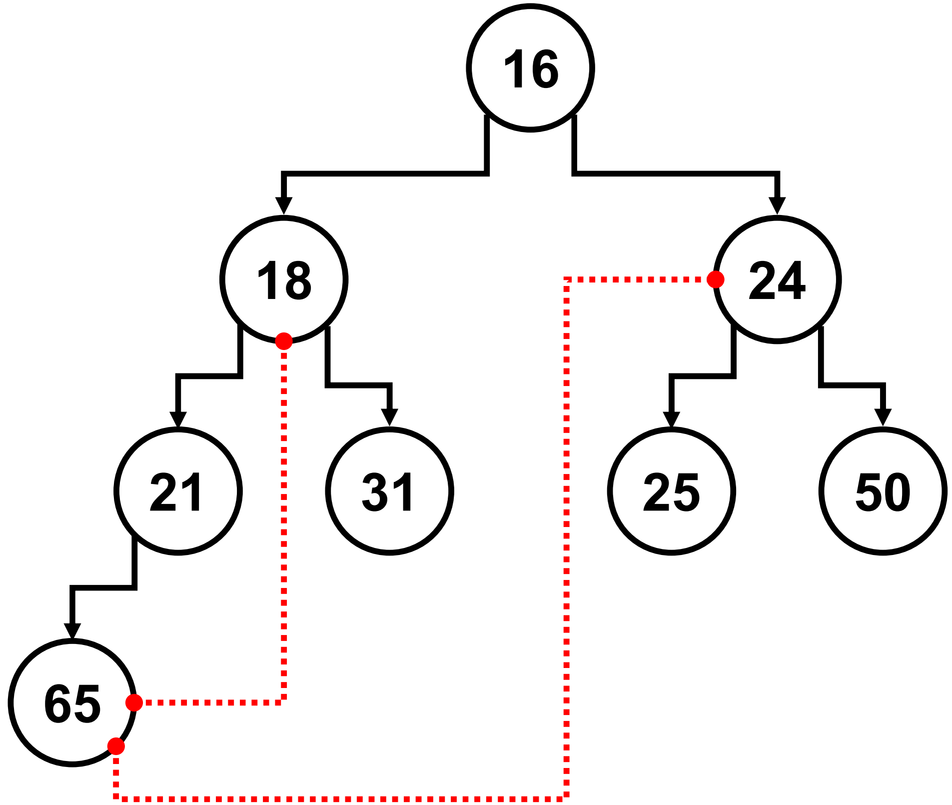
Exercise 1



it	0	1	2	3	4	5	6	7	8
1	13	18	16	21	31	25	50	65	24
2	16	18	16	21	31	25	50	65	24
3	16	18	24	21	31	25	50	65	24

Binary Heaps

Exercise 2

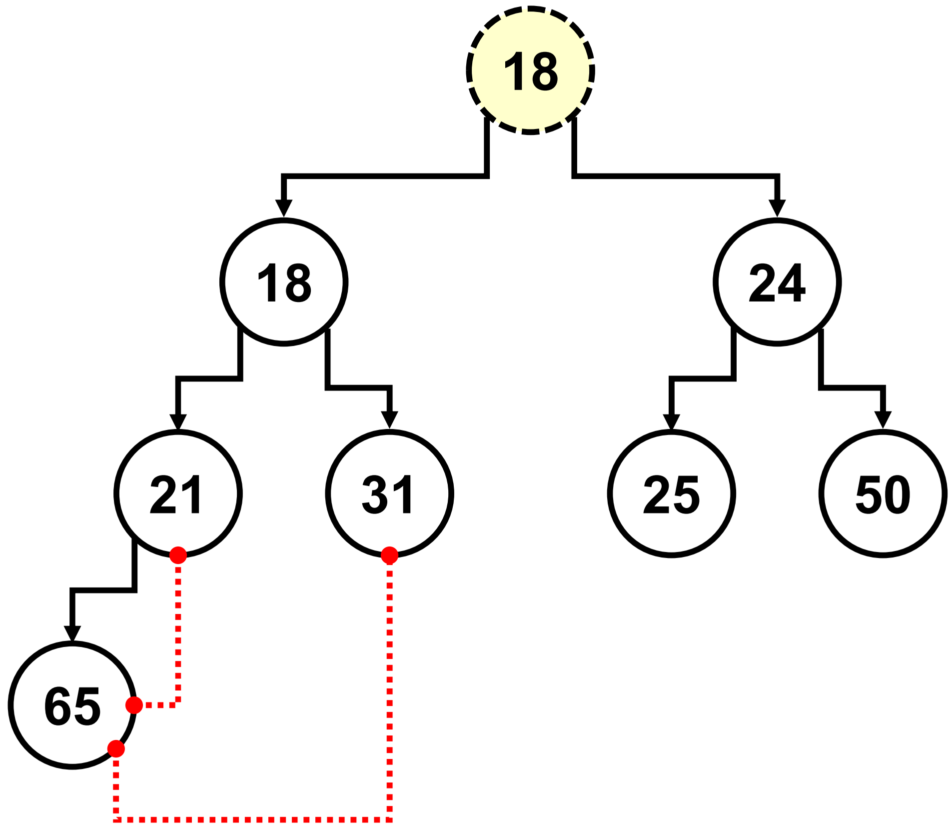


Compare with Slots 2(0)+1 and 2(0)+2

it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	

Binary Heaps

Exercise 2

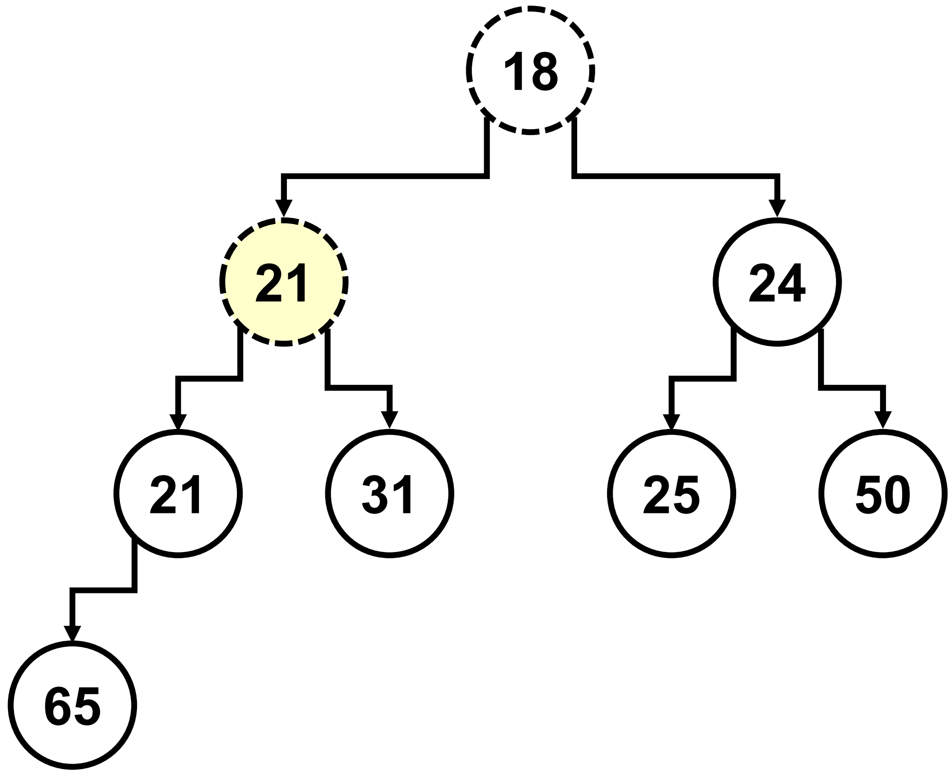


Compare with Slots 2(1)+1 and 2(1)+2

it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	

Binary Heaps

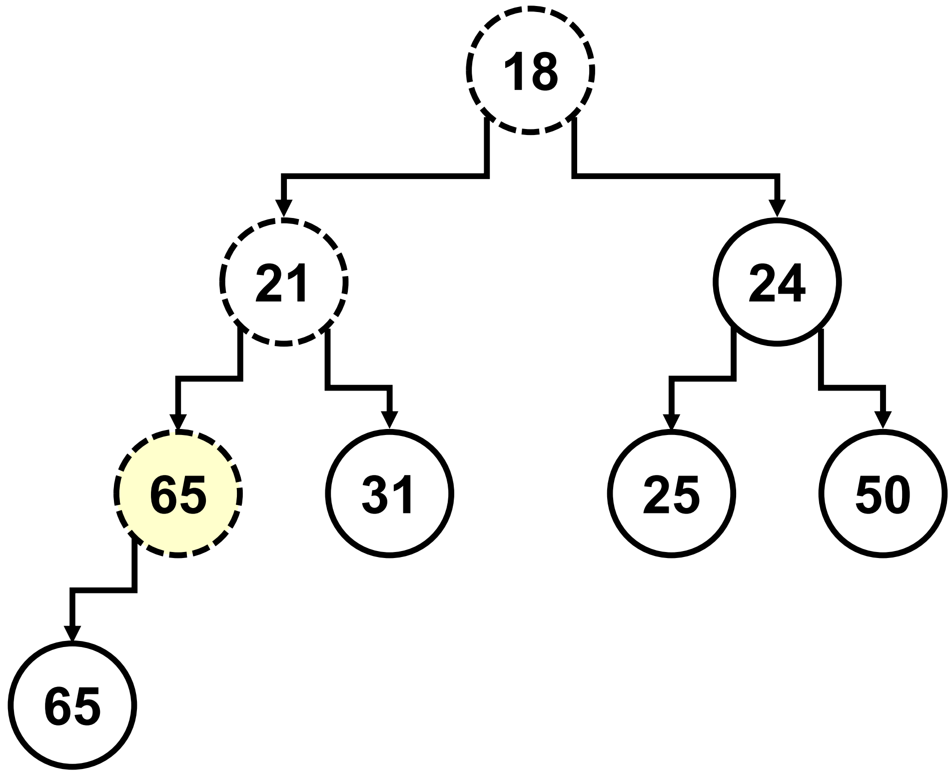
Exercise 2



it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	
3	18	21	24	21	31	25	50	65	

Binary Heaps

Exercise 2



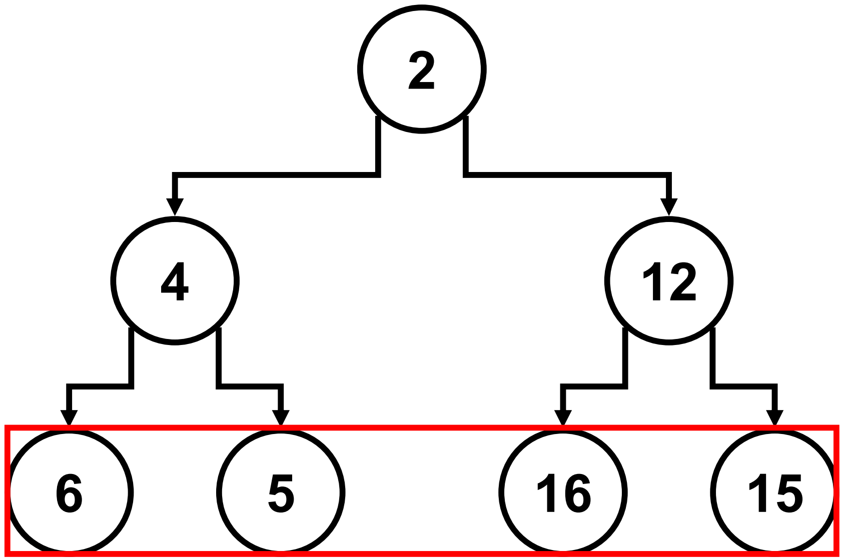
it	0	1	2	3	4	5	6	7	8
1	16	18	24	21	31	25	50	65	
2	18	18	24	21	31	25	50	65	
3	18	21	24	21	31	25	50	65	
4	18	21	24	65	31	25	50	65	

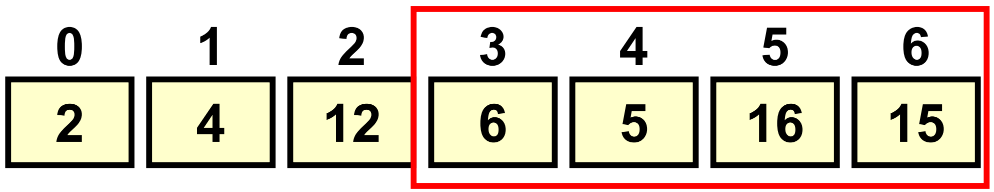
Special Operations using Heaps

Returning the item with the highest key	**O(n)**

```
Sequential search in the vector's area included in
the range:
    [size/2, size].
```

- Items with the greatest values are located in the tree's leaves
    - It is enough to explore only half of the vector.



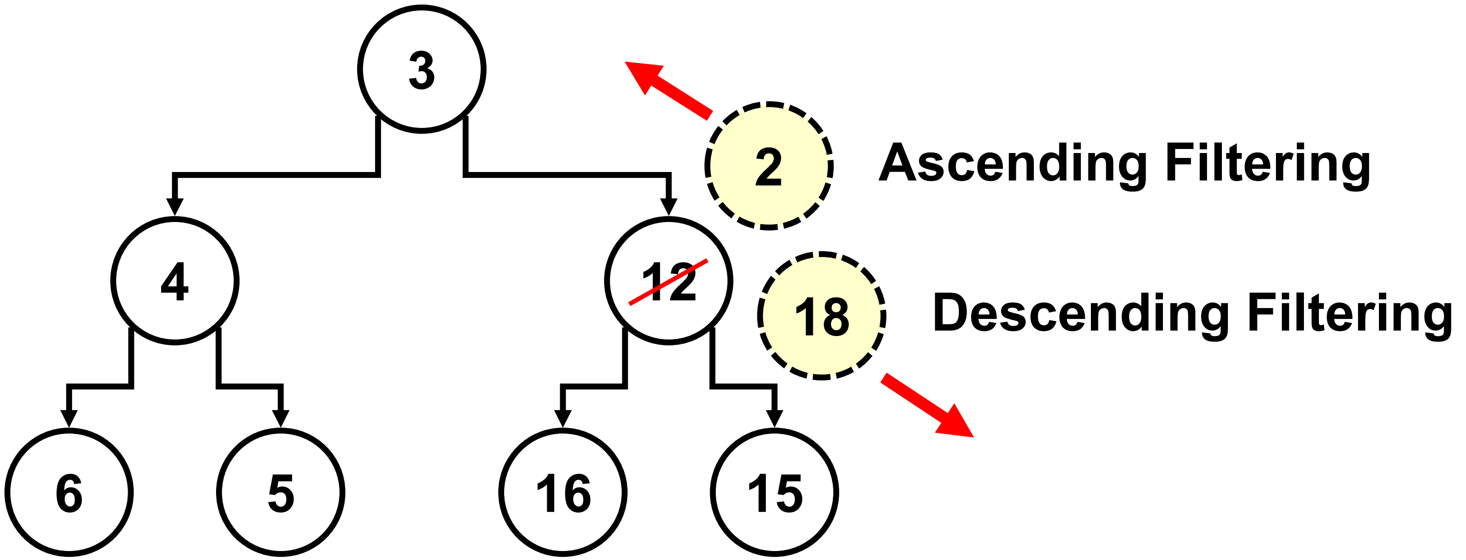


DISCUSSION: What is the temporal complexity of this algorithm?

Special Operations using Heaps

Changing the item's priority	$O(\log_2 n)$

```
1. Access to it and modify its priority.
2. If the new value is lower than the original
     •  Apply ascending filtering
     else
     •  Apply descending filtering.
```



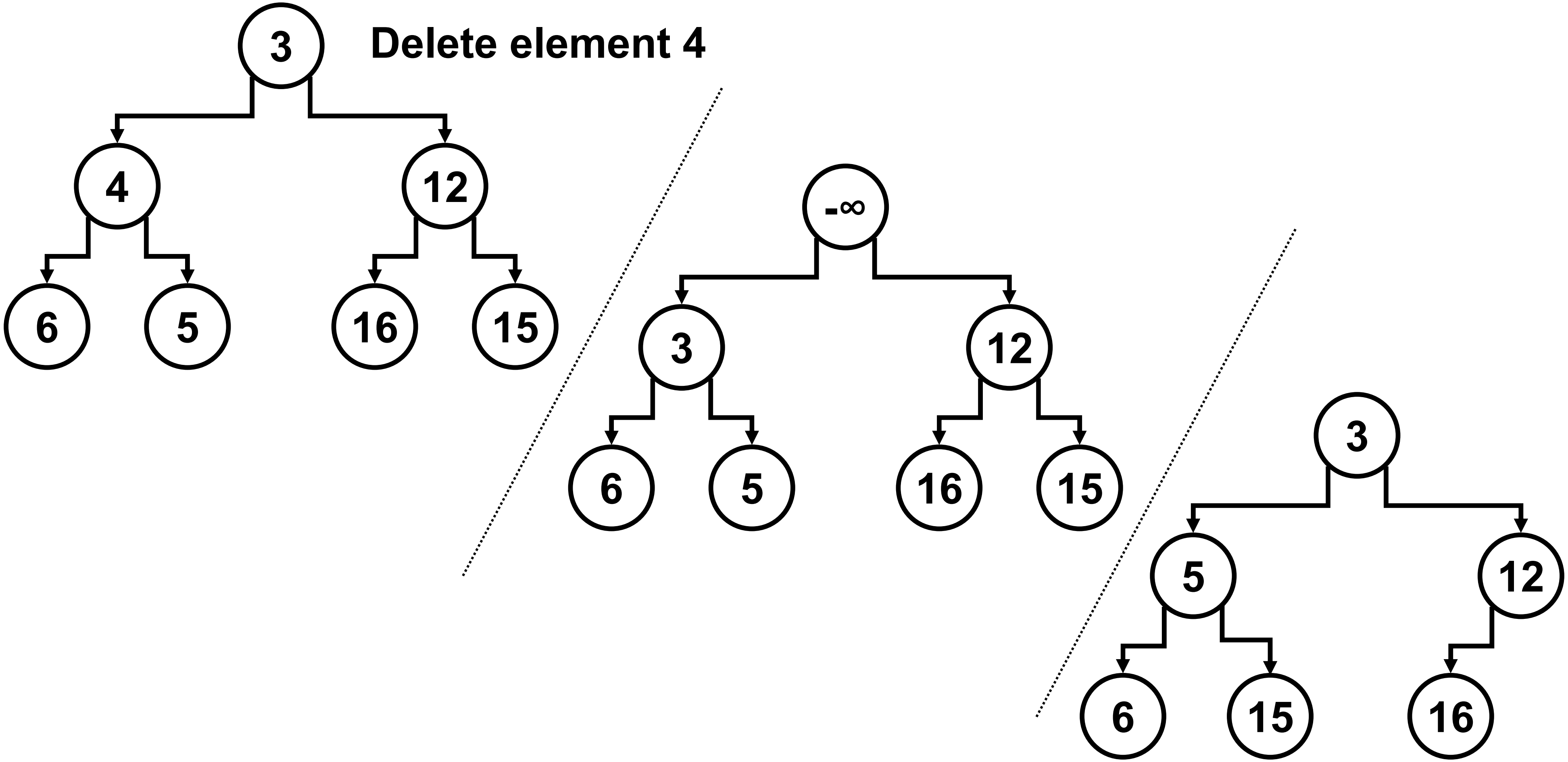
DISCUSSION: What is the temporal complexity of this algorithm?

Special Operations using Heaps

Deletion	$O(\log_2 n)$

```
1. Change the item's priority to -∞ in order to place
   it in the root's position.
2. Invoke the remove() method.
```

Delete element 4



DISCUSSION: What is the temporal complexity of this algorithm?

C60 Series

Dictionary Structures

Dr. Martin Gonzalez-Rodriguez

Dictionary Structures

Goal

- Save unrelated items in such way that it is possible to recover them in fastest possible way.
  - Gets the fastest access speed.
  - Uses huge amounts of memory.
  - Massively used in **cache systems, web catalogs** and **databases**.

Dictionary Structures

Goal

- Reaching a temporal complexity of **O(1)** for access tasks
  - Performance obtained in other operations is sacrificed.

Method	**Complexity**
Insert	O(1)
Search	O(1)
Deletion	O(1)
Print	O(n)
Get Highest	O(n)
Get Lowest	O(n)

Basic Elements

HashTable class

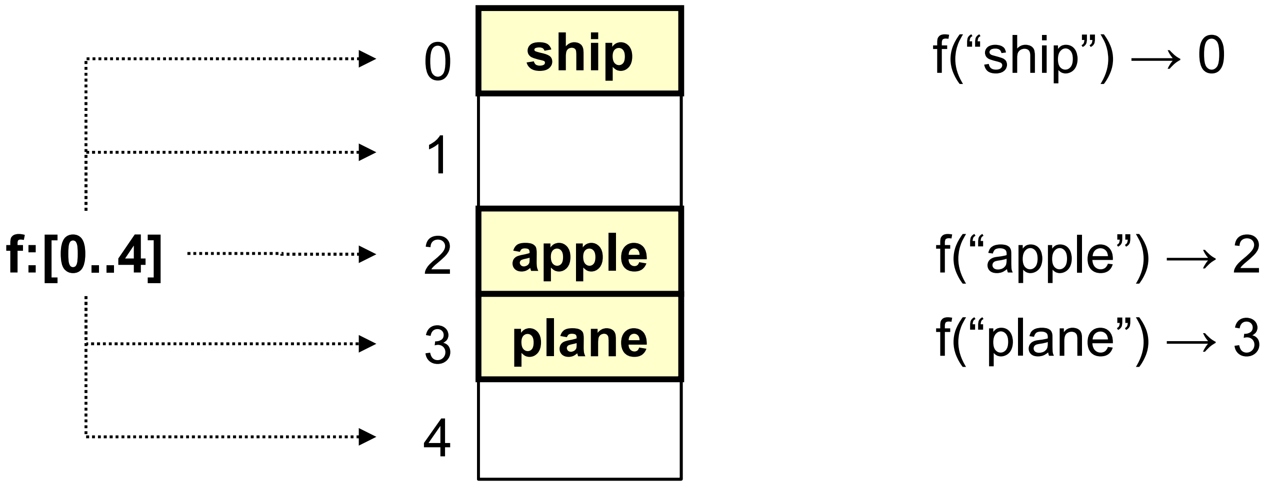
```
class HashTable<T> {
  private ArrayList<HashNode<T>> associativeArray;

  public HashTable(int B) {
        associativeArray = new ArrayList<HashNode<T>>(B);
  }

  private int f (T element){
        return (…);     // converts T to an int value in the range
                        // [0, B-1].
  }
}
```

Hash Function

Transform keys into indexes

- Receives the item's key.
  - Usually a *String* or *int*.

- Returns the slot number (index) where the item should be placed in the *associativeArray*.
  - Range for f: [0, B-1].



Hash Function

Function f for integer keys

```
private int f (T element)
{
  return (element.hashCode() % B); // Module operation
}
```

- Module operation has an excellent performance.
  - The elements are uniformly distributed if dealing with random keys.

$$\boxed{1 \% 4 = 1}$$
$$2 \% 4 = 2$$
$$3 \% 4 = 3$$
$$4 \% 4 = 0$$
$$\boxed{5 \% 4 = 1}$$

Collision

Hash Function

Collisions

- Two elements x and and are synonymous when…
  - f(x) == f (y)
  - **Synonymous** elements create **collisions** over the same vector's slot.

- Collision Management
  - **Active Protection**
    - Avoiding or delaying the collision (designing the perfect hash function).
  - **Passive Protection**
    - Two or more elements share the same vector's slot.
  - **Dynamic resizing**
    - Dynamically increasing (or decreasing) the size of the vector (B) depending on the number of used slots.

Hash Function

Perfect f function
$P(f(X_1)=0) == P(f(X_2)=1) = \ldots == P(f(X_m)=B-1) == 1/B$

- Ensures the lowest number of collisions.
  - If there are **n** elements in the vector, there will be an average of **n/B collisions**.

10 % 10 = $\boxed{0}$
20 % 10 = $\boxed{0}$
30 % 10 = $\boxed{0}$
40 % 10 = $\boxed{0}$
50 % 10 = $\boxed{0}$

10 % 7 = 3
20 % 7 = 6
30 % 7 = 2
40 % 7 = 5
50 % 7 = 1

- B **should be** a prime number!
  - Helps reducing the number of collisions when **there are not random keys**.

Hash Function

HashCode for Strings (Version 1)

```
public int convert (String t){ // <-> t.hashCode()
 int result = 0;

 for (int i=0; i<t.length(); i++)
  result += (int) t.charAt(i);

 return (result);
}

private int f (String element)
{
  return (convert(element) % B);
}
```

- Transforms the String key into an integer value which is used as the parameter of hash function.
  - The *convert* function is based on codes representing each character of the string (adding them).
    - (ASCII code, EBDIC, etc.).

Hash Function

Exercise

- Transform the String "PLANE" assuming that the code for the character A is 65.

Character	Code
P	80
L	76
A	65
N	78
E	69
Total	**368**

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Hash Function

Exercise

- Obtain the range for f assuming…
  - Maximum String size equals to 8 characters.
  - Code range [0, 127].
  - B = 10,007 slots.

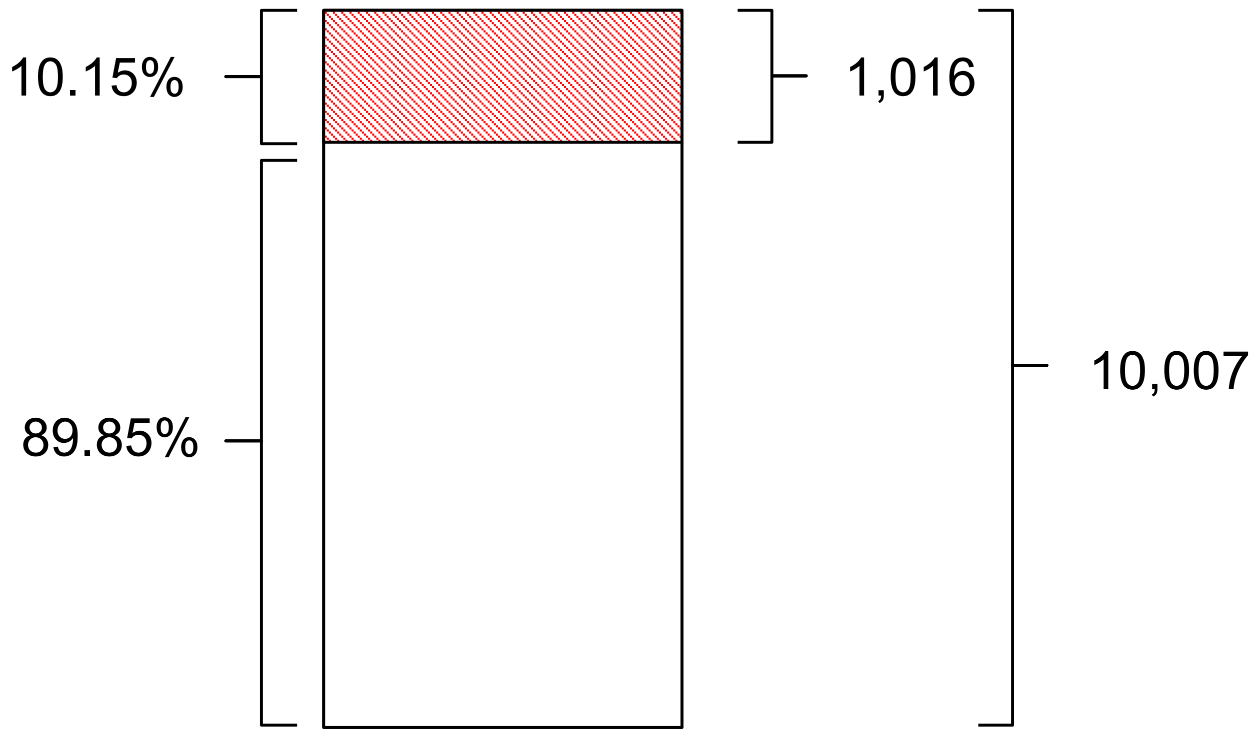
Range *convert (String t)*
`[8*0, 8*127] = [0; 1,016]`

Range *f (String t)*
`[0, 1,016] % 10,007 = [0; 1,016]`

Hash Function

Disadvantages

- If **B is a big number** and the **size of the key is small**, dispersion is concentrated in the upper area of the vector.
  - If the size is small, the sum of the codes will be small too.
  - When the module operator (%) is applied to a small number using a big B parameter, the result will be a really small figure.



Hash Function

HashCode for Strings (Version 2)

```
public int convert (String t){<-> t.hashCode()
 int result = 0;
 int k =(t.length()>3)?3:t.length();

 for (int i=0; i<k; i++)
  result += (int) Math.pow(27, 2-i) * (int) t.charAt(i);

 return (result);
}
```

- Assigns a weight to each character depending upon its position.
  - The weight value (27) is the same as the length of the alphabet.
    - The weight is 27^{2-i} being i the position of the character in the String.
  - It is possible to restrict the number of characters analyzed to a **limit of k** to improve the efficiency.
    - In the example, k <= 3.
    - The multiplication operation consumes much CPU time.

Convert (“PLANE”) = P * 27^2 + L * 27^1 + A * 27^0

Hash Function

Example

- Transform the String "PLANE" assuming that the code for the character A is 65.

Character	Weighted Code	Total
P	$80*27^2$	58,320
L	$76*27^1$	2,052
A	$65*27^0$	65
N	-	-
E	-	-
Total		**60,437**

60,437 % 10,007 = 395

Version 1 of *Convert("PLANE")* obtained 358 (358 % 10,007) = 358

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Hash Function

Disadvantages

- Words starting with the same character combination produce collisions.
  - "**PLA**NE", "**PLA**NING", "**PLA**STIC", etc.

- Assuming a vector size B = 10,007…
  - In **Theory**…
    - For k=3 there are 27*26*25 (17,550) different combinations for the beginning of a word (prior to invoke the *convert* function).
    - Since 17,550 > 10,007, the elements are distributed along the vector.
  - But in **Real Life**…
    - Only 2,851 combinations out of 17,550 makes sense in Spanish language.
      - » For example, there are not words starting with ZYV, ZVW, XYV, etc.
    - Those 2,851 valid words **only represent a 28.4%** of the 10,007 available slots in the vector.

It is necessary to explore all the characters in the String

Hash Function

HashCode for Strings (Version 3)

```
public long convert (String t){<-> t.hashCode()
 long result = 0;

 for (int i=0; i<t.length(); i++)
  result += (int) Math.pow(32, t.length()-i-1) * (int) t.charAt(i);

 return (result);
}
```

- How to optimize the algorithm to analyze the whole String?
  - **Using 32 as the weight**, instead of 27.
    - Multiplying by 32 is equivalent to a **shift of 5 bits at binary level** (shifting is faster than multiplying).
      - » $32 = 2^5$.

$$\text{Convert ("PLANE")} = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

DISCUSSION: How to reduce the number of calls to the Pow() method?

Hash Function

HashCode for Strings (Version 4)

```
public long convert (String t){<-> t.hashCode()
 long result = (int) t.charAt(0);

 for (int i=1; i<t.length(); i++)
  result = (32 * result) + (int) t.charAt(i);

 return (result);
}
```

- Using the **Horner's method**
  - Minimizes the number of multiplications using an alternative representation of the Polynomial.

$$Convert(\text{“PLANE”}) = P * 32^4 + L * 32^3 + A * 32^2 + N * 32^1 + E * 32^0$$

$$Convert_{Horner}(\text{“PLANE”}) = ((((P * 32) + L) * 32) + A) * 32) + N) * 32 + E$$

Hash Function

HashCode for Strings (Version 5)

```
public long convert (String t) {<-> t.hashCode()
 long result = (int) t.charAt(0);

 for (int i=1; i<t.length(); i++)
  result = ((32 * result) + (int) t.charAt(i)) % B;

 return (result);
}
```

- Avoiding *Overflows*
  - Calculation generates **such large numbers that can not be stored**.
  - The **module operator (%)** must be applied in every iteration in order to reduce the size of the figures.
    - Overflow is avoided at the cost of a temporary penalty.

f("PLANE") =

(((((P * 32) + L) % B * 32) + A) % B * 32) + N) % B * 32 + E) % B

Hash Function

Transform

- Transform the String “PLANE” assuming that the code for the character A is 65.

Character	Weighted Code	Total
P	$80*32^4$	83,886,080
L	$76*32^3$	2,490,368
A	$65*32^2$	66,560
N	$78*32^1$	2,496
E	$69*32^0$	69
Total		**86,445,573**

86,445,573 % 10,007 = 5,107

Versión 2 of *Convert(“PLANE”)* obtained 60,437 % 10,007 = 395

65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Passive Protection

Collisions are inevitable in the long term...

- **The smaller** the B **the greater** the probability of collision.
  - Certainty is achieved when…
    - B = 1.
    - Problem domains requires the use of duplicated keys.

- When two or more elements share the same vector slot…
  - There are several strategies to deal with collisions.

- Separate Chaining
- Open Addresing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Separate Chaining

Separate Chaining

- Each slot contains a dynamic data structure that stores the synonyms.
  - LinkedList.
  - AVLTree.

HashTable class	O(B) = O(1)

```
ppublic class HashTable<T>
{
 private int B = 10007;
 rivate ArrayList<AVLTree<T>> associativeArray;

 public HashTable(int B) {
  this.B = B;
  associativeArray = new ArrayList<AVLTree<T>>(B);

  for (int i=0; i<associativeArray.size(); i++)
   associativeArray.add(new AVLTree<T>());
 }
}
```

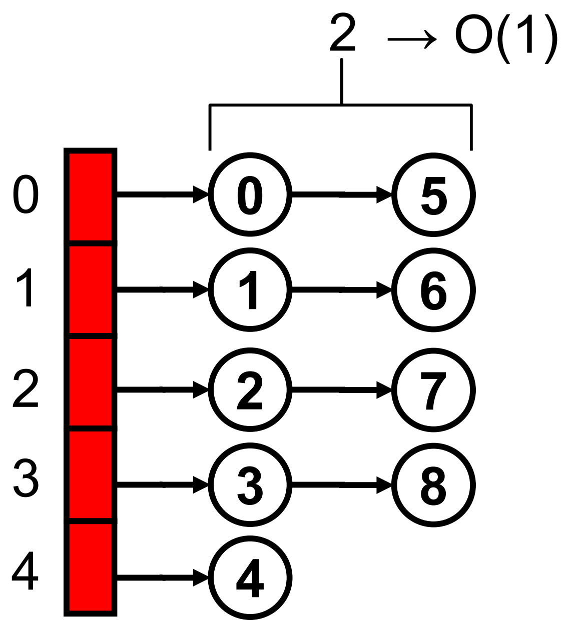
DISCUSSION: What is the temporal complexity of this constructor?

Separate Chaining

add	O(n/B) → O(1)

```
public void add (T a){
 if (!find(a))
  associativeArray.get(f(a.hashCode())).add(a);
}
```

find() and remove() behave in a similar way



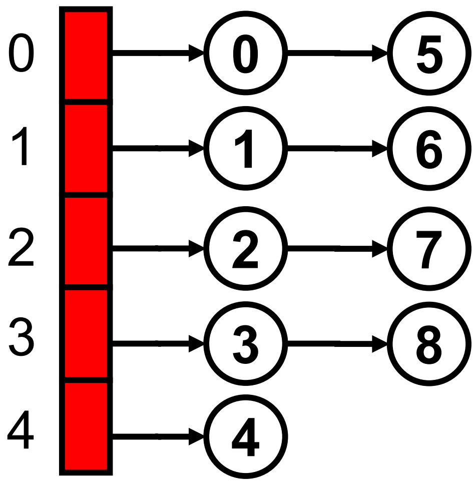

```
For (int i=0; i< 9; i++)
 table.add(new Integer(i));
```

DISCUSSION: What is the temporal complexity of this algorithm?

Separate Chaining

Load Factor (LF)

- It is calculated as the number of elements in the hash table divided by its size.
  - LF = n/B.
  - Represents the average size of each linked list.



LF = 9/5 = 1.8

Separate Chaining

An efficient LF

Search taks	Average of visited links
Unsuccessful	LF
Successful	1 + LF/2

- Ensuring a good performance in Hash Tables based on Separate Chaining requires **LF smaller or equal than one (LF <= 1)**
  - B = n (approximately).
  - Average size of the linked lists = 1.

Open Addressing

Open Addressing

- Each slot can contain only one item.
  - Whenever a collision is detected, the algorithm looks for an empty slot in the surrounding slots.
  - There are several different approaches to explore the vicinity.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.

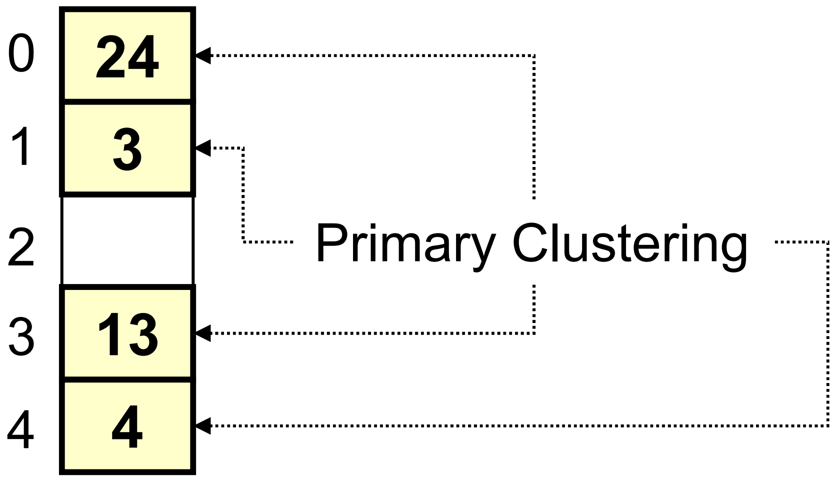
HashTable class

```
public class HashTable <T>
{
  private final static int B = 10007;
  private ArrayList<HashNode<T>> associativeArray;
}
```

Open Addressing

Linear Probing

- Consecutive search on the neighboring slots modifying the f function.
  - $f(x) = [x + i] \% B$.
    - Where i represents the number of attempts used to find an empty slot. It assumes the following values 0, 1, 2, 3…



add(4) → [4 + 0] % 5 = 4

add(13) → [13 + 0] % 5 = 3

add(24) → [24 + 0] % 5 = 4

add(24) → [24 + 1] % 5 = 0

add(3) → [3 + 0] % 5 = 3

add(3) → [3 + 1] % 5 = 4

add(3) → [3 + 2] % 5 = 0

add(3) → [3 + 3] % 5 = 1

Open Addressing

Clustering

- Set of interrelated occupied slots.
  - Clustering can be produced even on relatively empty hash tables.
  - Any key **distributed over a clustering area** requires several attempts to find its position in the vector.
    - And what it is worst… when the item is finally added, **it will join the clustering, which becomes larger and larger**.

- If the table is large enough, there will exist an empty slot for the element…
  - …but finding it will require much time!

Search	Approximate required attempt number
Unsuccessful	$(1 + 1/(1 - LF)^2)/2$
Successful	$(1 + 1/(1 - LF))/2$

Open Addresing

Theoretical studies about performance

LF	Attempts per insertion (average)
0.90	50
0.75	8.5
0.50	2.5

- It is recommended to use a LF <= 0.5
  - B should be **at least two times n**.
  - The increment is the use of extra memory is remarkable.

Recommendation for Separate Chaining: LF <= 1

Open Addresing

Lazy Deletion

- Clustering prevents simple deletion.
  - The element is **marked for deletion** but it is not deleted until its slot is selected to insert new items.
  - Marked elements are considered **empty during insertions but occupied during search tasks**.

0	**24**
1	**3**
2	
3	**13**
4	**4**

delete(24) → [24 + 0] % 5 = 4

delete(24) → [24 + 1] % 5 = 0

find(3) → [3 + 0] % 5 = 3

find(3) → [3 + 1] % 5 = 4

find(3) → [3 + 2] % 5 = 0

Access to the key 3 is lost!

Open Addresing

Lazy Deletion

HashTable class

```
public class HashNode <T>
{
    public final static byte EMPTY   = 0;
    public final static byte VALID   = 1;
    public final static byte DELETED = 2;

    private T element;
    private byte status = EMPTY;
}
```

Open Addressing

Lazy Deletion

HashTable class

```
public class HashTable<T>
{
  private final static int B = 10007;
  private ArrayList<HashNode<T>> associativeArray;
}
```

Before

0	**24**	VALID / DELETED
1	**3**	VALID
2		EMPTY
3	**13**	VALID
4	**4**	VALID

delete(24) → [24 + 0] % 5 = 4

delete(24) → [24 + 1] % 5 = 0

find(3) → [3 + 0] % 5 = 3

find(3) → [3 + 1] % 5 = 4

find(3) → [3 + 2] % 5 = 0

find(3) → [3 + 3] % 5 = 1

add(15) → [15 + 0] % 5 = 0

After

0	**15**	VALID
1	**3**	VALID
2		EMPTY
3	**13**	VALID
4	**4**	VALID

Open Addressing

Quadratic Probing

- When a collision is detected, the algorithm looks for an empty slot located at a **quadratic distance** from the first slot.
  - $f(x) = [x + i^2] \% B$.
    - Where i represents the attempt number. It assumes values of 0, 1, 2, 3…

0	**24**
1	
2	**3**
3	**13**
4	**4**

add(4) $\rightarrow [4 + 0^2] \% 5 = 4$

add(13) $\rightarrow [13 + 0^2] \% 5 = 3$

add(24) $\rightarrow [24 + 0^2] \% 5 = 4$

add(24) $\rightarrow [24 + 1^2] \% 5 = 0$

add(3) $\rightarrow [3 + 0^2] \% 5 = 3$

add(3) $\rightarrow [3 + 1^2] \% 5 = 4$

add(3) $\rightarrow [3 + 2^2] \% 5 = 2$

Open Addressing

Quadratic Probing

- Since the distance between verified slots (quadratic distance) is really big, **it is possible not to find an empty slot at all.**
  - Even though there may be free slots, exploring probing **can jump over them ignoring them!**

Quadrating Probing Theorem
`If using quadratic probing it holds that` **`B is a prime number`** `and` **`LF <= 0.5`**`, it is always possible to find a position to insert an item.`

- Quadratic Probing eliminates primary clustering…
  - … but it can produce secondary clustering.

- However secondary clustering may be acceptable…
  - Simulation studies show that in order to avoid secondary clustering **only one jump is needed** to find free slots.

Open Addressing

Double Hashing

- Uses two hashing functions.
  - $f(x) = [x + i*H_2(x)] \% B$.
    - Where i represents the attempt number. It assumes values of 0, 1, 2, 3…
    - Where H_2 is the jumping function. It can be anyone. The next one is frequently used for this purpose:
      - » $H_2(x) = R – x \% R$.
        - » Where R is the prime number predecessor of B.

0	
1	**3**
2	**24**
3	**13**
4	**4**

add(4) → [4 + 0*(3 – 4%3)] % 5 = 4

add(13) → [13 + 0*(3 – 13%3)] % 5 = 3

add(24) → [24 + 0*(3 – 24%3)] % 5 = 4

add(24) → [24 + 1*(3 – 24%3)] % 5 = 2

add(3) → [3 + 0*(3 – 3%3)] % 5 = 3

add(3) → [3 + 1*(3 – 3%3)] % 5 = 1

Solution: slots 4, 1, 3 and 0

TEST: Which slots are checked to insert item 19?

Open Addressing

Evaluation of Double Hashing

- Pros
  - Avoids clustering.
  - The number of attempts is really small.

- Cons
  - The use of a second mathematical function reduces the performance.

Dynamic Resizing

Dynamically changes the size of the hash table

- When LF increases to much…
  - The performance in the hash table drops down remarkably.
    - LF > 1 when using Separate Chaining.
  - Open Addressing stops as it may be impossible to find empty slots.
    - LF > 0.5 is the limit when using Open Addressing.

- Dynamic resizing recovers an acceptable LF as it moves the items to a bigger hash table.
  - The new B is designated as **the prime number immediately over the double of the original B parameter**.
  - All the elements in the old table are sequentially moved to the new one.

Dynamic Resizing

Exercise

- Execute Dynamic Resizing using Quadratic Probing

Prime number immediately over the double of 5 is 11

0	**24**
1	
2	**3**
3	**13**
4	**4**

add(24) $\rightarrow$ $[24 + 0^2]$ % 11 = 2

add(3) $\rightarrow$ $[3 + 0^2]$ % 11 = 3

add(13) $\rightarrow$ $[13 + 0^2]$ % 11 = 2

add(13) $\rightarrow$ $[13 + 1^2]$ % 11 = 3

add(13) $\rightarrow$ $[13 + 2^2]$ % 11 = 6

add(4) $\rightarrow$ $[4 + 0^2]$ % 11 = 4

O(n)

0	
1	
2	**24**
3	**3**
4	**4**
5	
6	**13**
7	
8	
9	
10	

DISCUSSION: What is the temporal complexity of this algorithm?

Dynamic Resizing

Triggering Dynamic Resizing

- Dynamic resizing may be triggered automatically whenever…
  a) Reaching a LF > 0.5.
  b) An insertion fails (there are not empty slots).
  c) When it exceeds a certain threshold defined in the constructor of the hash table.

- Inverse Resizing
  - Reduces the size of the hash table to save up memory when there have been many delete operations.

Type of Table	LF's threshold for Inverse Double Hashing
Separate Chaining	0.33
Open Addresing	0.16

Appendix A

To Know More

To Know More: Graphs

PLAYGROUND

- Visit the entry for the **Dijkstra Algorithm** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay attention on how the use of **Priority queues** can reduce the temporal complexity of this algorithm.
    - The *Priority Queue* data structure will be later studied in the Hierarchical Structures section.

To Know More: Graphs

PLAYGROUND

- Visit the entry for the **Floyd-Warshall Algorithm** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Find out what how the path reconstruction is done by this algorithm.
    - Pay attention to how the **Negative Cycles** are managed and how they can be detected by the algorithm.

To Know More: Graphs

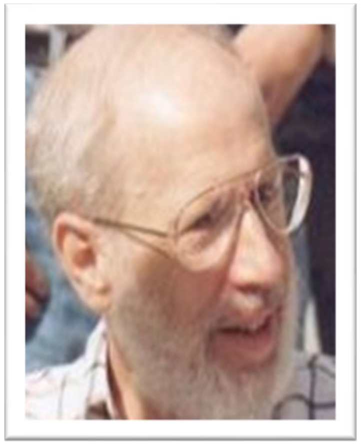
PLAYGROUND

- Visit the entry for the **Prim's Algorithm** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay special attention to the algorithm proof of correctness.

To Know More: Graphs

PLAYGROUND

- The problem of the minimum spanning tree was solved by the American researcher Joseph Kruskal too.
- Visit the entry for the **Kruskal's Algorithm** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay attention the differences between the Kruskal's Algorithm and the Prim's Algorithm.



Joseph Kruskal (Wikipedia)

To Know More: Trees

PLAYGROUND

- Visit the entry for the **Binary Search Tree** in the Wikipedia
  - **Carefully** read all the content for this entry.
  - Pay attention to the concept of **Optimal Binary Search Tree (OBST).**

To Know More: Trees

PLAYGROUND

- Visit the entry for the **AVL Tree** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay special attention to the comparison between AVL and red-black trees.

To Know More: Trees

PLAYGROUND

- Visit the entry for the **B Tree** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay attention to how this kind of trees can be used to provide concurrent access to the data.

To Know More: Binary Heaps

PLAYGROUND

- Visit the entry for the **Binary Heap** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay attention to how an amortized analysis demonstrates tha insertions may have a O(log n) complexity, while the delete operation may have O(1).

To Know More: Hash Tables

PLAYGROUND

- Visit the entry for the **Hash Table** in the Wikipedia
  - **Carefully** read all the content for this entry.
    - Pay special attention to how alternative hashing policies like **Robin Hood hashing** or **Cuckoo hashing** work.

Appendix B

References

Graph Theory

AHO, A; HOPCROFT, J; ULLMAN, D; (1988) *Estructuras de Datos and Algoritmos*. Addison-Wesley Iberoamericana. México [Cap 9].

JOYANES AGUILAR, Luis; ZAHONERO MARTÍNEZ, Ignacio; (1998) *Estructura de Datos: Algoritmos, Abstracción and Objetos*. Mc Graw Hill. ISBN: 84-481-2042-6. [Cap 14.]

ORTEGA F., Maruja; (1988) *Grafos and Algoritmos*. Universidad Metropolitana, Oficina Metrópolis.

WEISS, Mark Allen; (2000*) Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN 84-7829-035-4. [Cap 14.].

WEISS, Mark Allen; (1995) *Estructuras de Datos and Algoritmos* Addison-Wesley Iberoamericana. ISBN 0-201-62571-7. [Cap 9.].

Hierarchical Data Structures

HERNÁNDEZ, Roberto; LÁZARO, Juan Carlos; DORMIDO; Raquel, ROS, Salvador; (2001) *Estructuras de Datos and Algoritmos*. Prentice Hall. ISBN 84-205-2980-X [Cap. 5 and 6].

JOYANES AGUILAR, Luis; ZAHONERO MARTÍNEZ, Ignacio; (1998) *Estructura de Datos: Algoritmos, Abstracción and Objetos*. Mc Graw Hill. ISBN: 84-481-2042-6 [Cap. 10, 11 and 12].

ORTEGA F., Maruja; (1988) *Grafos and Algoritmos*. Universidad Metropolitana, Oficina Metrópolis.

WEISS, Mark Allen; (2000) *Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN84-7829-035-4.

WEISS, Mark Allen; (1995) *Estructuras de Datos and Algoritmos* Addison-Wesley Iberoamericana. ISBN 0-201-62571-7.

Hash Tables

BRASSARD G.; BRATLEY, P.; (1997) *Fundamentos de Algoritmia*. Prentice Hall. ISBN: 84-89660-00-X. [Cap. 5].

COLLADO M., MORALES R. and MORENO J. (1987) Estructuras de datos. Realización en Pascal. Ed. Díaz de Santos, 1987.

WEISS, Mark Allen; (2000) *Estructuras de Datos En Java 2*. Addison-Wesley Iberoamericana. ISBN84-7829-035-4. [Cap. 19].

WEISS, Mark Allen (1995) *Data Structures and Algorithm Analysis*. Addison-Wesley Iberoamericana. [Cap. 5].

Exercises

Martin Gonzalez-Rodriguez, Ph. D.

Unit 1

Algorithmics and Design

Recursion

Recursion

E1. Execute the next recursive function (factorial) for f(5):

- Factorial
  - F(0!) = 1
  - F(n!) = n(n-1)!

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	
2	2==0?	2*f(1)	

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	
2	2==0?	2*f(1)	
1	1==0?	1*f(0)	

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	
2	2==0?	2*f(1)	
1	1==0?	1*f(0)	
0	0==0?	1*1	1

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	
2	2==0?	2*f(1)	
1	1==0?	1*f(0)	1*1 = 1
0	0==0?	1*1	1

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	
2	2==0?	2*f(1)	2*1 = 2
1	1==0?	1*f(0)	1*1 = 1
0	0==0?	1*1	1

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	
3	3==0?	3*f(2)	3*2 = 6
2	2==0?	2*f(1)	2*1 = 2
1	1==0?	1*f(0)	1*1 = 1
0	0==0?	1*1	1

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	
4	4==0?	4*f(3)	4*6 = 24
3	3==0?	3*f(2)	3*2 = 6
2	2==0?	2*f(1)	2*1 = 2
1	1==0?	1*f(0)	1*1 = 1
0	0==0?	1*1	1

Recursion

E1. Execute the next recursive function (factorial):

n	Condition	n*f(n-1)	return
5	5==0?	5*f(4)	5*24 = 120
4	4==0?	4*f(3)	4*6 = 24
3	3==0?	3*f(2)	3*2 = 6
2	2==0?	2*f(1)	2*1 = 2
1	1==0?	1*f(0)	1*1 = 1
0	0==0?	1*1	1

Recursion

E2. Execute the next recursive function (sum) for f(2,5):

- Sum (a, b)
  - F(a, 0) = a
  - F(a, b) = 1 + f(a, b-1)

Recursion

E2. Execute the next recursive function (sum):

a	b	Condition	1 + f(a, b-1)	Return
2	5	5==0?	1 + f(2, 4)	
2	4	4==0?		

Recursion

E2. Execute the next recursive function (sum):

a	b	Condition	1 + f(a, b-1)	Return
2	5	5==0?	1 + f(2, 4)	
2	4	4==0?	1 + f(2, 3)	
2	3	3==0?	1 + f(2, 2)	
2	2	2==0?	1 + f(2, 1)	

Recursion

E2. Execute the next recursive function (sum):

a	b	Condition	1 + f(a, b-1)	Return
2	5	5==0?	1 + f(2, 4)	
2	4	4==0?	1 + f(2, 3)	
2	3	3==0?	1 + f(2, 2)	
2	2	2==0?	1 + f(2, 1)	
2	1	1==0?	1 + f(2, 0)	
2	0	0==0?		2

Recursion

E2. Execute the next recursive function (sum):

a	b	Condition	1 + f(a, b-1)	Return
2	5	5==0?	1 + f(2, 4)	7
2	4	4==0?	1 + f(2, 3)	6
2	3	3==0?	1 + f(2, 2)	5
2	2	2==0?	1 + f(2, 1)	4
2	1	1==0?	1 + f(2, 0)	3
2	0	0==0?		2

Recursion

E3. Execute the next recursive function (remainder) for f(15,4): (15%4) == 3

- Remainder (a, b)
  - F(a, b) = a when a-b<0
  - F(a, b) = f(a-b, b) when a-b >=0

Recursion

E3. Execute the next recursive function (remainder):

a	b	Condition	F(a-b, b)	Return
15	4	15-4<0?	f(11, 4)	
11	4	11-4<0?		

Recursion

E3. Execute the next recursive function (remainder):

a	b	Condition	F(a-b, b)	Return
15	4	15-4<0?	f(11, 4)	
11	4	11-4<0?	f(7,4)	
7	4	7-4<0?	f(3,4)	
3	4	3-4<0?		3

E3. Execute the next recursive function (remainder):

a	b	Condition	F(a-b, b)	Return
15	4	15-4<0?	f(11, 4)	3
11	4	11-4<0?	f(7,4)	3
7	4	7-4<0?	f(3,4)	3
3	4	3-4<0?		3

Recursion

E4. Execute the next recursive function (sum-array) for f({2, 5, 6, 8}, 4):

- Sum-array (a, b)
  - F(V, n) = V[0] when n == 1
  - F(V, n) = V[n-1] + f(V, n-1) when n > 1

Recursion

E4. Execute the next recursive function (sum-array):

V	n	Condition	V[n-1] + F(V, n-1)	Return
{2, 5, 6, 8}	4	4==1?	8 + f({2, 5, 6, 8}, 3)	
{2, 5, 6, 8}	3	3==1?		

It is your turn!

Recursion

E4. Execute the next recursive function (sum-array):

V	n	Condition	V[n-1] + F(V, n)	Return
{2, 5, 6, 8}	4	4==1?	8 + f({2, 5, 6, 8}, 3)	
{2, 5, 6, 8}	3	3==1?	6 + f({2, 5, 6, 8}, 2)	
{2, 5, 6, 8}	2	2==1?	5 + f({2, 5, 6, 8}, 1)	
{2, 5, 6, 8}	1	1==1?		2

Recursion

E4. Execute the next recursive function (sum-array):

V	n	Condition	V[n-1] + F(V, n)	Return
{2, 5, 6, 8}	4	4==1?	8 + f({2, 5, 6, 8}, 3)	21
{2, 5, 6, 8}	3	3==1?	6 + f({2, 5, 6, 8}, 2)	13
{2, 5, 6, 8}	2	2==1?	5 + f({2, 5, 6, 8}, 1)	7
{2, 5, 6, 8}	1	1==1?		2

Unit 2

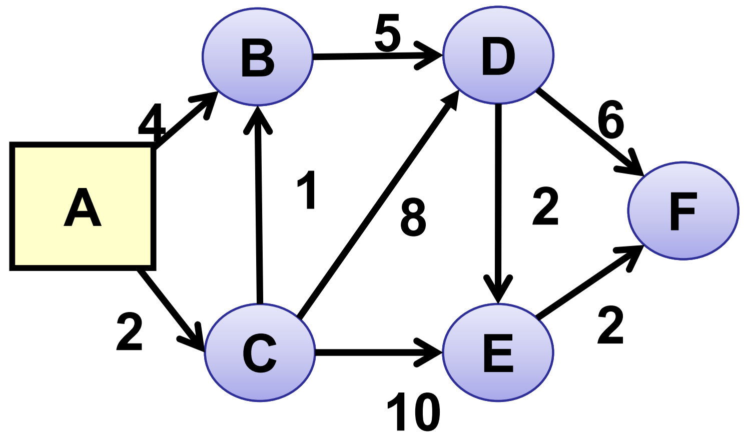
Network Structures

Dijkstra

Dijkstra Algorithm

E1. Minimum cost between A and F

- ❖ Cost from A.

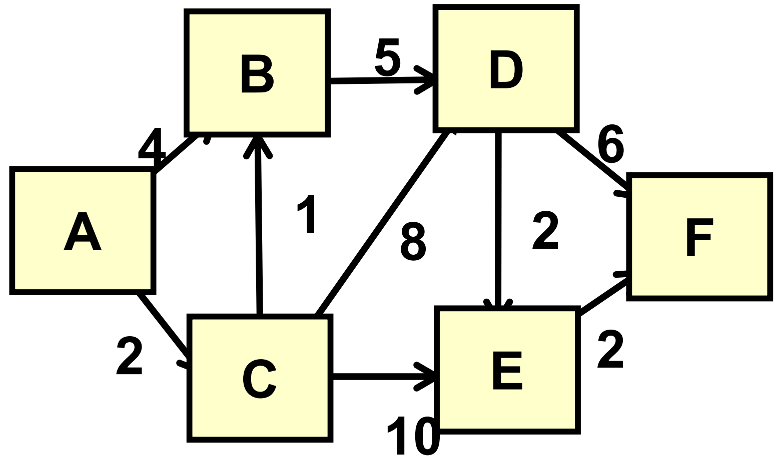


it	S	w	Vector D: B	C	D	E	F	Vector P: B	C	D	E	F
1	A		4	2	∞	∞	∞	A	A	-	-	-

Dijkstra Algorithm

E1. Minimum cost between A and F

- Cost from A.

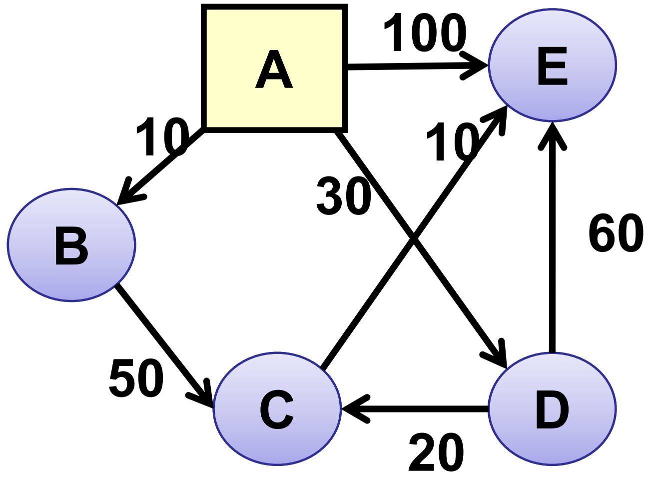


it	S	w	Vector D: B	C	D	E	F	Vector P: B	C	D	E	F
1	A		4	2	∞	∞	∞	A	A	-	-	-
2	A, C	C	3	2	10	12	∞	C	A	C	C	-
3	A, B, C	B	3	2	8	12	∞	C	A	B	C	-
4	A, B, C, D	D	3	2	8	10	14	C	A	B	D	D
5	A, B, C, D, E	E	3	2	8	10	12	C	A	B	D	E
6	A, B, C, D, E, F	F	3	2	8	10	12	C	A	B	D	E

Dijkstra Algorithm

E2. Minimum cost from A

- Cost from A.

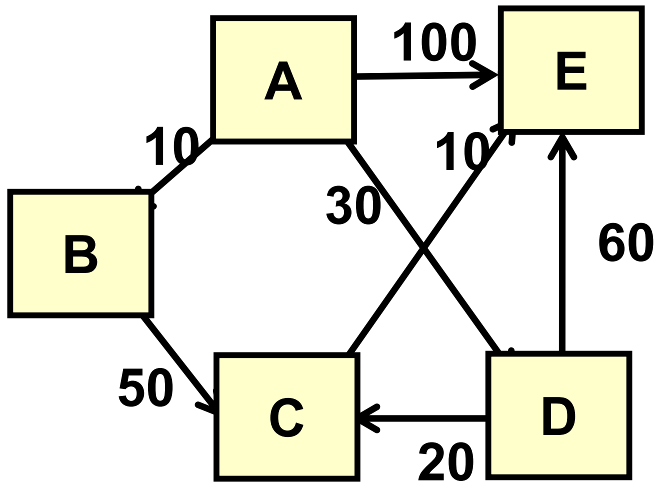


it	S	w	Vector D: B	C	D	E	Vector P: B	C	D	E
1	A		10	∞	30	100	A	-	A	A

Dijkstra Algorithm

E2. Minimum cost from A

- Cost from A.

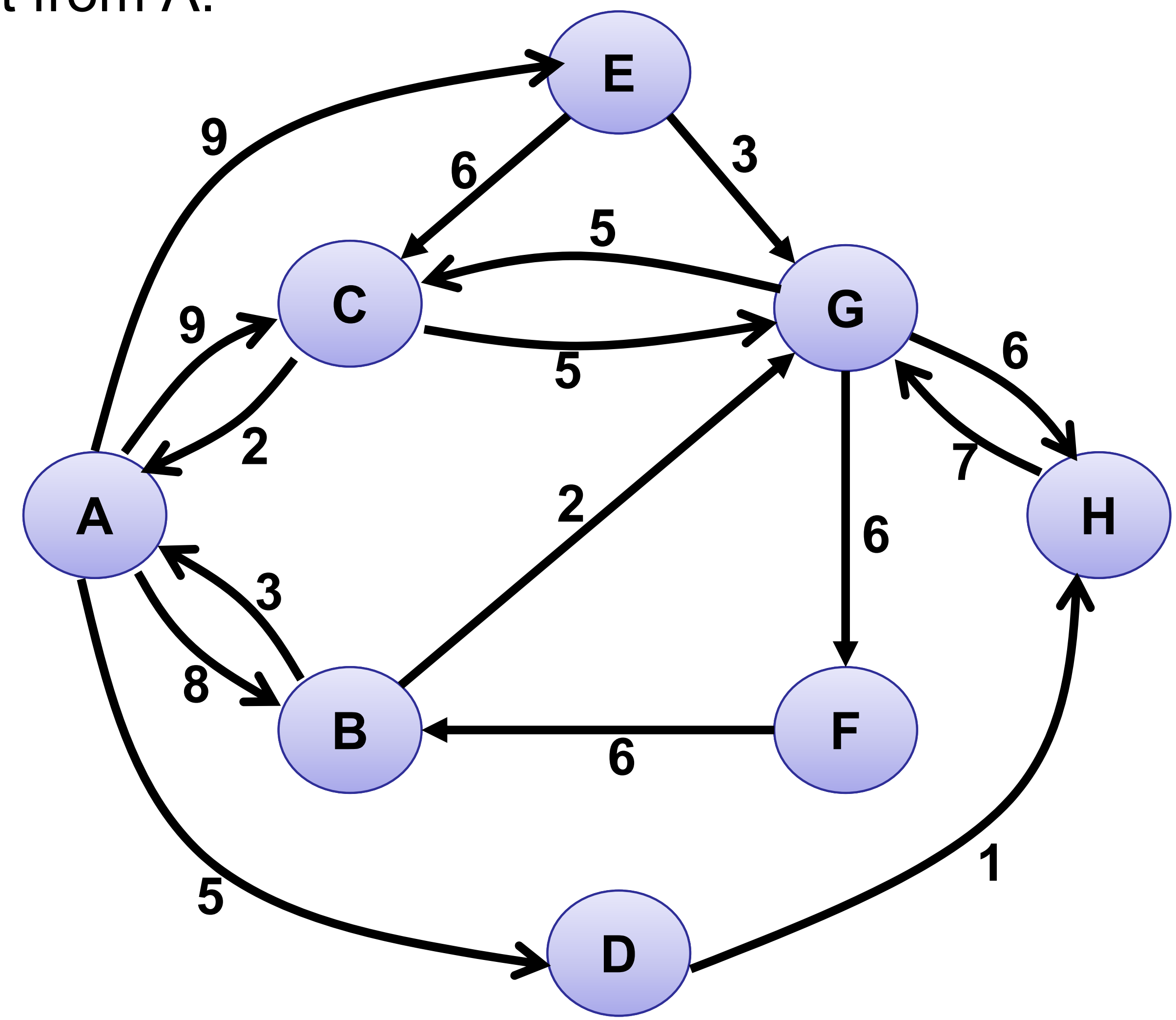


it	S	w	Vector D: B	C	D	E	Vector P: B	C	D	E
1	A		10	∞	30	100	A	-	A	A
2	A, B	B	10	60	30	100	A	B	A	A
3	A, B, D	D	10	50	30	90	A	D	A	D
4	A, B, C, D	C	10	50	30	60	A	D	A	C
5	A, B, C, D, E	E	10	50	30	60	A	D	A	C

Dijkstra Algorithm

E3. Minimum cost from A

- Cost from A.



Dijkstra Algorithm

E3. Minimum cost from A

- Init

	B	C	D	E	F	G	H	
D	8	9	5	9	∞	∞	∞	
P	A	A	A	A	-	-	-	

Dijkstra Algorithm

E3. Minimum cost from A

- Pivot D
- S = [A, D]

	B	C	D	E	F	G	H	
D	8	9	**5**	9	∞	∞	6	
P	A	A	A	A	-	-	D	

Dijkstra Algorithm

E3. Minimum cost from A

- <u>Pivot</u> H
- S = [A, D, H]

	B	C	D	E	F	G	H	
D	8	9	**5**	9	∞	13	**6**	
P	A	A	A	A	-	H	D	

Dijkstra Algorithm

E3. Minimum cost from A

- Pivot B
- S = [A, B, D, H]

	B	C	D	E	F	G	H	
D	**8**	9	**5**	9	∞	10	**6**	
P	A	A	A	A	-	B	D	

Dijkstra Algorithm

E3. Minimum cost from A

- Pivot C
- S = [A, B, C, D, H]

	B	C	D	E	F	G	H	
D	**8**	**9**	**5**	9	∞	10	**6**	
P	A	A	A	A	-	B	D	

Dijkstra Algorithm

E3. Minimum cost from A

- Pivot E
- S = [A, B, C, D, E, H]

	B	C	D	E	F	G	H	
D	**8**	**9**	**5**	**9**	∞	10	**6**	
P	A	A	A	A	-	B	D	

Dijkstra Algorithm

E3. Minimum cost from A

- Pivot G
- S = [A, B, C, D, E, G, H]

	B	C	D	E	F	G	H	
D	**8**	**9**	**5**	**9**	16	**10**	**6**	
P	A	A	A	A	G	B	D	

E3. Minimum cost from A

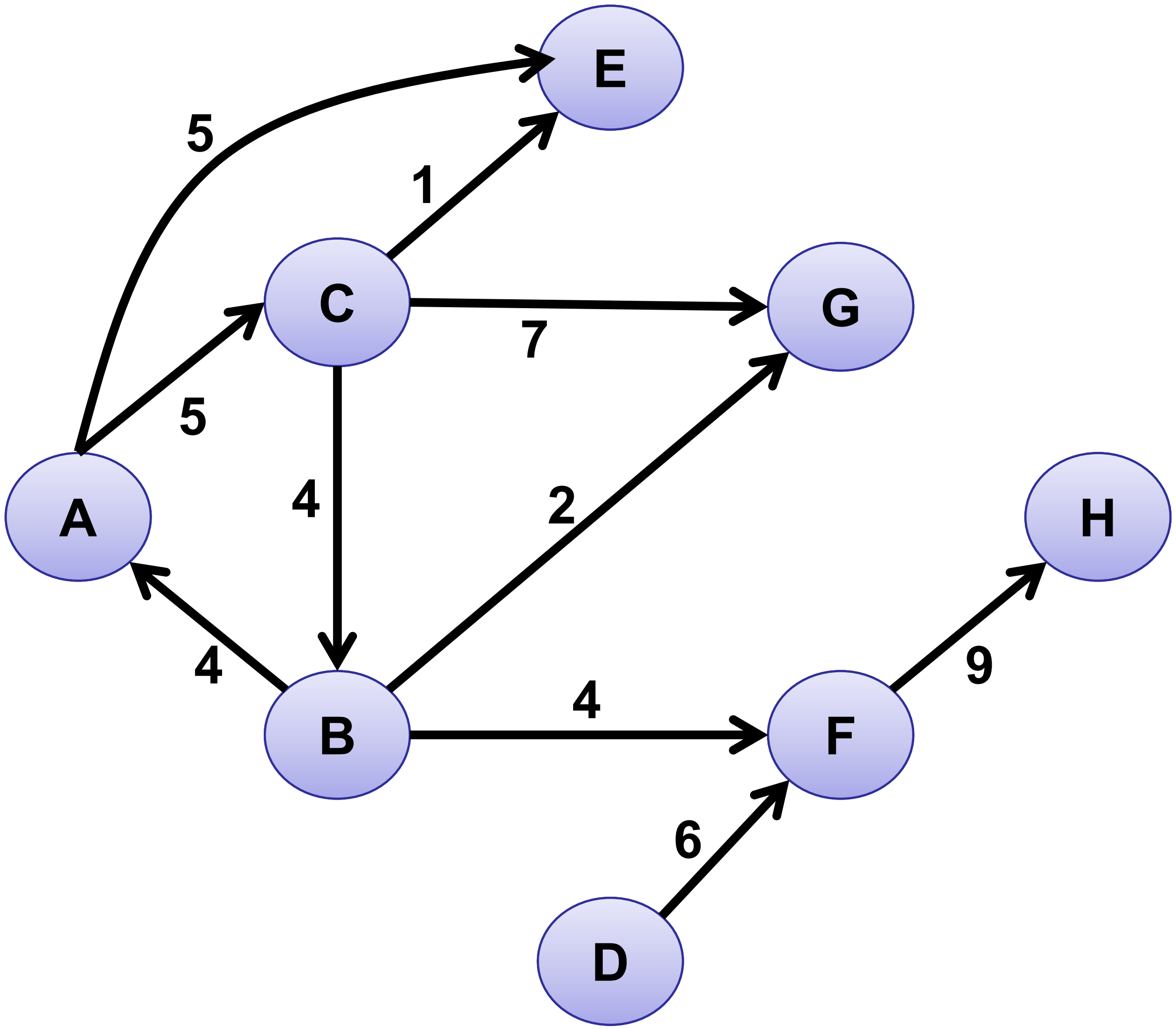
- Pivot F
- S = [A, B, C, D, E, F, G, H]

	B	C	D	E	F	G	H	
D	**8**	**9**	**5**	**9**	**16**	**10**	**6**	
P	A	A	A	A	G	B	D	

Dijkstra Algorithm

E4. Minimum cost from A

- Cost from A.



Dijkstra Algorithm

E4. Minimum cost from A

- Init

	B	C	D	E	F	G	H	
D	∞	5	∞	5	∞	∞	∞	
P	-	A	-	A	-	-	-	

E4. Minimum cost from A

- Pivot C
- S = [C]

	B	C	D	E	F	G	H	
D	9	**5**	∞	5	∞	12	∞	
P	C	A	-	A	-	C	-	

Dijkstra Algorithm

E4. Minimum cost from A

- Pivot E
- S = [C, E]

	B	C	D	E	F	G	H	
D	9	**5**	∞	**5**	∞	12	∞	
P	C	A	-	A	-	C	-	

Dijkstra Algorithm

E4. Minimum cost from A

- Pivot B
- S = [B, C, E]

	B	C	D	E	F	G	H	
D	**9**	**5**	∞	**5**	13	11	∞	
P	C	A	-	A	B	B	-	

Dijkstra Algorithm

E4. Minimum cost from A

- <u>Pivot</u> G
- S = [B, C, E, G]

	B	C	D	E	F	G	H	
D	**9**	**5**	∞	**5**	13	**11**	∞	
P	C	A	-	A	B	B	-	

Dijkstra Algorithm

E4. Minimum cost from A

- Pivot F
- S = [B, C, E, F, G]

	B	C	D	E	F	G	H	
D	**9**	**5**	∞	**5**	**13**	**11**	22	
P	C	A	-	A	B	B	F	

Dijkstra Algorithm

E4. Minimum cost from A

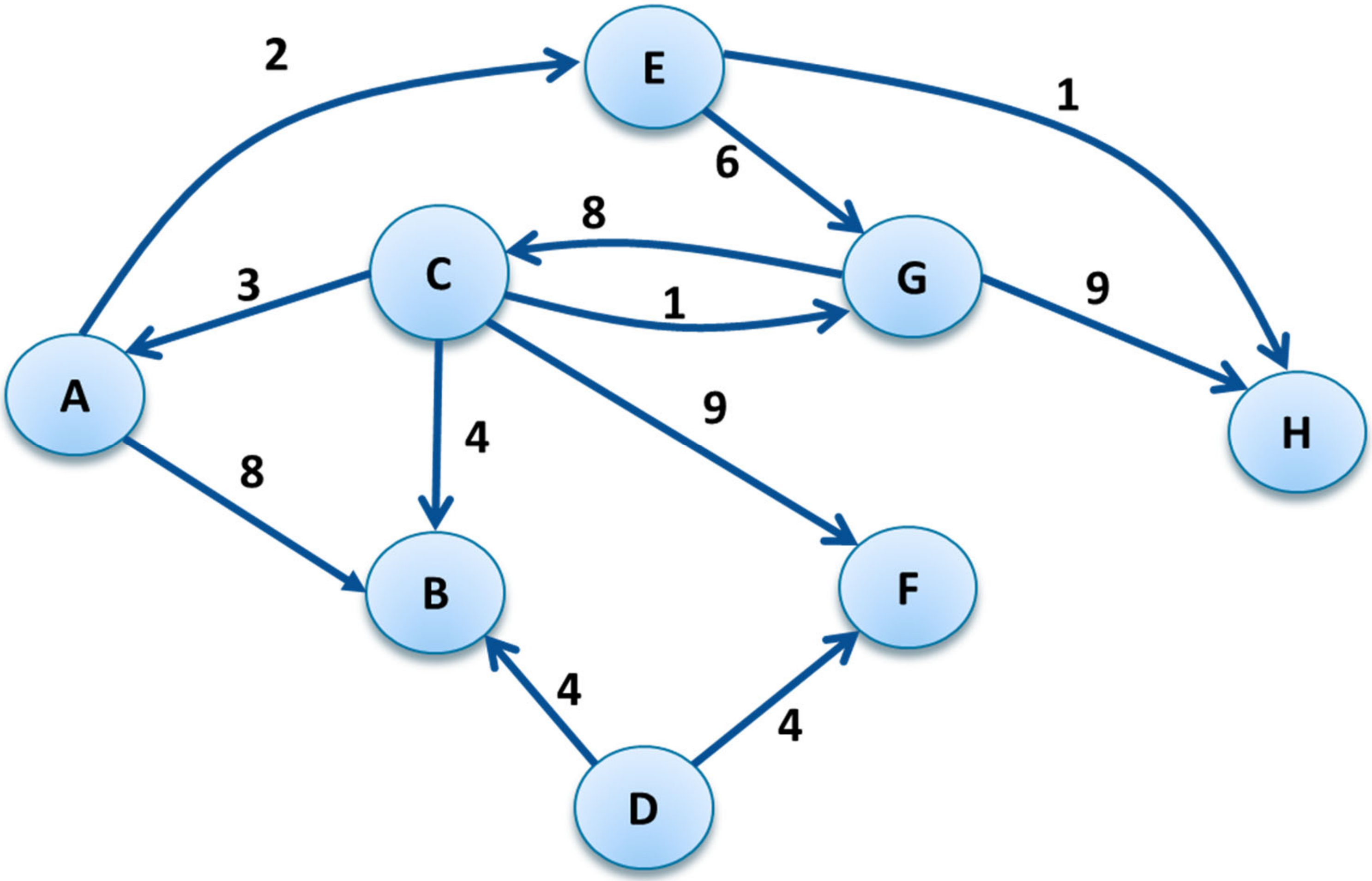
- Pivot H
- S = [B, C, E, F, G, H]

	B	C	D	E	F	G	H	
D	**9**	**5**	∞	**5**	**13**	**11**	**22**	
P	C	A	-	A	B	B	F	

Dijkstra Algorithm

E5. Minimum cost from E

- Cost from E.



Dijkstra Algorithm

E5. Minimum cost from E

- Init

	A	B	C	D		F	G	H
D	∞	∞	∞	∞		∞	6	1
P	-	-	-	-		-	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot H
- S = [E, H]

	A	B	C	D		F	G	H
D	∞	∞	∞	∞		∞	6	**1**
P	-	-	-	-		-	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot G
- S = [E, H, G]

	A	B	C	D		F	G	H
D	∞	∞	14	∞		∞	**6**	**1**
P	-	-	G	-		-	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot C
- S = [C, E, H, G]

	A	**B**	**C**	**D**		**F**	**G**	**H**
D	17	18	**14**	∞		23	**6**	**1**
P	C	C	G	-		C	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot A
- S = [A, C, E, H, G]

	A	B	C	D		F	G	H
D	**17**	18	**14**	∞		23	**6**	**1**
P	C	C	G	-		C	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot B
- S = [A, B, C, E, H, G]

	A	B	C	D		F	G	H
D	**17**	**18**	**14**	∞		23	**6**	**1**
P	C	C	G	-		C	E	E

Dijkstra Algorithm

E5. Minimum cost from E

- Pivot F
- S = [A, B, C, E, F, H, G]

	A	B	C	D		F	G	H
D	**17**	**18**	**14**	∞		**23**	**6**	**1**
P	C	C	G	-		C	E	E

Unit 3

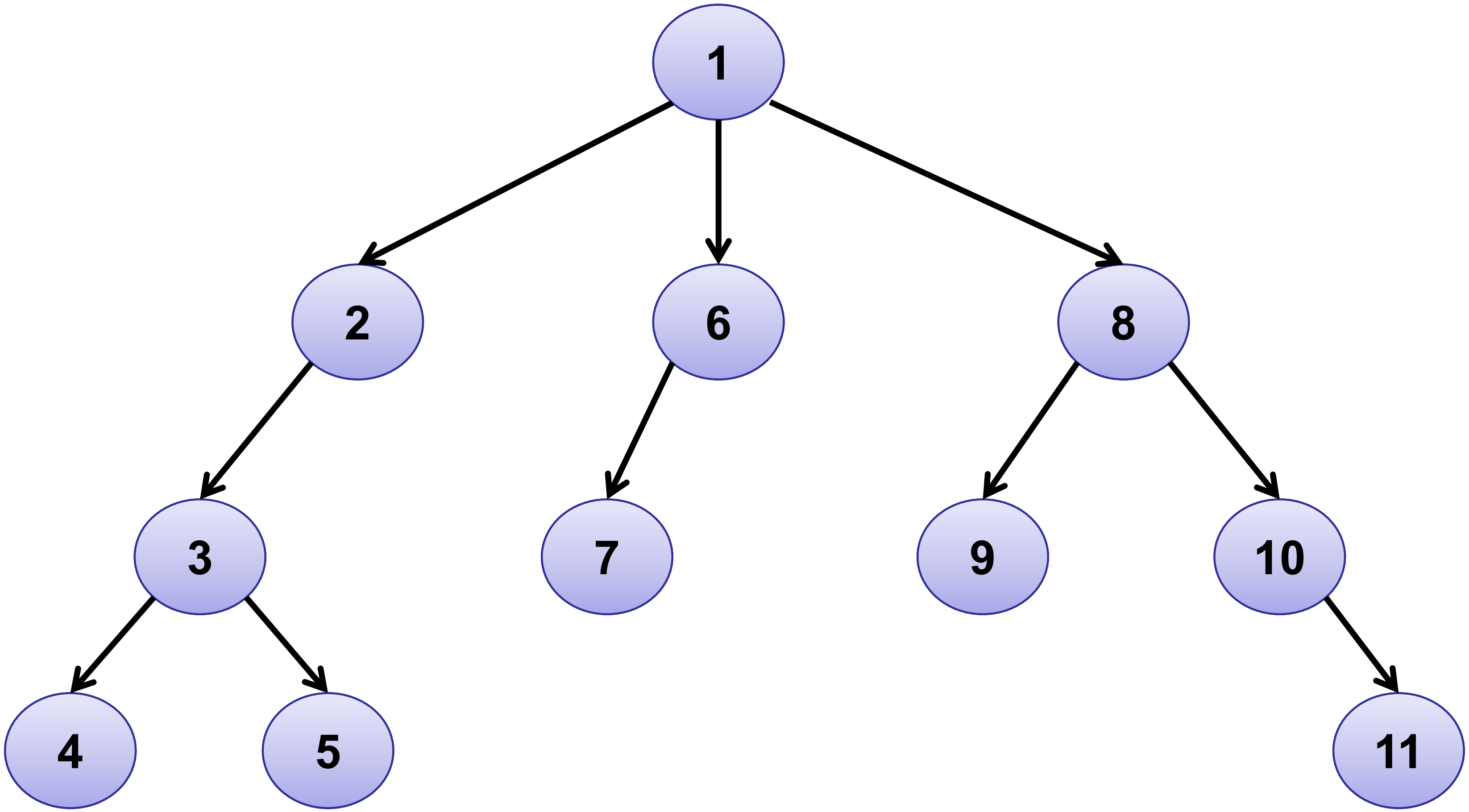
Network Structures

Search & Floyd

Depth First Search

E1. Draw the path starting navigation from node 1.

- Assume that nodes were inserted in order.



Depth First Search

E1. Draw the path starting navigation from node A.

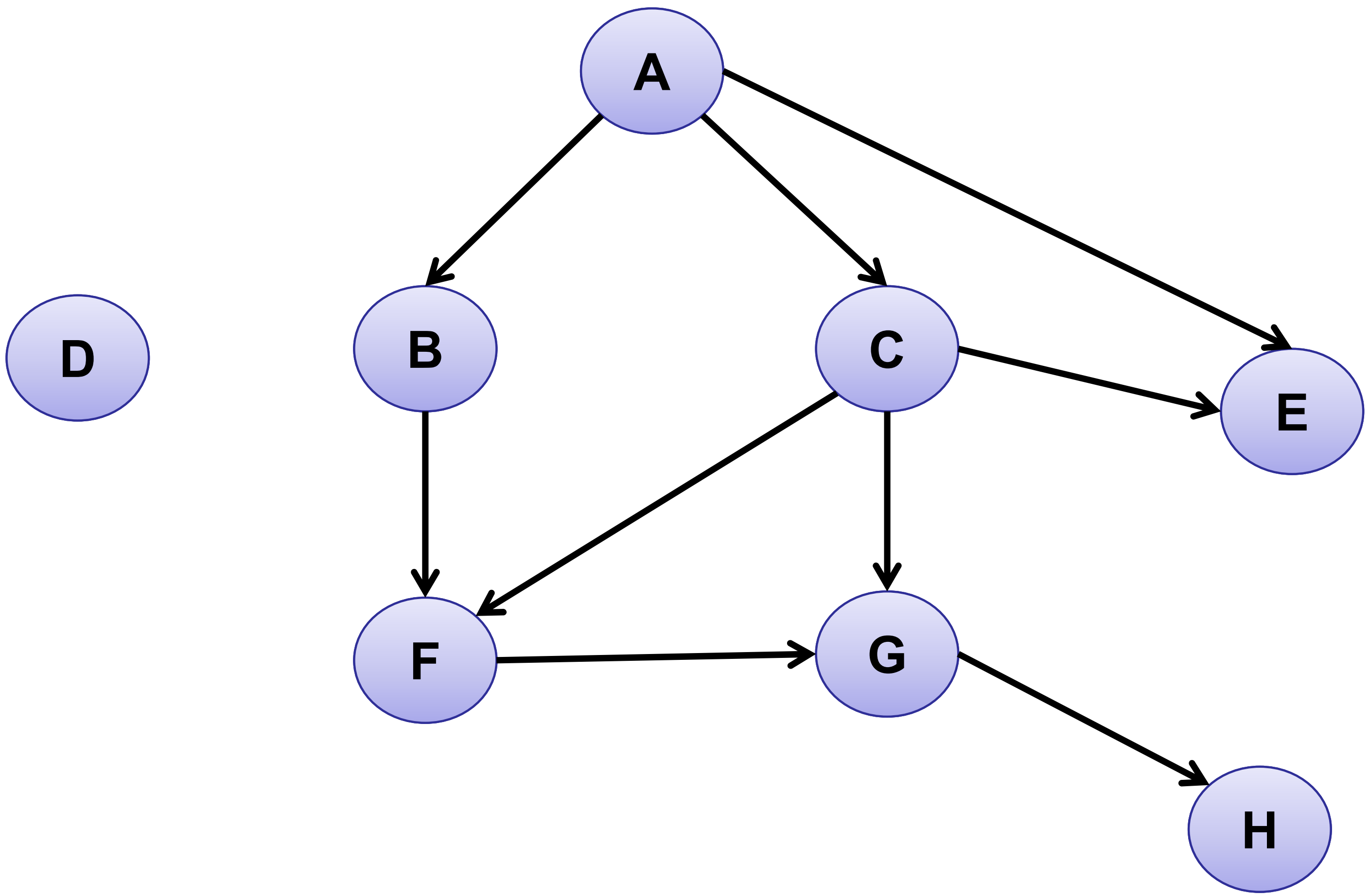
- Assume that nodes were inserted in order.

Path	
{1}	
{1, 2}	
{1, 2, 3}	
{1, 2, 3, 4}	
{1, 2, 3, 4, 5}	
{1, 2, 3, 4, 5, 6}	
{1, 2, 3, 4, 5, 6, 7}	
{1, 2, 3, 4, 5, 6, 7, 8}	
{1, 2, 3, 4, 5, 6, 7, 8, 9}	
{1, 2, 3, 4, 5, 6, 7, 8, 10}	
{1, 2, 3, 4, 5, 6, 7, 8, 10, 11}	

Depth First Search

E2. Draw the path starting navigation from node A.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E2. Draw the path starting navigation from node A.

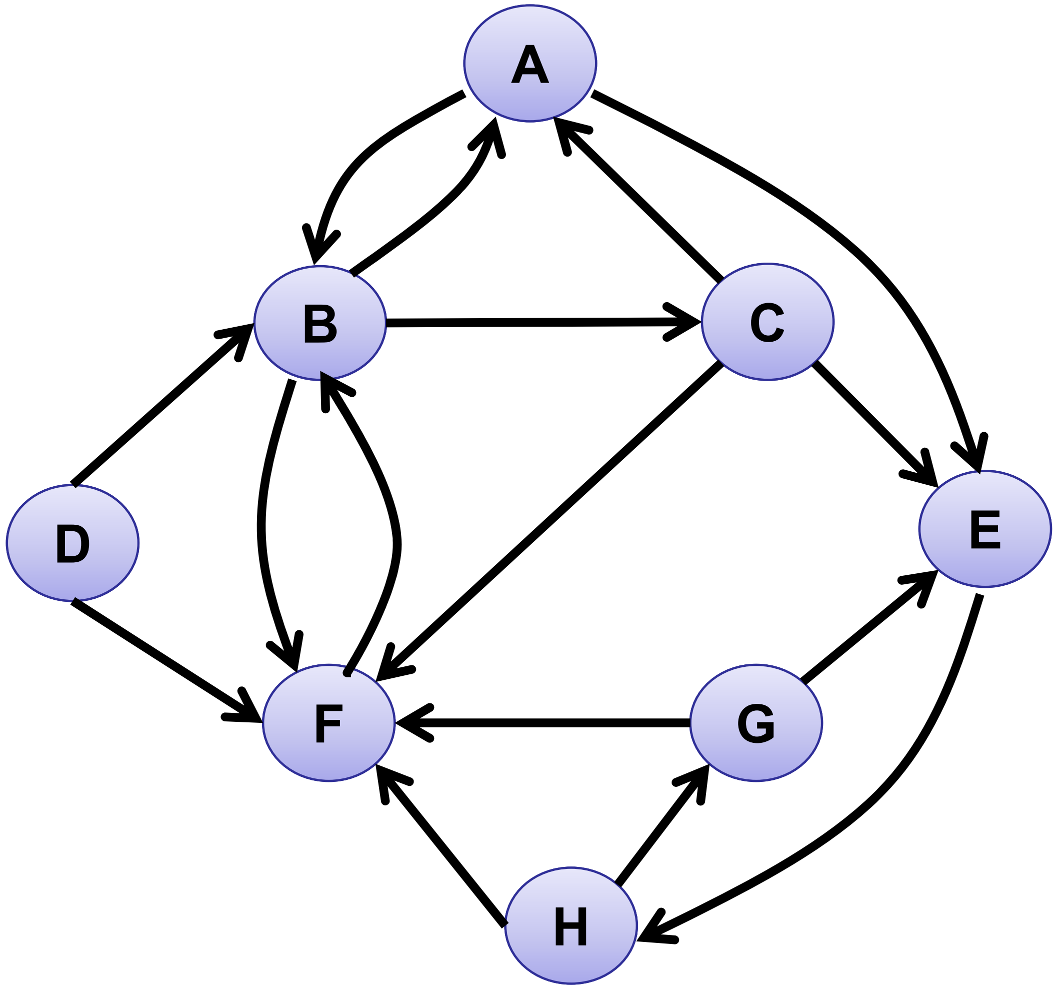
- Assume that nodes were inserted in alphabetical order.

Path	
{A}	
{A, B}	
{A, B, F}	
{A, B, F, G}	
{A, B, F, G, H}	
{A, B, F, C, H, C}	
{A, B, F, C, H, C, E}	

Depth First Search

E3. Draw the path starting navigation from node C.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E3. Draw the path starting navigation from node C.

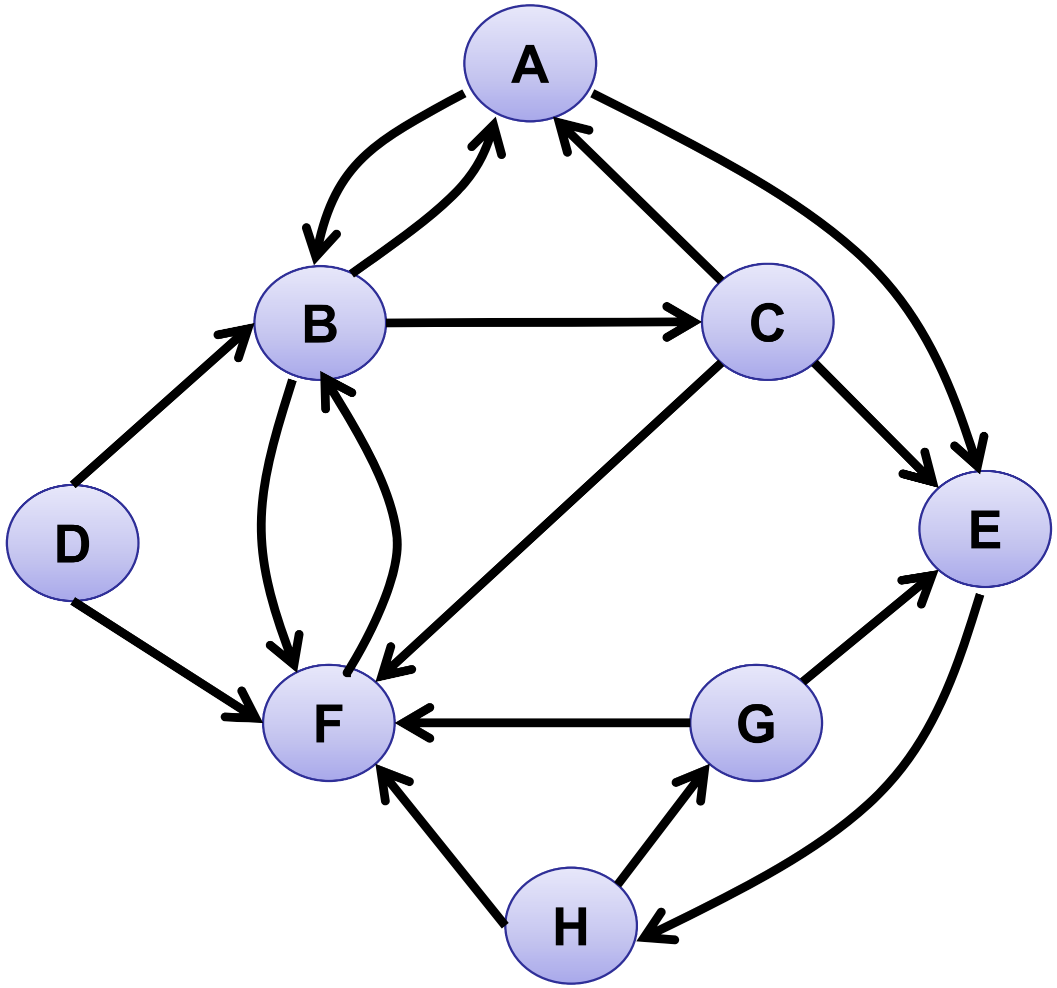
- Assume that nodes were inserted in alphabetical order.

Path	
{C}	
{C, A}	
{C, A, B}	
{C, A, B, F}	
{C, A, B, F, E}	
{C, A, B, F, E, H}	
{C, A, B, F, E, H, G}	

Depth First Search

E4. Draw the path starting navigation from node D.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E4. Draw the path starting navigation from node d.

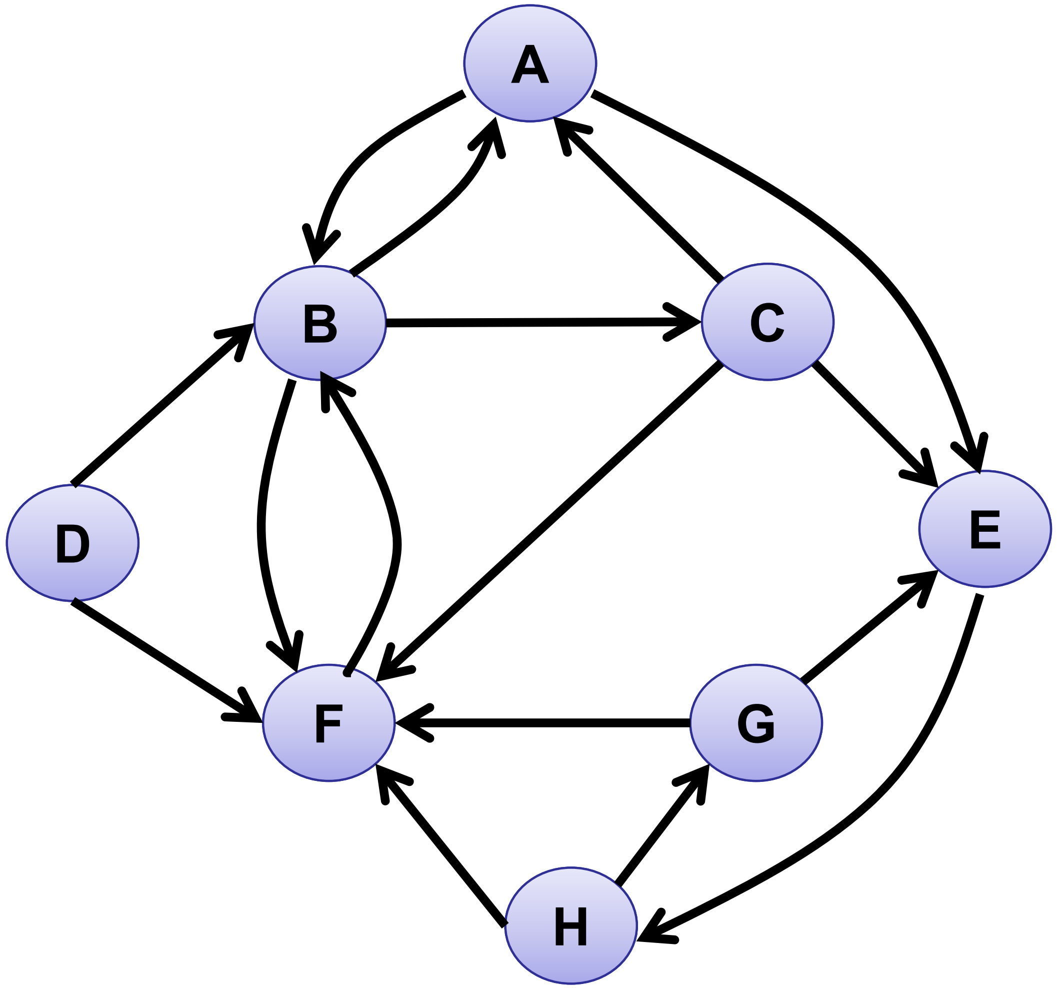
- Assume that nodes were inserted in alphabetical order.

Path	
{D}	
{D, B}	
{D, B, A}	
{D, B, A, E}	
{D, B, A, E, H}	
{D, B, A, E, H, F}	
{D, B, A, E, H, F, G}	
{D, B, A, E, H, F, G, C}	

Depth First Search

E5. Draw the path starting navigation from node H.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E5. Draw the path starting navigation from node H.

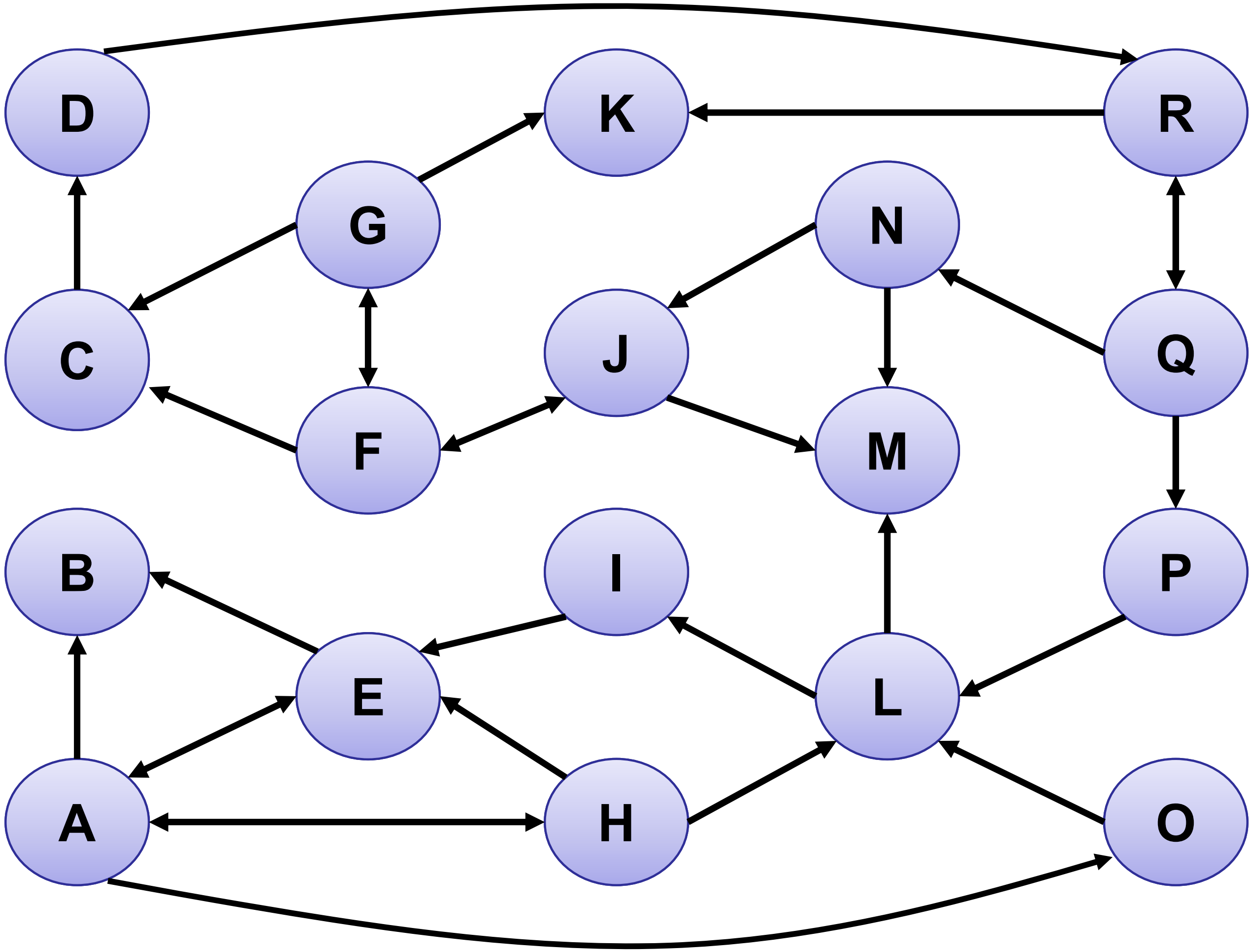
- Assume that nodes were inserted in alphabetical order.

Path	
{H}	
{H, G}	
{H, G, E}	
{H, G, E, F}	
{H, G, E, F, B}	
{H, G, E, F, B, A}	
{H, G, E, F, B, A, C}	

Depth First Search

E6. Draw the path starting navigation from node A.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E6. Draw the path starting navigation from node A.

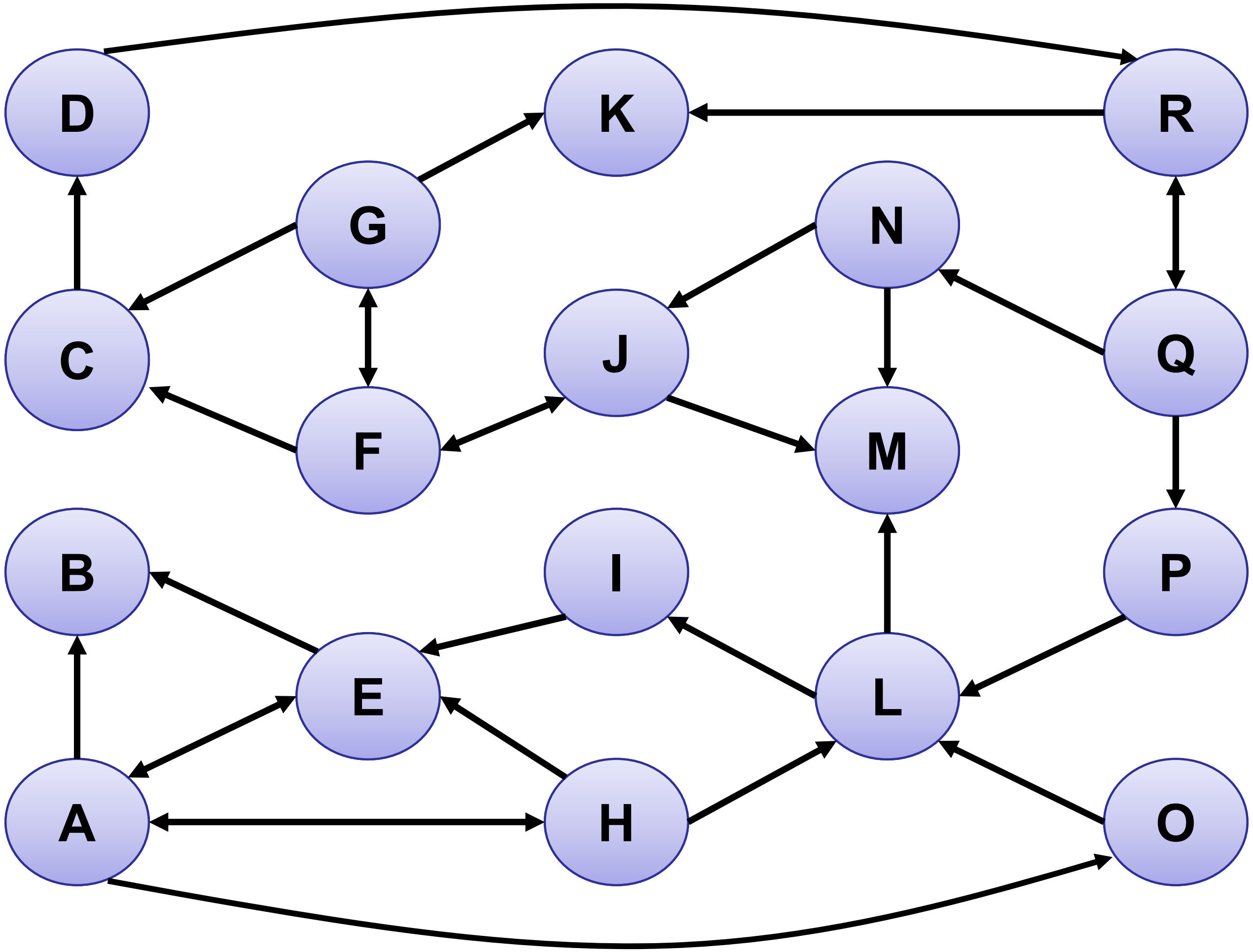
- Assume that nodes were inserted in alphabetical order.

Path	
{A}	
{A, B}	
{A, B, E}	
{A, B, E, H}	
{A, B, E, H, L}	
{A, B, E, H, L, I}	
{A, B, E, H, L, I, M}	
{A, B, E, H, L, I, M, O}	

Depth First Search

E7. Draw the path starting navigation from node E.

- Assume that nodes were inserted in alphabetical order.



Depth First Search

E7. Draw the path starting navigation from node E.

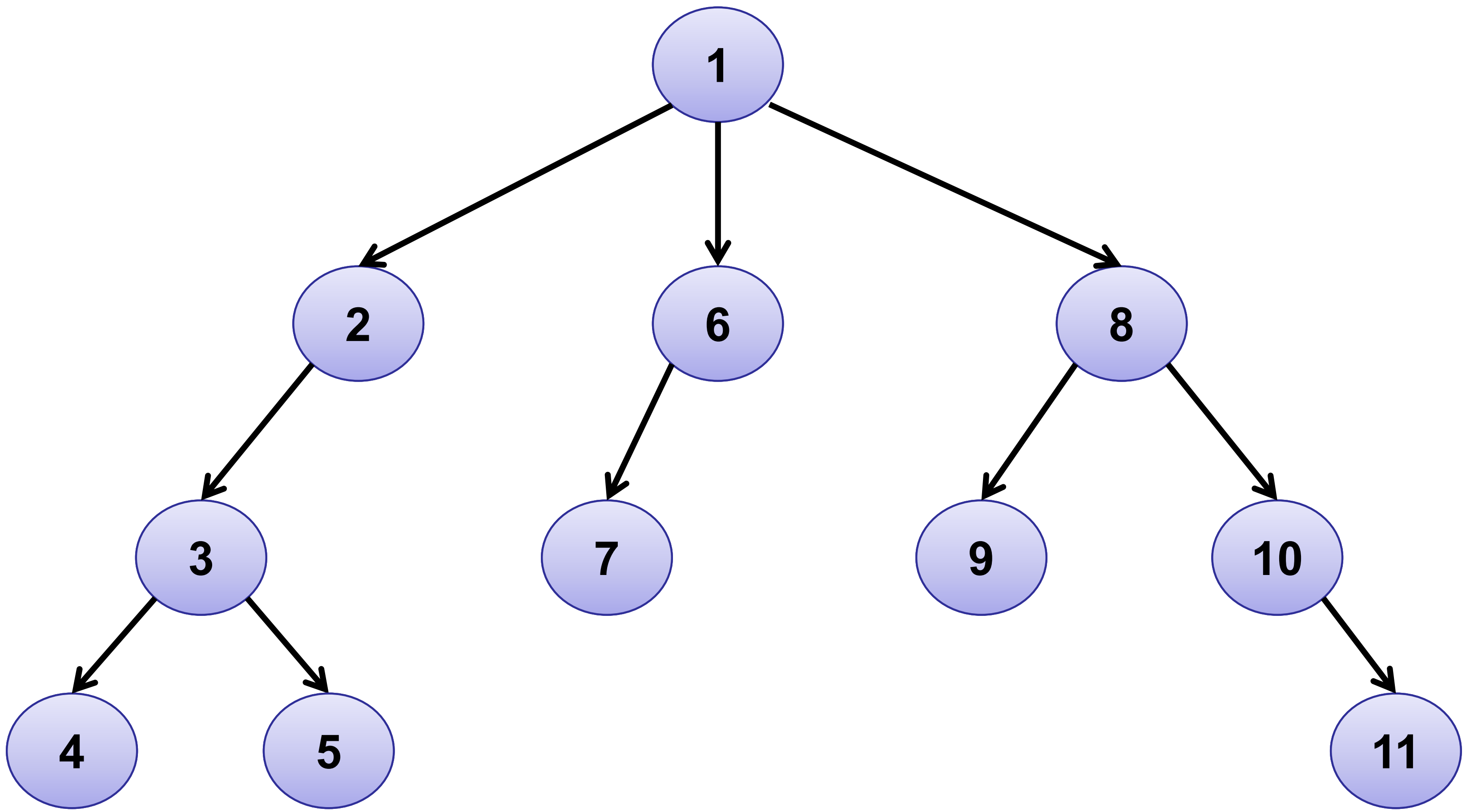
- Assume that nodes were inserted in alphabetical order.

Path	
{E}	
{E, A}	
{E, A, B}	
{E, A, B, H}	
{E, A, B, H, L}	
{E, A, B, H, L, I}	
{E, A, B, H, L, I, M}	
{E, A, B, H, L, I, M, O}	

Width First Search

E8. Draw the path starting navigation from node 1.

- Assume that nodes were inserted in alphabetical order.



Width First Search

E8. Draw the path starting navigation from node 1.

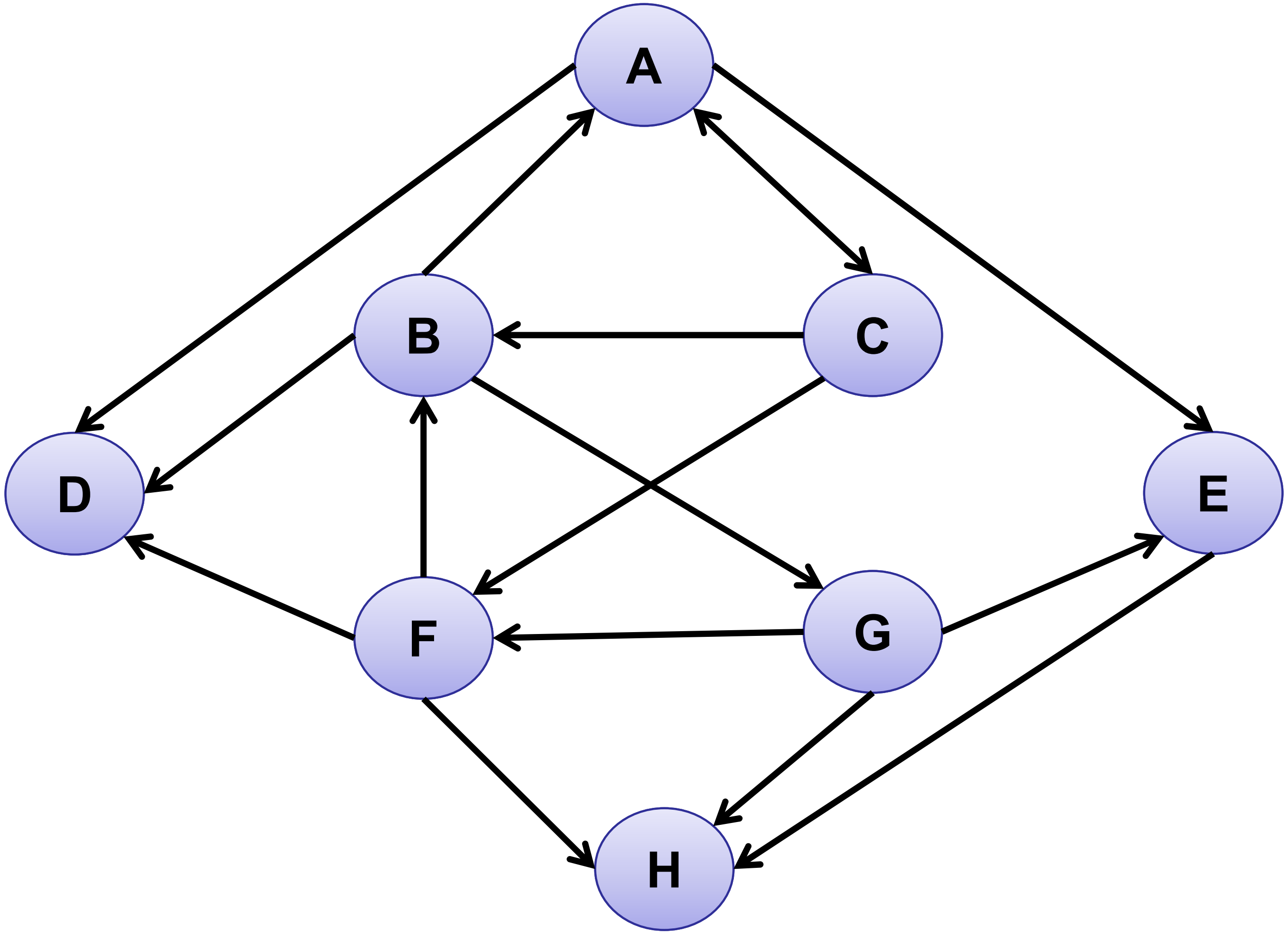
- Assume that nodes were inserted in order.

Path	Candidates
{1}	{2, 6, 8}
{1, 2}	{6, 8, 3}
{1, 2, 6}	{8, 3, 7}
{1, 2, 6, 8}	{3, 7, 9, 10}
{1, 2, 6, 8, 3}	{7, 9, 10, 4, 5}
{1, 2, 6, 8, 3, 7}	{9, 10, 4, 5}
{1, 2, 6, 8, 3, 7, 9}	{10, 4, 5}
{1, 2, 6, 8, 3, 7, 9, 10}	{4, 5, 11}
{1, 2, 6, 8, 3, 7, 9, 10, 4}	{5, 11}
{1, 2, 6, 8, 3, 7, 9, 10, 4, 5}	{11}
{1, 2, 6, 8, 3, 7, 9, 10, 4, 5, 11}	{}

Width First Search

E9. Draw the path starting navigation from node A.

- Assume that nodes were inserted in alphabetical order.



Width First Search

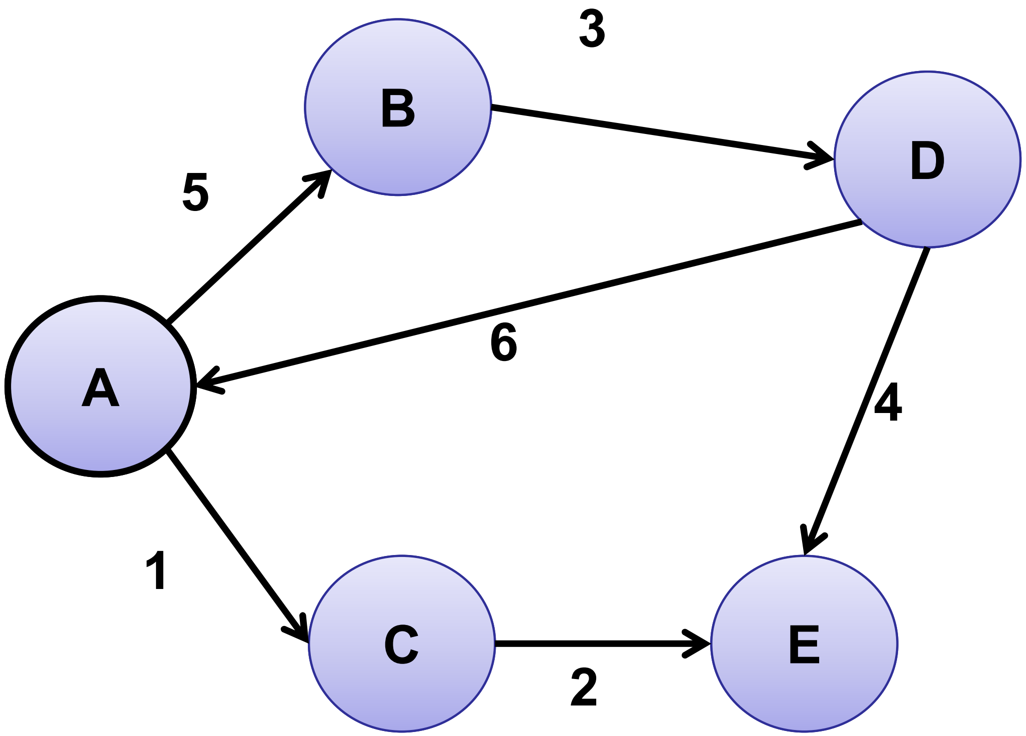
E9. Draw the path starting navigation from node A.

- Assume that nodes were inserted in alphabetical order.

Path	Candidates
{A}	{C, D, E}
{A, C}	{D, E, B, F}
{A, C, D}	{E, B, F, H}
{A, C, D, E}	{B, F, H}
{A, C, D, E, B}	{F, H, G}
{A, C, D, E, B, F}	{H, G}
{A, C, D, E, B, F, H}	{G}
{A, C, D, E, B, F, H, G}	{}

Floyd-Warshall

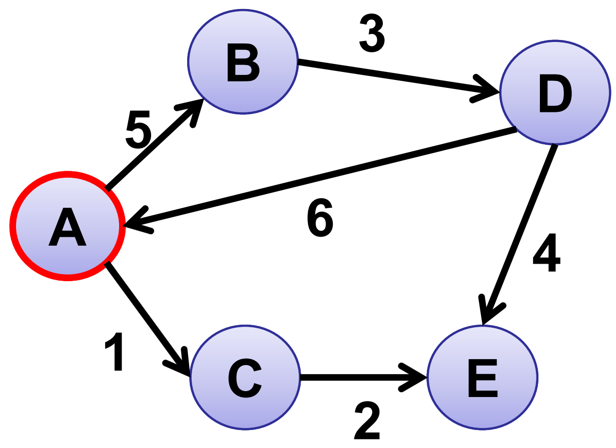
E10. Execute the Floyd-Warshall algorithm on this graph



Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Initialization.



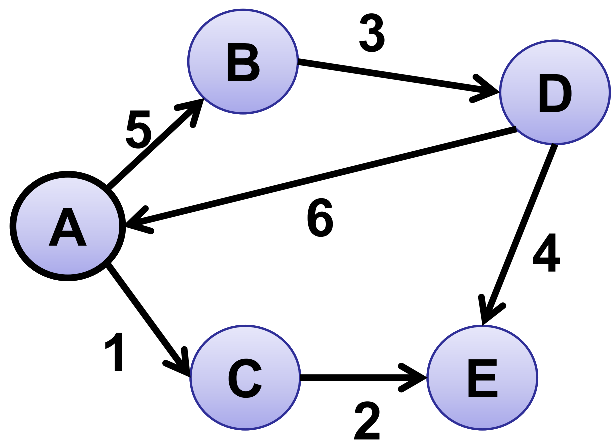
	A	B	C	D	E
A	0	5	1	∞	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	∞	∞	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A					
B					
C					
D					
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 1 (node A).



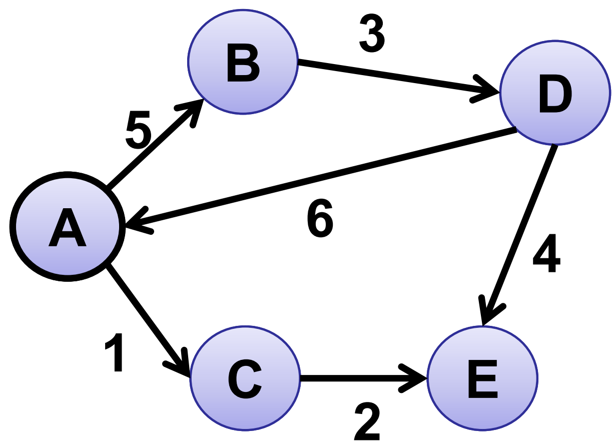
	A	B	C	D	E
A	0	5	1	∞	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	∞	∞	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A					
B					
C					
D					
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 1 (node A).
- AFTER ITERATION COMPLETION.



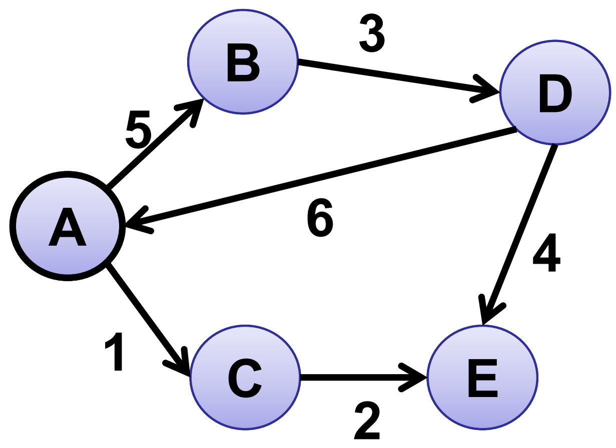
	A	B	C	D	E
A	0	5	1	∞	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A	0				
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 2 (node B).



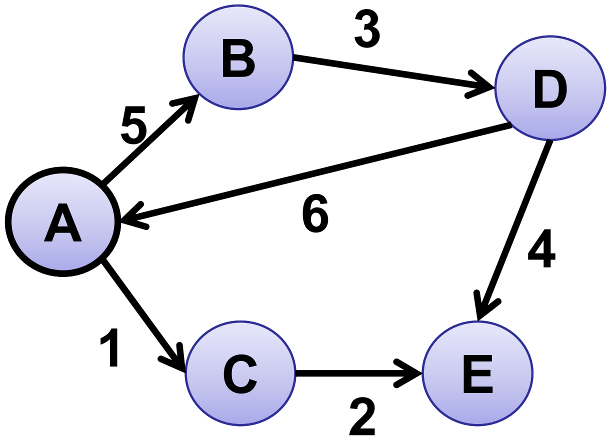
	A	B	C	D	E
A	0	5	1	∞	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A					
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 2 (node B).
- AFTER ITERATION COMPLETION.



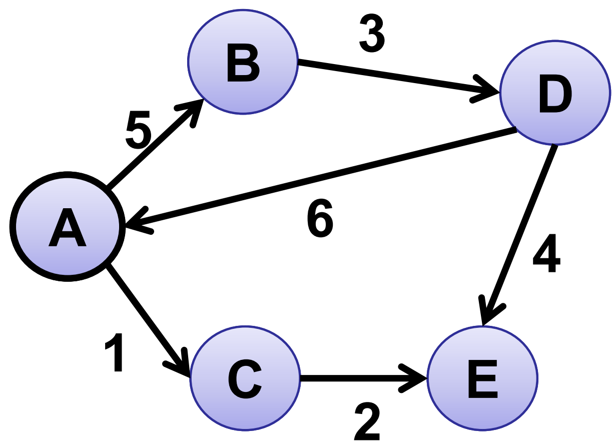
	A	B	C	D	E
A	0	5	1	8	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 3 (node C).



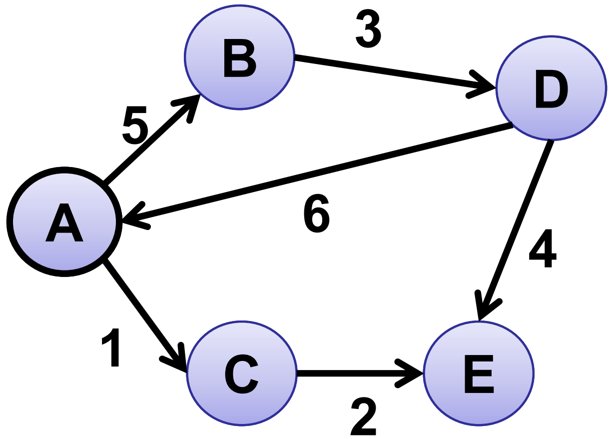
	A	B	C	D	E
A	0	5	1	8	∞
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 3 (node C).
- AFTER ITERATION COMPLETION.



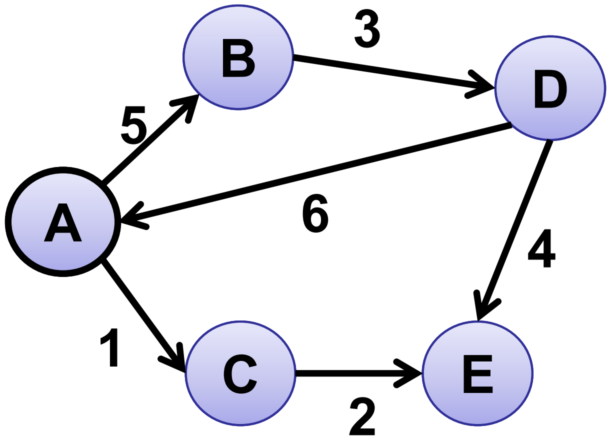
	A	B	C	D	E
A	0	5	1	8	3
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	C
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 4 (node D).



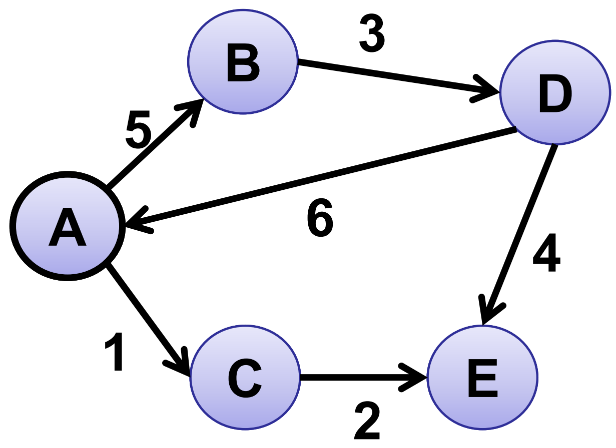
	A	B	C	D	E
A	0	5	1	8	3
B	∞	0	∞	3	∞
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	C
B					
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 4 (node D).
- AFTER ITERATION COMPLETION.



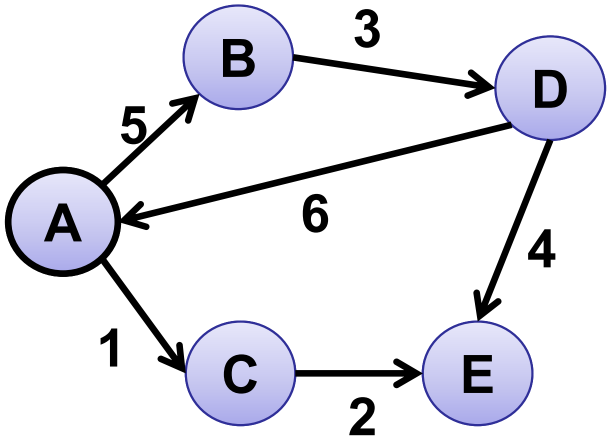
	A	B	C	D	E
A	0	5	1	8	3
B	**9**	0	**10**	3	**7**
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	C
B	**D**		**D**		**D**
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 5 (node E).



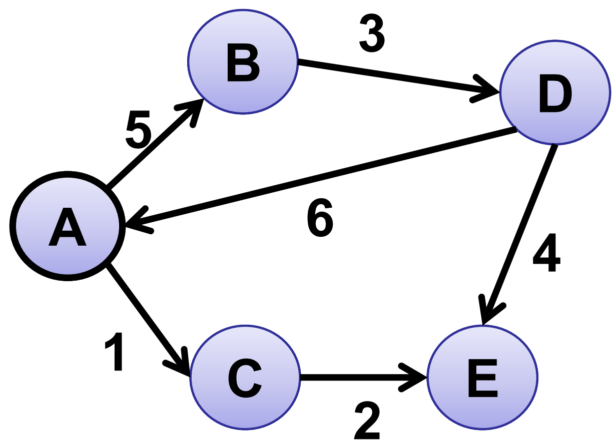
	A	B	C	D	E
A	0	5	1	8	3
B	9	0	10	3	7
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	C
B	D		D		D
C					
D		A	A		
E					

Floyd-Warshall

E10. Execute the Floyd-Warshall algorithm on this graph

- Iteration 5 (node E).
- AFTER ITERATION COMPLETION.
- END OF THE EXECUTION.



	A	B	C	D	E
A	0	5	1	8	3
B	9	0	10	3	7
C	∞	∞	0	∞	2
D	6	11	7	0	4
E	∞	∞	∞	∞	0

	A	B	C	D	E
A				B	C
B	D		D		D
C					
D		A	A		
E					

Floyd-Warshall

E11. Execute Print Path between B and C

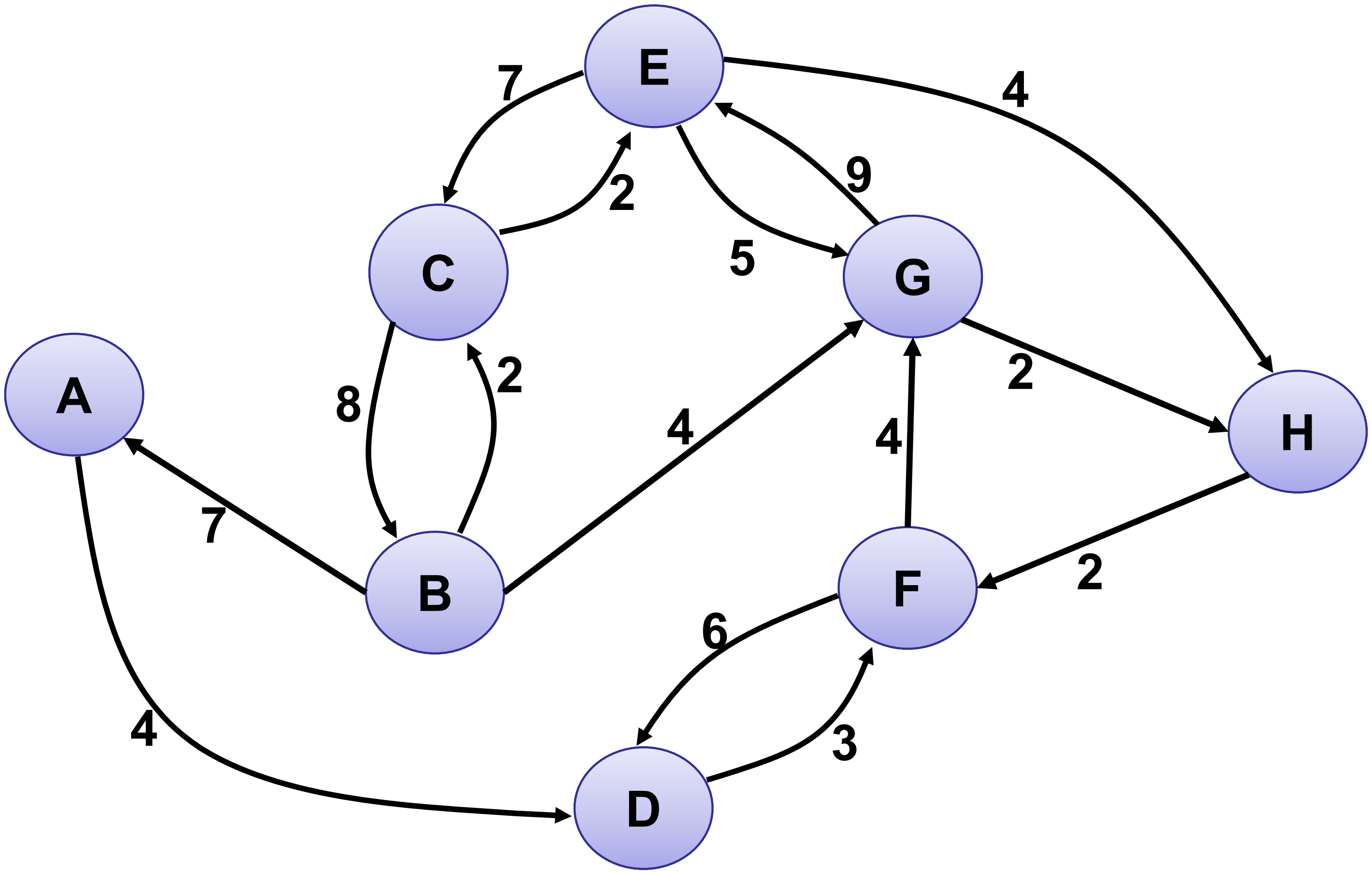
printPath (fragment)

```
private void printPath(int i, int j)
{
 int k = P[i][j];
 if (k>0) {
  printPath (i, k);
  System.out.print ('-' + k);
  printPath (k, j);
 }
}

System.out.print (departure);
printPath (departure, arrival);
System.out.println ('-' + arrival);
```

	A	B	C	D	E
A				B	C
B	D		D		D
C					
D		A	A		
E					

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph



Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Initialization.

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	∞	∞	∞
B	7	0	2	∞	∞	∞	4	∞
C	∞	8	0	∞	2	∞	∞	∞
D	∞	∞	∞	0	∞	3	∞	∞
E	∞	∞	7	∞	0	∞	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	∞	∞	∞	∞	9	∞	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 1 (node A).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	∞	∞	∞
B	7	0	2	**11A**	∞	∞	4	∞
C	∞	8	0	∞	2	∞	∞	∞
D	∞	∞	∞	0	∞	3	∞	∞
E	∞	∞	7	∞	0	∞	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	∞	∞	∞	∞	9	∞	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 2 (node B).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	∞	∞	∞
B	7	0	2	**11A**	∞	∞	4	∞
C	**15B**	8	0	**19B**	2	∞	**12B**	∞
D	∞	∞	∞	0	∞	3	∞	∞
E	∞	∞	7	∞	0	∞	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	∞	∞	∞	∞	9	∞	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 3 (node C).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	∞	∞	∞
B	7	0	2	**11A**	**4C**	∞	4	∞
C	**15B**	8	0	**19B**	2	∞	**12B**	∞
D	∞	∞	∞	0	∞	3	∞	∞
E	**22C**	**15C**	7	**26C**	0	∞	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	∞	∞	∞	∞	9	∞	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 4 (node D).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	**07D**	∞	∞
B	7	0	2	**11A**	**4C**	**14D**	4	∞
C	**15B**	8	0	**19B**	2	**22D**	**12B**	∞
D	∞	∞	∞	0	∞	3	∞	∞
E	**22C**	**15C**	7	**26C**	0	**29D**	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	∞	∞	∞	∞	9	∞	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 5 (node E).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	**07D**	∞	∞
B	7	0	2	**11A**	**4C**	**14D**	4	**08E**
C	**15B**	8	0	**19B**	2	**22D**	**07E**	**06E**
D	∞	∞	∞	0	∞	3	∞	∞
E	**22C**	**15C**	7	**26C**	0	**29D**	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	**31E**	**24E**	**16E**	**35E**	9	**38E**	0	2
H	∞	∞	∞	∞	∞	2	∞	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 6 (node F).

	A	B	C	D	E	F	G	H
A	0	∞	∞	4	∞	**07D**	**11F**	∞
B	7	0	2	**11A**	**4C**	**14D**	4	**08E**
C	**15B**	8	0	**19B**	2	**22D**	**07E**	**06E**
D	∞	∞	∞	0	∞	3	**07F**	∞
E	**22C**	**15C**	7	**26C**	0	**29D**	5	4
F	∞	∞	∞	6	∞	0	4	∞
G	**31E**	**24E**	**16E**	**35E**	9	**38E**	0	2
H	∞	∞	∞	**08F**	∞	2	**06F**	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 7 (node G).

	A	B	C	D	E	F	G	H
A	0	**35G**	**27G**	4	**20G**	**07D**	**11F**	**13G**
B	7	0	2	**11A**	**4C**	**14D**	4	**06G**
C	**15B**	8	0	**19B**	2	**22D**	**07E**	**06E**
D	**38G**	**31G**	**23G**	0	**16G**	3	**07F**	**09G**
E	**22C**	**15C**	7	**26C**	0	**29D**	5	4
F	**35G**	**28G**	**20G**	6	**13G**	0	4	**06G**
G	**31E**	**24E**	**16E**	**35E**	9	**38E**	0	2
H	**37G**	**30G**	**22G**	**08F**	**15G**	2	**06F**	∞

Floyd-Warshall

E12. Execute the Floyd-Warshall algorithm on this graph

- Iteration 8 (node H).

	A	B	C	D	E	F	G	H
A	0	**35G**	**27G**	4	**20G**	**07D**	**11F**	**13G**
B	7	0	2	**11A**	**4C**	**08H**	4	**06G**
C	**15B**	8	0	**14H**	2	**08H**	**07E**	**06E**
D	**38G**	**31G**	**23G**	0	**16G**	3	**07F**	**09G**
E	**22C**	**15C**	7	**12H**	0	**06H**	5	4
F	**35G**	**28G**	**20G**	6	**13G**	0	4	**06G**
G	**31E**	**24E**	**16E**	**10H**	9	**04H**	0	2
H	**37G**	**30G**	**22G**	**08F**	**15G**	2	**06F**	∞

Unit 4

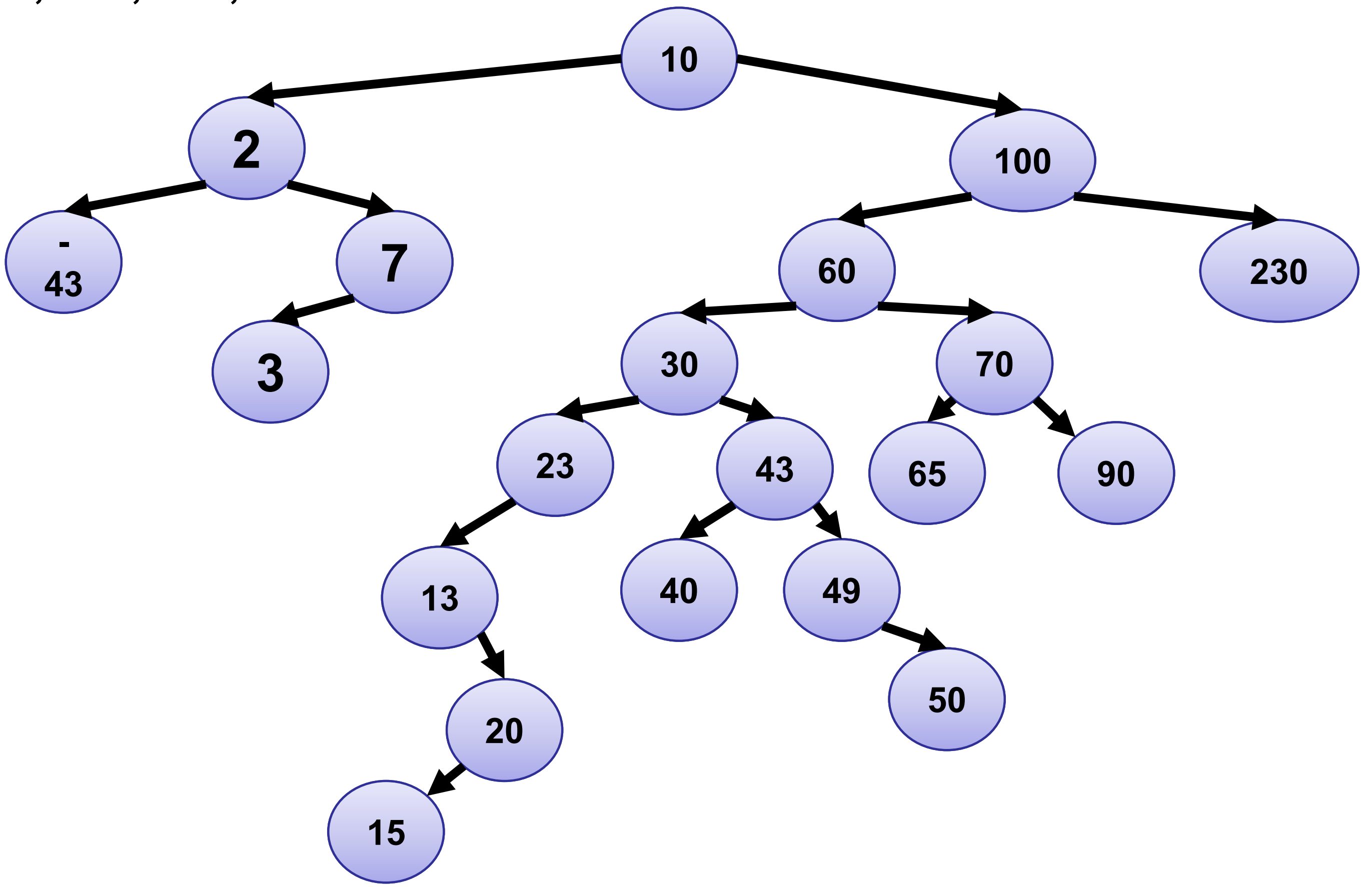
Hierarchical Structures

BST and AVL trees

BST Trees

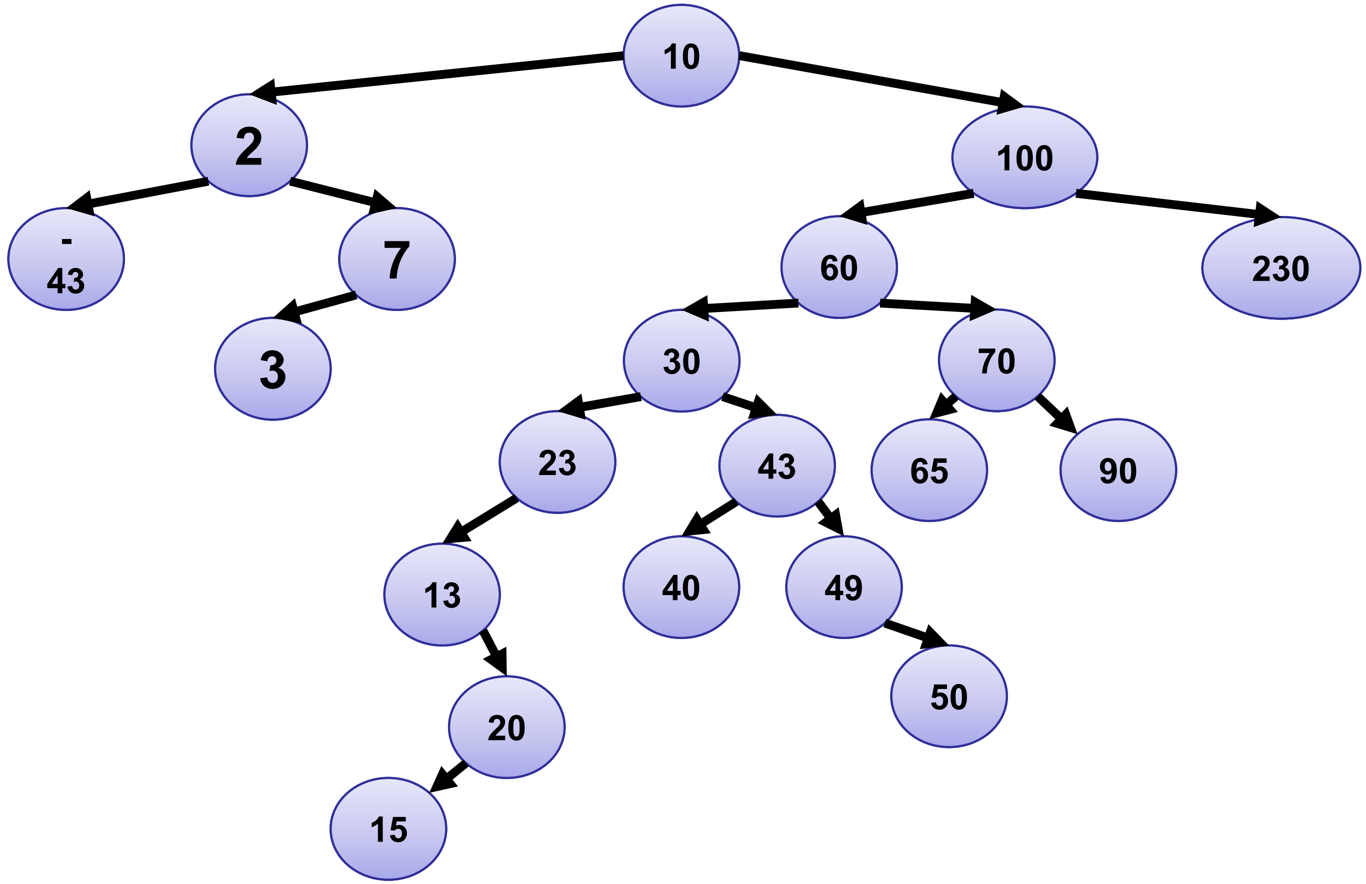
E1. Create a Binary Search Tree and add the following elements

- 10, 100, 60, 30, 2, -43, 70, 90, 23, 43, 65, 13, 230, 49, 7, 40, 50, 20, 15, 3



BST Trees

E2. Navigate in preorder, inorder and postorder.



Preorder: 10, 2, -43, 7, 3, 100, 60, 30, 23, 13, 20, 15, 43, 40, 49, 50, 70, 65, 90, 230

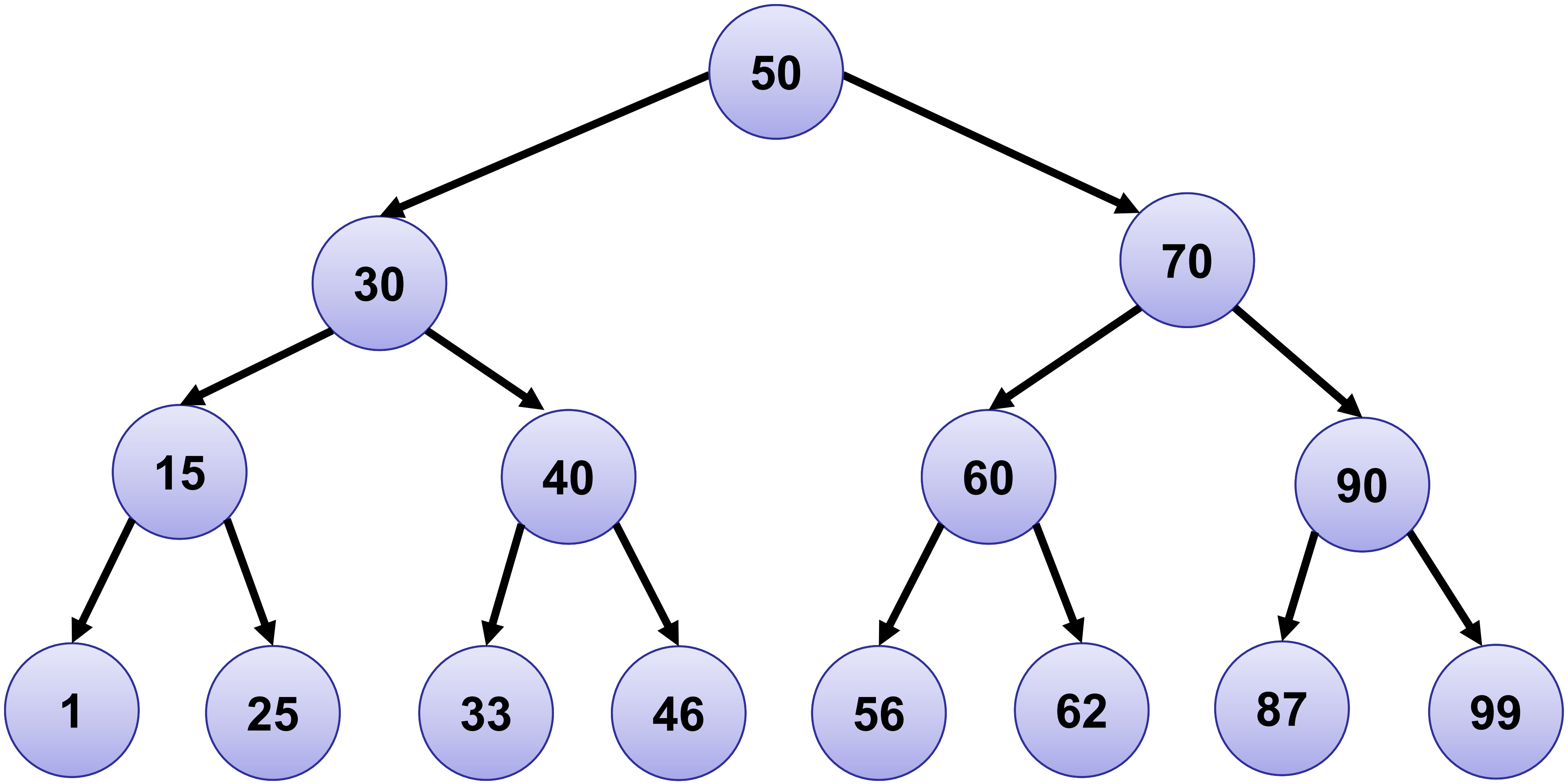
Inorder: -43, 2, 3, 7, 10, 13, 15, 20, 23, 30, 40, 43, 49, 50, 60, 65, 70, 90, 100, 230

Postorder: -43, 3, 7, 2 15, 20, 13, 23, 40, 50, 49, 43, 30, 65, 90, 70, 60, 230, 100, 10

BST Trees

E3. Delete the next elements

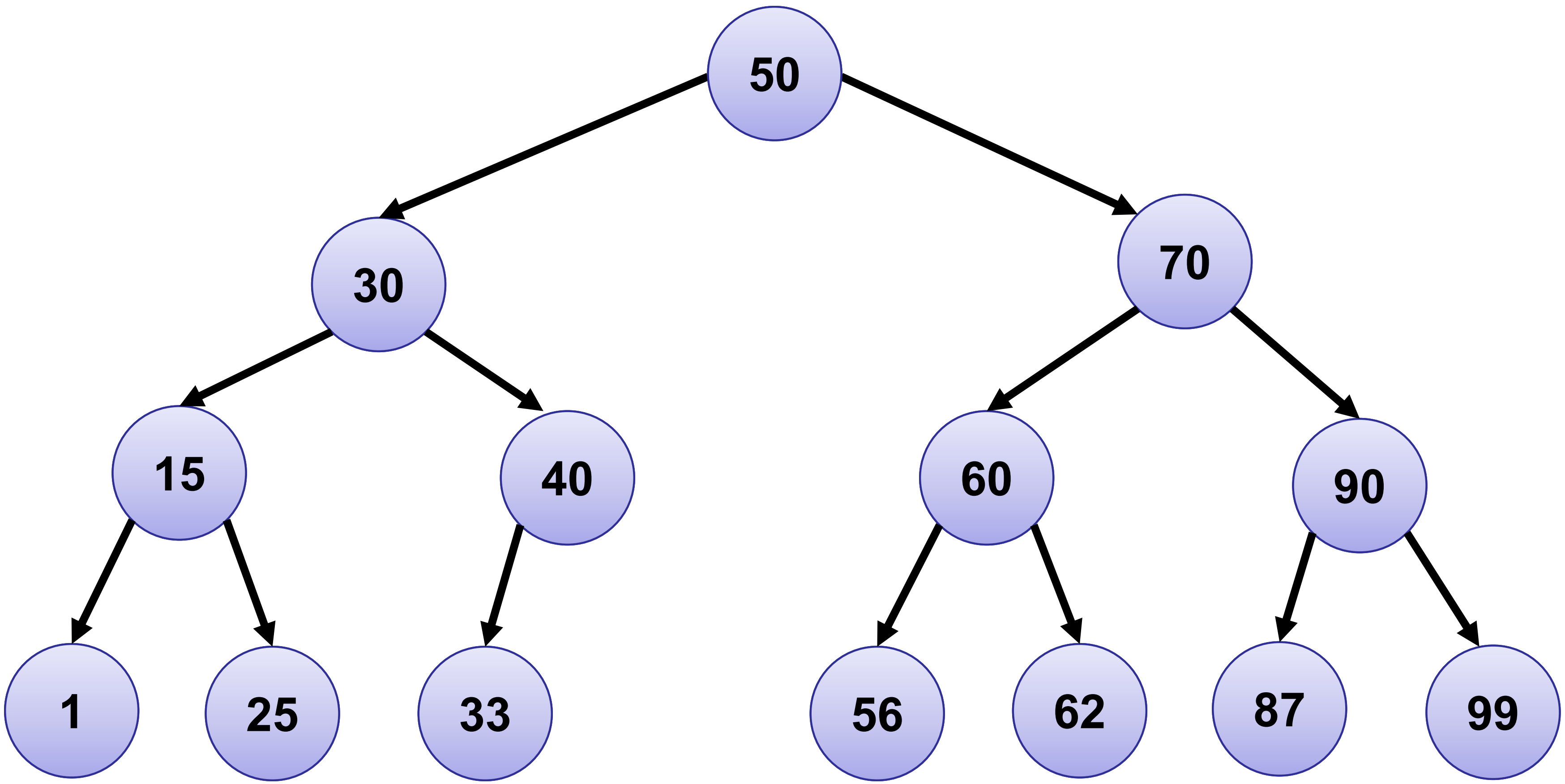
- 46



BST Trees

E3. Delete the next elements

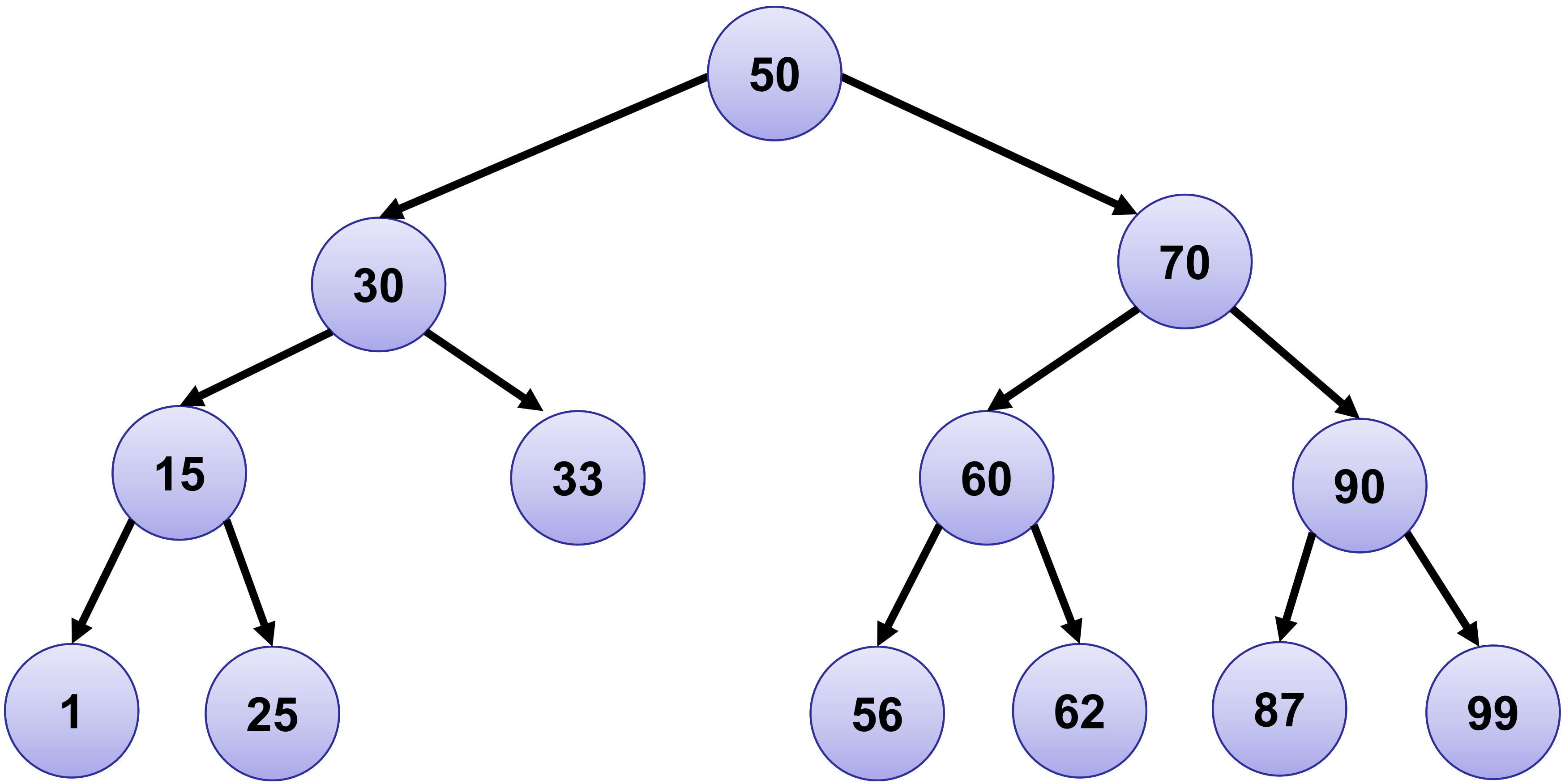
- 40



BST Trees

E3. Delete the next elements

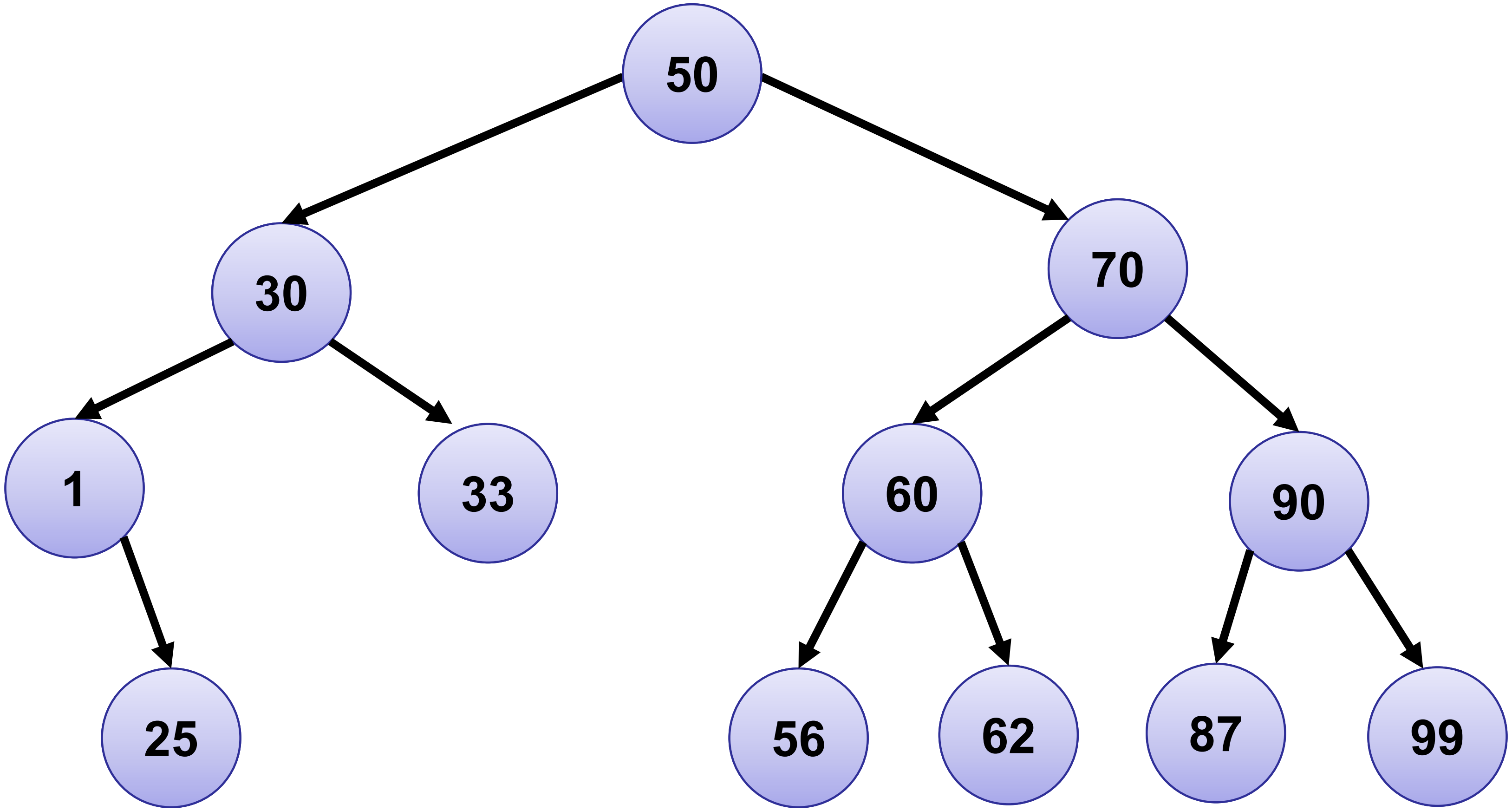
- 15



BST Trees

E3. Delete the next elements

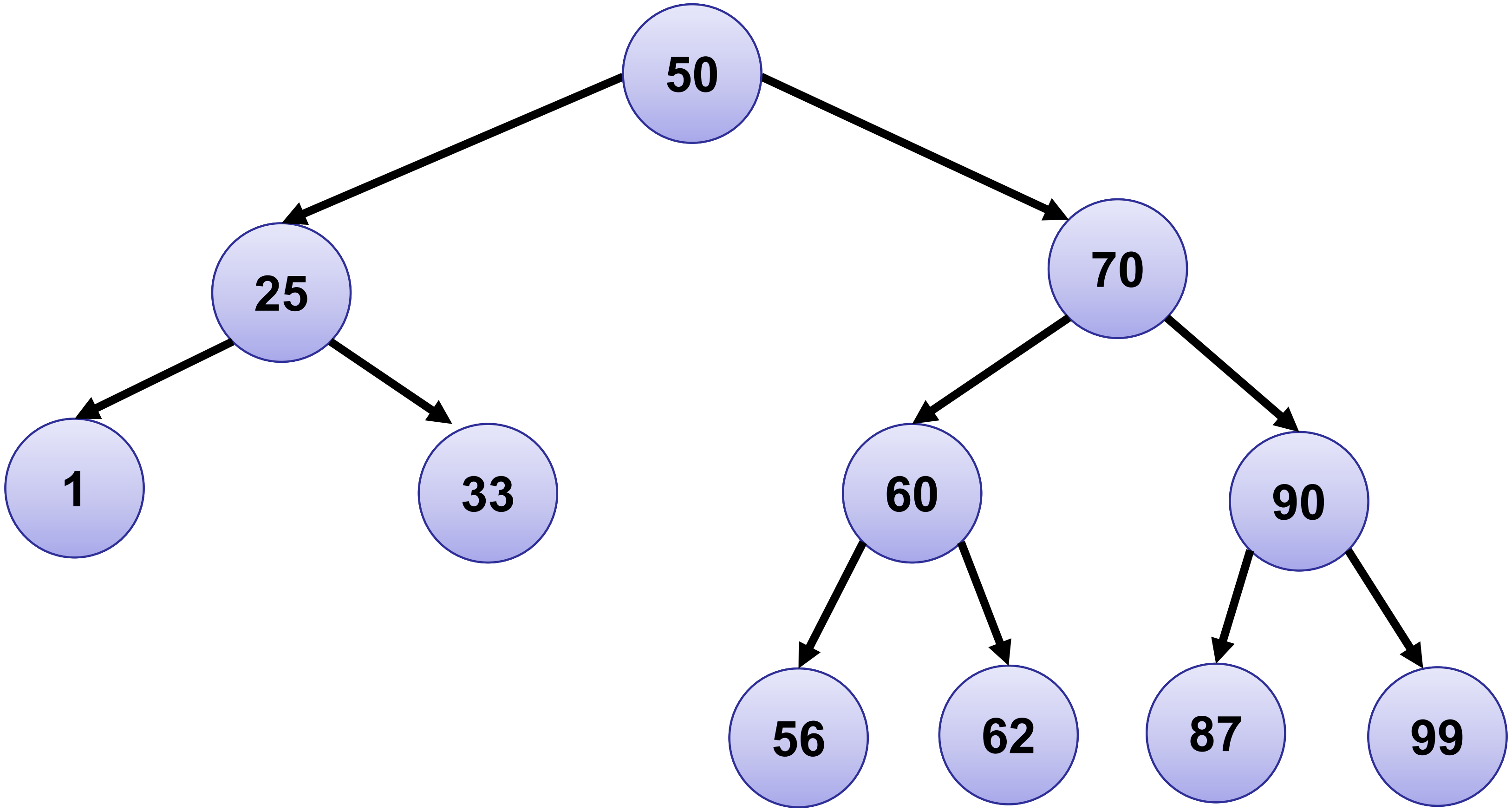
- 30



BST Trees

E3. Delete the next elements

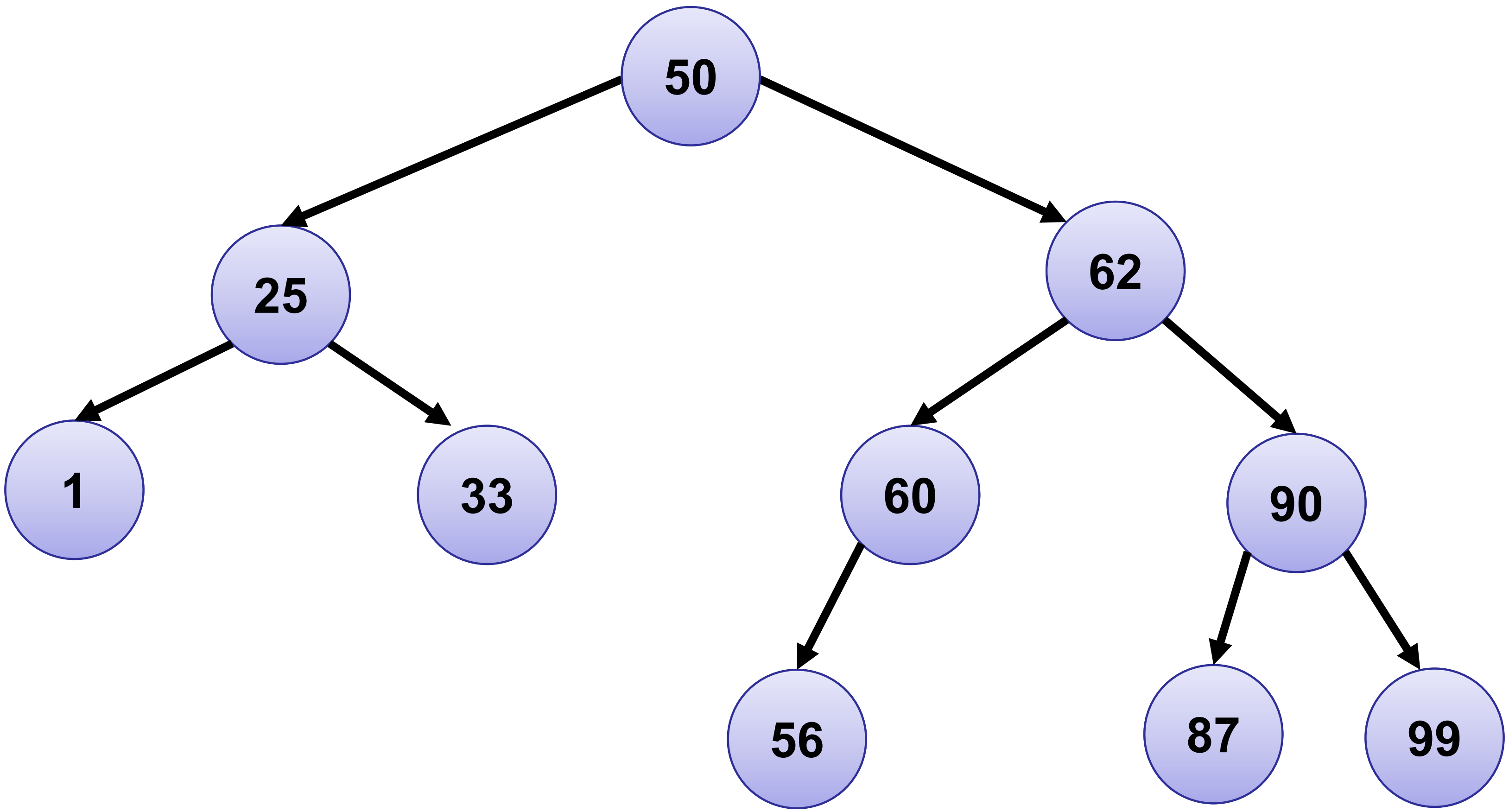
- 70



BST Trees

E3. Delete the next elements

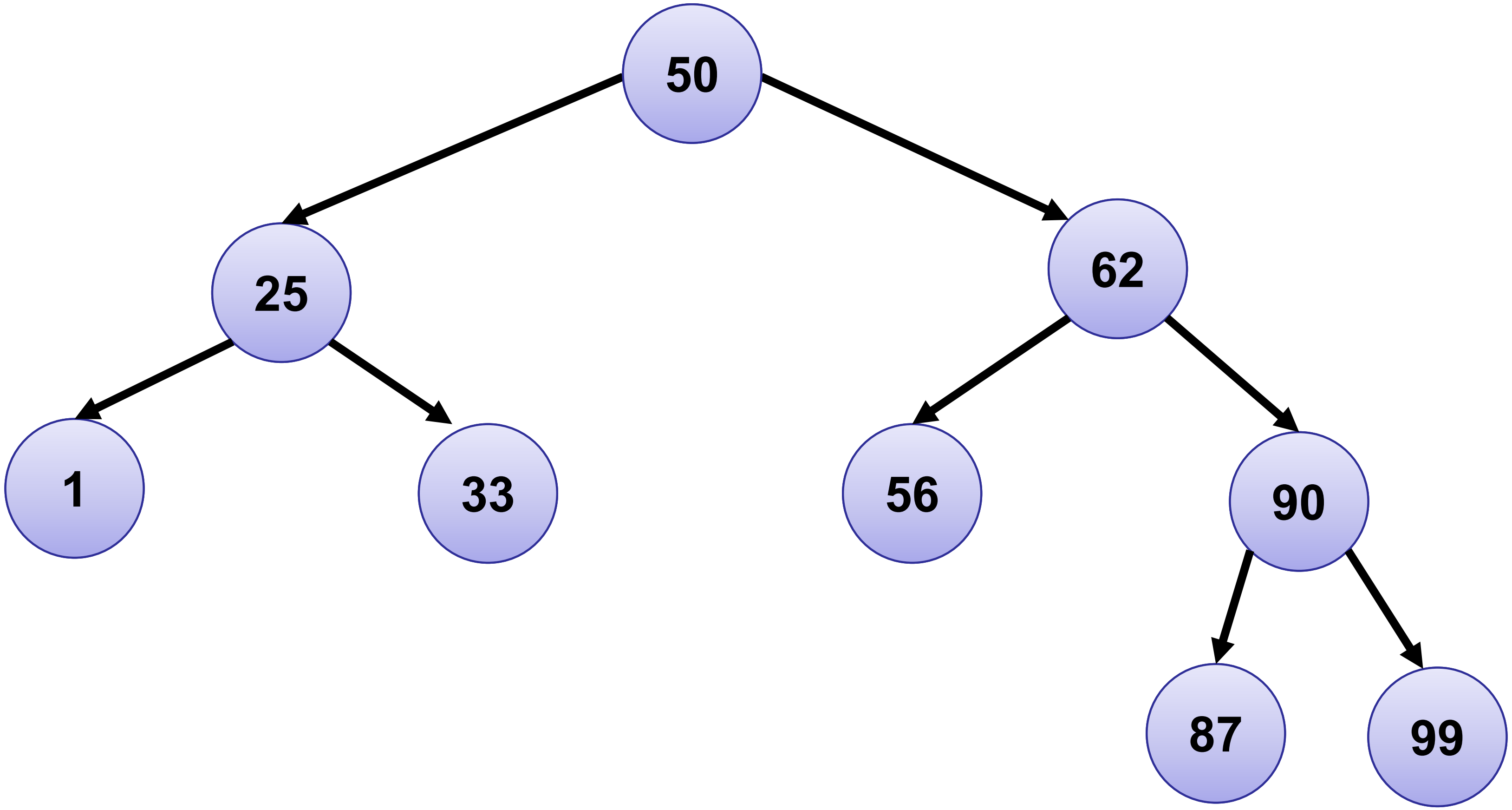
- 60



BST Trees

E3. Delete the next elements

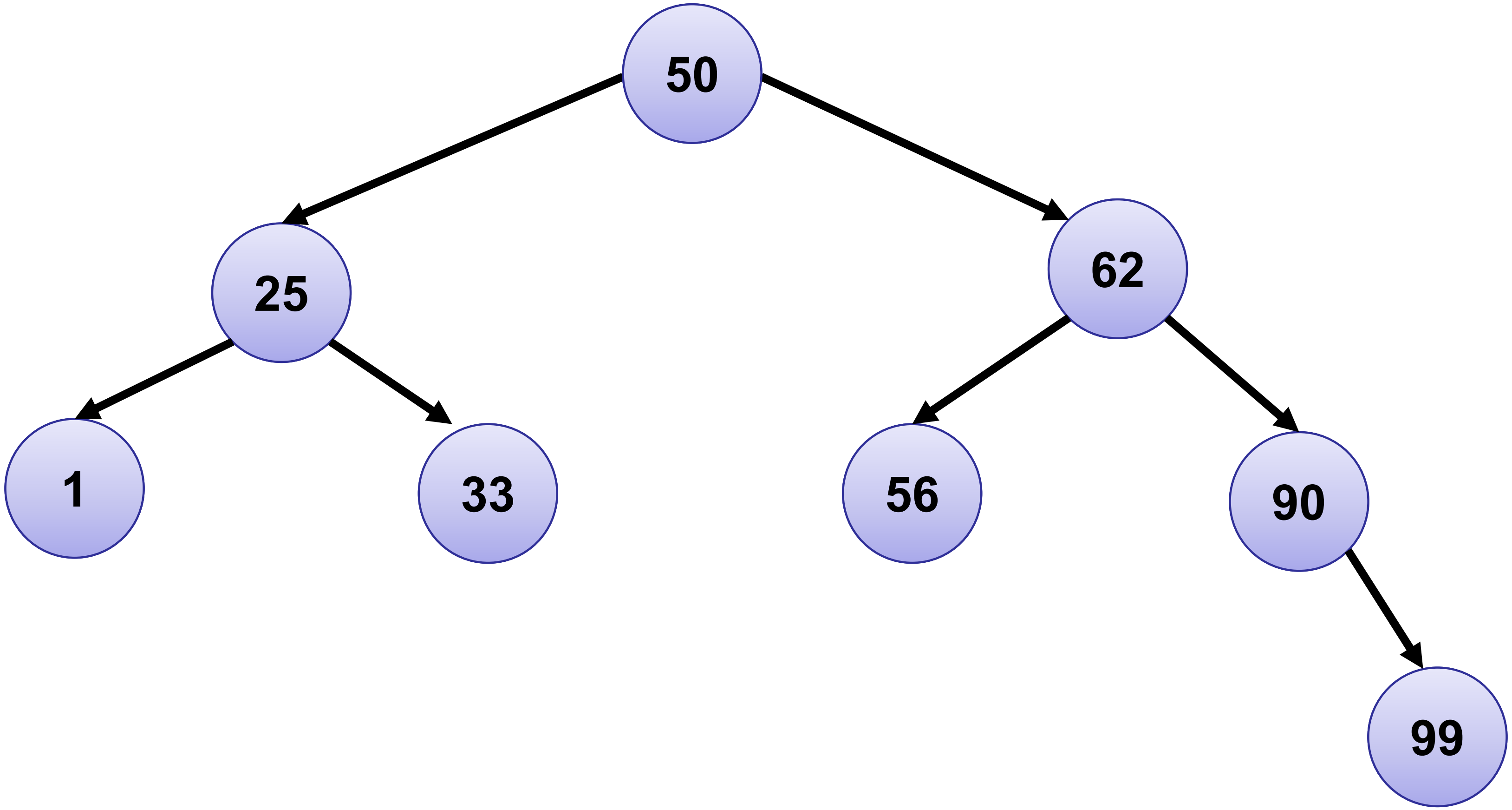
- 87



BST Trees

E3. Delete the next elements

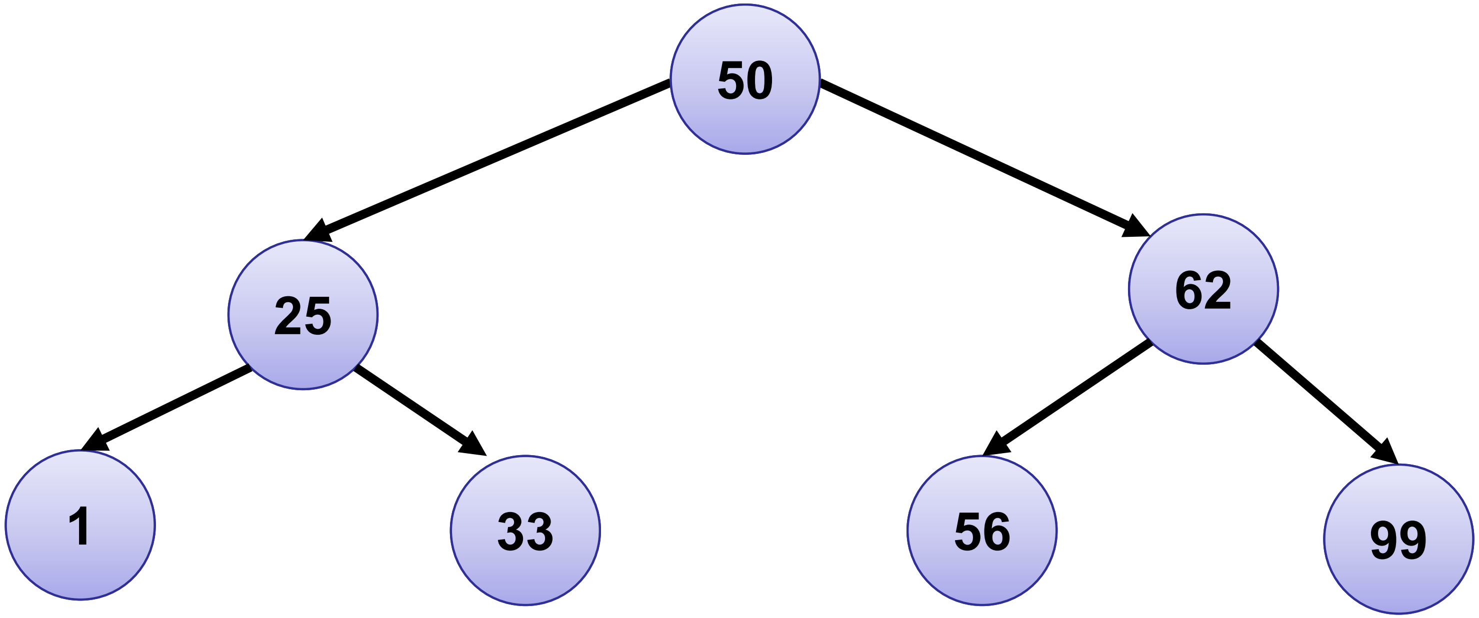
- 90



BST Trees

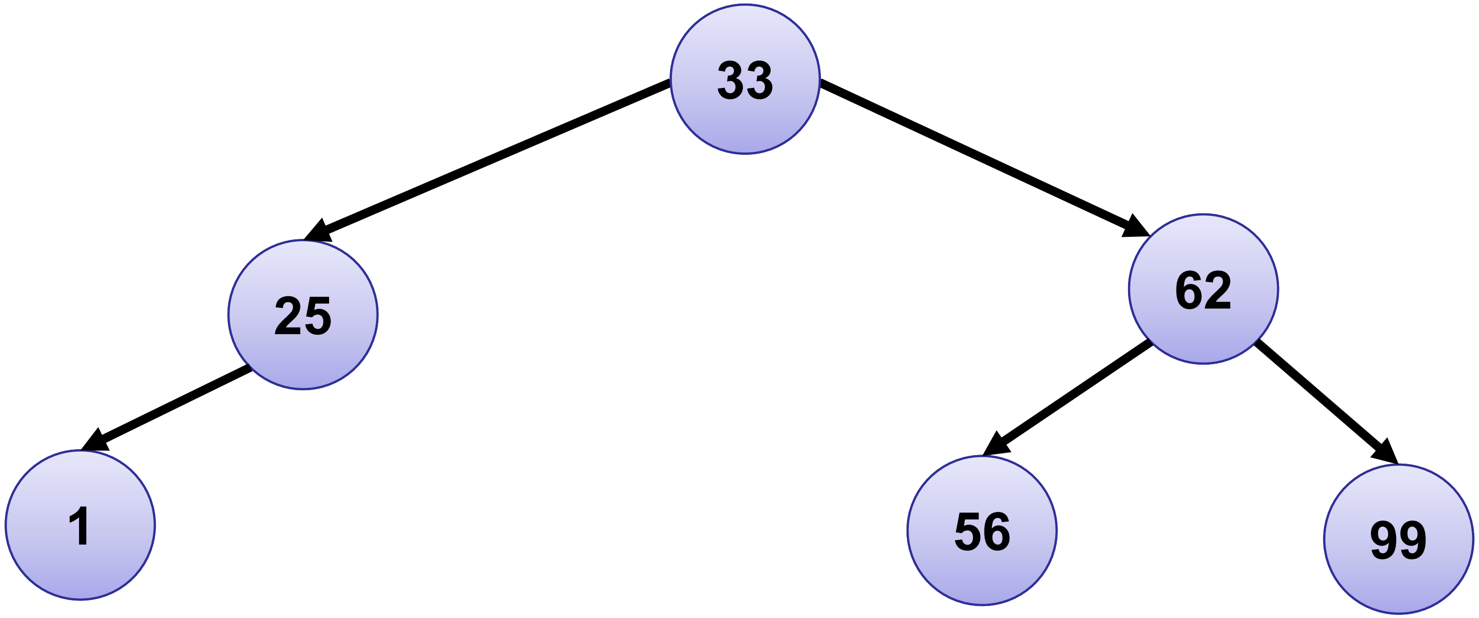
E3. Delete the next elements

- 50



BST Trees

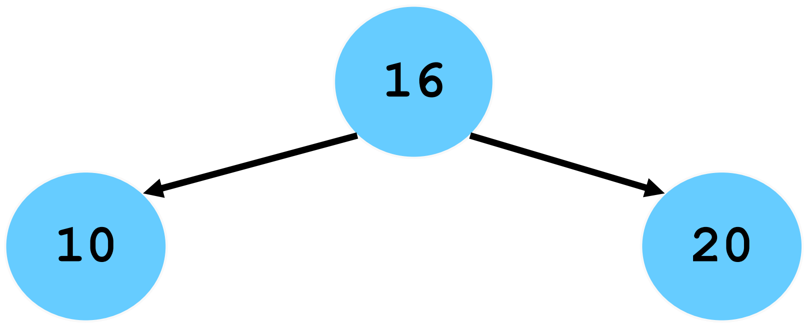
E3. End



AVL Trees

E4. Create an AVL tree and add the following elements

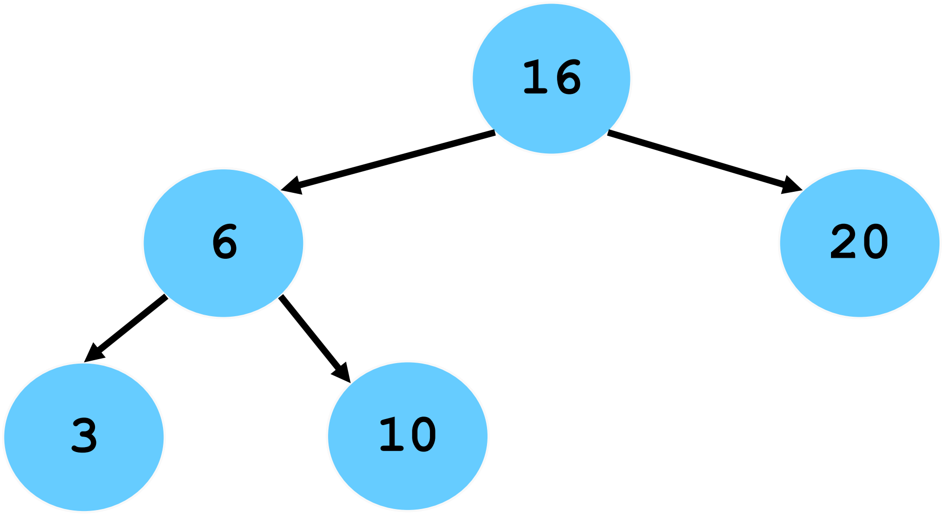
- 10, 16, 20



AVL Trees

E4. Insert the following elements

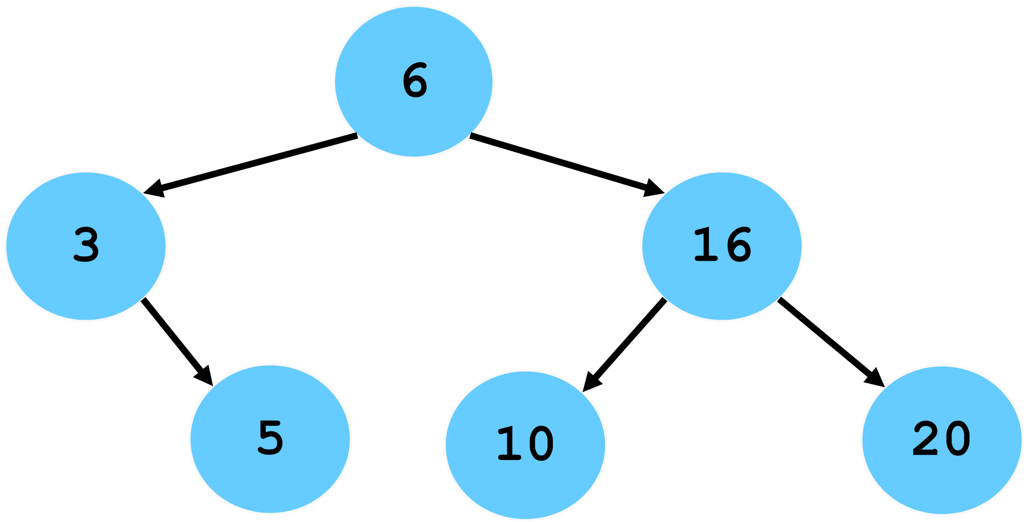
- 6, 3



AVL Trees

E4. Insert the following elements

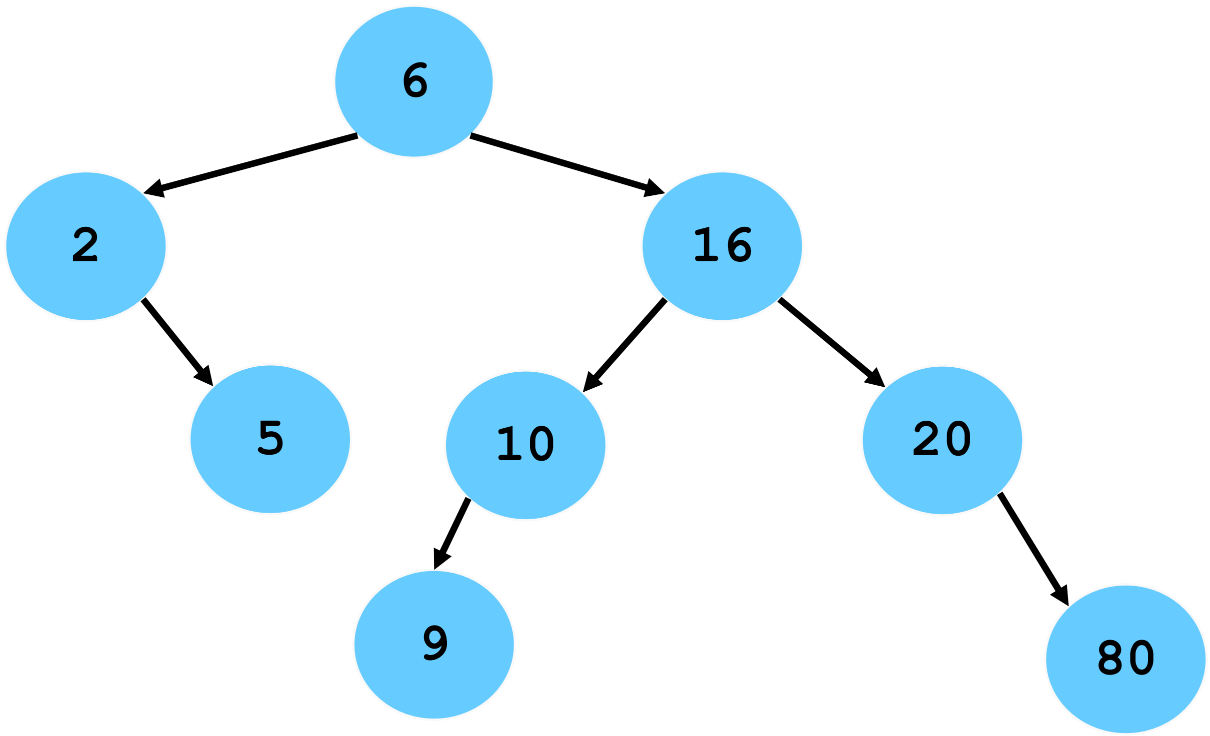
- 5



AVL Trees

E4. Insert the following elements

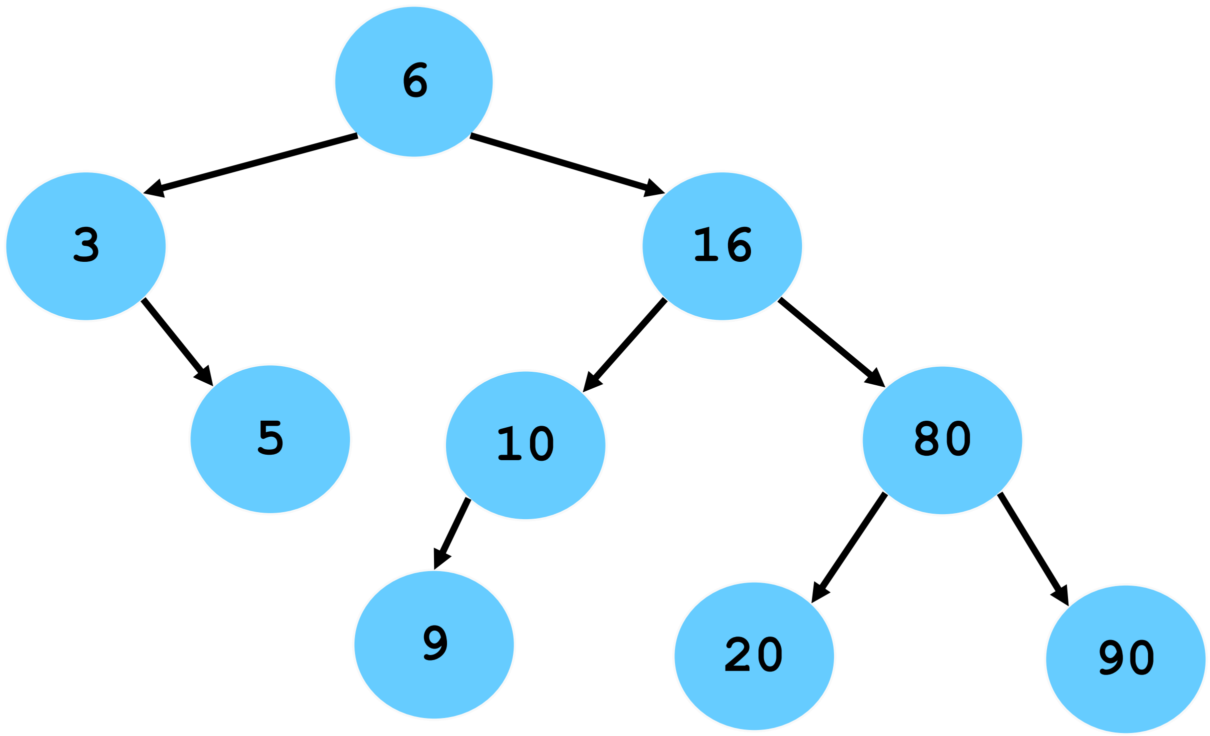
- 9, 80



AVL Trees

E4. Insert the following elements

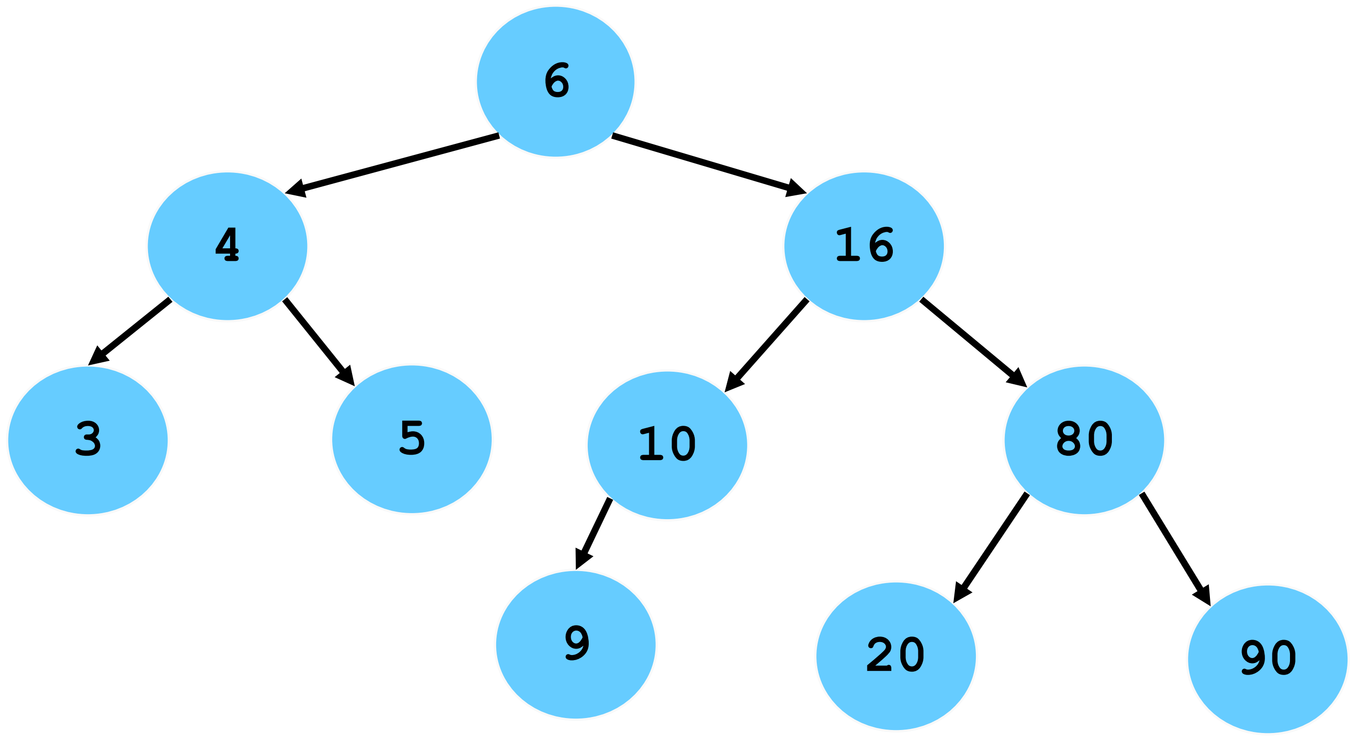
- 90



AVL Trees

E4. Insert the following elements

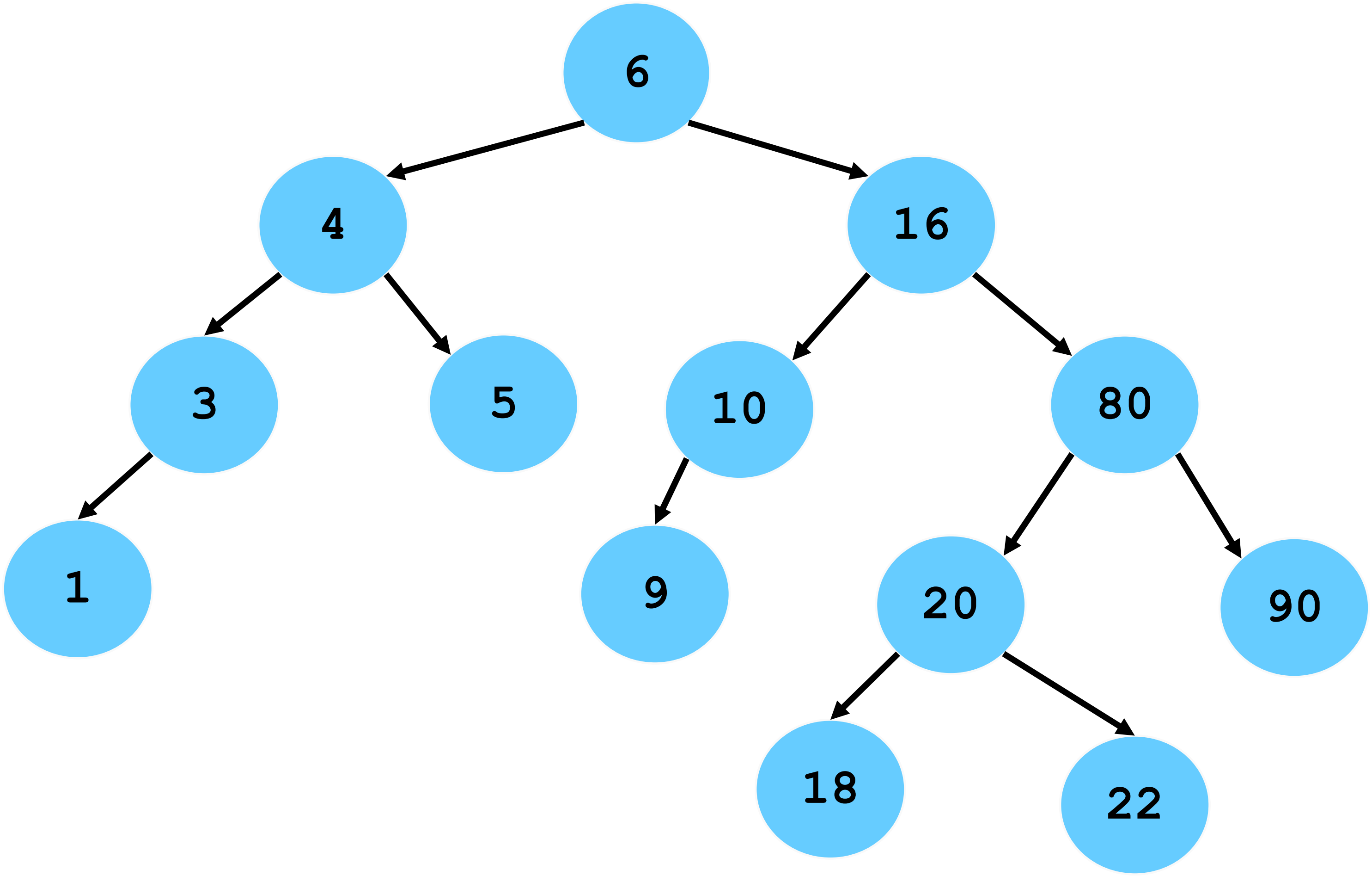
- 4



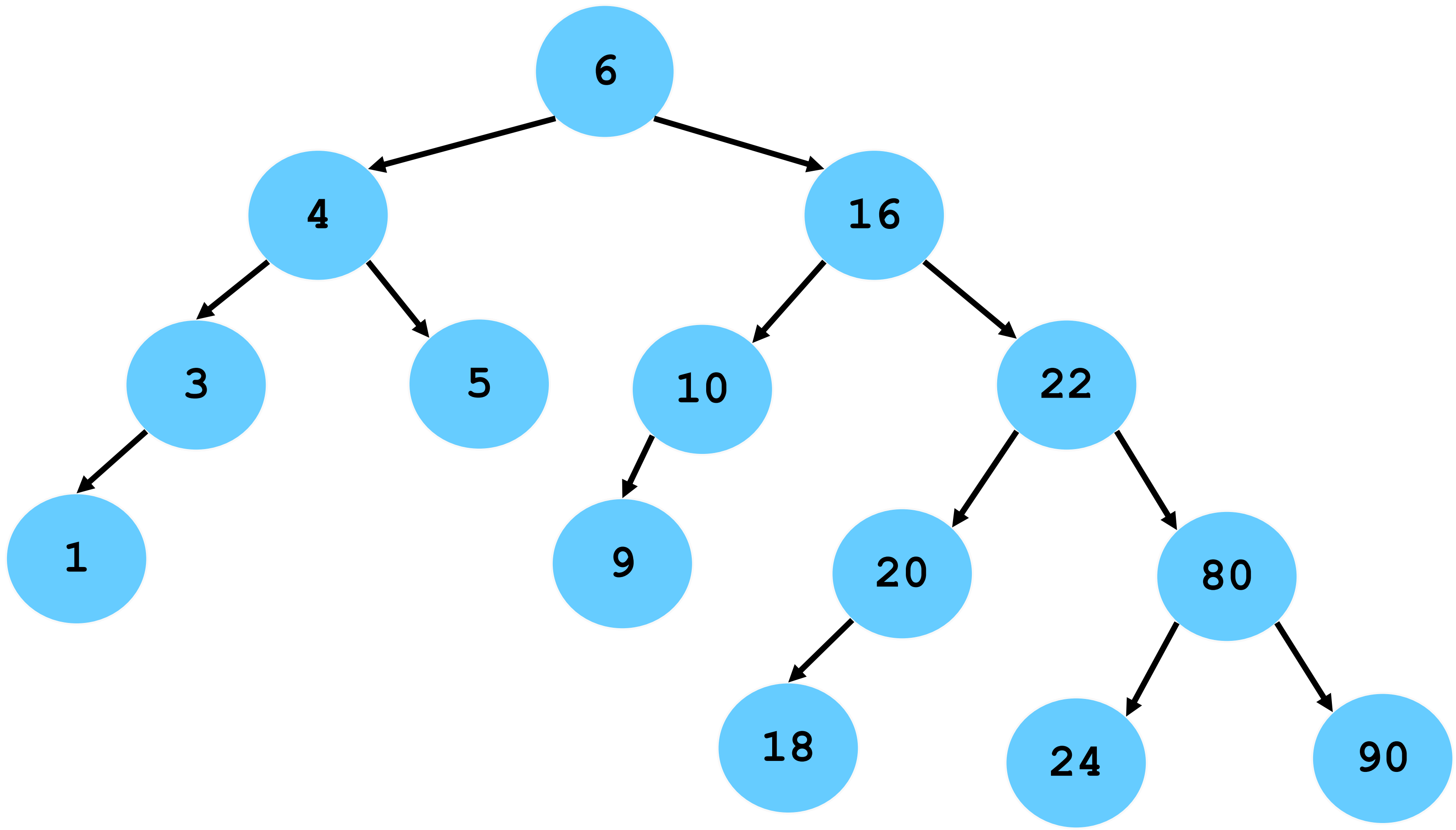
AVL Trees

E4. Insert the following elements

- 1, 18, 22



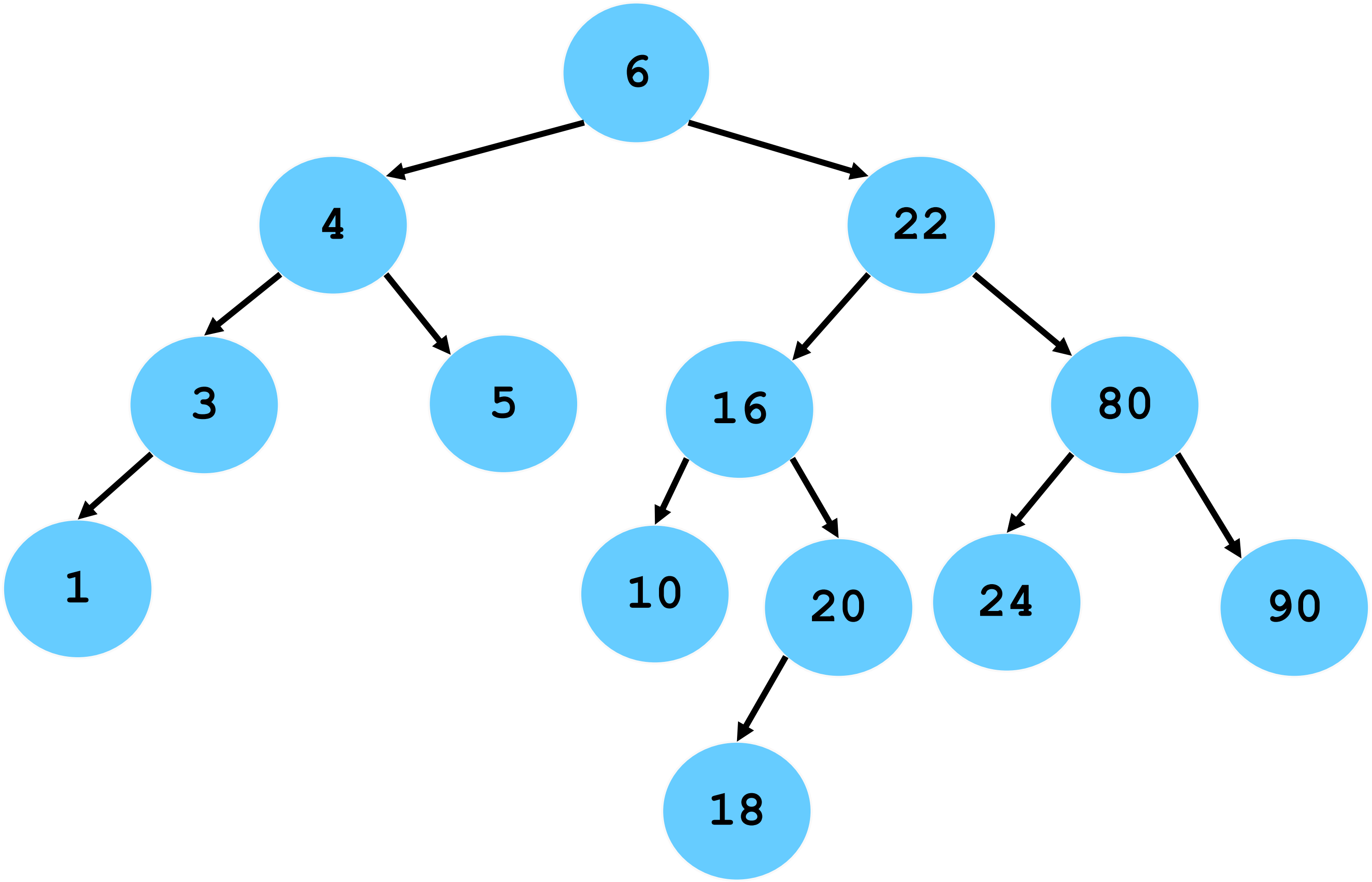
E4. Insert 24



AVL Trees

E5. Delete the following elements

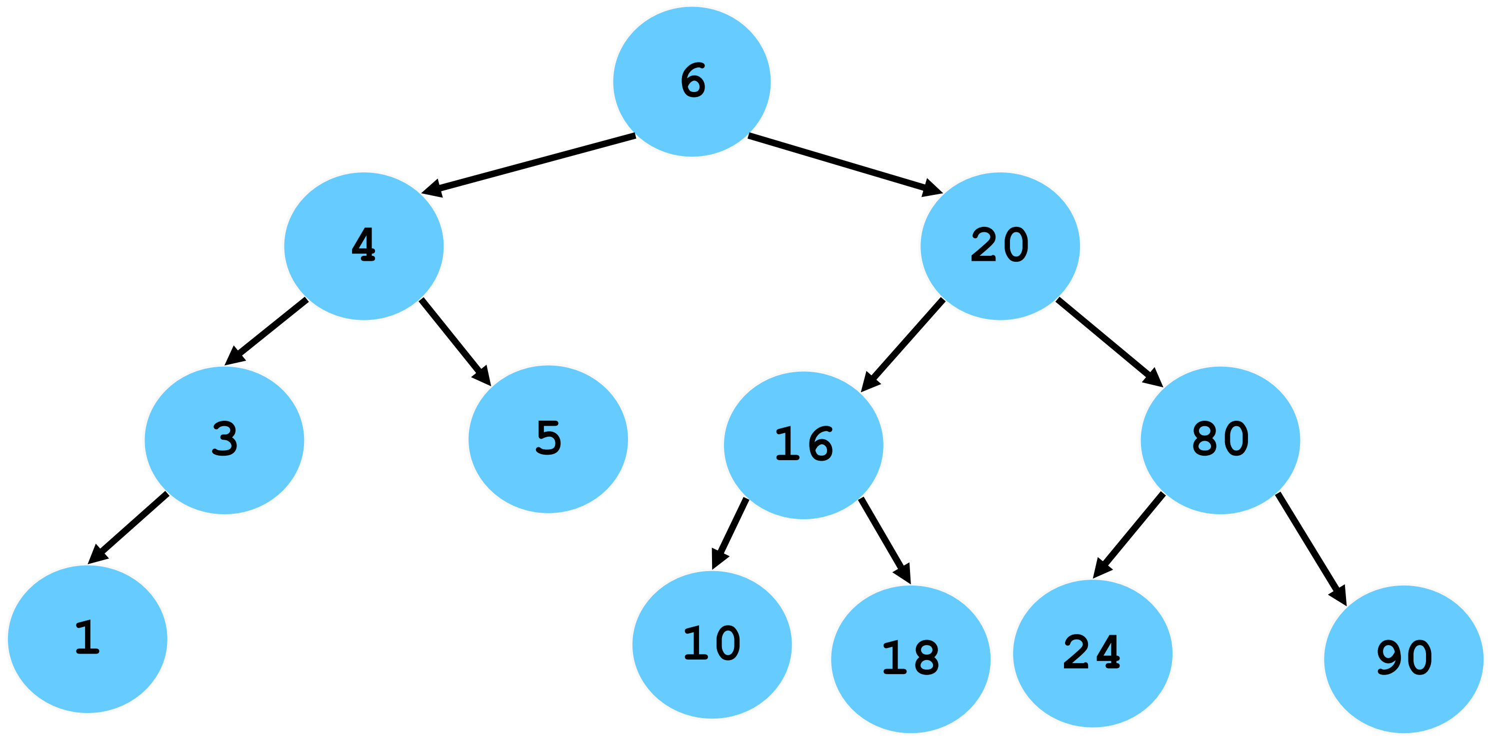
- 9



AVL Trees

E5. Delete the following elements

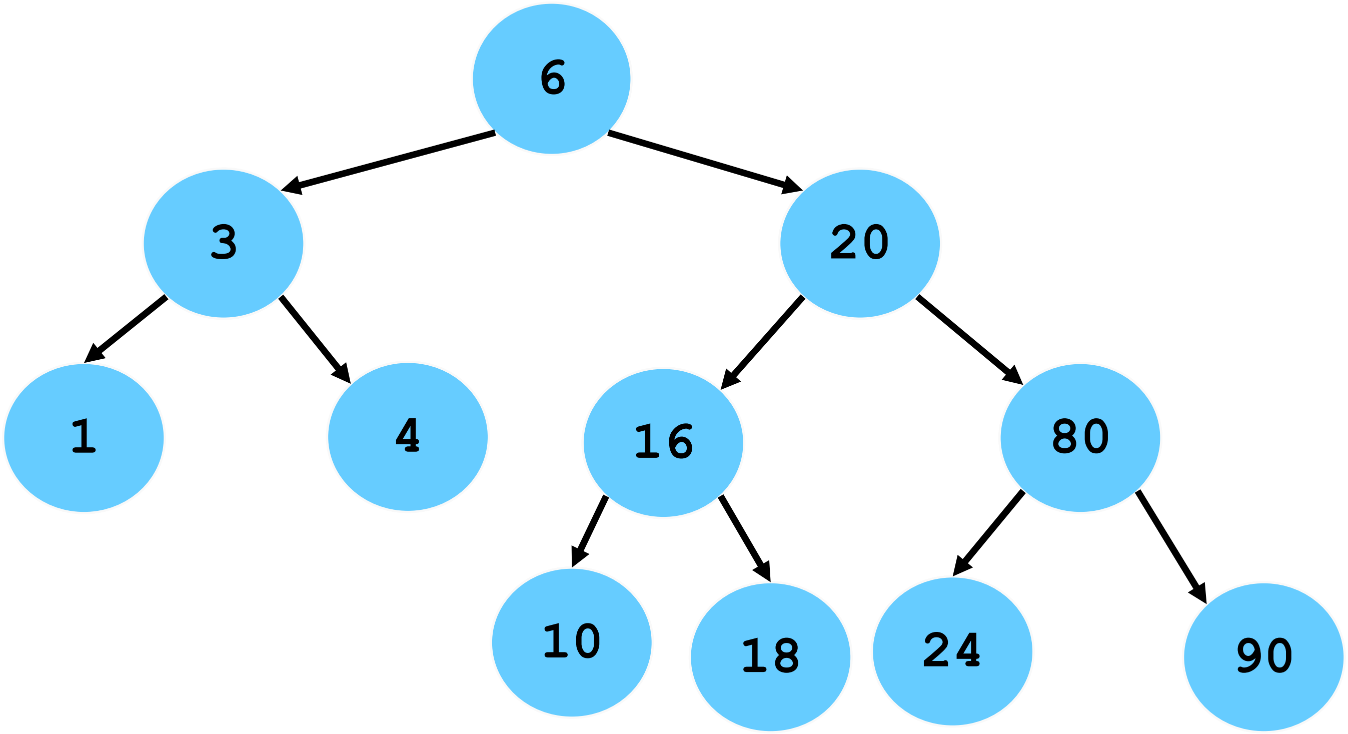
- 22



AVL Trees

E5. Delete the following elements

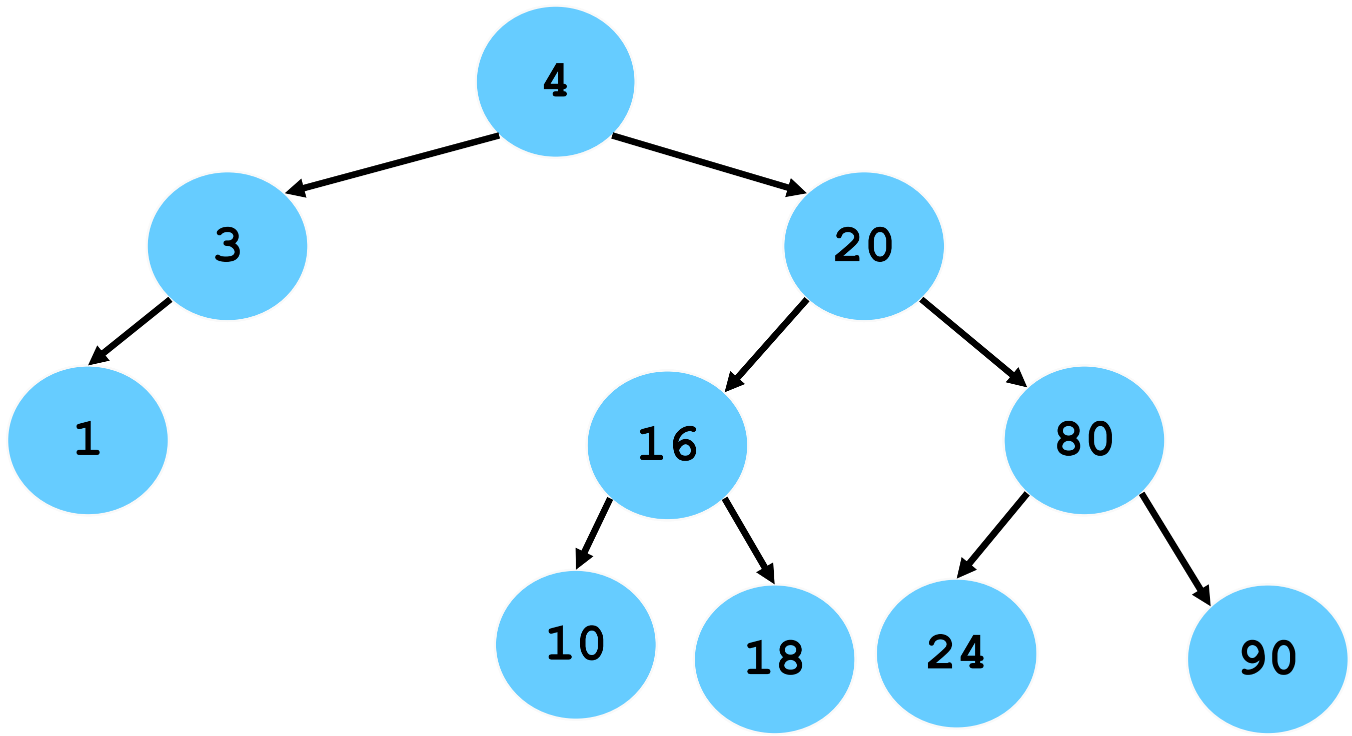
- 5



AVL Trees

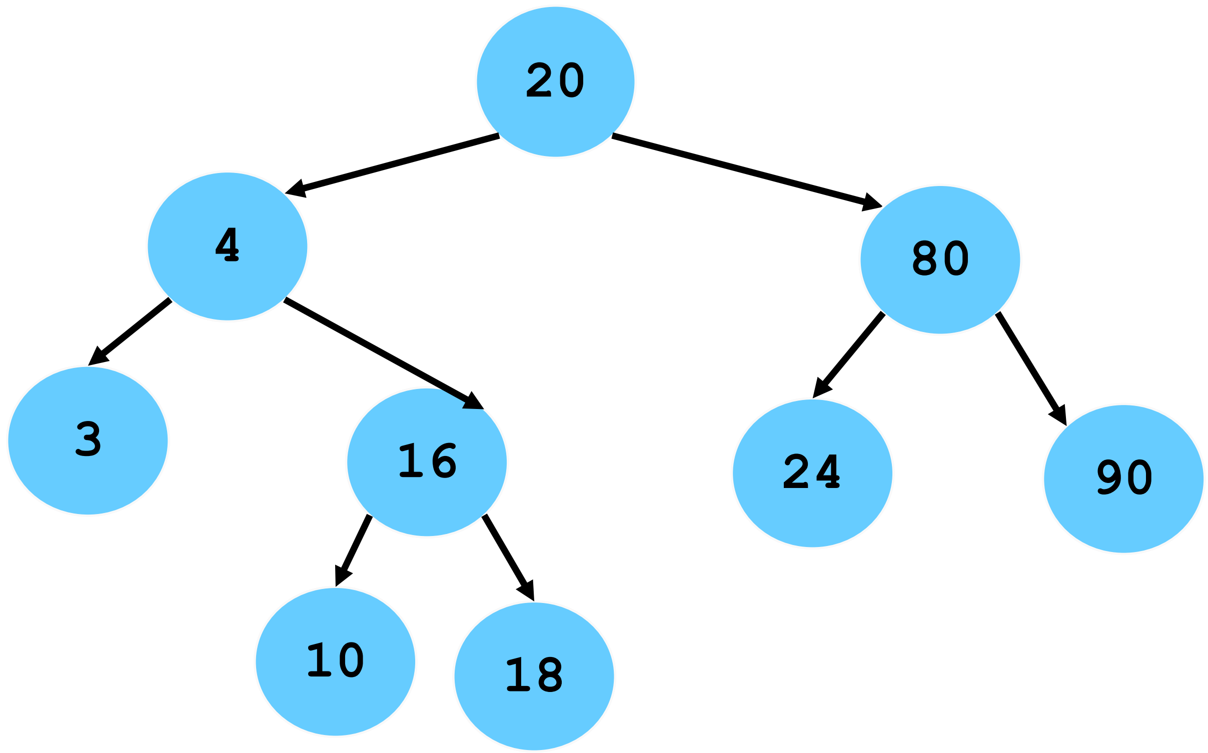
E5. Delete the following elements

- 6



E5. Delete the following elements

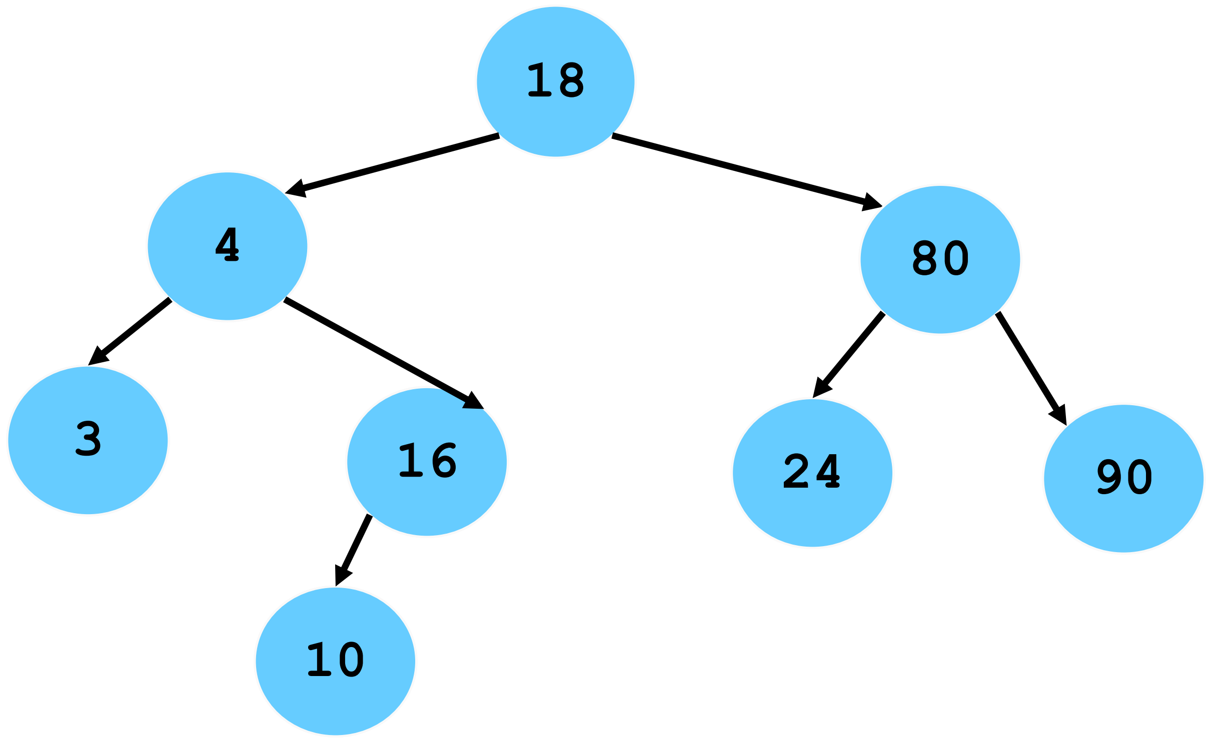
- 1



AVL Trees

E5. Delete the following elements

- 20



E5. End

Unit 5

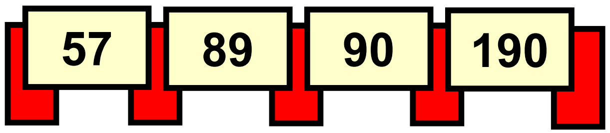
Hierarchical Structures

B Trees & Priority Queues

B Trees

E1. Create a B2 and insert the following elements

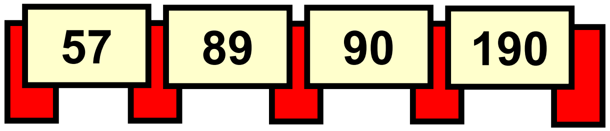
- 190, 57, 89, 90

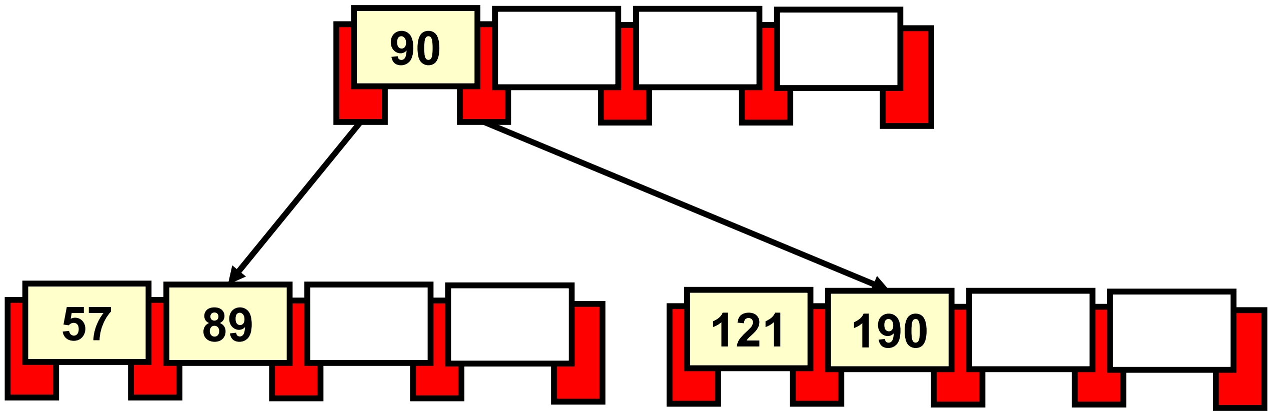


B Trees

E1. Insert the following elements

- 121

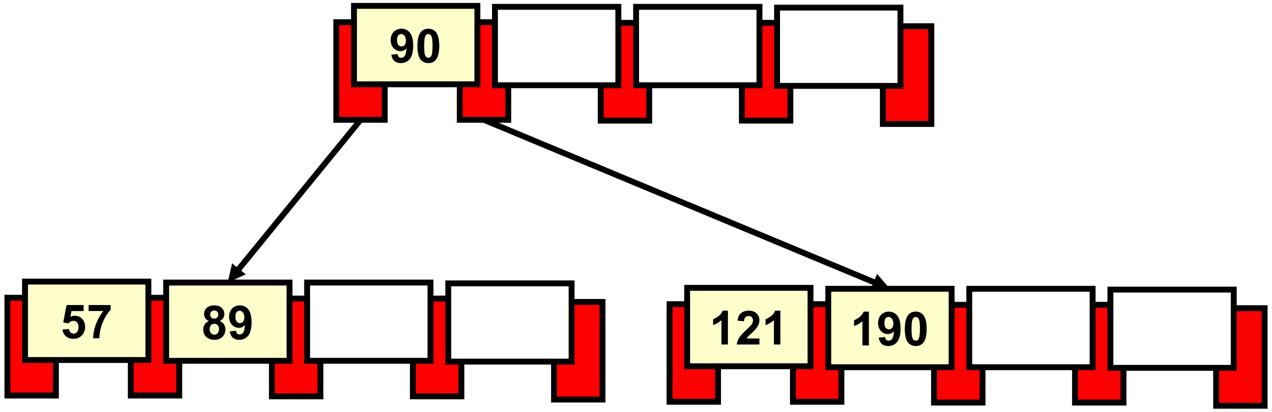


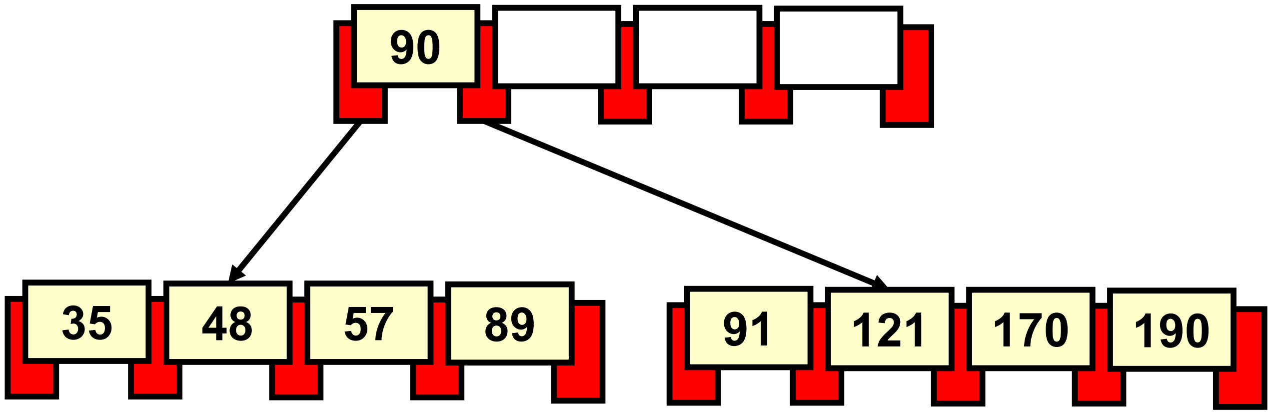


B Trees

E1. Insert the following elements

- 170, 35, 48, 91

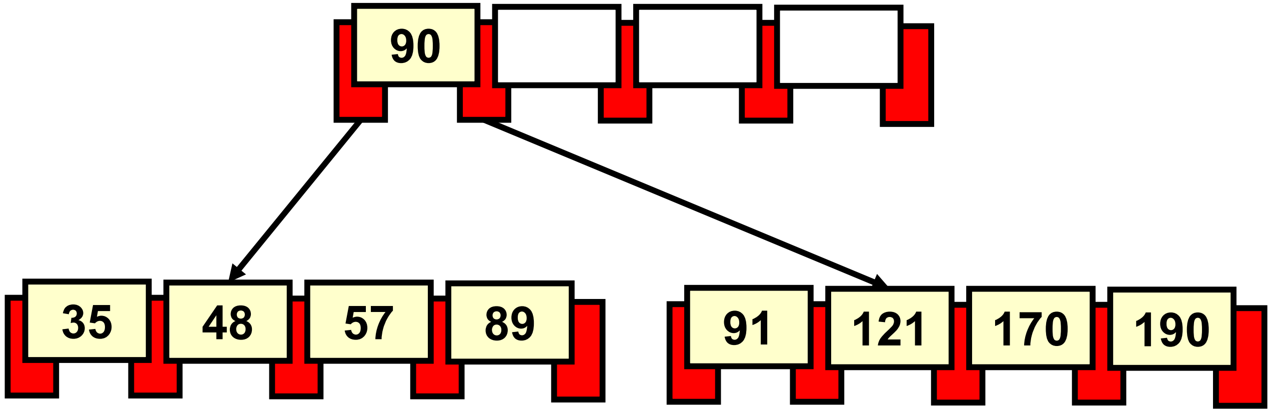


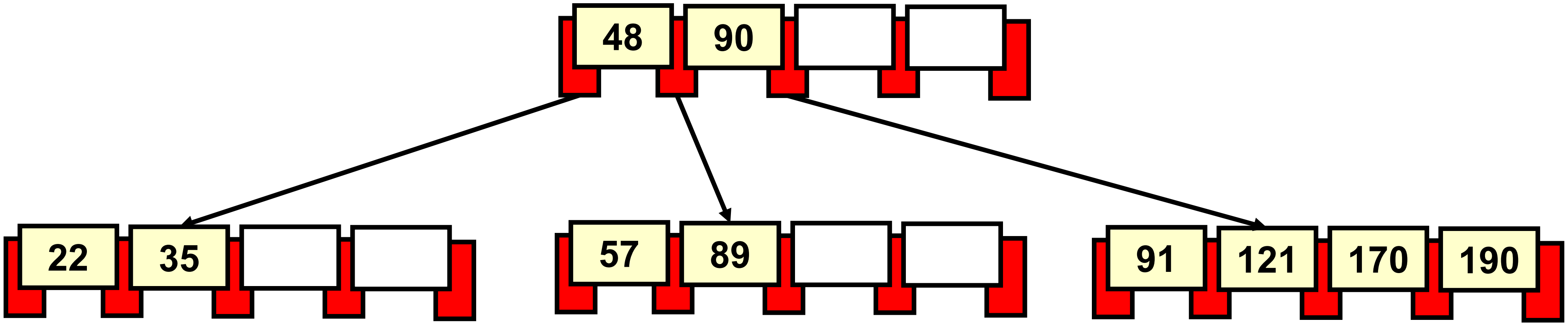


B Trees

E1. Insert the following elements

- 22

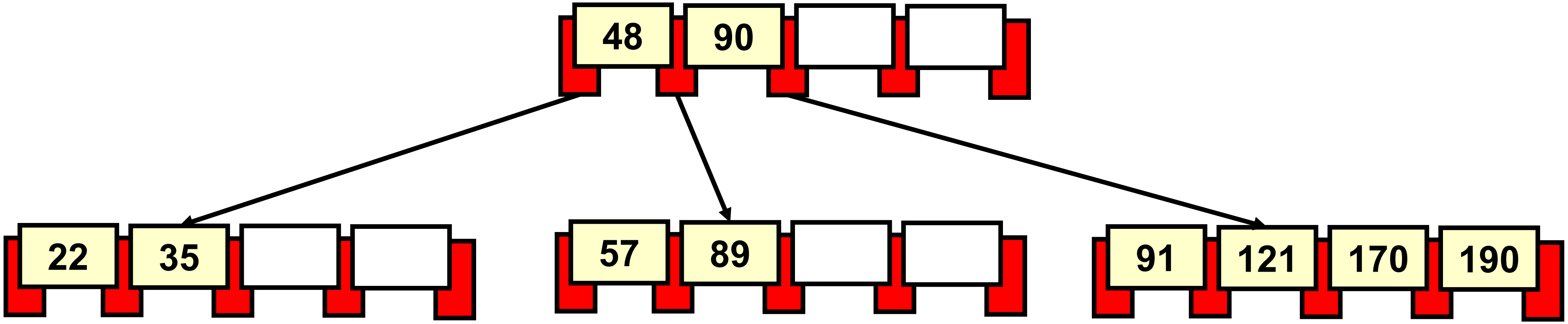


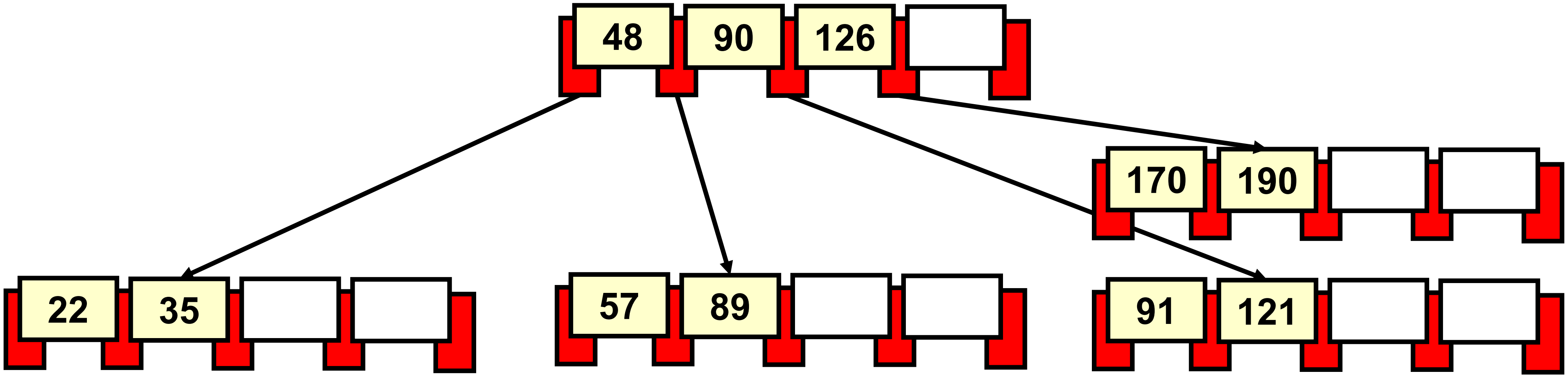


B Trees

E1. Insert the following elements

- 126

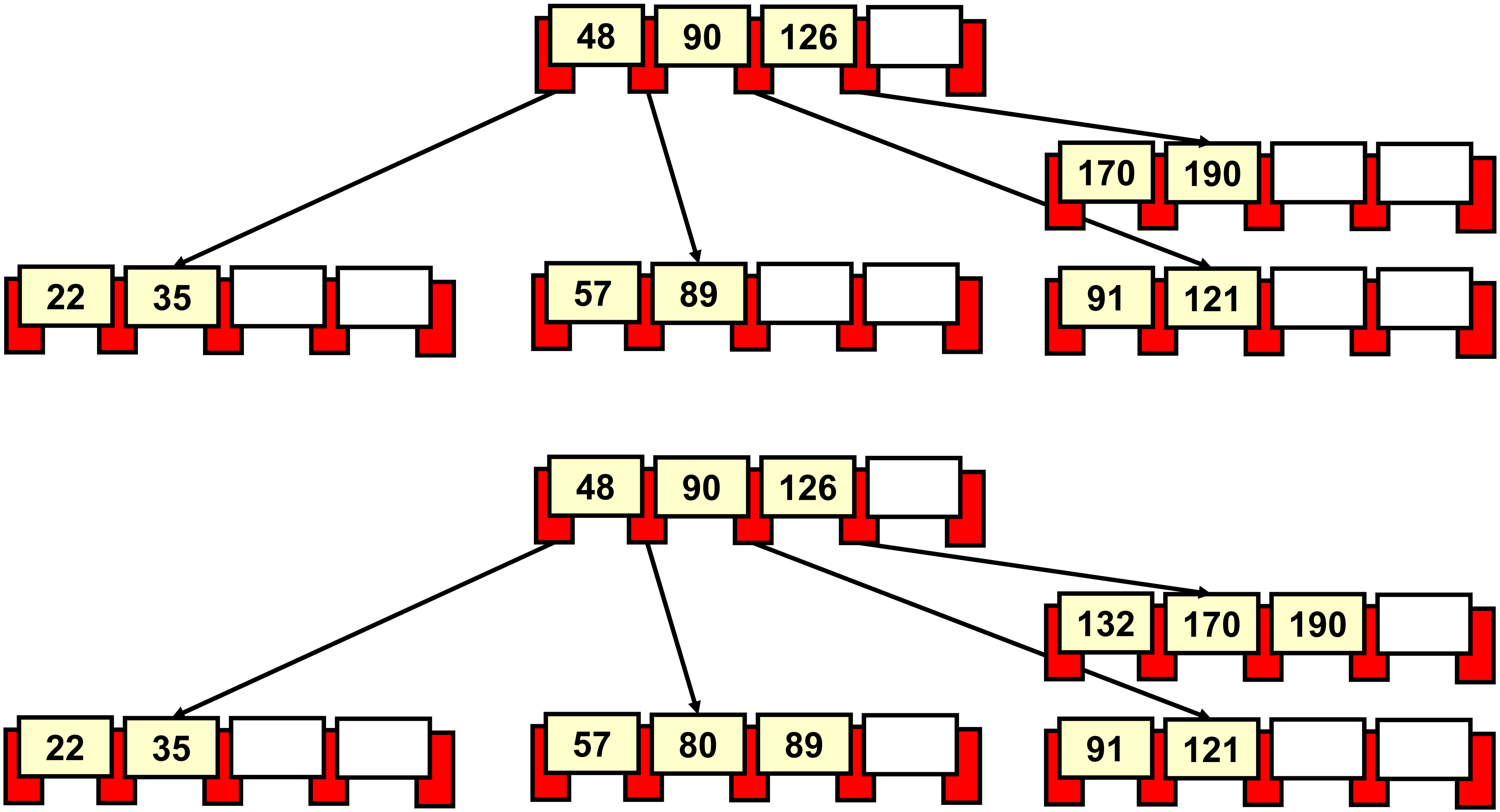




B Trees

E1. Insert the following elements

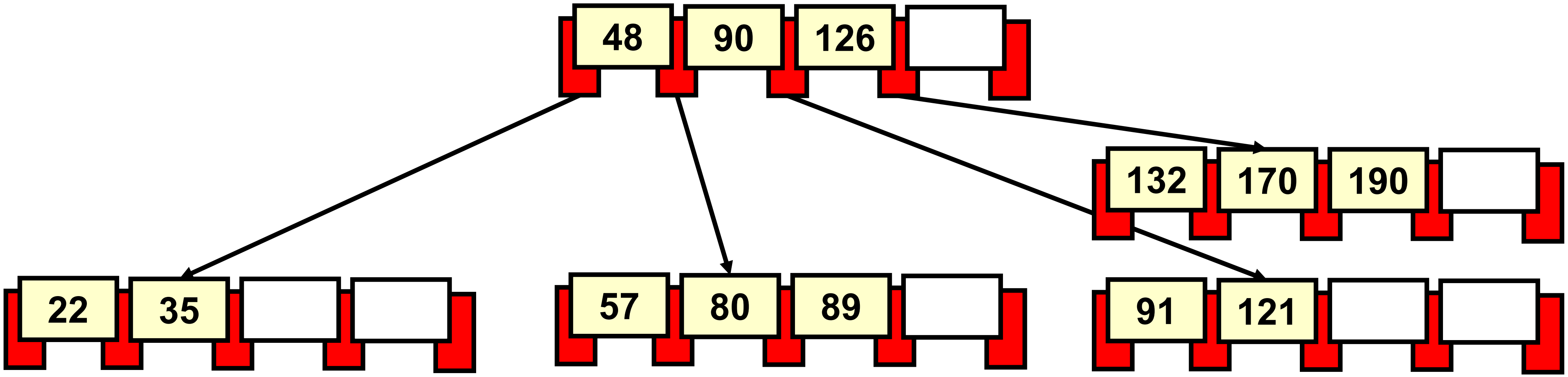
- 132, 80

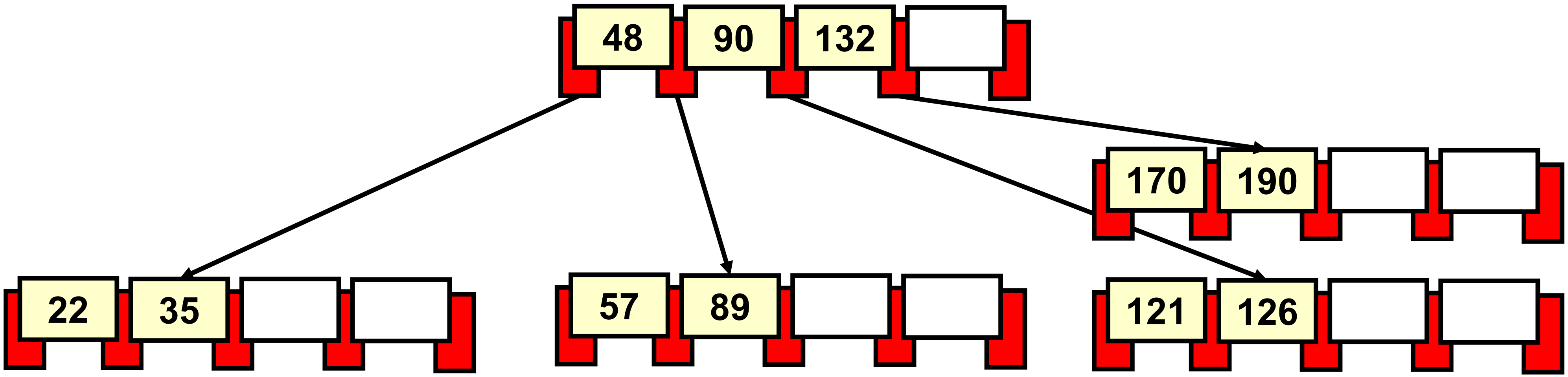


B Trees

E1. Delete the following elements

- 80, 91

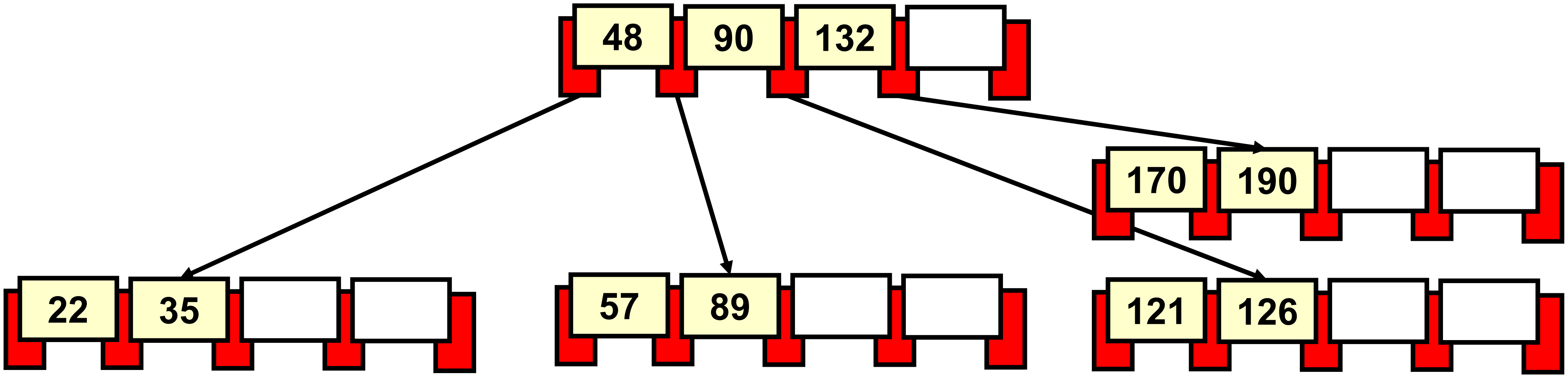


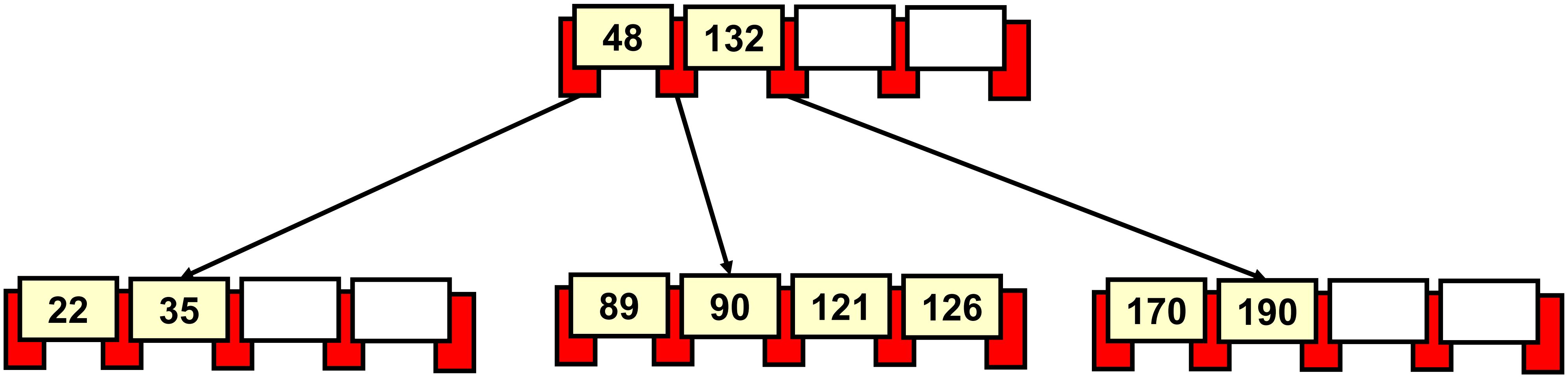


B Trees

E1. Delete the following elements

- 57

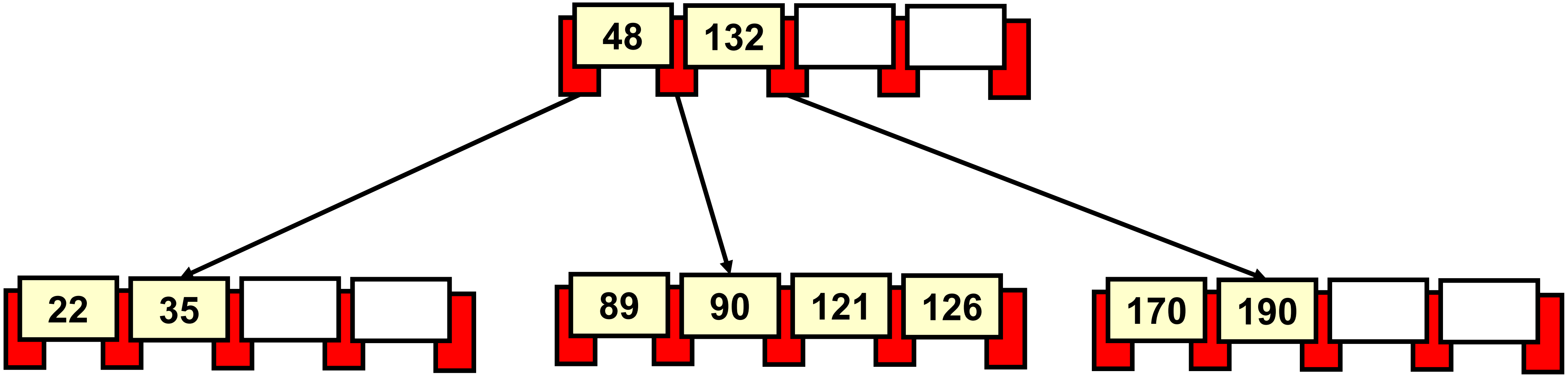


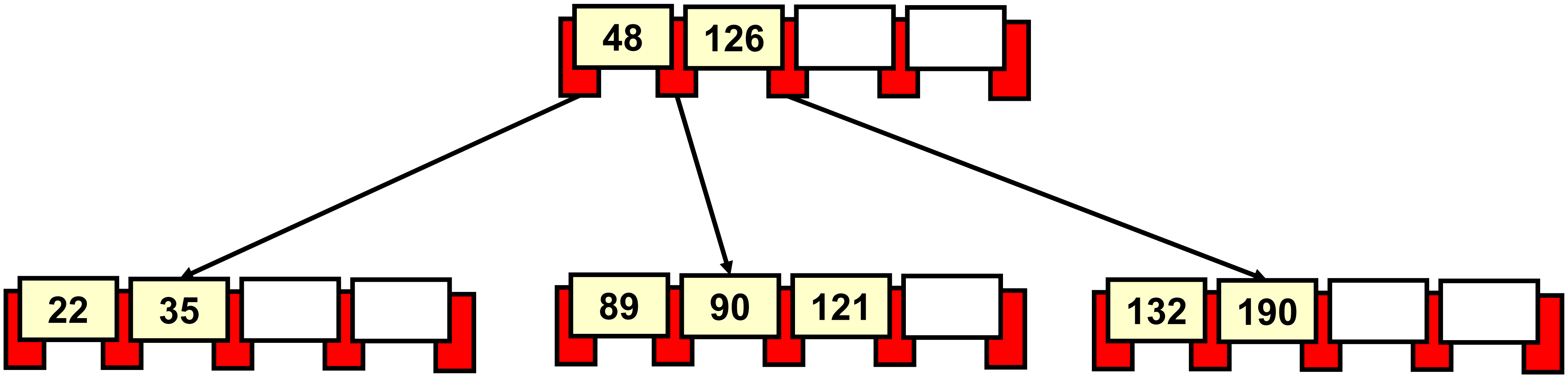


B Trees

E1. Delete the following elements

- 170

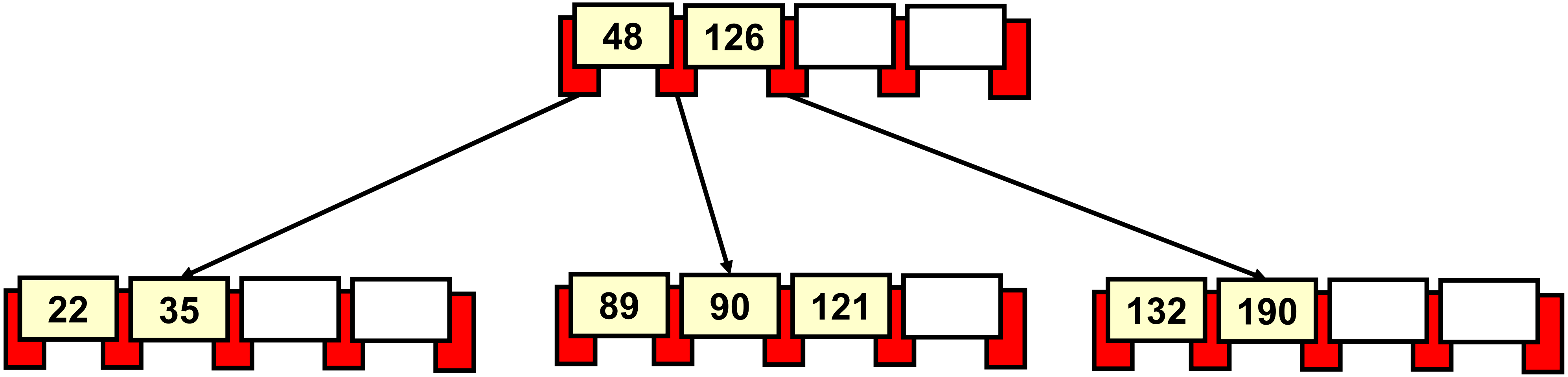


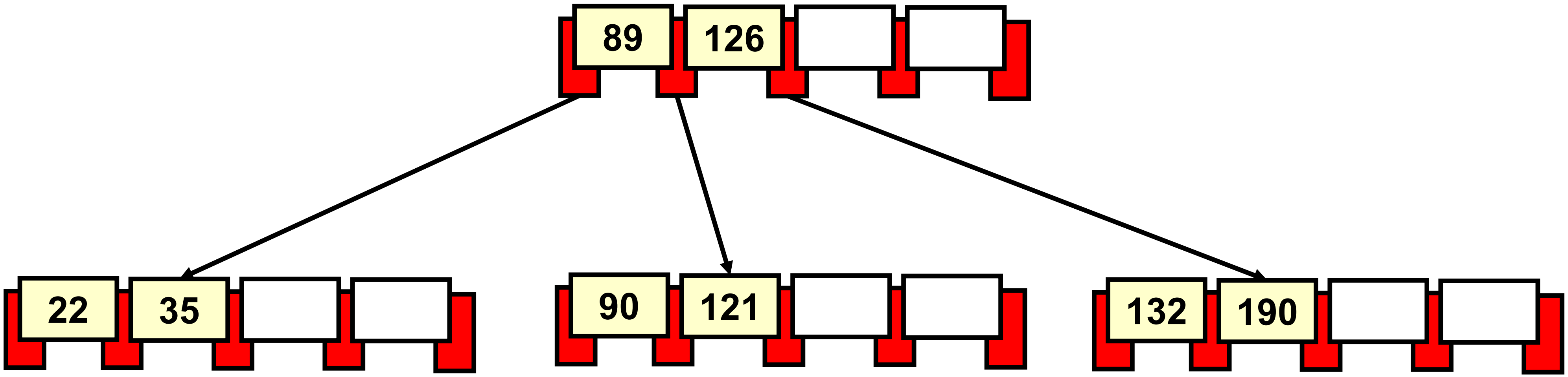


B Trees

E1. Delete the following elements

- 48

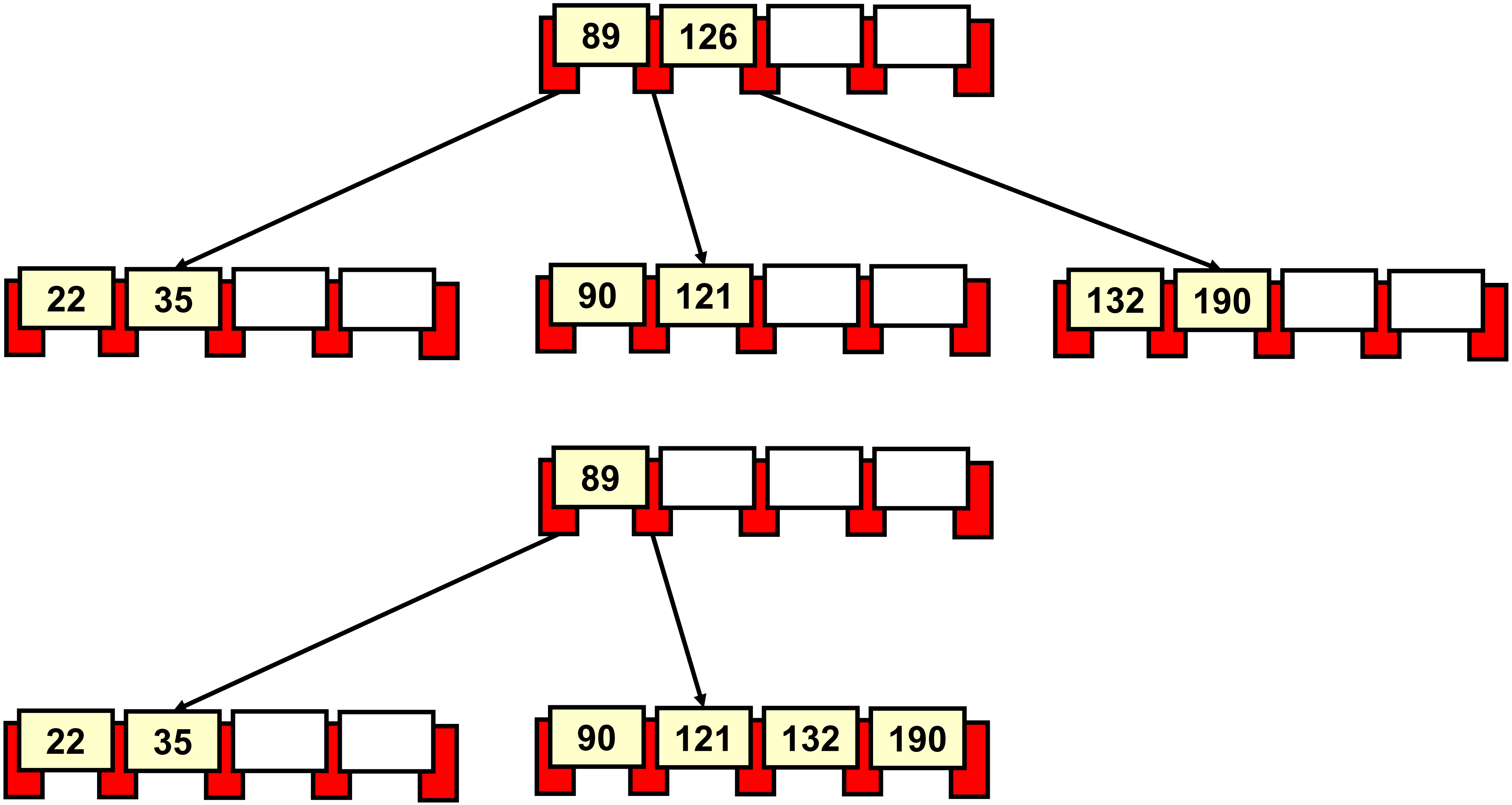




B Trees

E1. Delete the following elements

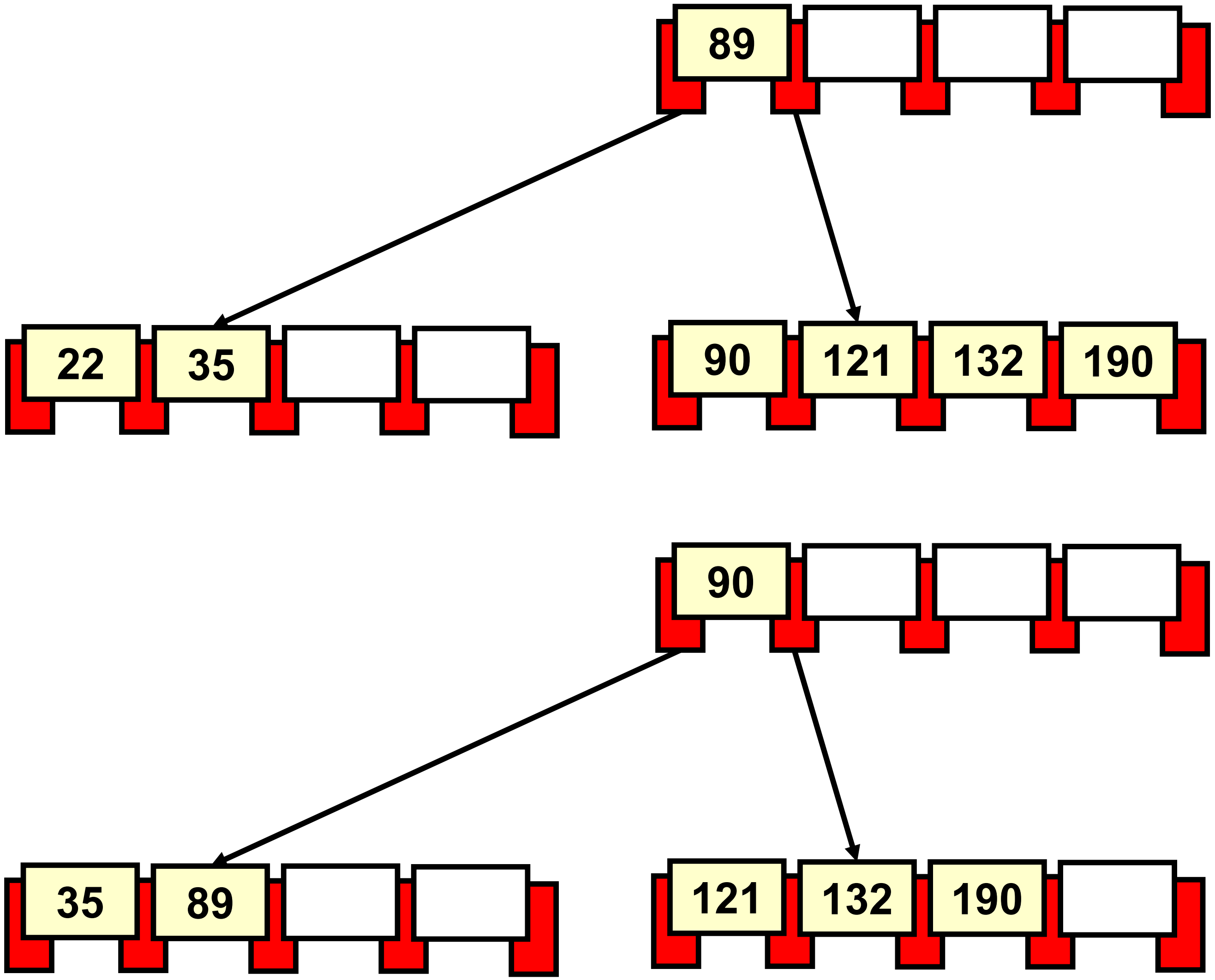
- 126



B Trees

E1. Delete the following elements

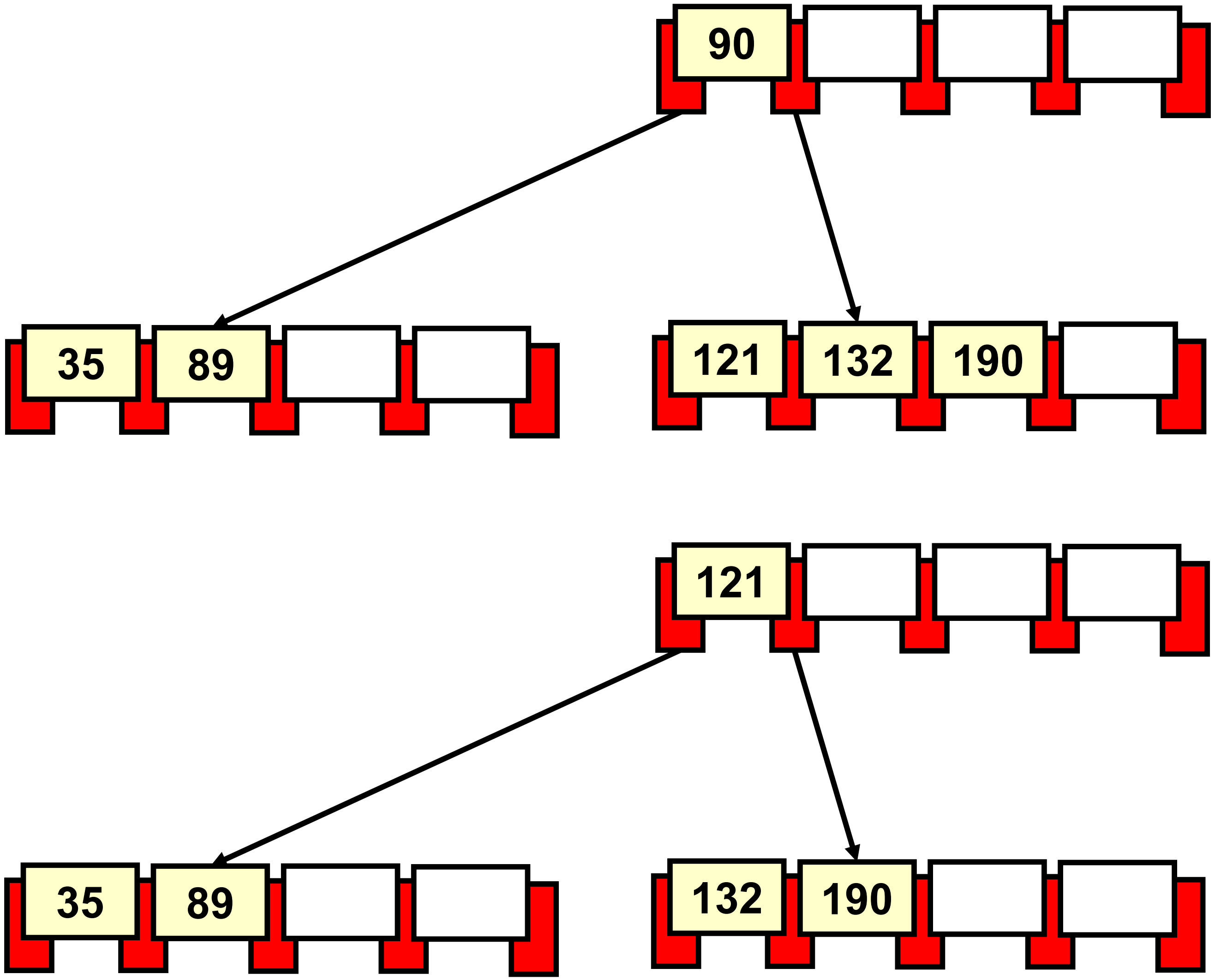
- 22



B Trees

E1. Delete the following elements

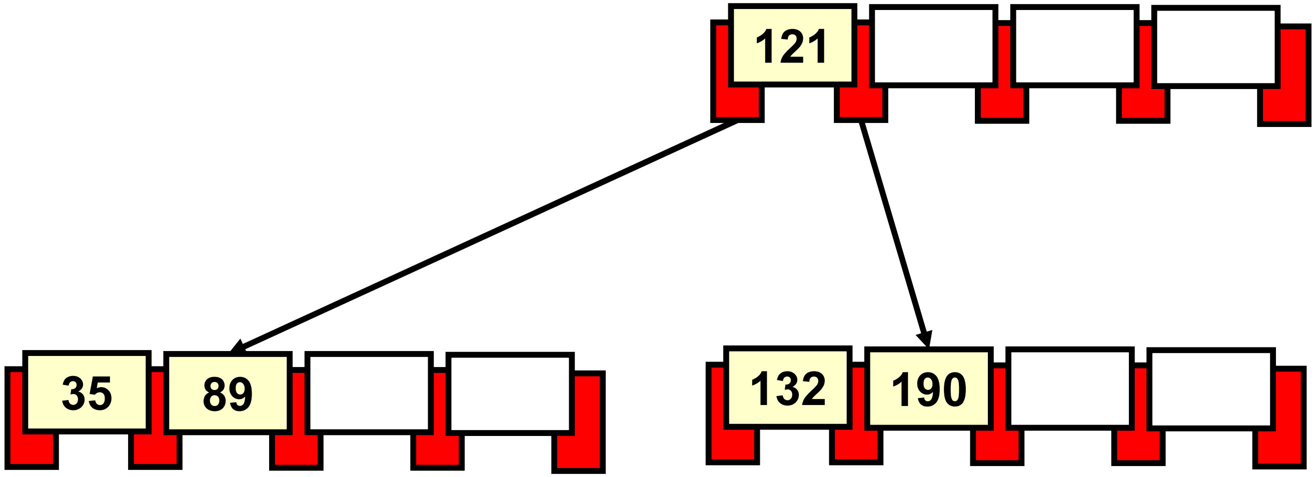
- 90

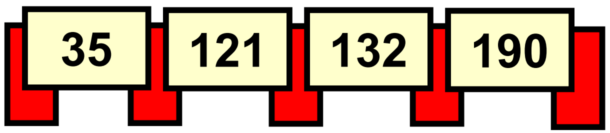


B Trees

E1. Delete the following elements

- 89





Priority Queues

E2. Create a Priority Queue based on the following parameters:

- Minimums.
- Size 16.

Priority Queues

E2. Add the following elements:

- 60.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
60															

Priority Queues

E2. Add the following elements:

- 48.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
60															

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
48	60														

Priority Queues

E2. Add the following elements:

- 80, 20.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
48	60														

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	80	60												

Priority Queues

E2. Add the following elements:

- 55, 65.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	80	60												

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	65	60	55	80										

Priority Queues

E2. Add the following elements:

- 63, 51.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	65	60	55	80										

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	63	51	55	80	65	60								

Priority Queues

E2. Add the following elements:

- 75, 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
20	48	63	51	55	80	65	60								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	20	63	51	48	80	65	60	75	55						

Priority Queues

E2. Add the following elements:

- 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	20	63	51	48	80	65	60	75	55						

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	63	51	20	80	65	60	75	55	48					

Priority Queues

E2. Add the following elements:

- 90, 95, 100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	63	51	20	80	65	60	75	55	48					

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	63	51	20	80	65	60	75	55	48	90	95	100		

Priority Queues

E2. Add the following elements:

- 41

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	63	51	20	80	65	60	75	55	48	90	95	100		

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	51	20	80	63	60	75	55	48	90	95	100	65	

Priority Queues

E2. Add the following elements:

- 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	51	20	80	63	60	75	55	48	90	95	100	65	

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	63	51	75	55	48	90	95	100	65	60

Priority Queues

E2. Delete the following elements:

- 100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	63	51	75	55	48	90	95	100	65	60

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	63	51	75	55	48	90	95	60	65	**60**

Priority Queues

E2. Delete the following elements:

- 60

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	63	51	75	55	48	90	95	60	65	**60**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	65	51	75	55	48	90	95	65	**65**	**60**

Priority Queues

E2. Delete the following elements:

- 63

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	63	51	75	55	48	90	95	65	**65**	**60**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	65	51	75	55	48	90	95	**65**	**65**	**60**

Priority Queues

E2. Execute the ***remove*** method

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	41	42	20	80	65	51	75	55	48	90	95	**65**	**65**	**60**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	20	41	42	48	80	65	51	75	55	95	90	**95**	**65**	**65**	**60**

Unit 6

Dictionary Structures

Open Addressing Hash Tables

Separate Chaining

E1. Create a Hash Table using the following parameters:

- Separate Chaining
- Size 13.
- Dynamic Resizing:
  - Increasing: LF >1
  - Inverse Dynamic Resizing: LF<0.33.

Separate Chaining

E1. Add the following elements:

- 1, 10, 15, 20

0	1	2	3	4	5	6	7	8	9	10	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	15					20			10		

LF: 0.31

Separate Chaining

E1. Add the following elements:

- 7

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	15					20			10		

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	15					20			10		
							7					

LF: 0,38

Separate Chaining

E1. Add the following elements:

- 13, 3, 2, 4, 6, 8, 18, 11

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	15					20			10		
							7					

0	1	2	3	4	5	6	7	8	9	10	11	12
13	1	15	3	4	18	6	20	8		10	11	
		2					7					

LF: 1.00

Separate Chaining

E1. Add the following elements:

- 12

0	1	2	3	4	5	6	7	8	9	10	11	12
13	1	15	3	4	18	6	20	8		10	11	
		2					7					

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	1	2	3	4		6	7	8		10	11	12	13	

15	16	17	18	19	20	21	22	23	24	25	26	27	28
15			18		20								

LF: 0.48

Open Addressing

E2. Create a Hash Table using the following parameters

- Open Addressing
- Size 7.
- Linear Probing.
- Lazy deletion:
  - **E**mpty.
  - **V**alid.
  - **D**eleted.

Open Addressing

E2. Add the following elements:

- 4

0	1	2	3	4	5	6
E	E	E	E	E	E	E

0	1	2	3	4	5	6
				4		
E	E	E	E	V	E	E

LF: 0.14

Open Addressing

E2. Add the following elements:

- 10

0	1	2	3	4	5	6
				4		
E	E	E	E	V	E	E

0	1	2	3	4	5	6
			10	4		
E	E	E	V	V	E	E

LF: 0.29

Open Addressing

E2. Add the following elements:

- 12

0	1	2	3	4	5	6
			10	4		
E	E	E	V	V	E	E

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

LF: 0.43

Open Addressing

E2. Add the following elements:

- 3

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

0	1	2	3	4	5	6
			10	4	12	3
E	E	E	V	V	V	V

Collisions at 3, 4 and 5

LF: 0.57

Open Addressing

E2. Add the following elements:

- 17

0	1	2	3	4	5	6
			10	4	12	3
E	E	E	V	V	V	V

0	1	2	3	4	5	6
17			10	4	12	3
V	E	E	V	V	V	V

Collisions at 3, 4, 5, and 6

LF: 0.71

Open Addressing

E2. Add the following elements:

- 15

0	1	2	3	4	5	6
17			10	4	12	3
V	E	E	V	V	V	V

0	1	2	3	4	5	6
17	15		10	4	12	3
V	V	E	V	V	V	V

LF: 0.85

Open Addressing

E2. Add the following elements:

- 14

0	1	2	3	4	5	6
17	15		10	4	12	3
V	V	E	V	V	V	V

0	1	2	3	4	5	6
17	15	14	10	4	12	3
V	V	V	V	V	V	V

Collisions at 0, 1 and 2

LF: 1.00

Open Addressing

E3. Create a Hash Table using the following parameters

- Open Addressing
- Size 7.
- Quadratic Probing.
- Lazy deletion:
  - **E**mpty.
  - **V**alid.
  - **D**eleted.

Open Addressing

E3. Add the following elements:

- 4, 10 and 12

0	1	2	3	4	5	6
E	E	E	E	E	E	E

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

LF: 0.43

Open Addressing

E3. Add the following elements:

- 17

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

0	1	2	3	4	5	6
17			10	4	12	
V	E	E	V	V	V	E

Collisions at 3 and 4

LF: 0.57

Open Addressing

E3. Add the following elements:

- 3

0	1	2	3	4	5	6
17			10	4	12	
V	E	E	V	V	V	E

Collisions at 3, 4, 0, 5, 5, …

LF: 0.57

Open Addressing

E4. Create a Hash Table using the following parameters

- Open Addressing
- Size 7.
- Double Hashing.
- Lazy deletion:
  - **E**mpty.
  - **V**alid.
  - **D**eleted.

Open Addressing

E4. Add the following elements:

- 4, 10 and 12

0	1	2	3	4	5	6
E	E	E	E	E	E	E

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

LF: 0.43

Open Addressing

E4. Add the following elements:

- 17

0	1	2	3	4	5	6
			10	4	12	
E	E	E	V	V	V	E

0	1	2	3	4	5	6
			10	4	12	17
E	E	E	V	V	V	V

Collision at 3

LF: 0.57

Open Addressing

E4. Add the following elements:

- 3

0	1	2	3	4	5	6
			10	4	12	17
E	E	E	V	V	V	V

0	1	2	3	4	5	6
3			10	4	12	17
V	E	E	V	V	V	V

Collisions at 3 and 6

LF: 0.71

Open Addressing

E4. Add the following elements:

- 5

0	1	2	3	4	5	6
3			10	4	12	17
V	E	E	V	V	V	V

0	1	2	3	4	5	6
3	5		10	4	12	17
V	V	E	V	V	V	V

Collisions at 5 and 3

LF: 0.86

Open Addressing

E4. Add the following elements:

- 7

0	1	2	3	4	5	6
3	5		10	4	12	17
V	V	E	V	V	V	V

0	1	2	3	4	5	6
3	5	17	10	4	12	17
V	V	V	V	V	V	V

Collisions at 0, 3, 6 and 2

LF: 1.0

Open Addressing

E5. Create a Hash Table using the following parameters

- Open Addressing
- Size 23.
- Linear Probing.
- Lazy deletion:
  - **E**mpty.
  - **V**alid.
  - **D**eleted.
- Dynamic Resizing:
  - Increasing: LF >0.5
  - Inverse Dynamic Resizing: LF<0.16.

Open Addressing

E5. Add the following elements:

- 1, 2, 10, 11, 12, 13, 15, 16, 17, 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2								10	11	12	13		15	16	17		19			
E	V	V	E	E	E	E	E	E	E	V	V	V	V	E	V	V	V	E	V	E	E	E

LF: 0.43

Open Addressing

E5. Delete the following elements:

- 2, 13, 19, 16, 10

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2								10	11	12	13		15	16	17		19			
E	V	V	E	E	E	E	E	E	E	V	V	V	V	E	V	V	V	E	V	E	E	E

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2								10	11	12	13		15	16	17		19			
E	V	D	E	E	E	E	E	E	E	D	V	V	D	E	V	D	V	E	D	E	E	E

LF: 0.22

Open Addressing

E5. Add the following elements:

- 21, 9, 33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2								10	11	12	13		15	16	17		19			
E	V	D	E	E	E	E	E	E	E	D	V	V	D	E	V	D	V	E	D	E	E	E

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2							9	33	11	12	13		15	16	17		19		21	
E	V	D	E	E	E	E	E	E	V	V	V	V	D	E	V	D	V	E	D	E	V	E

LF: 0.35

Open Addressing

E5. Delete the following elements:

- 1, 33, 21, 9, 11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2							9	33	11	12	13		15	16	17		19		21	
E	V	D	E	E	E	E	E	E	V	V	V	V	D	E	V	D	V	E	D	E	V	E

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2							9	33	11	12	13		15	16	17		19		21	
E	D	D	E	E	E	E	E	E	D	D	D	V	D	E	V	D	V	E	D	E	D	E

Inverse Resizing

LF: 0.13

Open Addressing

E5. (Inverse resizing)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
	1	2							9	33	11	12	13		15	16	17		19		21	
E	D	D	E	E	E	E	E	E	D	D	D	V	D	E	V	D	V	E	D	E	D	E

0	1	2	3	4	5	6	7	8	9	10
	12			15		17				
E	V	E	E	V	E	V	E	E	E	E

LF: 0.27

Open Addressing

E5. Add the following elements:

- 3, 9, 4

0	1	2	3	4	5	6	7	8	9	10
	12			15		17				
E	V	E	E	V	E	V	E	E	E	E

0	1	2	3	4	5	6	7	8	9	10
	12		3	15	4	17			9	
E	V	E	V	V	V	V	E	E	V	E

Dynamic Resizing

LF: 0.54

Open Addressing

E5. Dynamic Resizing

0	1	2	3	4	5	6	7	8	9	10
	12		3	15	4	17			9	
E	V	E	V	V	V	V	E	E	V	E

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
			3	4					9			12			15		17					
E	E	E	V	V	E	E	E	E	V	E	E	V	E	E	V	E	V	E	E	E	E	E

LF: 0.22