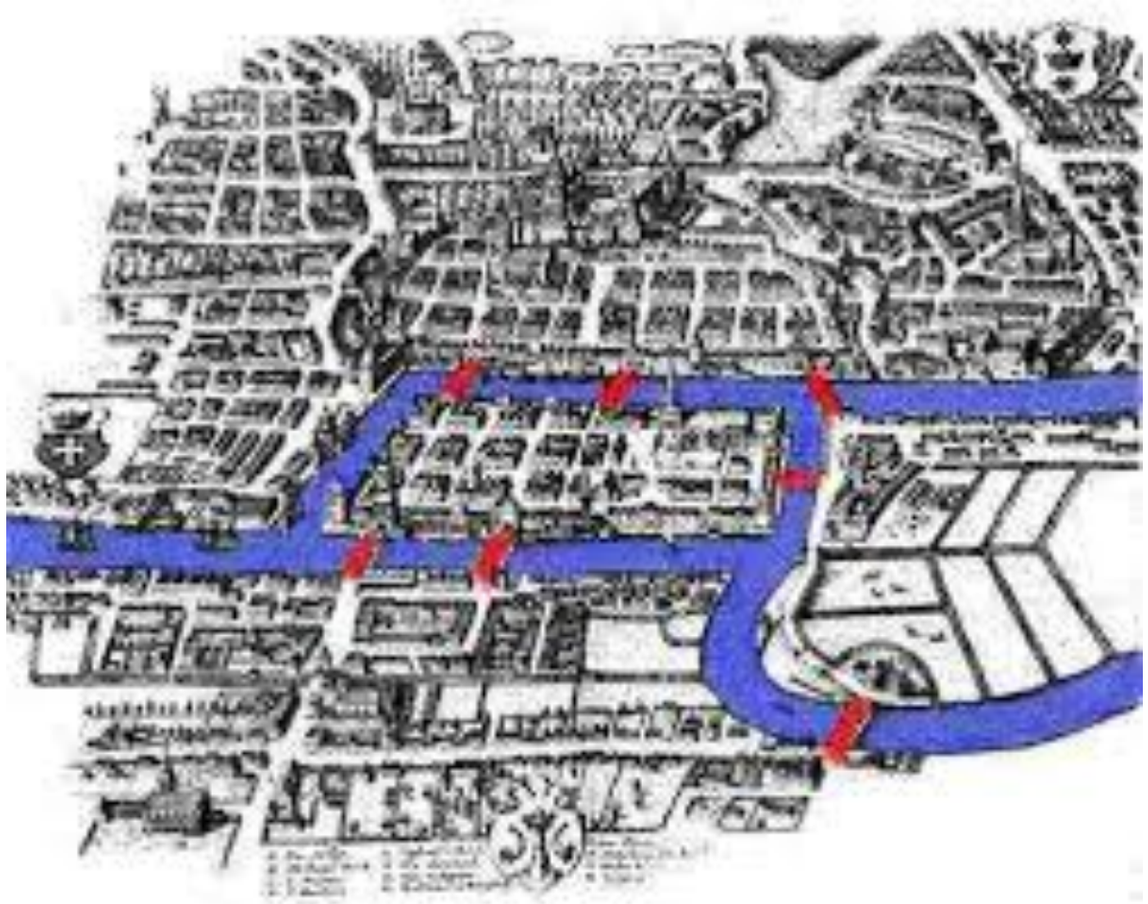
Self-Balancing Trees and Graphs



Self-Balancing Trees

Refer to Chp. 19

Array Implementation

Binary search trees offer significant advantages in terms of searching speed.
As we have seen, BSTs merge the advantages of link-based implementations with the advantages of binary search with random-access arrays.

A drawback, is the performance degradation a tree undergo if it is heavily unbalanced.

Can you figure out what is the most extreme case of unbalance and what is the consequence from the point of view of time computational complexity?

In order to deal with unbalance, **balanced search trees** were introduced.

2-3 Trees

Definition:
a 2-3 Tree is a tree where each node (which is not a leaf) can have either 2 or 3 children at most.
Nodes with 2 (3) children are called 2-nodes (3-nodes).

Observations:
1) A 2-3 tree is not in general a binary tree. If a 2-3 tree contains only 2-nodes, it is equivalent to a binary tree.
In this respect, a binary tree is a special case of 2-3-tree.

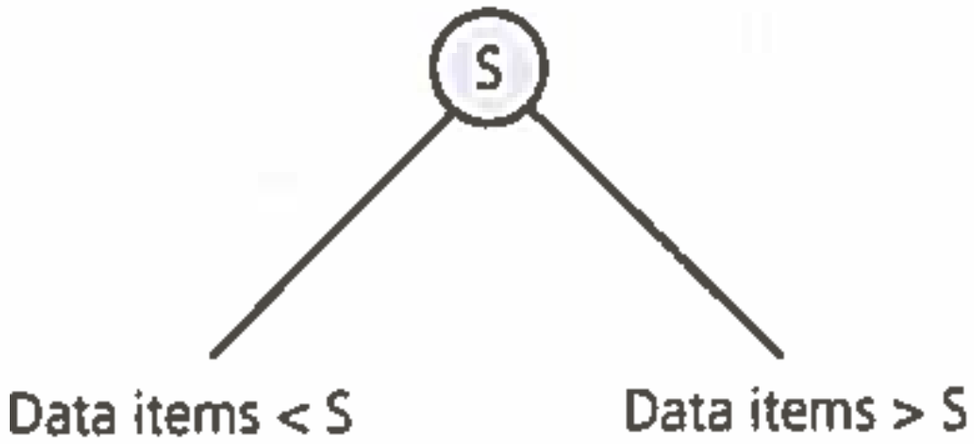
2) A 2-3- tree of height h contains at least as many nodes as a complete binary tree of height h.

Said in another way, a 2-3-tree with N nodes has an height which is never bigger than an N-node binary tree: $H<\log_2(N+1)$.

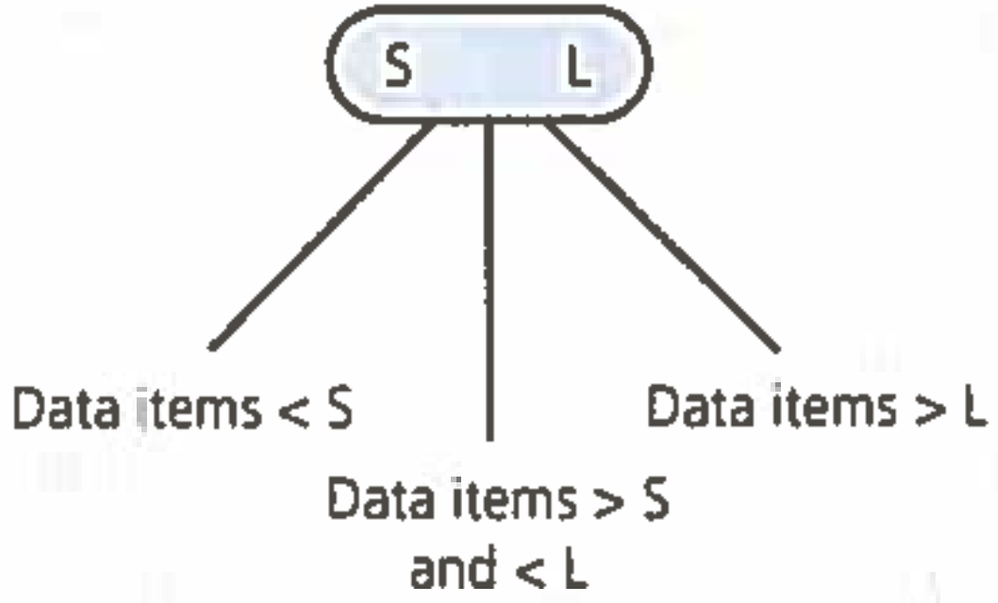
3) We can distinguish generic 2-3 trees and 2-3 search trees.

2-3 Trees Filling Rules

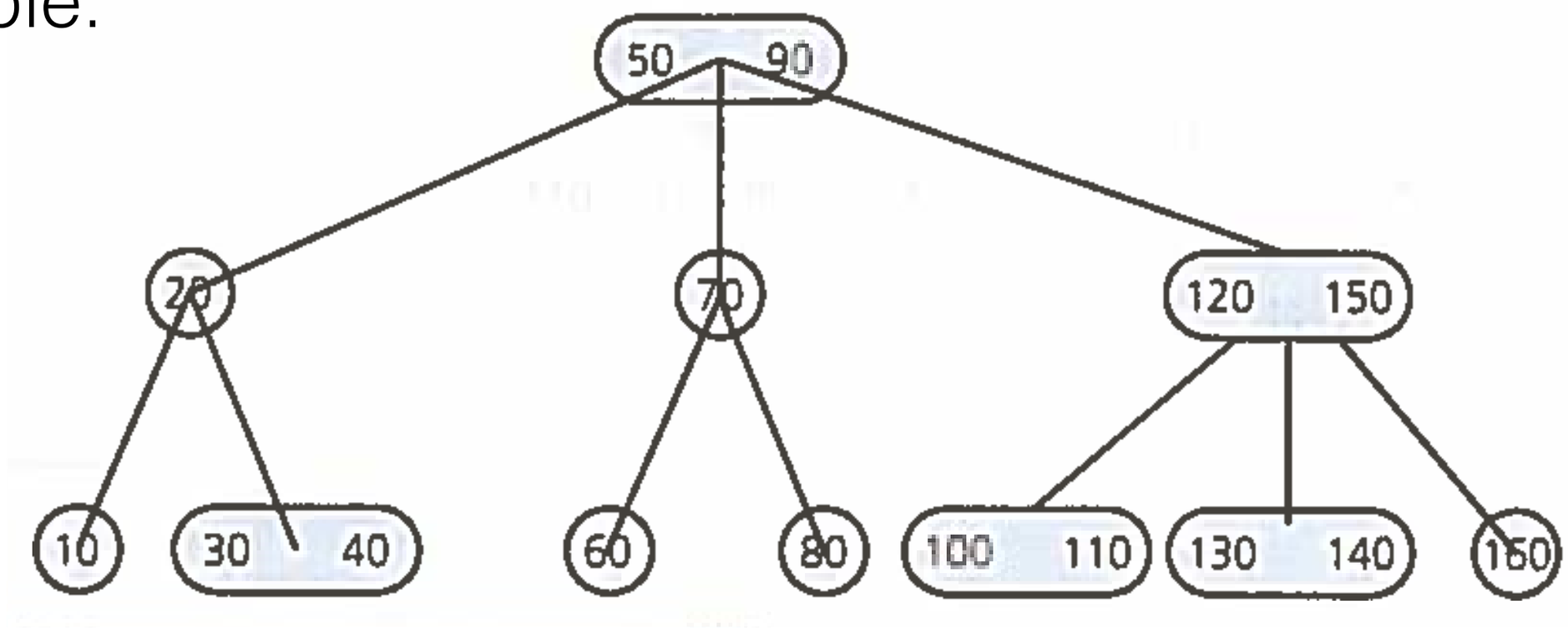
2-nodes contain 1 data items



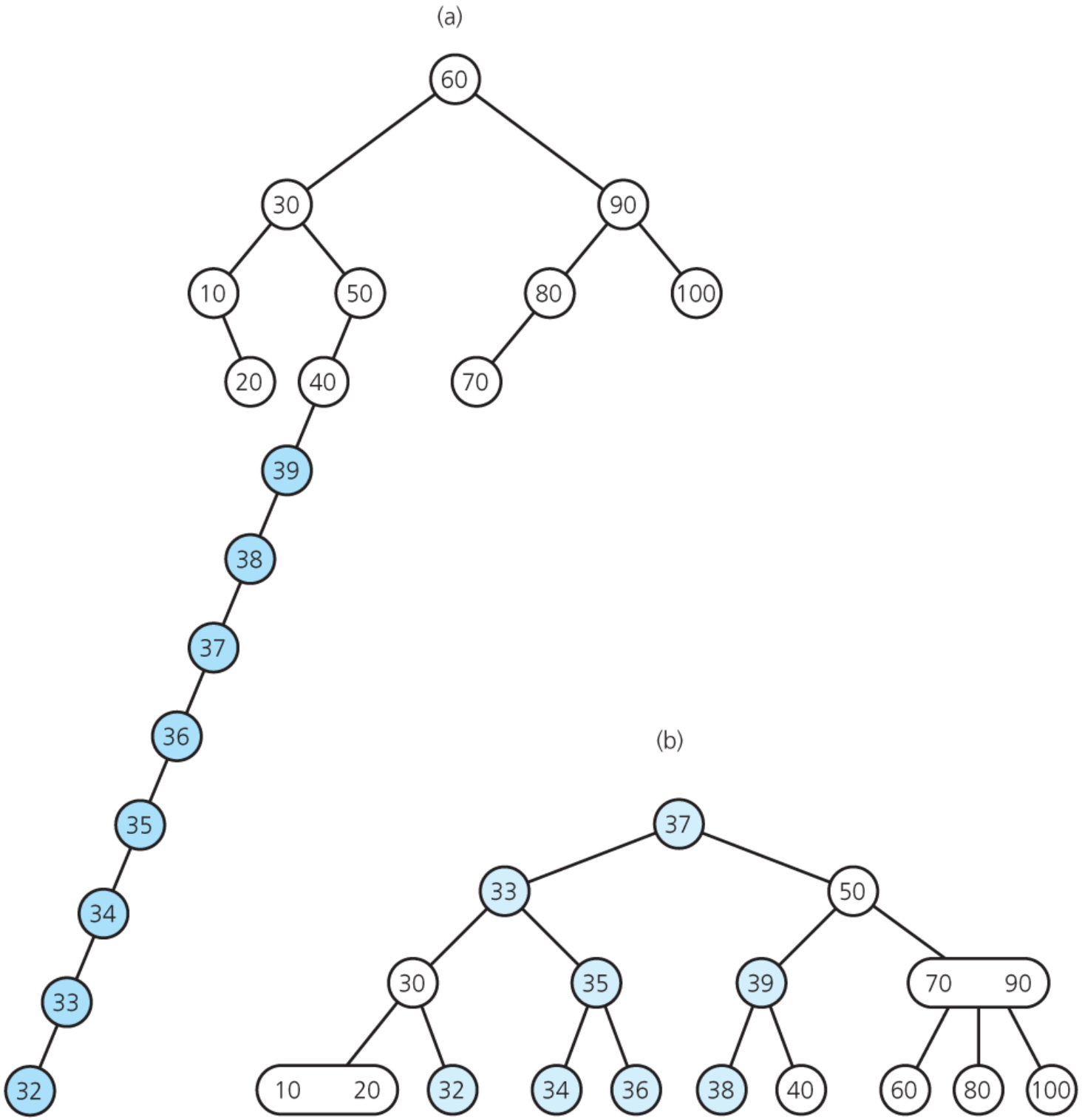
3-nodes contain 2 data items



Example:



2-3 Tree Self-Balancing



2-3 Tree Node

```
template<class ItemType>
class TriNode
{
private:
   ItemType smallItem, largeItem;    // Data portion
   TriNode<ItemType>* leftChildPtr;  // Left-child pointer
   TriNode<ItemType>* midChildPtr;   // Middle-child pointer
   TriNode<ItemType>* rightChildPtr; // Right-child pointer

public:
   TriNode();
   TriNode(const ItemType& anItem);
   TriNode(const ItemType& anItem, TriNode<ItemType>* leftPtr,
           TriNode<ItemType>* midPtr, TriNode<ItemType>* rightPtr);

   bool isLeaf() const;
   bool isTwoNode() const;
   bool isThreeNode() const;
\
   ItemType getSmallItem() const;
   ItemType getLargeItem() const;

   void setSmallItem(const ItemType& anItem);
   void setLargeItem(const ItemType& anItem);

   TriNode<ItemType>* getLeftChildPtr() const;
   TriNode<ItemType>* getMidChildPtr() const;
   TriNode<ItemType>* getRightChildPtr() const;

   void setLeftChildPtr(TriNode<ItemType>* leftPtr);
   void setMidChildPtr(TriNode<ItemType>* midPtr);
   void setRightChildPtr(TriNode<ItemType>* rightPtr);
};
```

2-3 Tree Traversal

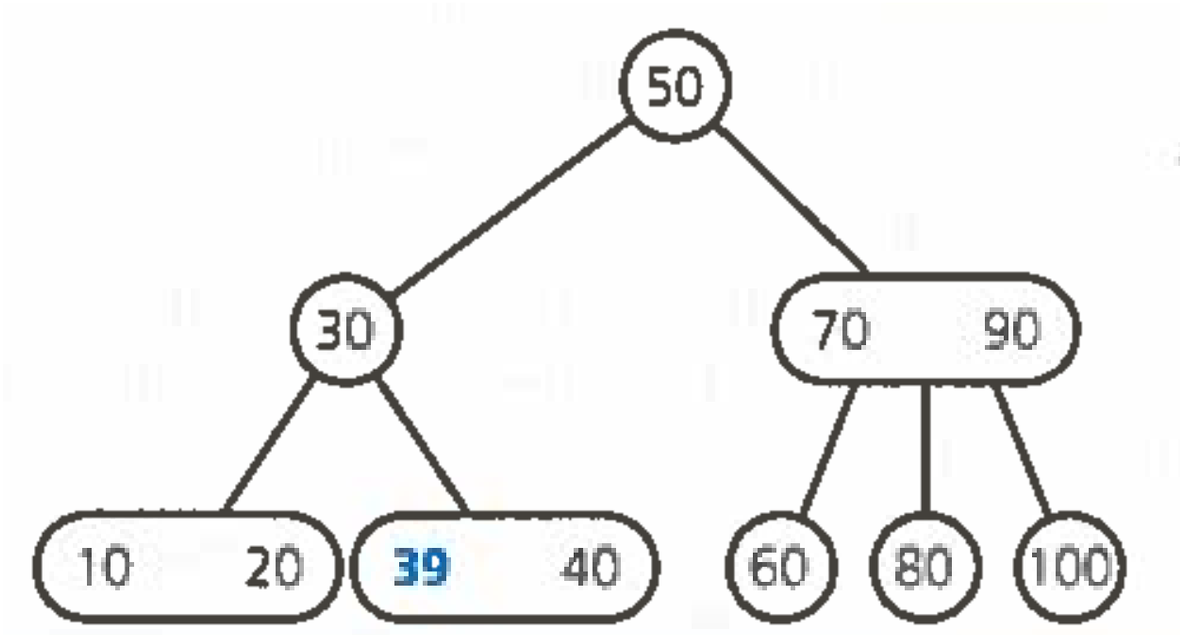
```
//PSEUDOCODE
void InOrderTraversal(23Tree: TwoThreeTree){

    if (23Tree node r is a leaf) visit data item;
    else if (r has 2 items){ //r is a 3-node
        InOrderTraversal(left subtree);
        visit first data item;
        InOrderTraversal(middle subtree);
        visit second data item;
        InOrderTraversal(right subtree);
    }
    else { //r has 1 item (it is a 2-node)
        InOrderTraversal(left subtree);
        visit data item;
        InOrderTraversal(right subtree);
    }
}
```

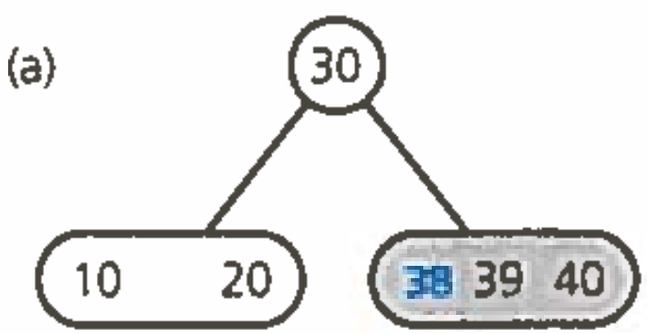
2-3 Tree Search

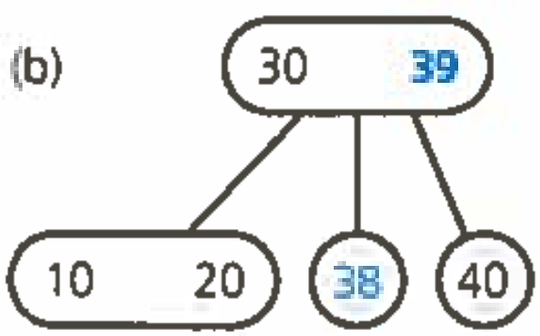
```
//PSEUDOCODE
T Search(TwoThreeTree, Item target){

   if (target is in root r) return item //base case
   else if (r is a leaf) return failure //not found
   else if (r has 2 data items){ //it is a 3-node
      if (target < smaller item in r)
         return Search(r left subtree, target);
      else if (target<largest item in r)
         return Search(r middle subtree, target);
      else
         return Search(r right subtree, target);
      }
   else { //r has 1 item (it is a 2-node)
      if (target < data item in r)
         return Search(r left subtree, target);
      else
         return Search(r right subtree, target);
   }
}
```

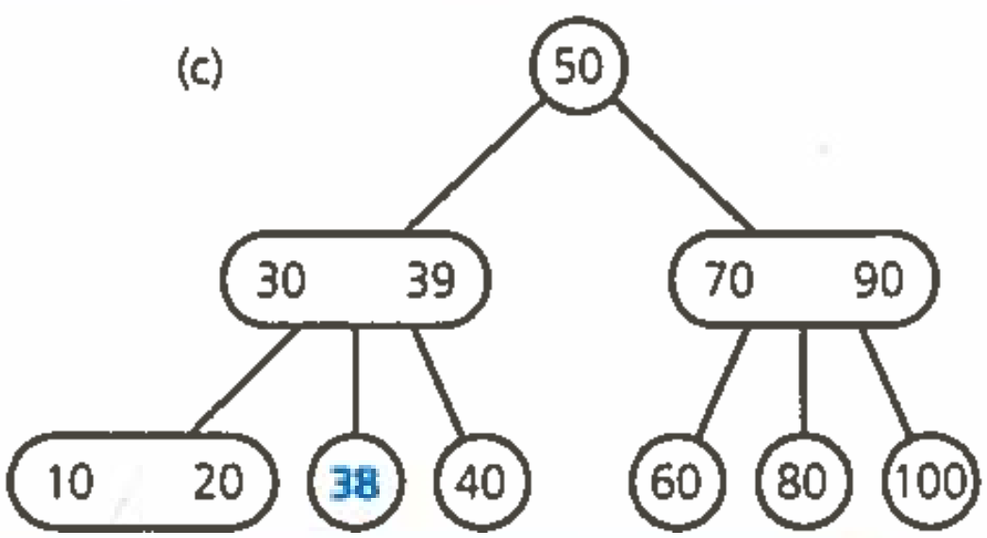
The search is still $O(\log_2 N)$ like for a BST: this is because although you visit less nodes on average, you do more comparisons due to the presence of 3-nodes!

2-3 Tree Insertion









2-3 Tree Insertion

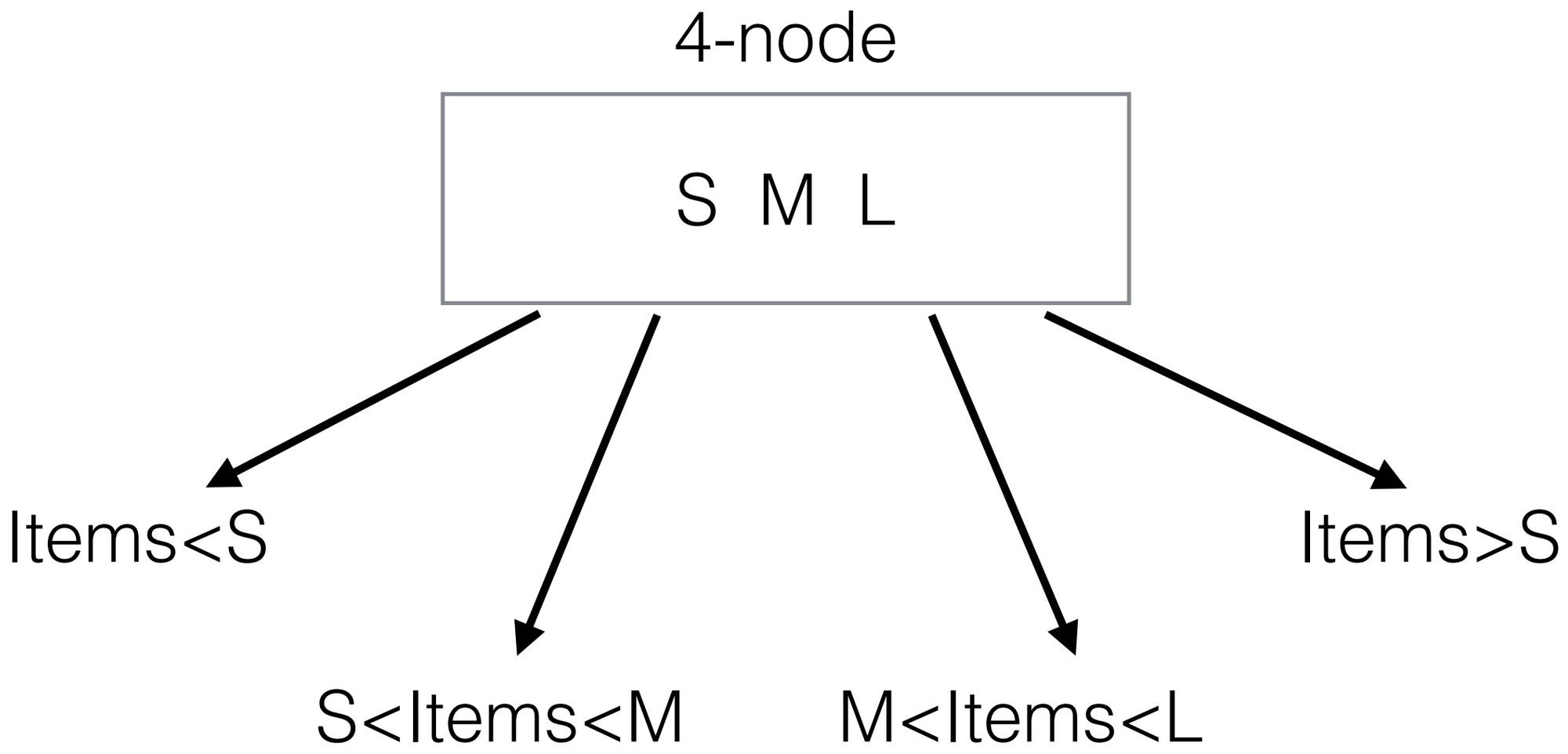
```
insertItem(23Tree, T item){
    Locate (e.g. using search) the leaf for item
    Add item to the leaf

    if (leaf has 3 items) split(leaf)
}
```

```
split(23Node n){
    if (n is root) create new node p;
    else let p be the parent of n;

    Replace n with n1 and n2 with p the parent;
    smaller value in n1;
    largest value in n2;

    if (n is not a leaf){
        n1 becomes the parent of n's two leftmost children;
        n2 becomes the parent of n's two rightmost children;
    }

    Move item in n with middle value up to p;
    if (p has 3 items) split(p);
}
```


What about removal?

2-3-4 Trees

A 2-3-4 Tree is like a 2-3 Tree, but 4-nodes are allowed.
4-nodes contain 3 data items and have 4 children.



2-3 Trees vs 2-3-4 Trees vs N-Trees

2-3-4 Trees have an advantage over 2-3 Trees in terms of insertion and removal algorithms. Both trees are self-balancing anyway.

The advantage of 2-3-4 trees is that the algorithm searching for the leaf where to place a new item splits every 4-node it encounters. So the next procedure of backing up will not need any splitting.

Something similar happens for the merging procedure of removal. While searching for the item to remove, you can always merge 2-nodes into 3- or 4-nodes. In this way, when you find the item to remove, you just remove it!

The higher efficiency of insertion and removal makes the 2-3-4 tree a preferred choice over the 2-3 tree.

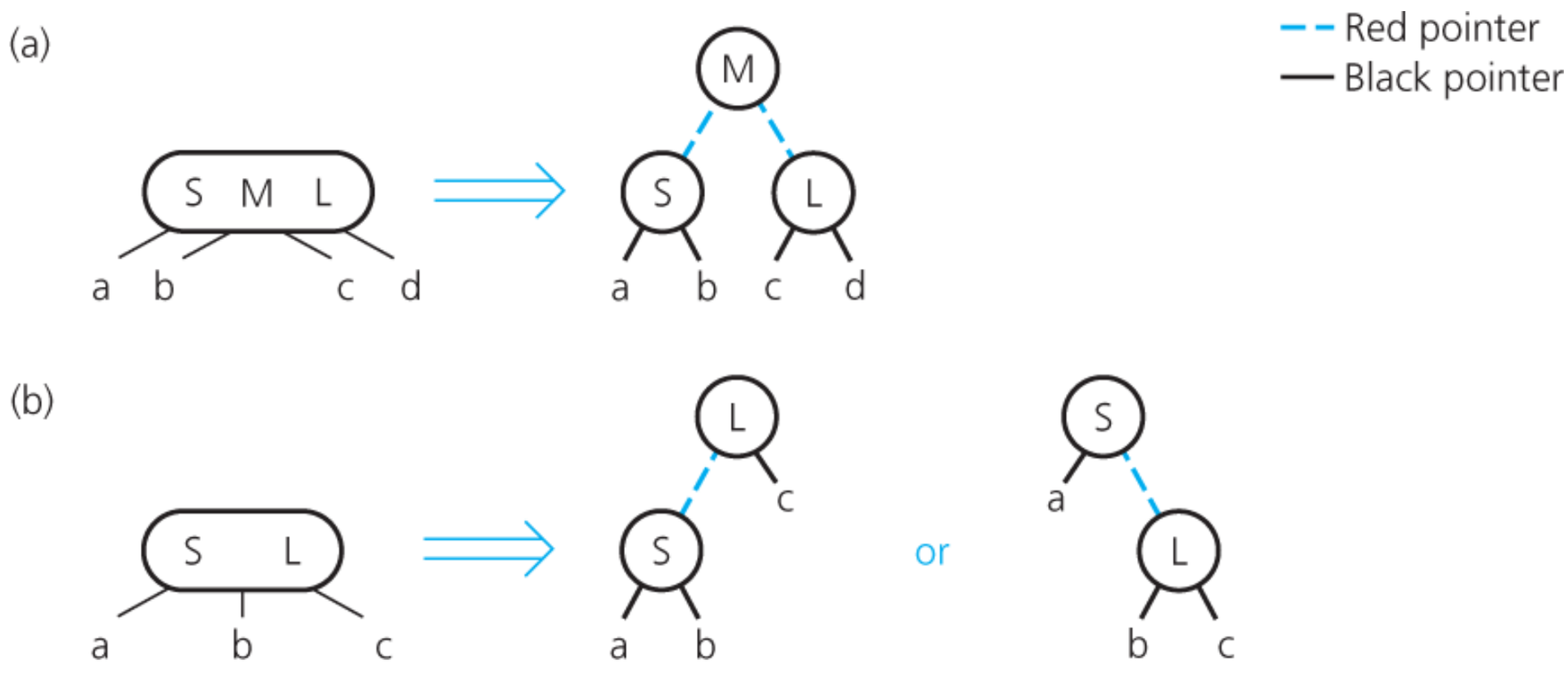
What about 2-3-4-5 or higher order trees?
Such trees can give you an advantage, since they have a reduced height at the expense of more and more comparisons. The efficiency of the tree traversal and comparisons must be carefully estimated for deciding the maximum number of children's of a node.

Red-Black Trees

A potential disadvantage of 2-3-4 trees is the increased storage requirement over binary search trees.
The problem is solved by **red-black Trees** which are binary trees with the advantages of 2-3-4 Trees.

The idea is to distinguish between 2-nodes appearing in the original 2-3-4-tree and 2-nodes generated from 3- and 4-nodes.

The distinction is made “coloring” pointers with two different “colors”: red and black. Let’s say that all the child’s pointers in the original 2-3-4-tree were black. We will use red pointers to link 2-nodes which result from splitting 3- and 4-nodes.

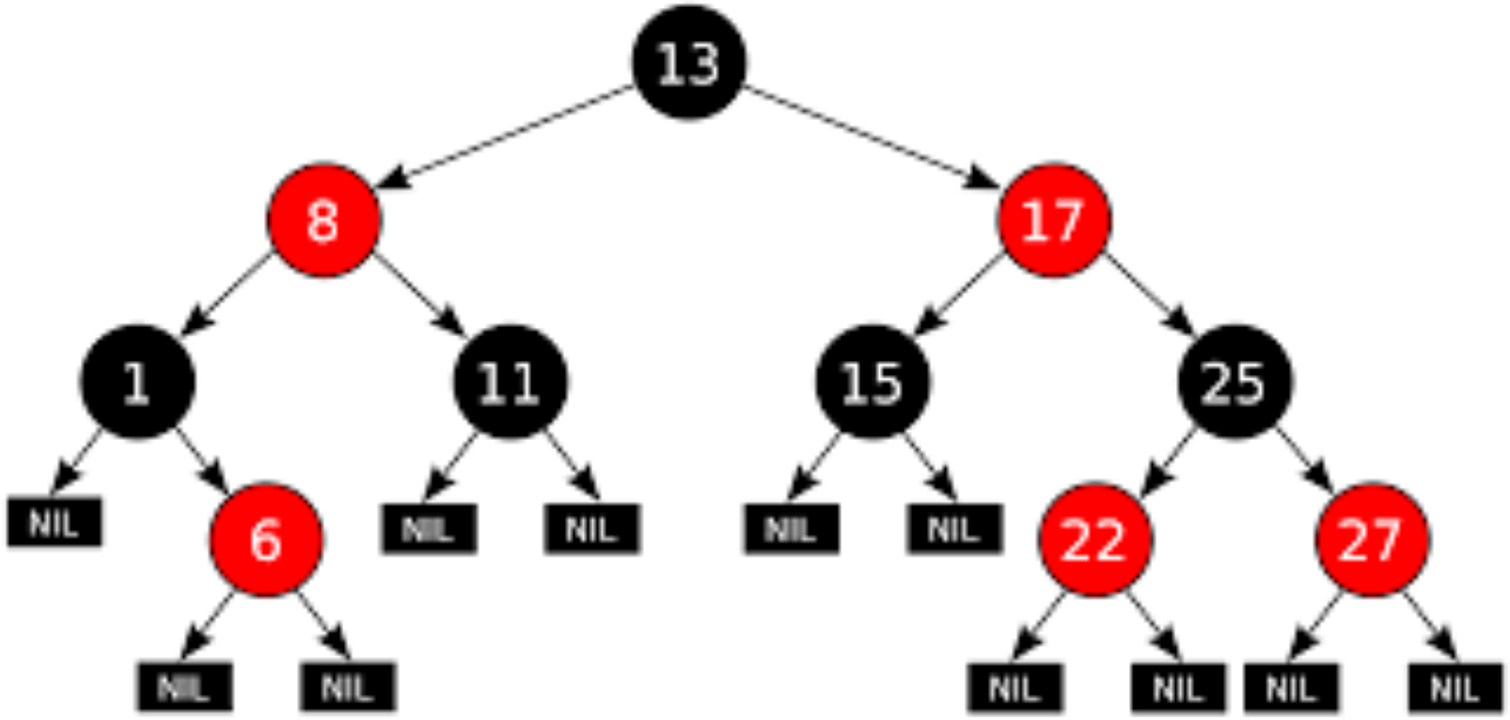


Red-Black Trees

Another (simpler) way to see RBTs is to consider all the nodes associated with a color (R or B) together with the following rules:

- Every node is Red or Black
- The root is always black
- If a node is Red, the children must be black (the converse is not necessarily true).
- Every path from the root to a leaf must contain the same number of black nodes (the so-called “black height”).

These basic rules enforce tree balancing while performing insertions.



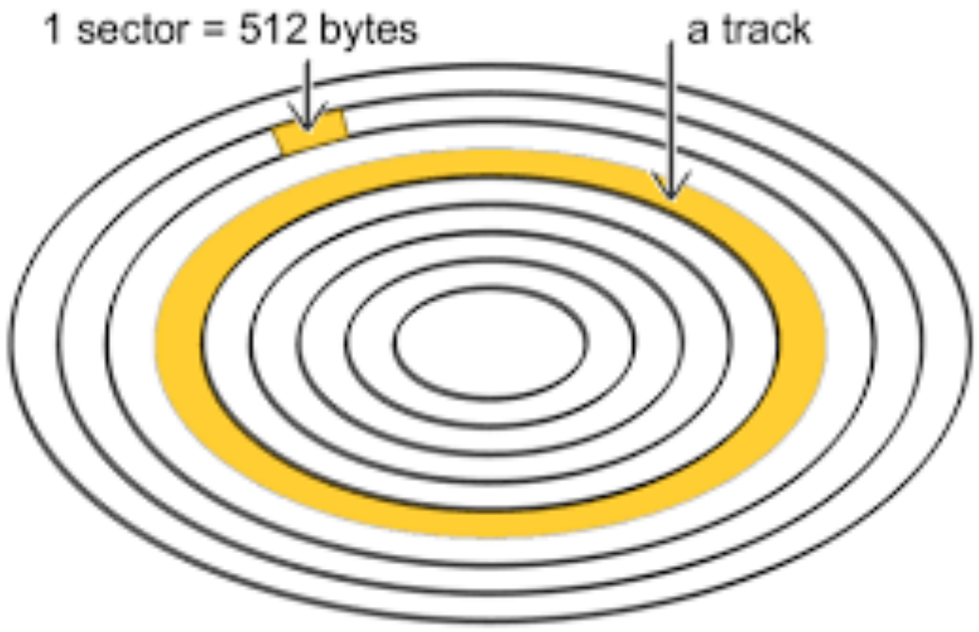
B-Trees and External Storage

B-trees are generalizations of 2-3 and 2-3-4 trees. Actually, 2-3 trees are B-trees of order 3 and 2-3-4 trees are B-trees of order 4 …).

B-trees are widely used in external storage applications. Given the way disks are read (block by block), it is convenient to organize the data according to a B-tree.
It makes sense to allow a number of children per node which matches the size of a block (caveat: we have to take into account the storage of pointers..).

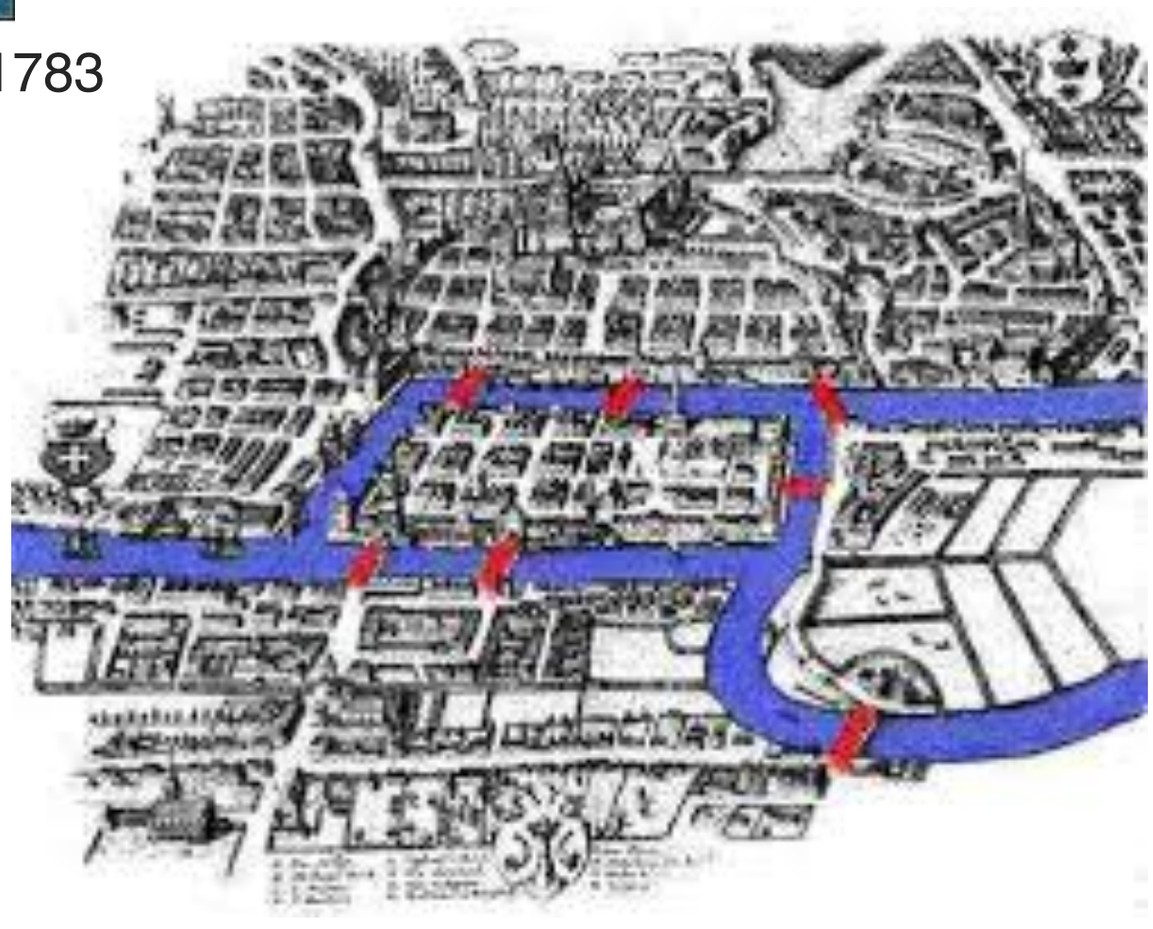
Insertion in a B-tree is done in a way similar to 2-3-trees, since 2-3-4-trees leave too many non-full nodes and this is not optimal for a disk.:

- When a node is split, half data goes into the new node and half stays.
- Node splits happen bottom-up (like in 2-3-trees.)
- The item promoted up is not the middle one, but the middle in the sequence formed by the items in the node, plus the new one (like in 2-3- trees).



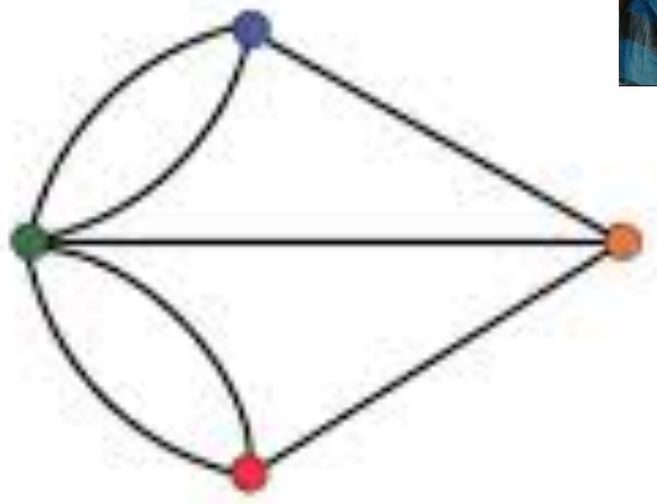
Graphs

Leonhard Euler 1707 – 1783



Refer to Chp. 20

The Bridges of Koenigsberg

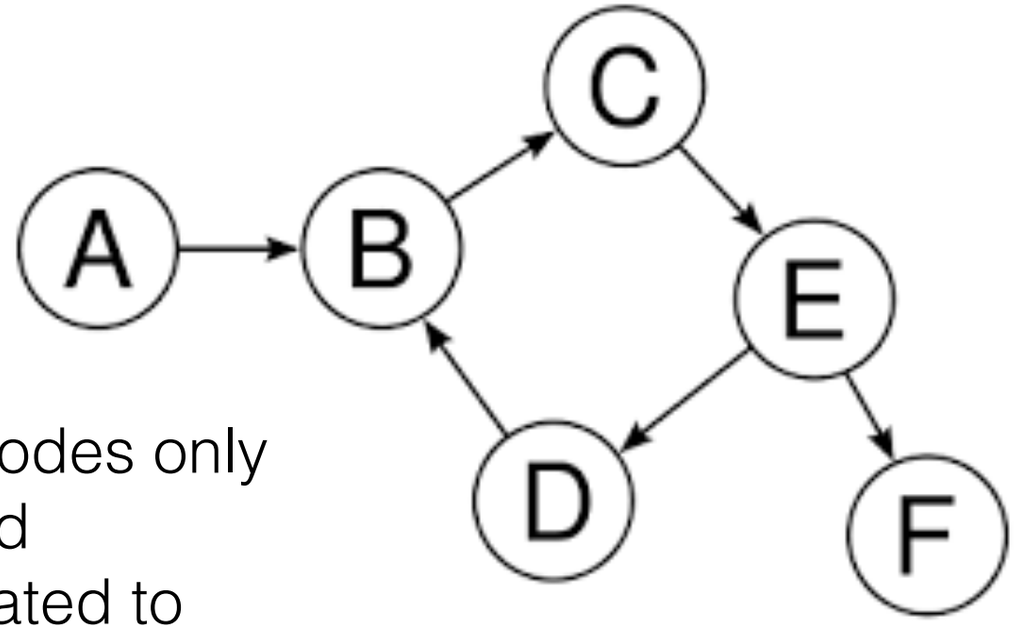


The problem is to find a walk that would cross each bridge once and only once, with the constraint that the islands could only be reached by the bridges and every bridge once accessed must be crossed to its other end.
The starting and ending points of the walk need not be the same.

Euler proved that the problem has no solution.
He proved that with a graph-based analysis and this is probably the beginning of graph theory!

Graphs: Basic Definitions

A graph G consists on two sets:
- V: the set of **vertices**
- E: the set of **edges**



If the edges allow the transitions among nodes only in a particular direction, the graph is called **directed**. If there is also a quantity associated to the edges, the graph is **weighted**.

A **subgraph** is a subset of vertices and edges.

Two vertices are **adjacent** if joined by an edge.

A **path** between two vertices is a sequence of edges joining them

A graph is **connected** if every pair of vertices has a path joining them.

A graph is **complete** if every pair of vertices has an edge joining them.

What is the relationship between Graphs and Trees?

A Graph Interface

```
template<class LabelType>
class GraphInterface
{
public:
     virtual int getNumVertices() const = 0;

     virtual int getNumEdges() const = 0;

     virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;

     virtual bool remove(LabelType start, LabelType end) = 0;

     virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;

     virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

     virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
};
```

Graph Description

There are two main ways to represent a graph:

1) Adjacency Matrix
2) Adjacency List

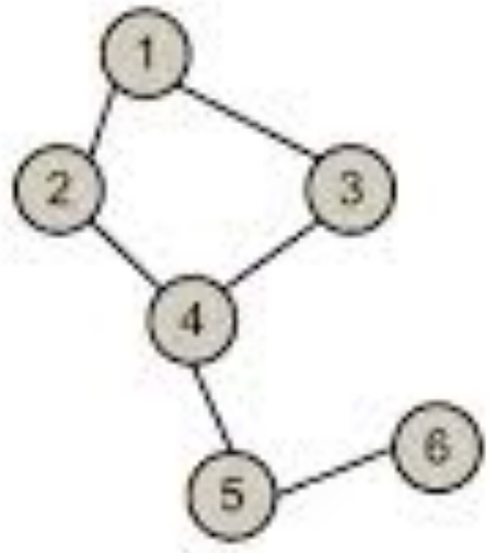
Every methods has advantages and disadvantages.

Adjacency Matrix

The adjacency matrix is a matrix which describes the relationship among vertices in a graph.

Example:
Let A[i][j] be the adjacency matrix.
We can set A[i][j] = 1 if the node i is connected with the node j. A[i][j] = 0 otherwise.

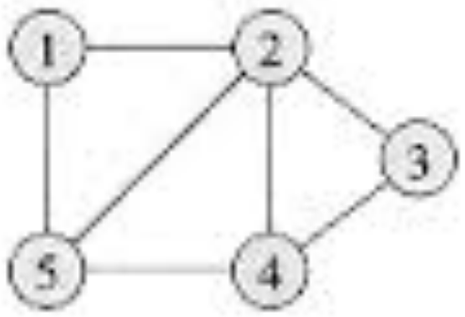
If the graph is weighted, we can substitute 1 with the appropriate weight.

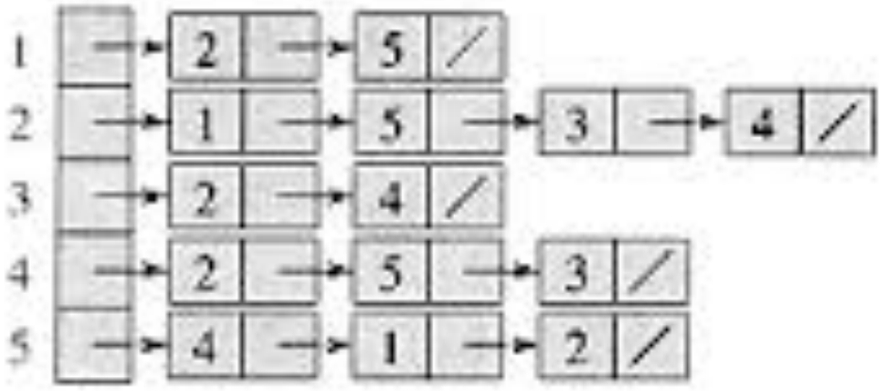


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Adjacency List

This graph description is based on linked chains. The i-th chain contains all the vertices connected to the vertex i.





Adjacency List

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

Adjacency Matrix

Matrix or List?

Depends from:

- The operations you most frequently would like to perform on the graph
- The type of graph.

Two common operations are:
1) Determine if vertices i and j are connected.
2) Find all vertices connected to the vertex i.

Clearly, operation 1) is faster with a matrix, while 2) is faster with a list.

Moreover, the matrix has always V2 entries, while the list has entries equal to the number of edges. So, if the graph is highly connected, a matrix can be convenient. If the graph is “sparse”, a list is a better choice from the memory allocation point of view.

Question:

Remember the “connecting flights” problem? How is it connected with graphs? Which graph implementation would you use in that problem?

Graph Traversal: Depth-First Search

This kind of search in a graph tries to go as deep as possible into the graph, following a path through adjacent vertices. When a vertex is visited, it should be marked, in order to avoid infinite loops!

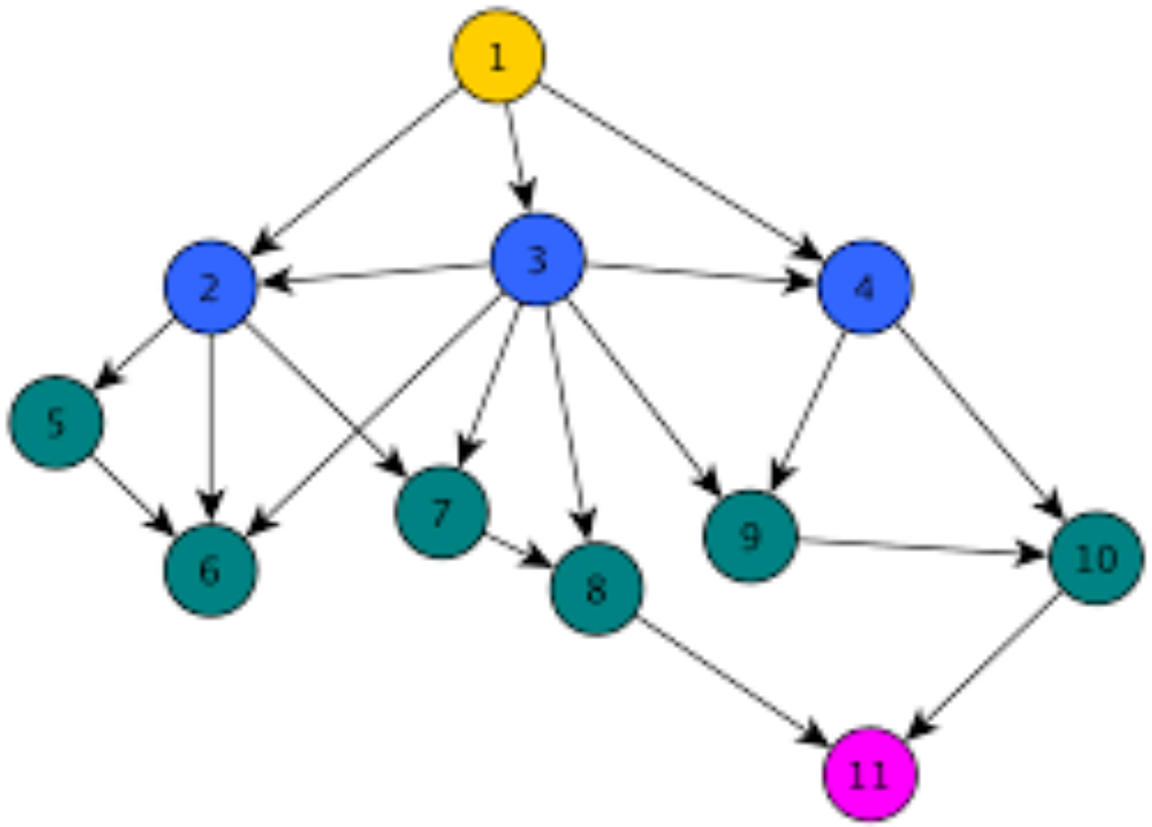
The (recursive or stack-based) algorithm is the following:

```
DFS(vertex v){
    Mark v as visited;
    for (each unvisited u adjacent to v){
        DFS(vertex u);
    }

}
```

If you remember, the search strategy was similar in the “connecting flights” problem. There are some differences though: which ones?
DFS is also a common strategy in many computer-based games (e.g. chess,..).

Graph Traversal: Breadth-First Search

BFS starts from one vertex and visits first all the adjacent vertices.
This algorithm is based on a queue:



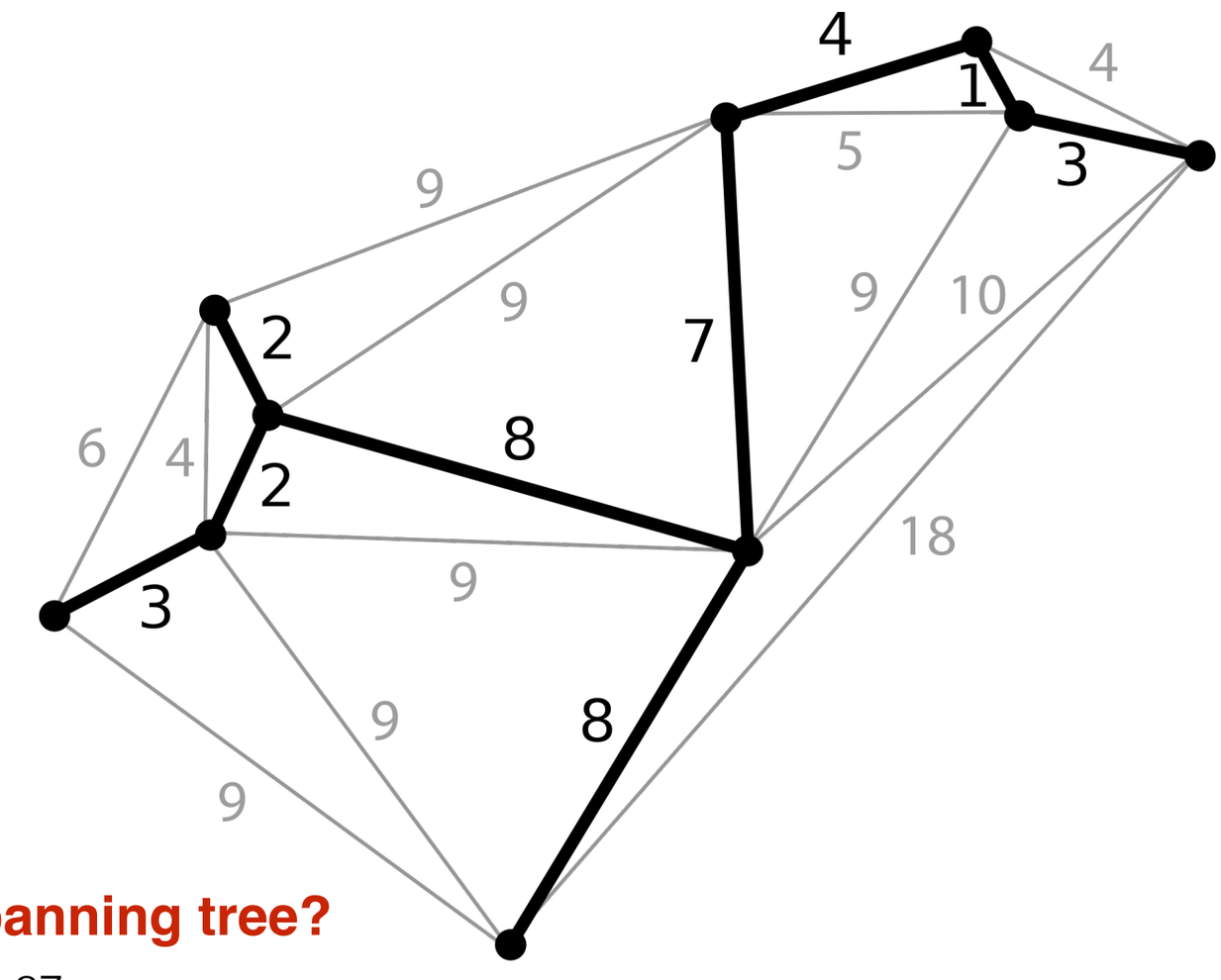

```
BFS(vertex v){
    Define a queue Q;
    Q.enqueue(v);
    Mark v as visited;

    while (!q.isEmpty()){
        q.dequeue(w);
        for (each unvisited vertex u adjacent to w){
            Mark u as visited;
            q.enqueue(u);
        }
    }
}
```

Spanning Trees

Example: Chip Design.
Imagine you would like to design an integrated circuit where it must be possible in principle to reach each pin from any other one. You can picture the circuit with a graph where every vertex is a pin and every connection is an edge.

From an industrial point of view, it would be useful to construct the minimum amount of connections needed for realizing the goal of connecting all the pins.
To this purpose, it is useful to transform the graph to an associated **minimum spanning tree.**

Another example could be the telephone connection among N cities. It is sufficient that you can reach with a cable (the edges) every city from every starting city: so you do not need to connect every city with every other city (or build a complete graph!).



Given a graph: how to construct a minimum spanning tree?

Minimum Spanning Trees

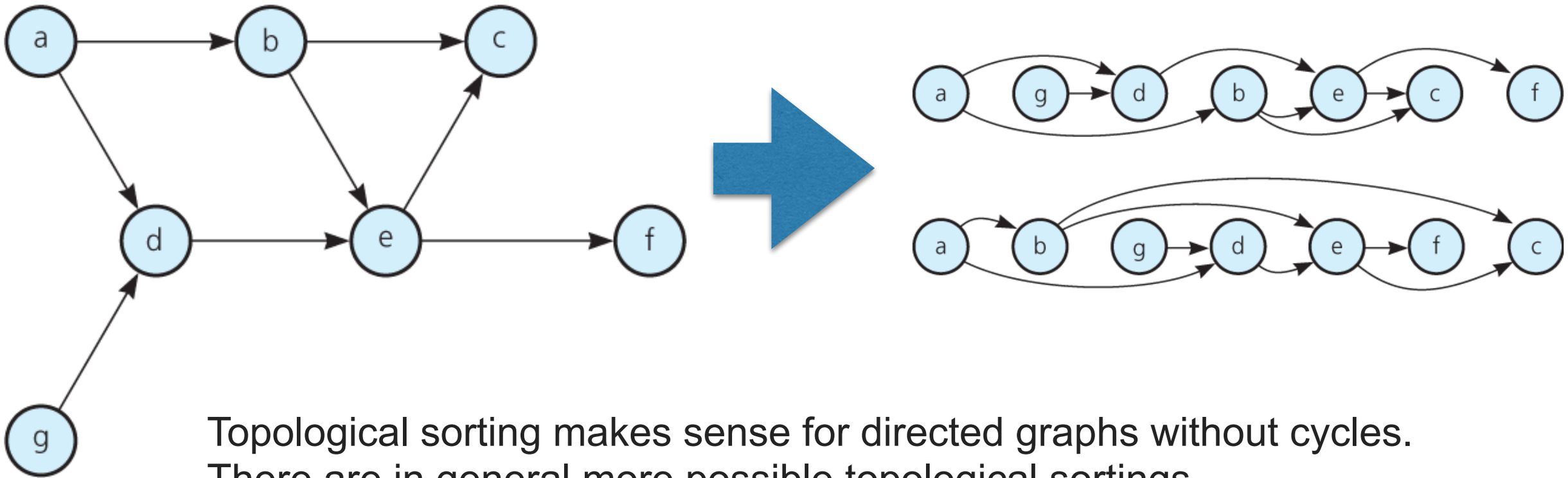
A way to find spanning trees is simply to remember the path taken by the BFS and DFS algorithms while they visit a graph. There is of course more than one spanning tree. Considering weighted graphs, we can consider the spanning tree which has the smallest possible cost (or sum of weights). Such a tree is called **minimum spanning tree**.
The classical "greedy" algorithm for finding a MST is **Prim's Algorithm**.

```
Prim(vertex root){
    Mark root as visited and include it in the MST;
    while (there are unvisited vertices){

        Find the least-cost edge (v,u) from a visited vertex v
        to an unvisited vertex u;

        Mark u as visited;

        Add u and the edge (v,u) to the MST;

    }
}
```

Topological Sorting

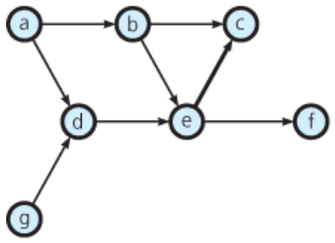
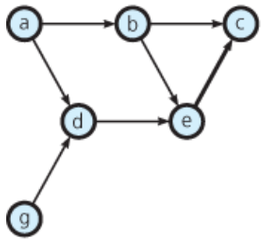
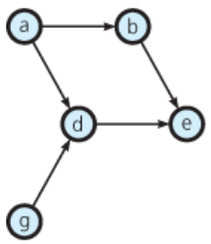
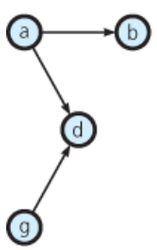
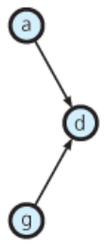
Topological Sorting of a directed graph is a linear ordering of its vertices such that for every directed edge (u,v) from vertex u to vertex v, u comes before v in the **ordering**.

EXAMPLE:
For completing your degree, you have to take a certain amount of courses. The order of the courses is not random though: there are requirements. Your path to the degree can be encoded into a graph:



Topological sorting makes sense for directed graphs without cycles.
There are in general more possible topological sortings.

Topological Sorting Algorithms: DFS

Graph `theGraph`	List `aList`
	
Remove f from `theGraph`; add it to `aList`	
	f
Remove c from `theGraph`; add it to `aList`	
	c f
Remove e from `theGraph`; add it to `aList`	
	e c f
Remove b from `theGraph`; add it to `aList`	
	b e c f
Remove d from `theGraph`; add it to `aList`	
	d b e c f
Remove g from `theGraph`; add it to `aList`	
	g d b e c f
Remove a from `theGraph`; add it to `aList`	
	a g d b e c f

Shortest Path: Dijkstra's Algorithm

In a weighted graph, a natural (and useful!) question is: what is the path between two vertices for which the sum of the weights is minimal? This is the shortest path problem and is solved by the famous **Dijkstra's Algorithm.** **Google Maps uses something similar to this!**

```
shortestPath(Graph G, double *weight){
   Create a set of vertices SV with only vertex 0;
   N = number of vertices in G;

   for (v=0 ; v<N-1 ; i++)
      weight[v] = G[0][v];//G is the adjacency matrix

   for (step = 2 ; step<=N ; step++){
      Find the smallest weight[v] such that v is not in SV;
      Add v to SV;

      for (all vertices u not in SV){
         if (weight[u]>weight[v]+G[v][u])
            weight[u] = weight[v] + G[v][u];
      }
   }
}
```

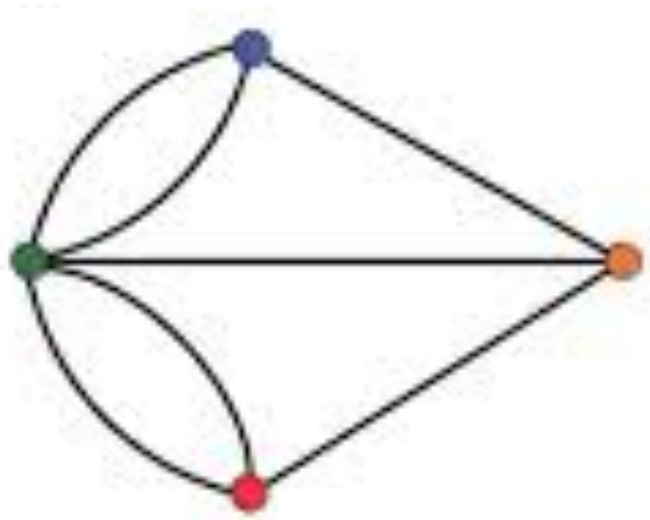

weights are >0 in this implementation.

Circuits

A **circuit** is a path in a graph that begins and ends at the same vertex.
A circuit that visit all the edges of a graph going through each edge only once is called **Euler circuit.**

Euler proved that such a circuit exists if and only if each vertex touches an even number of edges.

Remember for example the Koenigsberg bridges problem:



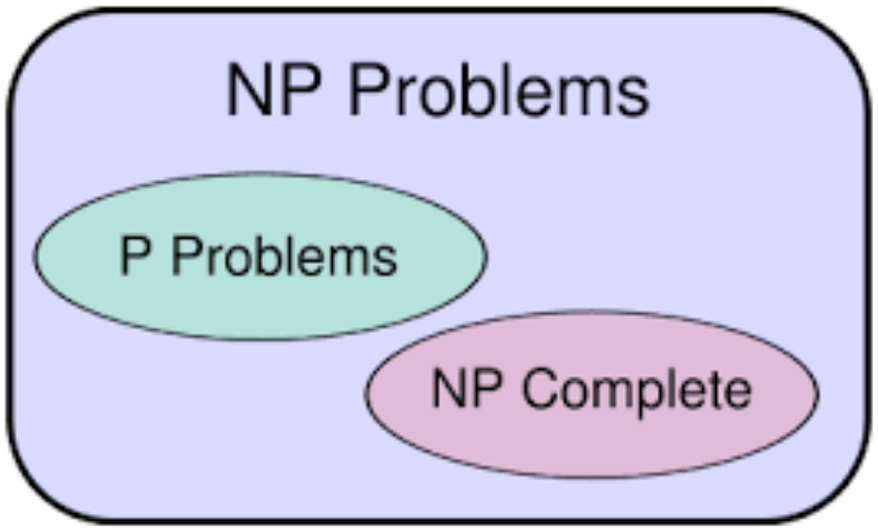
Circuits

A circuit that visit all the vertices of a graph going through each vertex only once is called **Hamilton circuit.**
A famous problem connected to Hamilton circuits is the **Traveling salesperson problem** (TSP). A variation of the problem involves a weighted graph and wTSP involves the less expensive Hamilton circuit.

TSP is considered a “difficult” problem. This means that there is no fast solution (O(poly)) to it.

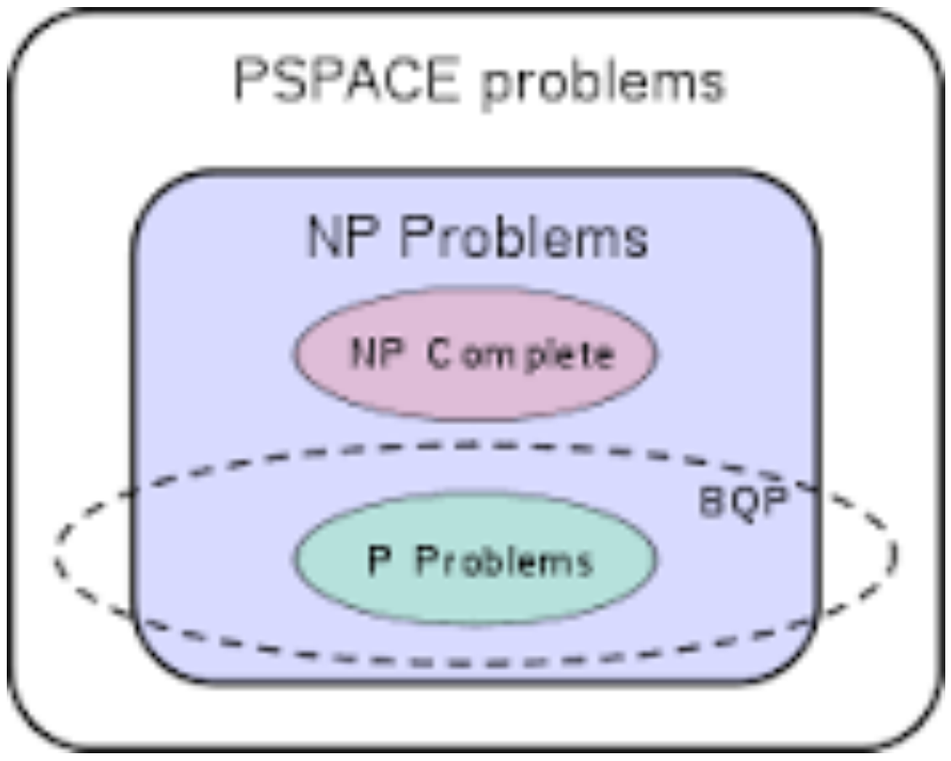
In fact, TSP is an example of **NP-complete** problem and as such it is one of the most difficult decision problems.

Can we classify the difficulty of problems?

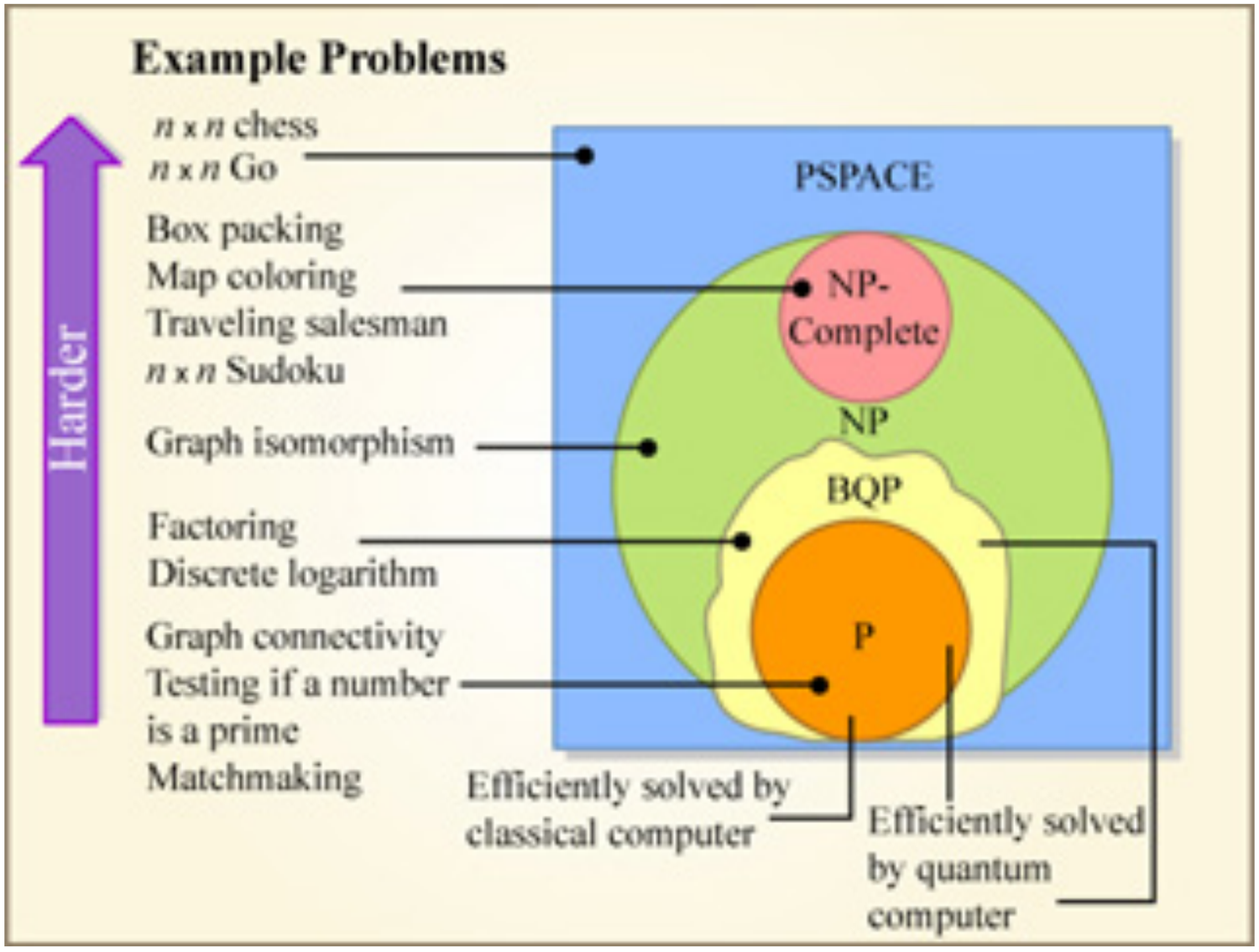


Classical view

Quantum Extension



Complexity Theory



…and much, much more!