University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Tutorial problems
Implementing graph data structures

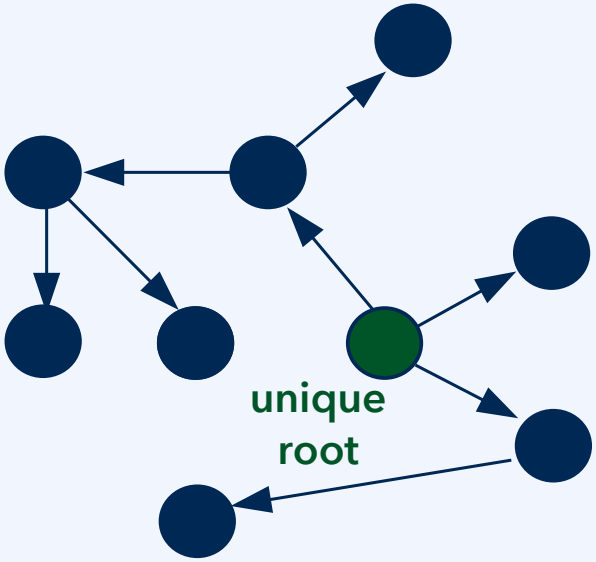
Where opportunity creates success

University of Central Lancashire
UCLan

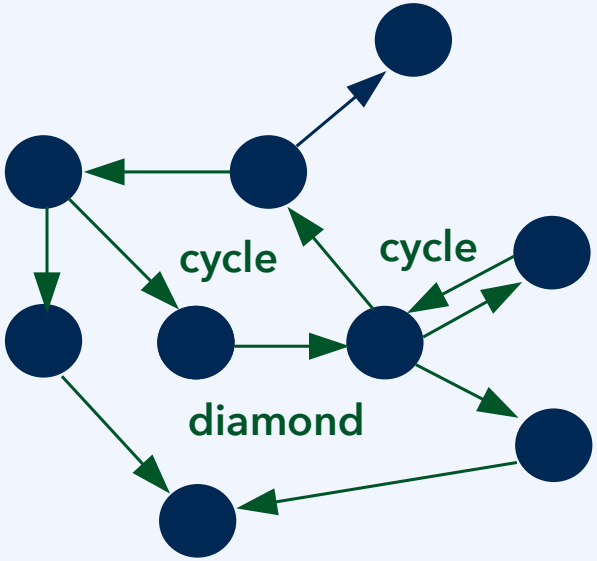
Main concepts from the previous lecture

Trees as a special kind of graph, and graphs as a generalization of trees

tree (a kind of graph)



graph



Remark

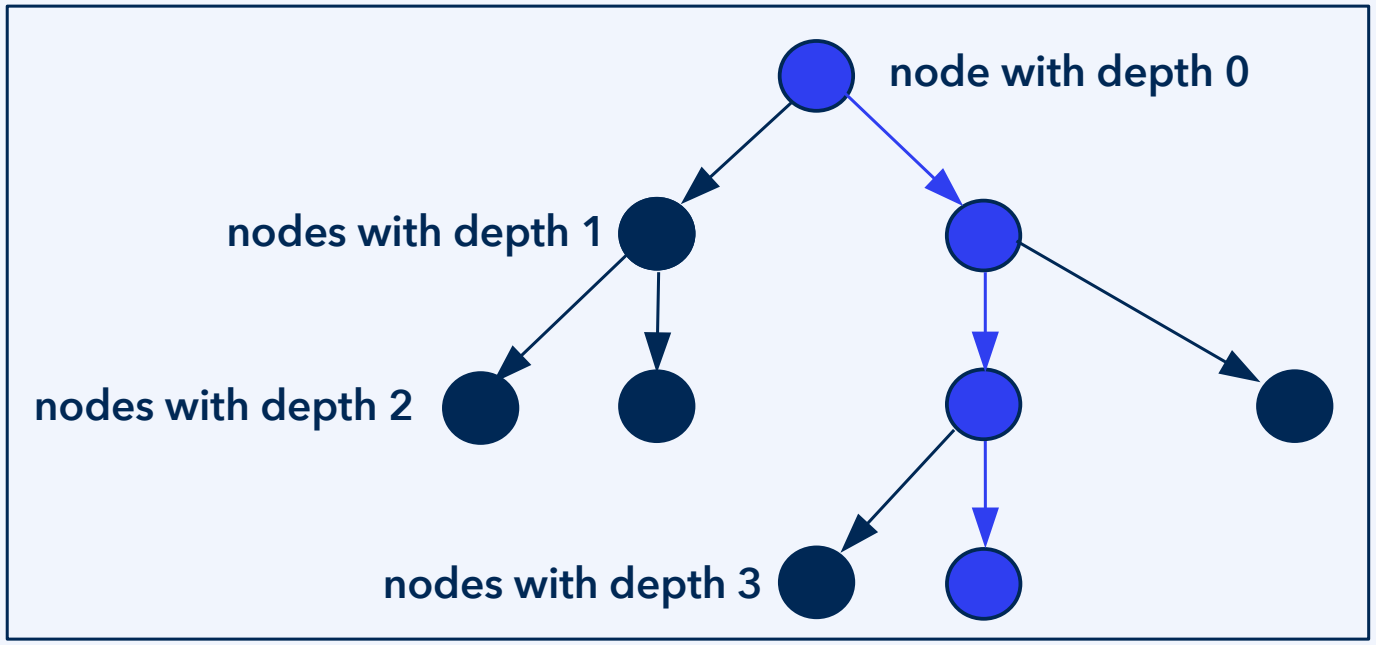
Trees are graphs, and all that is said about graphs applies to trees also.

University of Central Lancashire
UCLan

Main concepts from the previous lecture

Depth of a tree and (re-)balancing of a search tree

To **search for an element** or to **insert a new element**, links have to be followed downward. The number of these links is bounded by the **depth of the tree**.



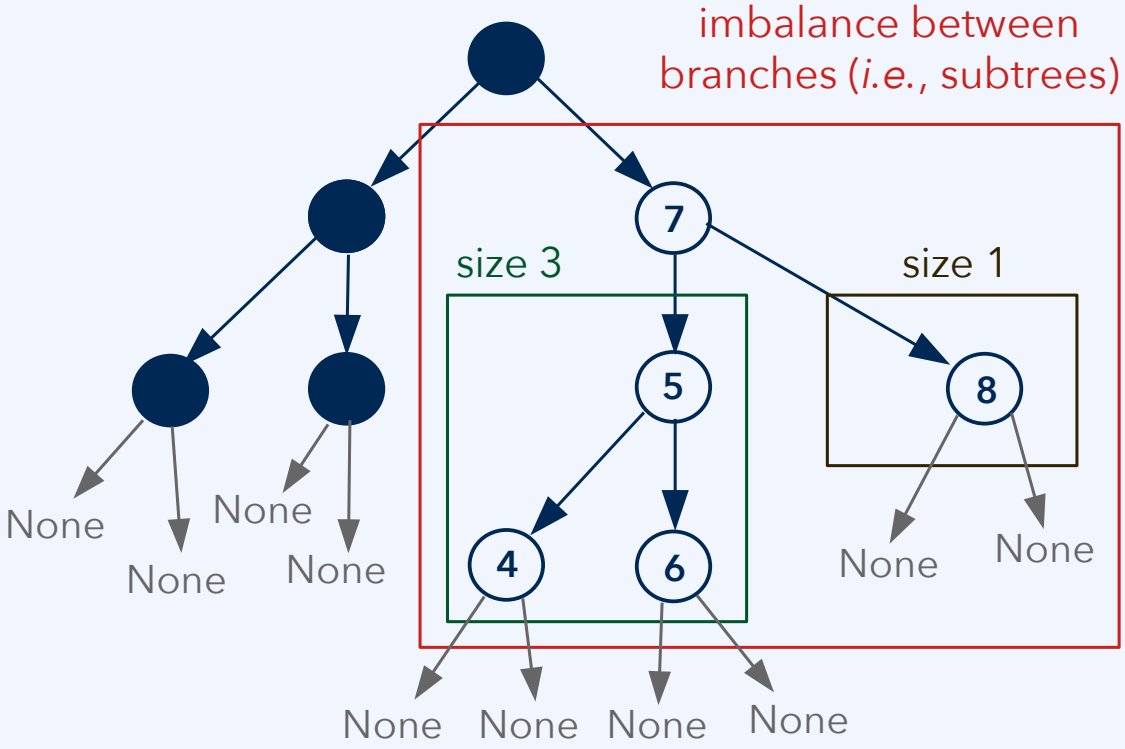
depth of a tree: length of the long-est path

binary search tree with depth 3

Main concepts from the previous lecture

Depth of a tree and (re-)balancing of a search tree

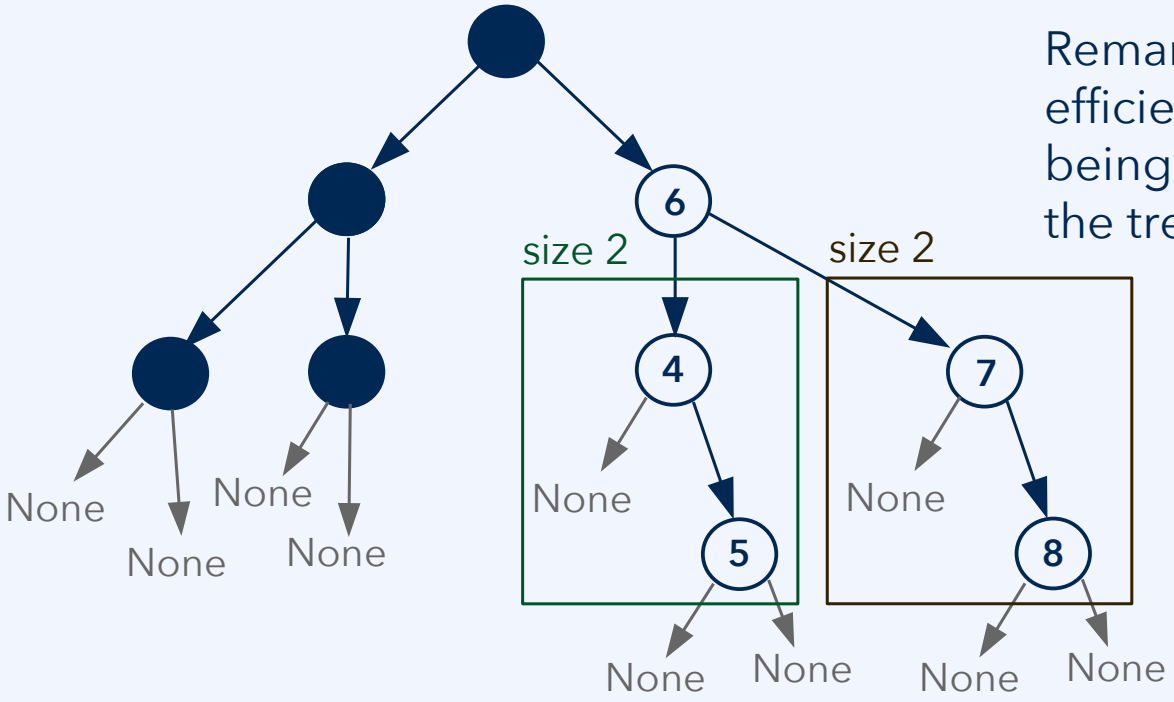
To **search for an element** or to **insert a new element**, links have to be followed downward. The number of these links is bounded by the **depth of the tree**.



Main concepts from the previous lecture

Depth of a tree and (re-)balancing of a search tree

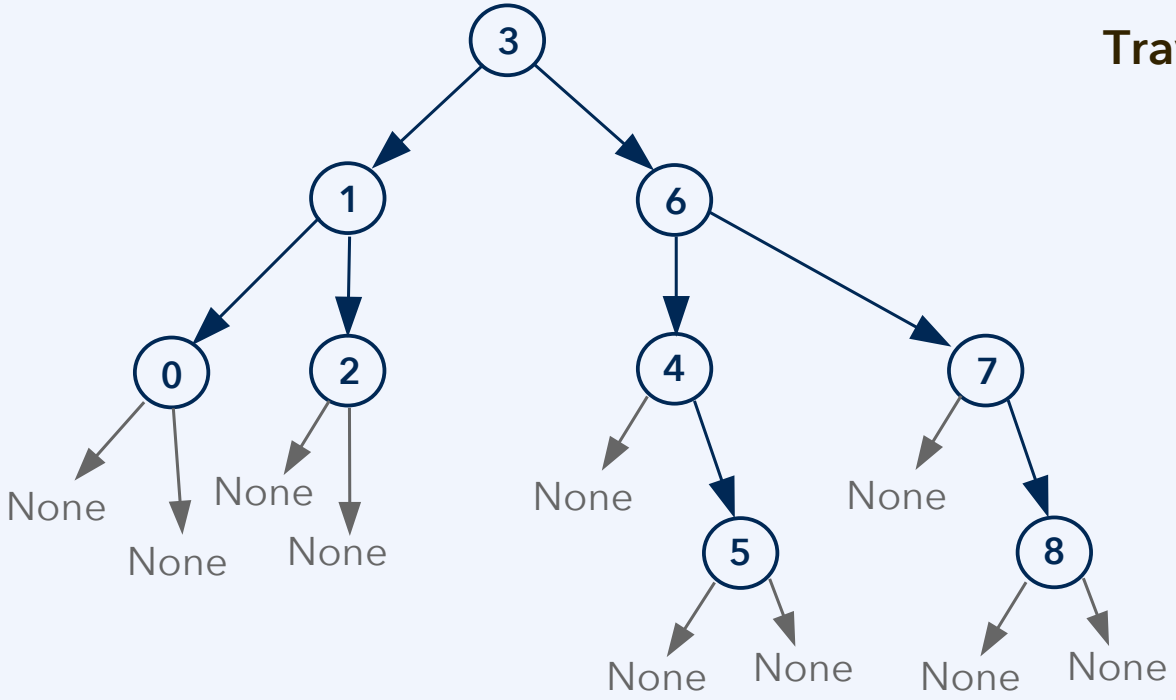
To **search for an element** or to **insert a new element**, links have to be followed downward. The number of these links is bounded by the **depth of the tree**.



Remark: Doing this efficiently relies on being able to traverse the tree in order.

Main concepts from the previous lecture

Traversal of trees and graphs: Binary search trees



Traversal algorithm:

- traverse the left subordinate branch (if there is one)
- visit the root node
- traverse the right subordinate branch (if there is one)

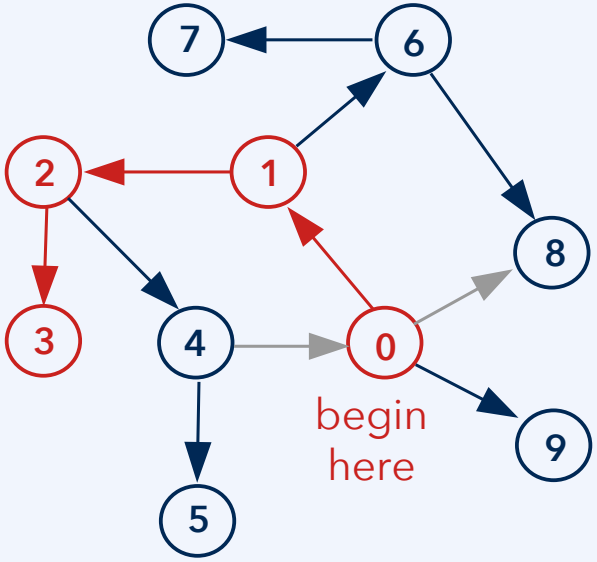
University of Central Lancashire
UCLan

Main concepts from the previous lecture

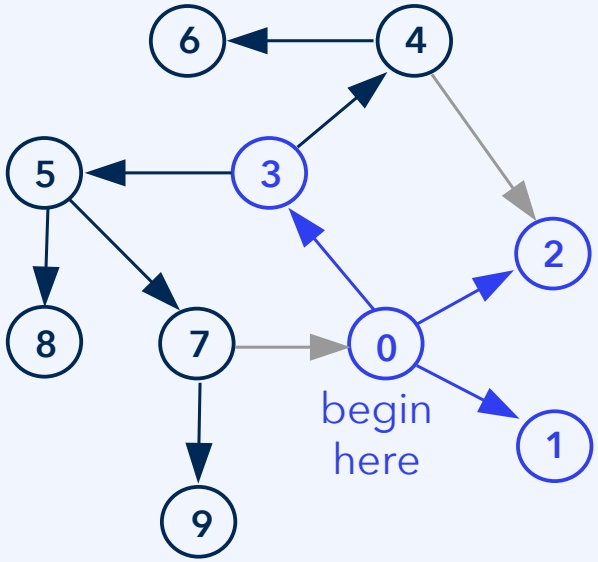
Traversal of trees and graphs: Depth-first search and breadth-first search

DFS always proceeds from the most recently detected node (LIFO).
BFS always proceeds from the node that was detected earliest (FIFO).

depth-first search (DFS)



breadth-first search (BFS)



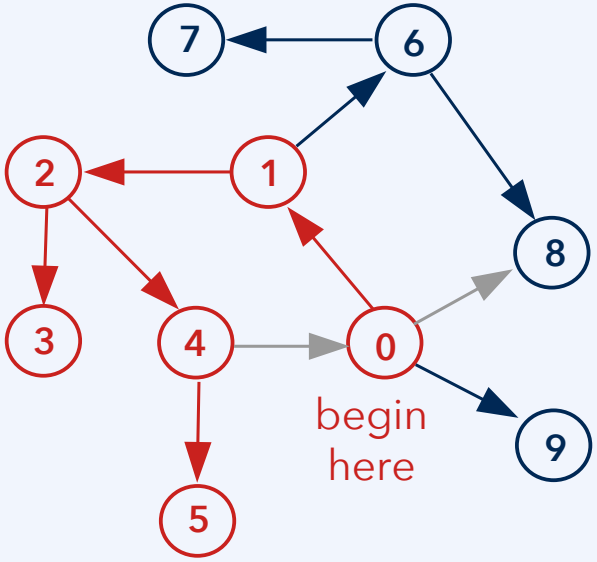
University of Central Lancashire
UCLan

Main concepts from the previous lecture

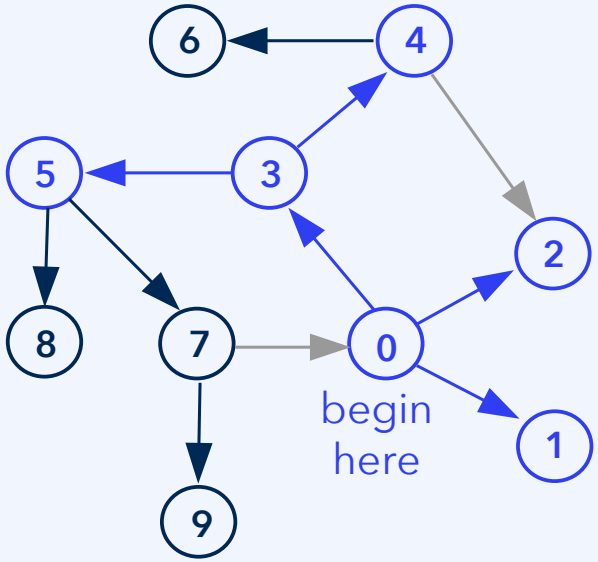
Traversal of trees and graphs: Depth-first search and breadth-first search

DFS always proceeds from the most recently detected node (LIFO).
BFS always proceeds from the node that was detected earliest (FIFO).

depth-first search (DFS)



breadth-first search (BFS)

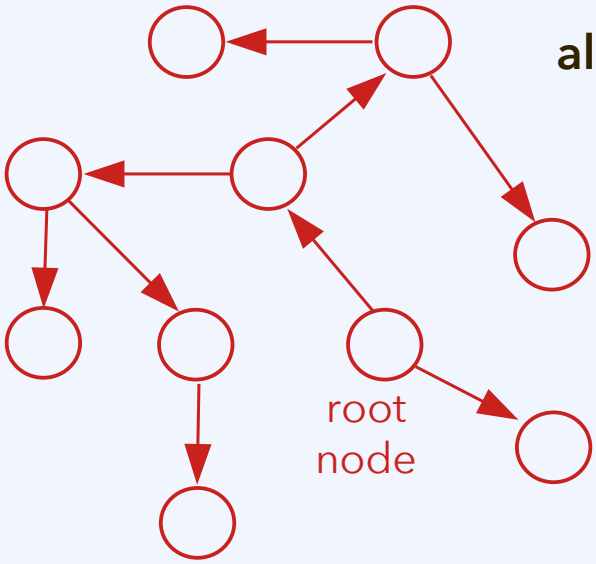


Note: Only elements *to which there is a path from the initial node* can be found.

Related concept: Spanning tree

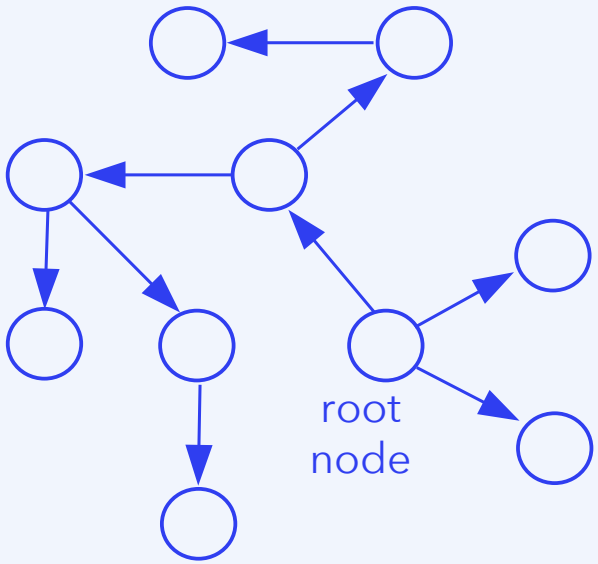
A graph that is not a tree can be reduced to a tree by *eliminating edges*. Such a tree is called a **spanning tree** if it **covers all nodes**. When needed, this is often done by DFS or BFS, retaining only the edges followed for visiting nodes.

DFS spanning tree



greedy algorithm(s)

BFS spanning tree



This construction is only feasible *if there are paths to all nodes from the root*.

University of Central Lancashire
UCLan

Tutorial problems

CO2412

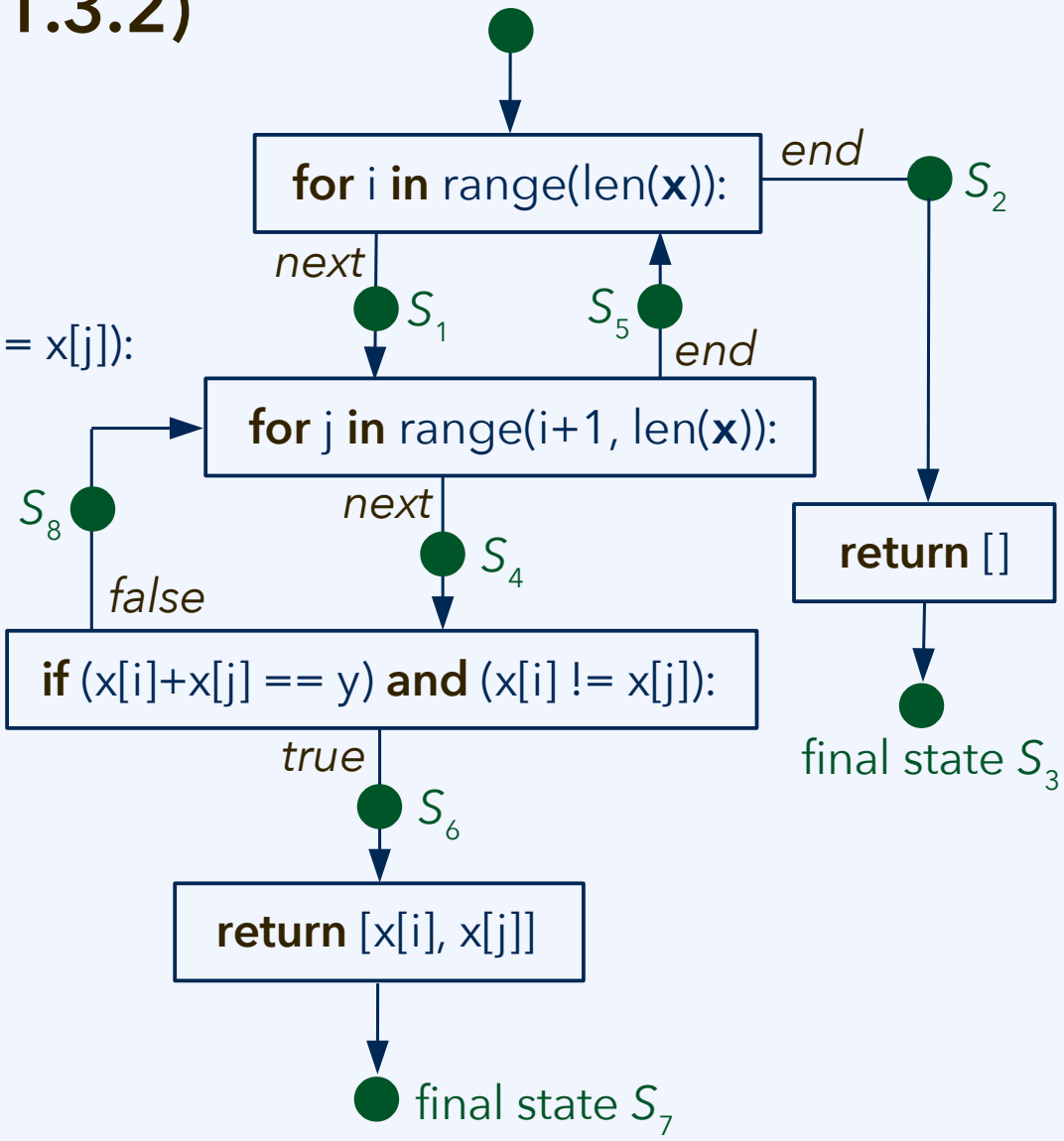
7th December 2021

Number matching (T1.3.2)

```
def natmatch(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

Specification

The function takes a list **x** and a natural number *y* as arguments. If in the list **x**, there are elements a and b which are not equal and add up to y, the list [a, b] is returned; otherwise, [] is returned.



Number matching (T1.3.2)

```
def natmatch(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

Note: Input size *n* given by len(x)

loop executed O(*n*) times:

- loop executed O(*n*) times:
  - O(1) instructions
  - O(1) optional instructions

O(1) optional instructions

$O(n) \cdot O(n \cdot 1) + O(1) = O(n^2)$ instructions

$O(n^2)$ time efficiency

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

Python dictionaries and sets could be used to this effect equivalently.

Example, **x** = [6, 4, 5, 3, 9], y = 11:

6 → 11 – 6 = 5 not found in storage
insert 6 into storage

4 → 11 – 4 = 7 not found in storage
insert 4 into storage

5 → 11 – 5 = 6 found in storage
return [6, 5]

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

Python dictionaries and sets are implemented as dynamically resized **hash tables**:

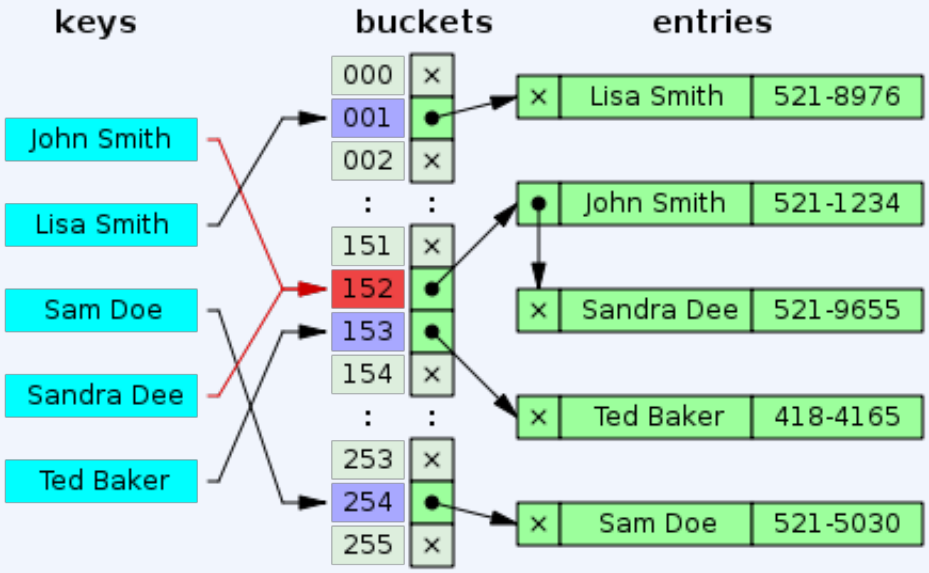


Fig. from Wikipedia, "Hash table"

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan; worst case still $O(n^2)$:

```
def natmatch(x, y):
    mydict = {}
    for i in range(len(x)):
        c = y - x[i]
        if c in mydict:
            return [c, x[i]]
        mydict[x[i]] = i
    return []
```

In the worst case, this data structure has $O(n)$ time for search and insertion. For the average case, it is highly efficient.

Python dictionaries and sets are implemented as dynamically resized **hash tables**:

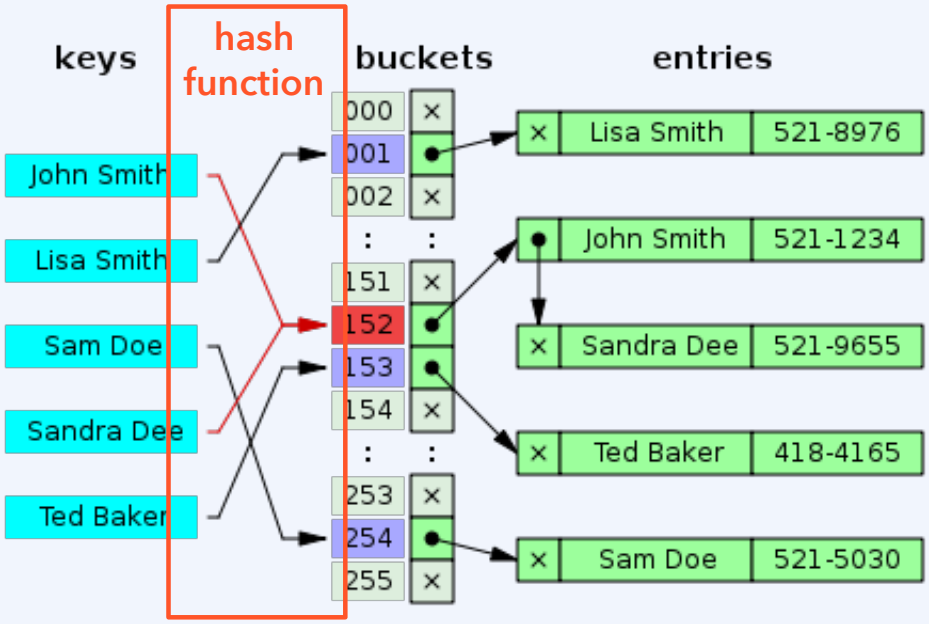


Fig. from Wikipedia, "Hash table"

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search** operation and one **insertion** operation.

What is the time efficiency? How does it depend on the employed data structure?

Example, **x** = [6, 4, 5, 3, 9], y = 11:

6 → 11 – 6 = 5 not found in storage
insert 6 into storage

4 → 11 – 4 = 7 not found in storage
insert 4 into storage

5 → 11 – 5 = 6 found in storage
return [6, 5]

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search** operation and one **insertion** operation.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
- A sorted dynamic array?
- A sorted linked list or an unbalanced search tree?
- A balanced search tree?
- Python sets or dicts?

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

Each with one **search (s)** operation and one **insertion (i)** operation.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
  s done in O(*n*), **i** done in O(1).
- A sorted dynamic array?
  s done in O(log *n*), **i** done in O(*n*).
- A sorted linked list or an unbalanced search tree?
  s done in O(*n*), **i** done in O(*n*).
- A balanced search tree?
  s and **i** both done in O(log *n*).
- Python sets or dicts?
  Worst case O(*n*) for both **s** and **i**.

Number matching (T1.3.2)

Improved algorithm implemented by Harry Rowan:

```
def natmatch(x, y):
    initialize empty storage
    for i in range(len(x)):
        c = y - x[i]
        if storage.contains(c):
            return [c, x[i]]
        storage.insert(x[i])
    return []
```

O(*n*) loop operations.

- O(log *n*) time per iteration.

O(*n* log *n*) with a balanced tree.

What is the time efficiency? How does it depend on the employed data structure?

How about:

- Unsorted linked list or dyn. array?
  s done in O(*n*), **i** done in O(1).
- A sorted dynamic array?
  s done in O(log *n*), **i** done in O(*n*).
- A sorted linked list or an unbalanced search tree?
  s done in O(*n*), **i** done in O(*n*).
- A balanced search tree?
  s and **i** both done in O(log *n*).
- Python sets or dicts?
  Worst case O(*n*) for both **s** and **i**.

University of Central Lancashire
UCLan

Cashier problem (T2.2)

The cashier problem is specified as follows. The function solving the problem has two arguments:

1) first, a natural number, given in the smallest currency unit (*e.g.*, pence), representing an **amount of money** that is to be paid out;
2) second, a sorted list with the **values of the existing coin types**, in the same currency unit (we assume that "1" is always among these values).

As the function's return value, we expect a **list containing coin values** that add up to the requested amount; this must be the **shortest possible list**, *i.e.*, we want to use as few coins as possible.

Note that as a precondition it is assumed that the list passed as the function's second argument is already sorted.

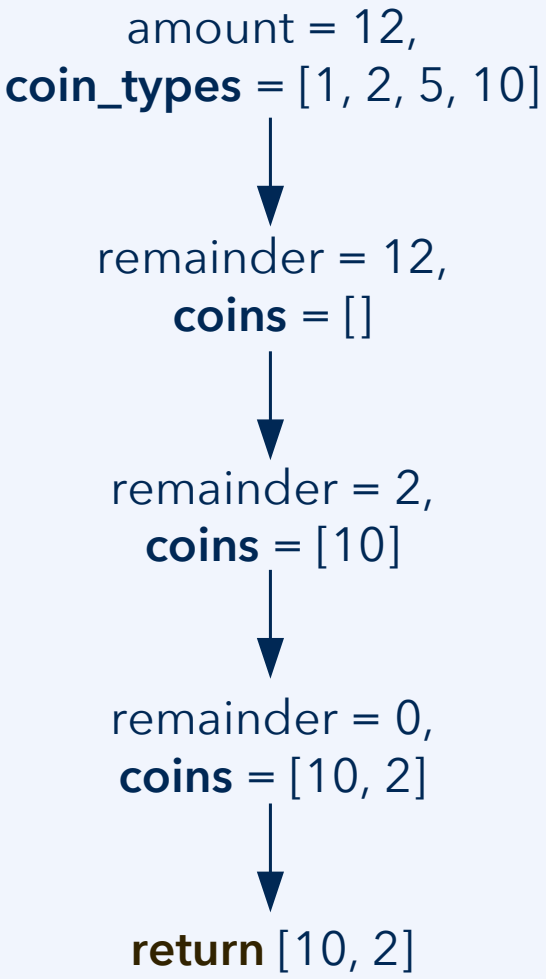
Cashier problem (T2.2)

greedy algorithm

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount

    while remainder >= coin_types[0]:

        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

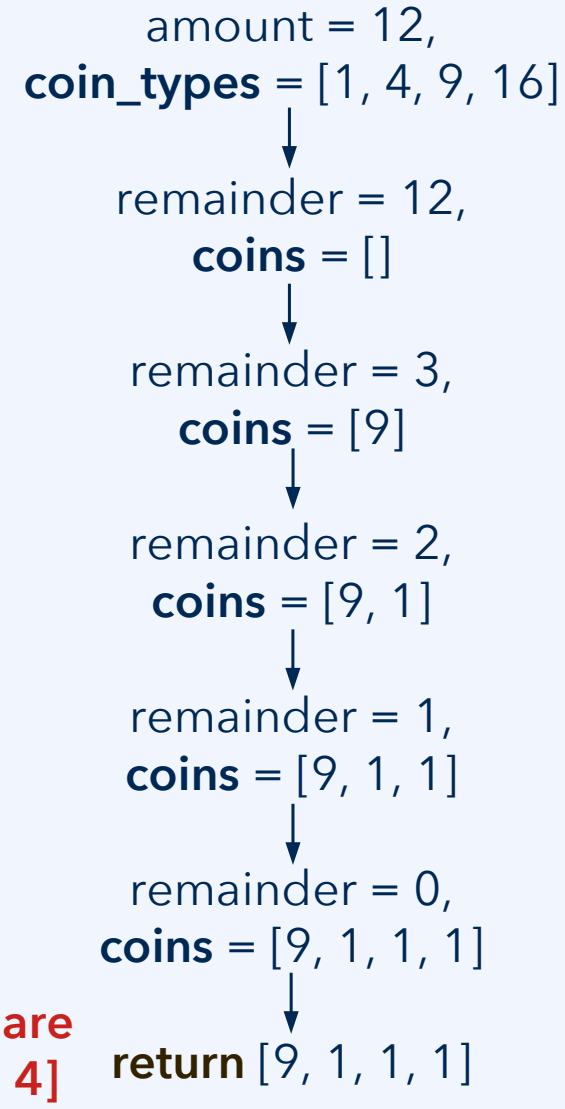


Cashier problem (T2.2)

Condition for the greedy algorithm:

The shortest sum containing coin *x* never consists of more coins than the shortest equivalent sum containing only coins < *x*.

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount
    while remainder >= coin_types[0]:
        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```



Cashier problem (T2.2)

greedy algorithm

Let n = amount and k = len(**coin_types**).

```
def cashier(amount, coin_types):
    coins = []                                  O(1) instructions
    remainder = amount

    while remainder >= coin_types[0]:           Upper bound: O(n) iterations

        for i in range(len(coin_types)-1, -1, -1):    – Upper bound: O(k) iterations
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

- O(1) time on average, for a well-managed **dyn. array**. O(1) time worst case if we were using a **linked list**.

Cashier problem (T2.2)

greedy algorithm

Let n = amount and k = len(**coin_types**).

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount

    while remainder >= coin_types[0]:

        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

O(1) instructions

Upper bound: O(n) iterations

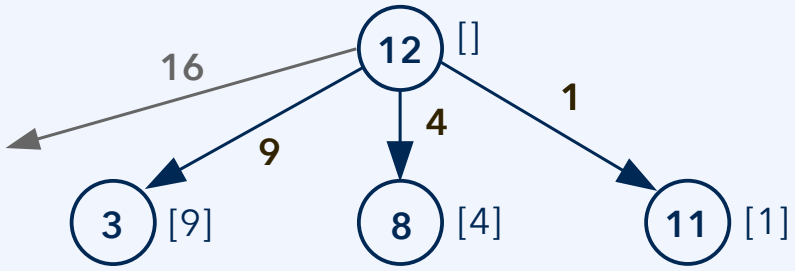
- Upper bound: O(k) iterations
  - O(1) time* per iteration
    [*rigorous with a linked list]

O(kn) time efficiency

that is O(n) if k is treated as constant

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem

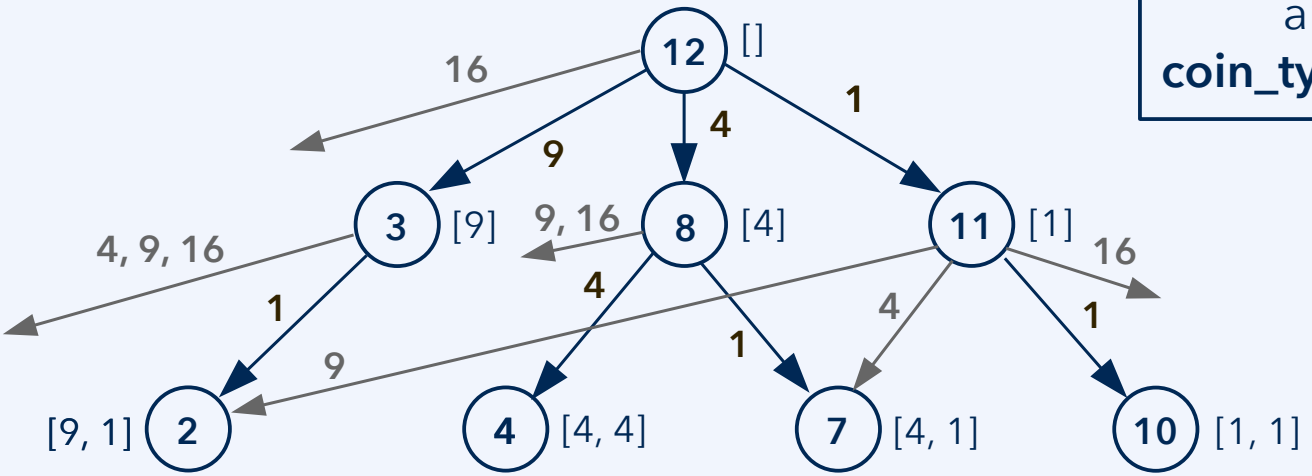


amount = 12,
coin_types = [1, 4, 9, 16]

If we return a 9-valued coin, the remainder reduces to 3.
If we return a 4-valued coin, the remainder reduces to 8.
If we return a 1-valued coin, the remainder reduces to 11.

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem



amount = 12,
coin_types = [1, 4, 9, 16]

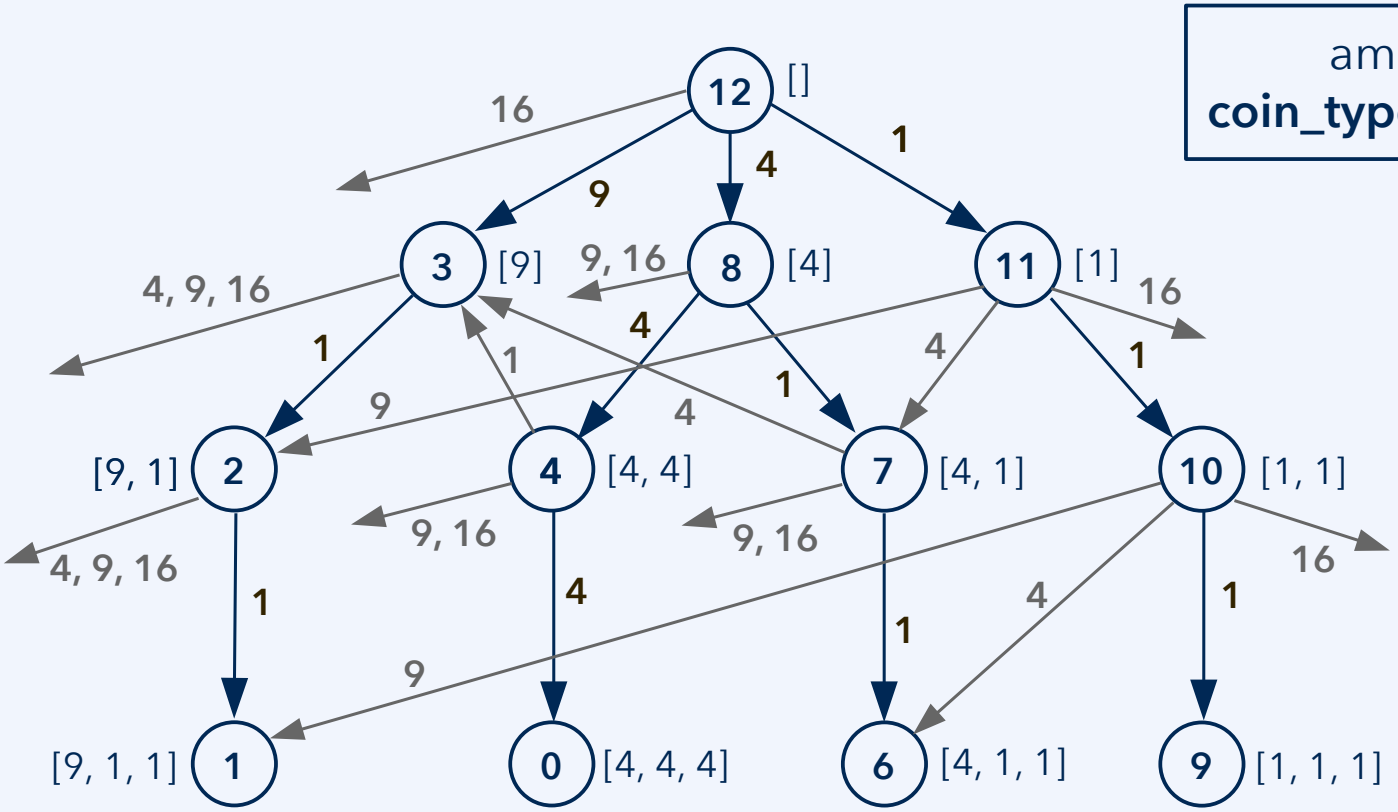
one coin required

two coins required

A remainder of 12 currency units can be reached using zero coins.
A remainder of 3, 8, or 11 can be reached using one coin.
A remainder of 2, 4, 7, or 10 can be reached using two coins.

Cashier problem (T2.2)

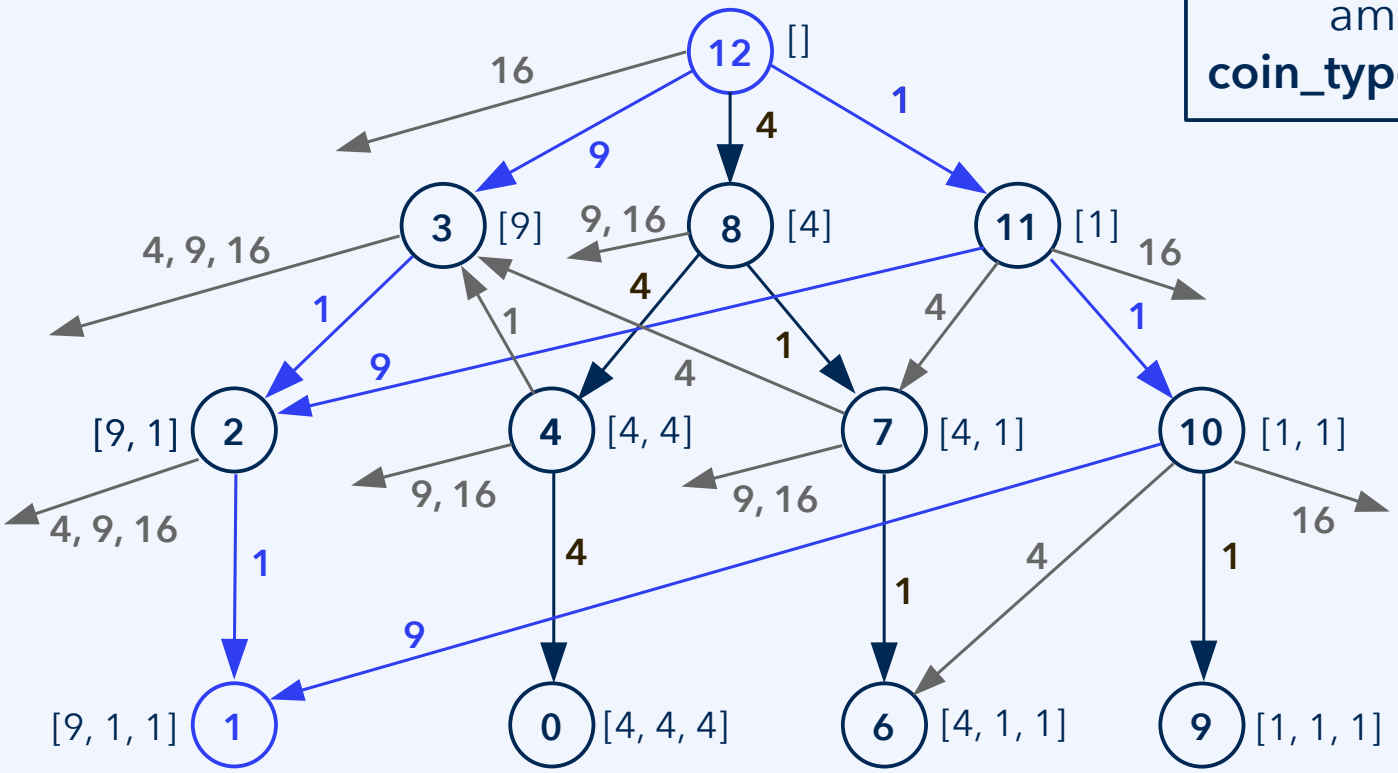
Illustration: Dynamic programming algorithm for the cashier problem



Cashier problem (T2.2)

Overlapping subproblems, *e.g.*, equivalence of [9, 1, 1], [1, 9, 1], and [1, 1, 9]

amount = 12,
coin_types = [1, 4, 9, 16]



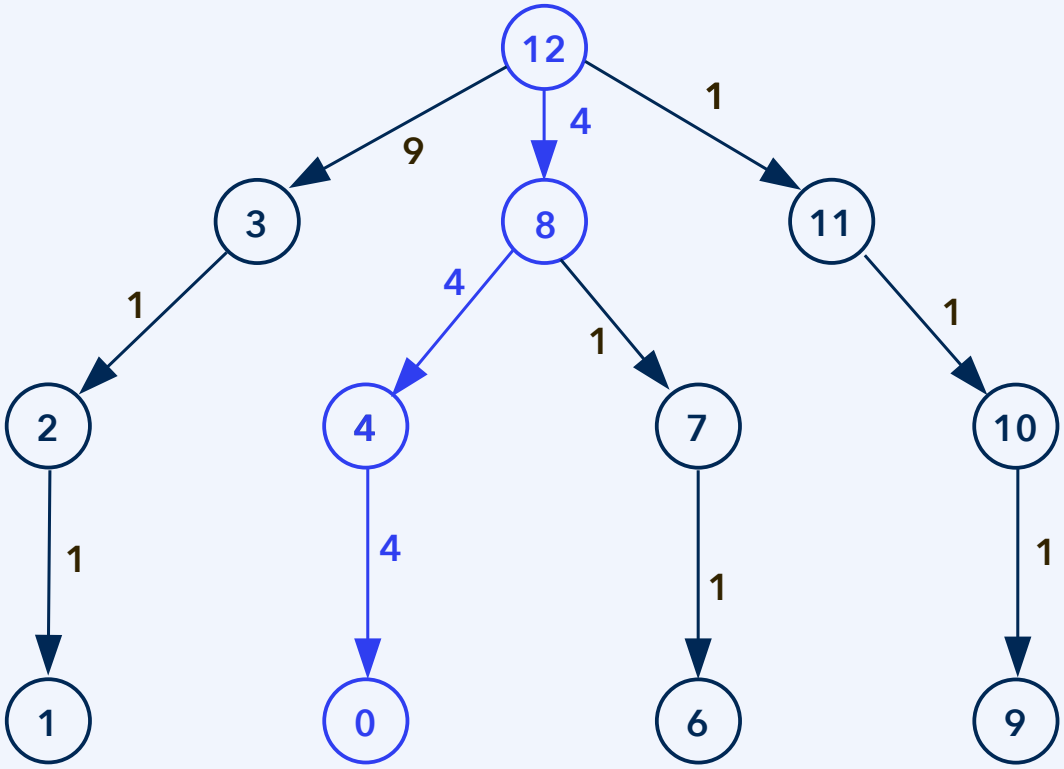
one coin required

two coins required

three coins required

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem



This reduces to computing a **BFS spanning tree**,

over a graph with at most $n+1$ nodes,

with node out-degree upper bound of k.

Time efficiency $O(kn)$, same as for the greedy algorithm.

University of Central Lancashire
UCLan

Implementing graph data structures

CO2412

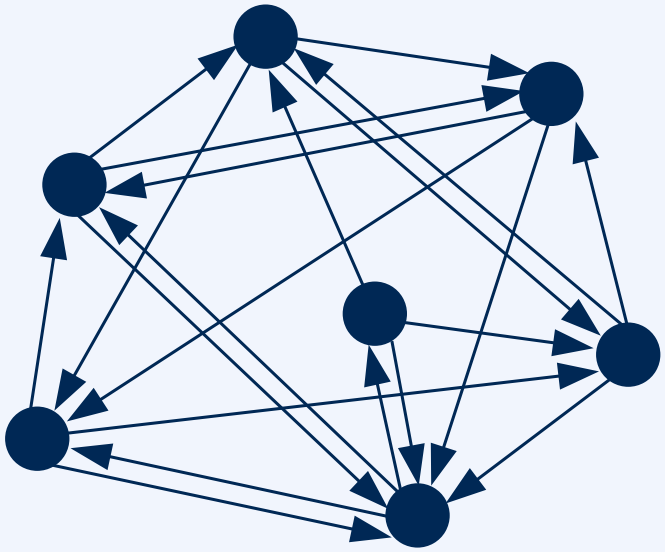
7th December 2021

University of Central Lancashire
UCLan

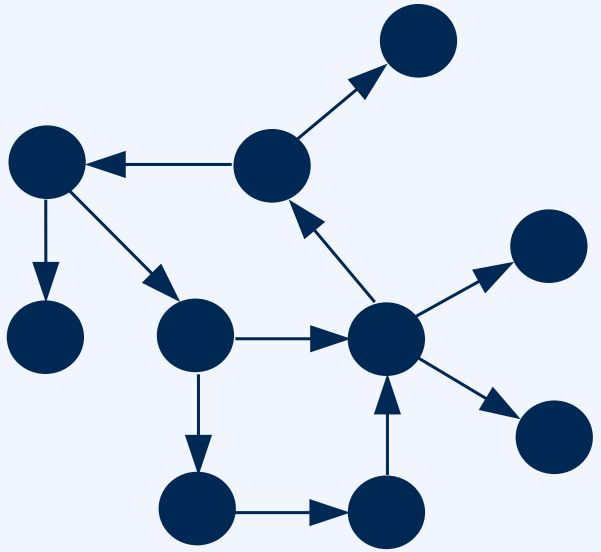
Graphs as data structures: Implementation

Neighbour lists, implemented as **adjacency or incidence lists**, are most suitable for **sparse graphs**. Matrix-like data structures are best for **dense graphs**.

dense graphs

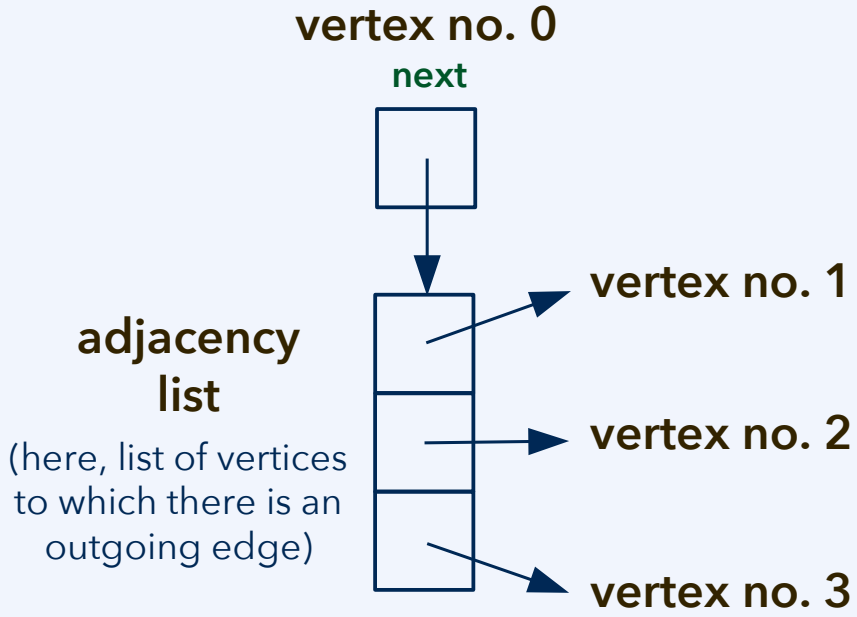


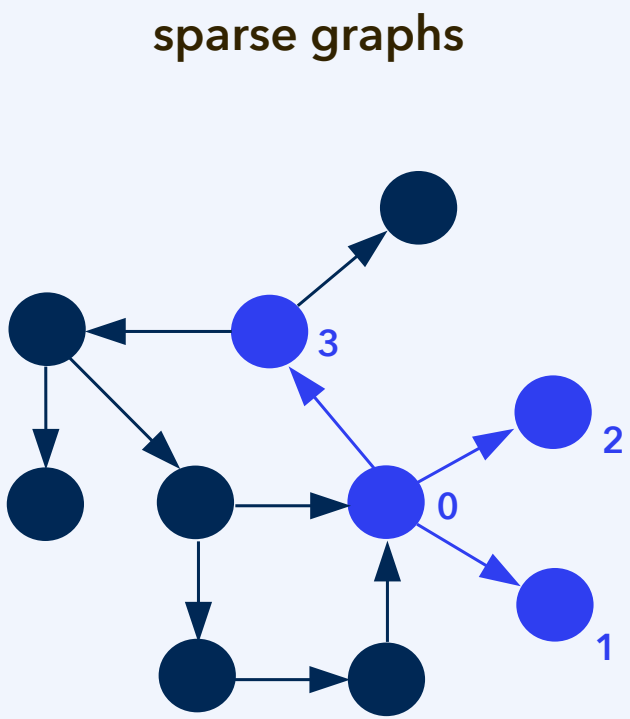
sparse graphs



Graphs as data structures: Implementation

There are a variety of ways of implementing graphs. It depends on the kind of graph and the use case which of them is the most suitable one.

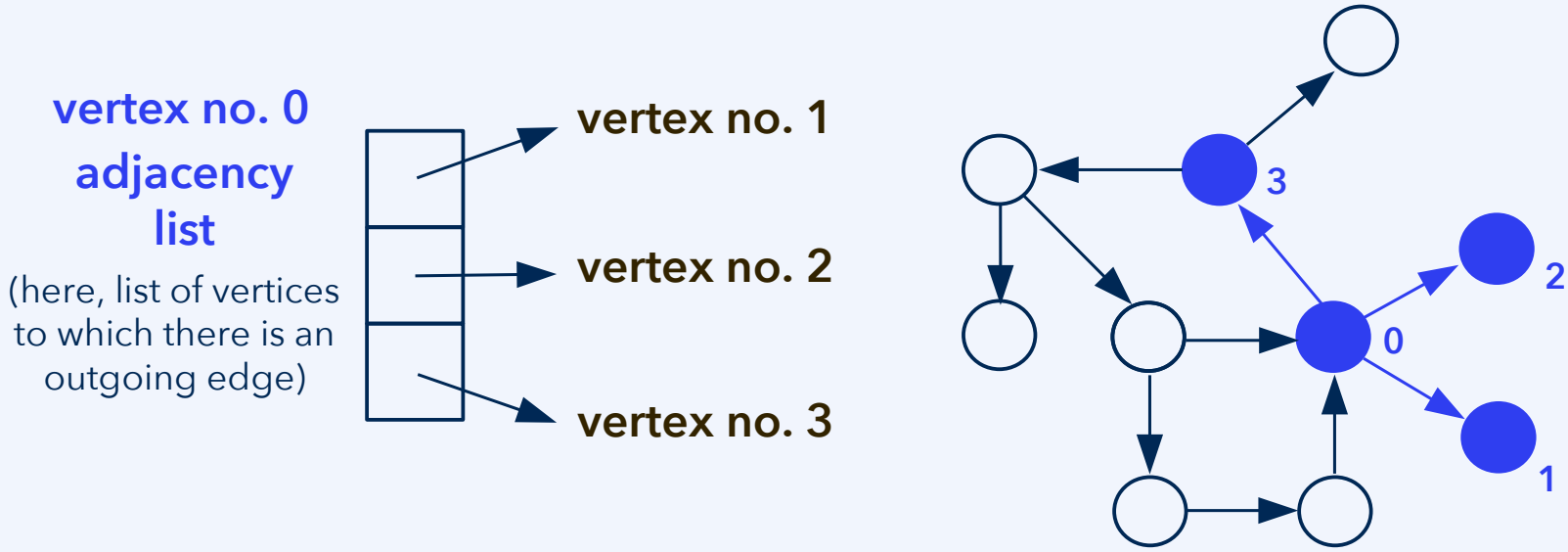




Graphs as data structures: Implementation

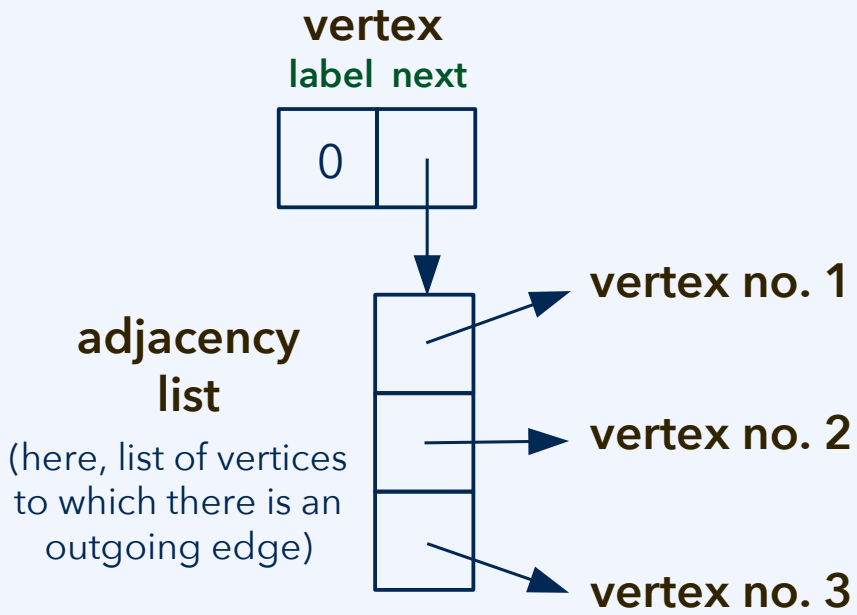
Remark: If the vertices do not have any specific properties at all, there is no strict need to distinguish them from their outgoing adjacency lists.

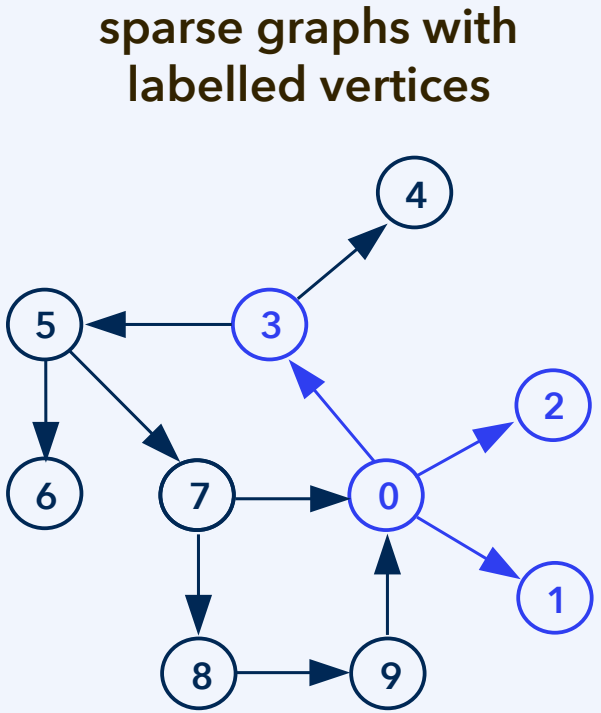
sparse graphs



Graphs as data structures: Implementation

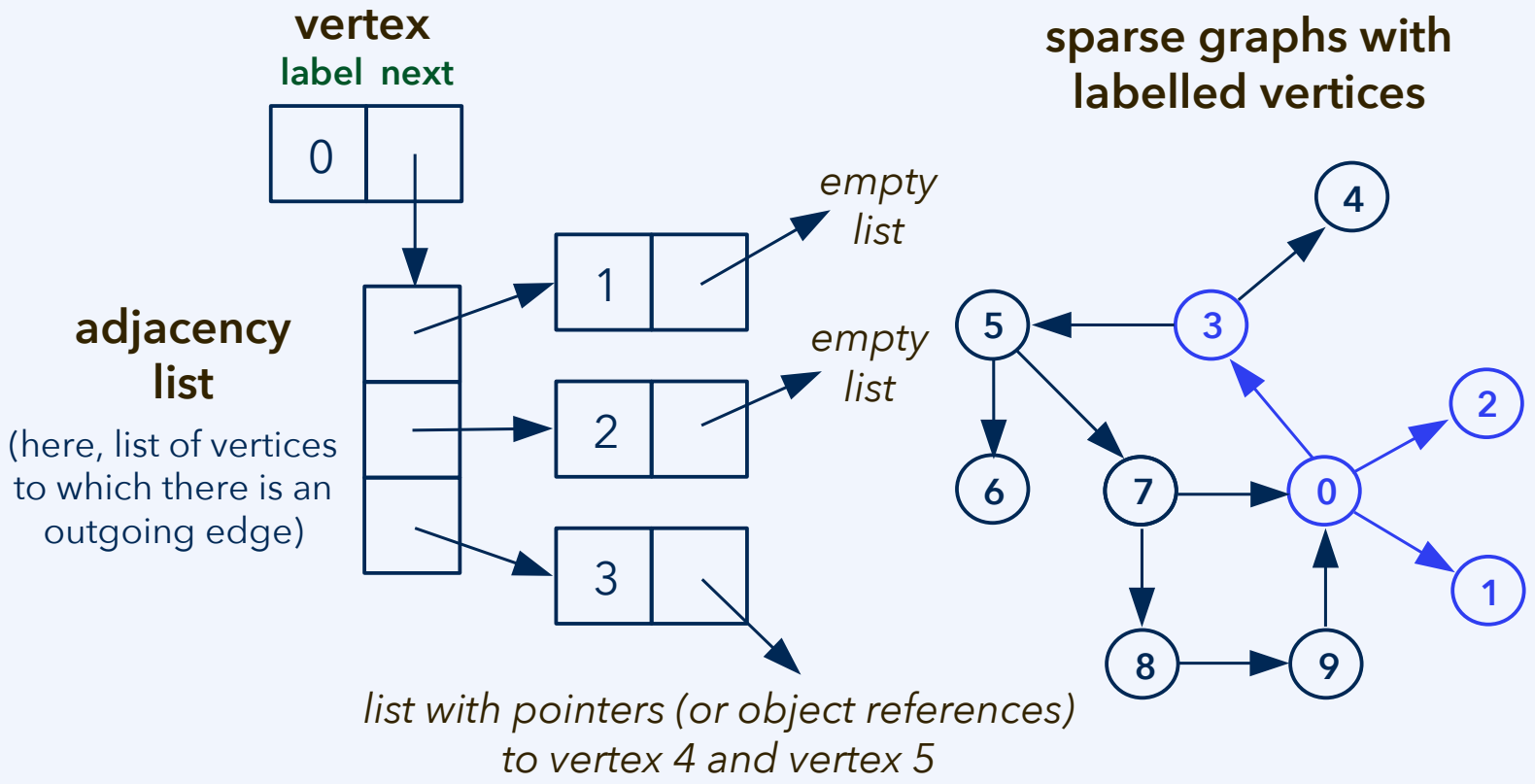
Most technically relevant use cases work with labelled graphs, where at least the vertices (often also the edges) are associated with a data item, the label.





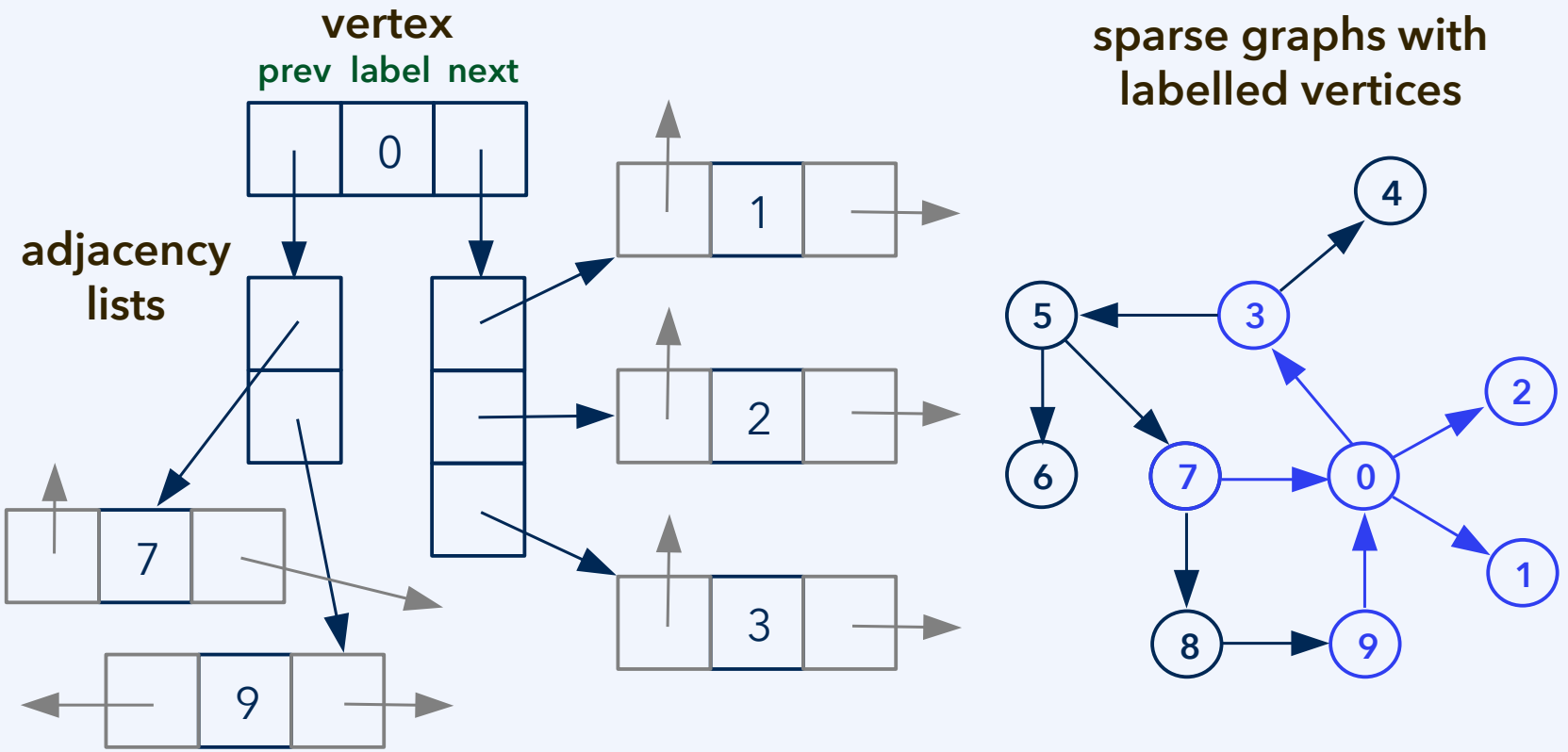
Graphs as data structures: Implementation

Remark: This construction is particularly suitable for tree data structures, since trees are by definition sparse graphs, and they normally contain data items.



Graphs as data structures: Implementation

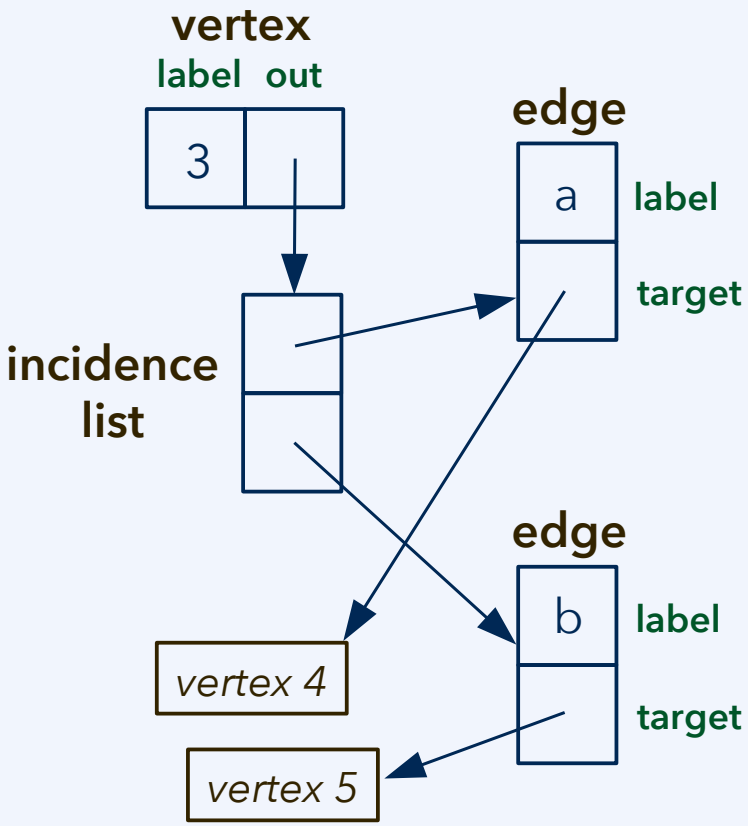
Instead of singly list data structures, doubly linked data structures can also be used; *e.g.*, with an additional adjacency list pointing to predecessor nodes.

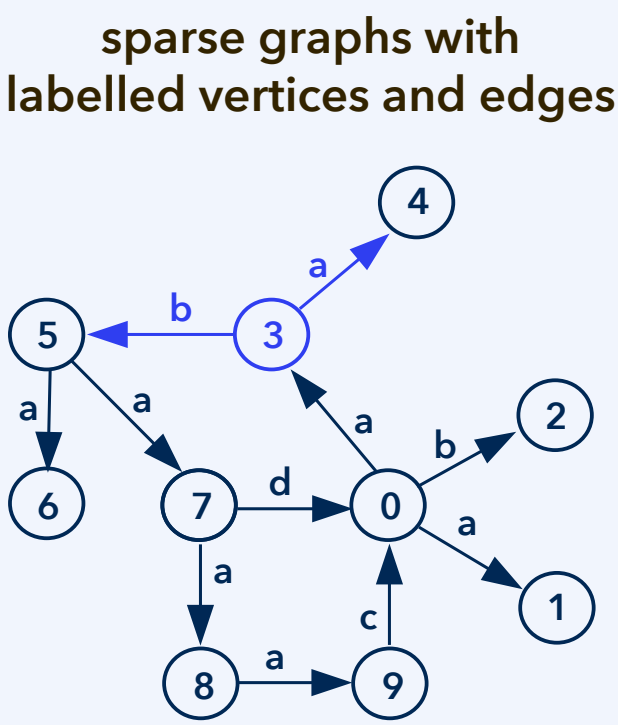


University of Central Lancashire
UCLan

Graphs as data structures: Implementation

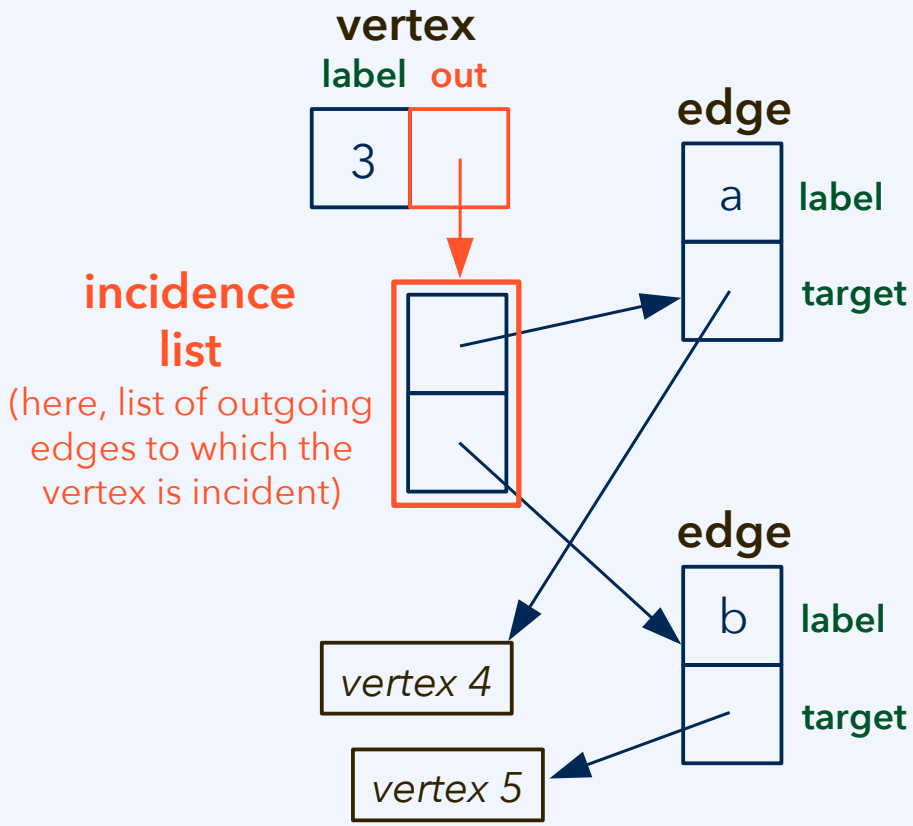
For adjacency lists or incidence lists, a variety of data structures can be used, *e.g.*, dynamic arrays. They need not be sequential data structures.

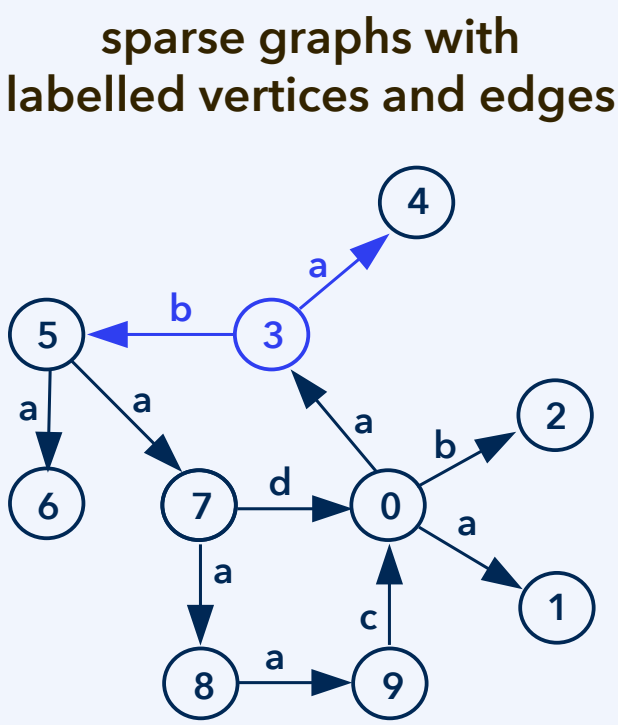




Graphs as data structures: Implementation

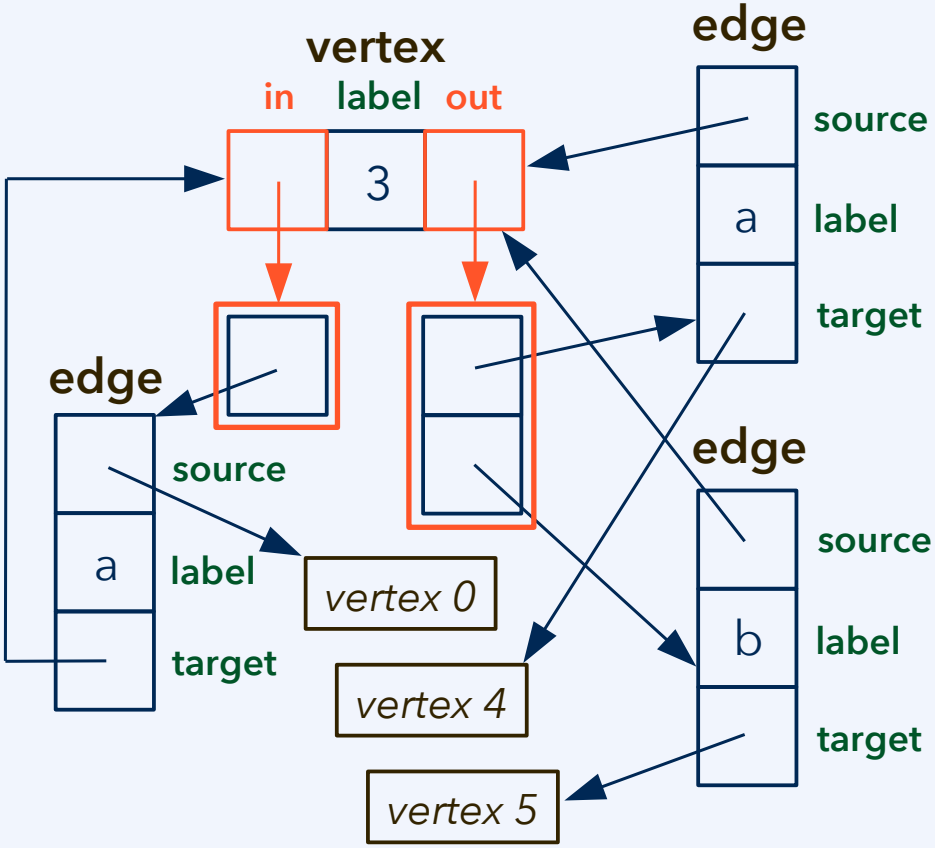
For adjacency lists or incidence lists, a variety of data structures can be used, *e.g.*, dynamic arrays. They need not be sequential data structures.

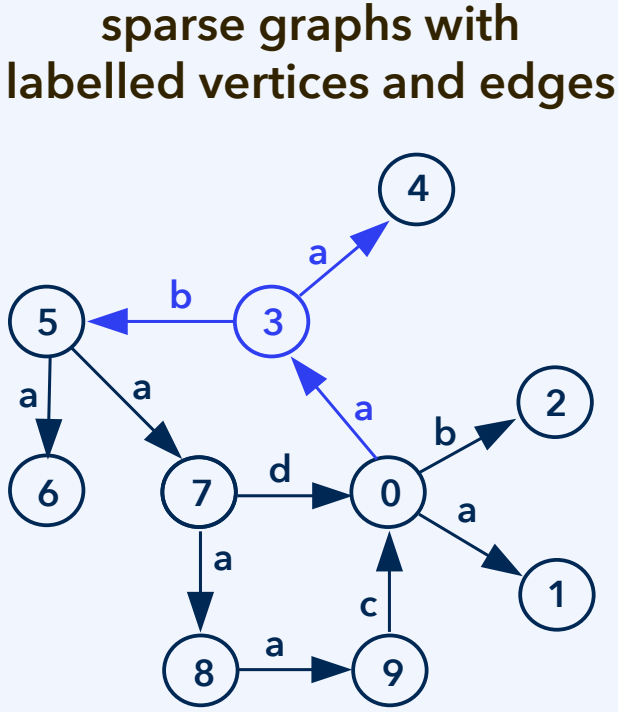




Graphs as data structures: Implementation

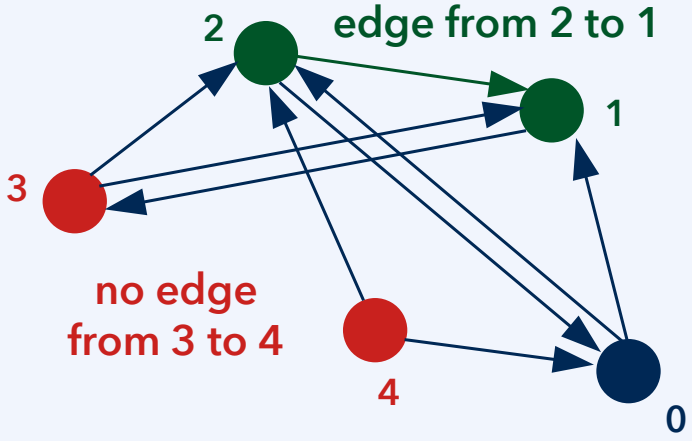
Instead of singly list data structures, the corresponding doubly linked alternatives can always be used; *e.g.*, with links to source nodes and incoming edges.





Graphs as data structures: Implementation

Matrix-like data structures in Python include lists of lists (*i.e.*, 2D dynamic arrays), if the numpy library is used, two-dimensional static arrays. For graphs, the most relevant data structure of this type is the **adjacency matrix**.




```
adj = [  [0,  1,  1,  0,  0],
         [0,  0,  0,  1,  0],
         [1,  1,  0,  0,  0],
         [0,  1,  1,  0,  0],
         [1,  0,  1,  0,  0]  ]
```

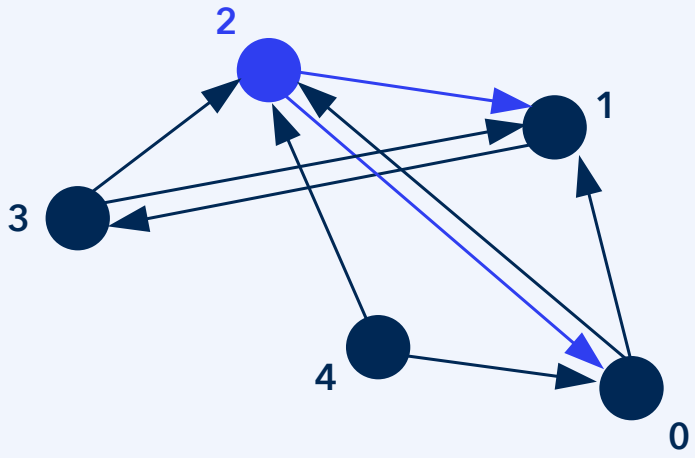
adj[2][1] = **1**, or **True**

adj[3][4] = **0**, or **False**

University of Central Lancashire
UCLan

Graphs as data structures: Implementation

Matrix-like data structures in Python include lists of lists (*i.e.*, 2D dynamic arrays), if the numpy library is used, two-dimensional static arrays. For graphs, the most relevant data structure of this type is the **adjacency matrix**.




```
adj = [  [0,  1,  1,  0,  0],   out of node 0
         [0,  0,  0,  1,  0],   out of node 1
         [1,  1,  0,  0,  0],   out of node 2
         [0,  1,  1,  0,  0],   out of node 3
         [1,  0,  1,  0,  0]  ]
```

For a sparse graph, the vast majority of entries in the 2D array/matrix is zero. Adjacency matrices are commonly only used when expecting a **dense graph**.

University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Tutorial problems
Implementing graph data structures

Where opportunity creates success