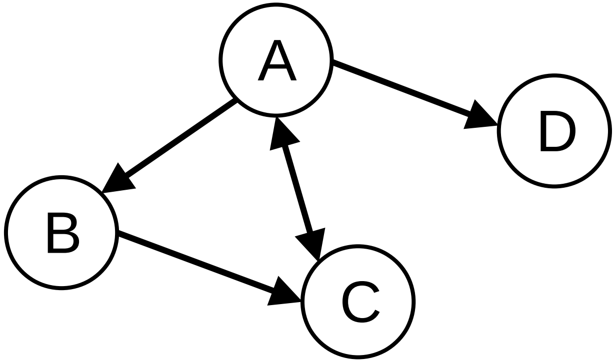
C03 Graph Traversal

Graphs and Trees
Traversal

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

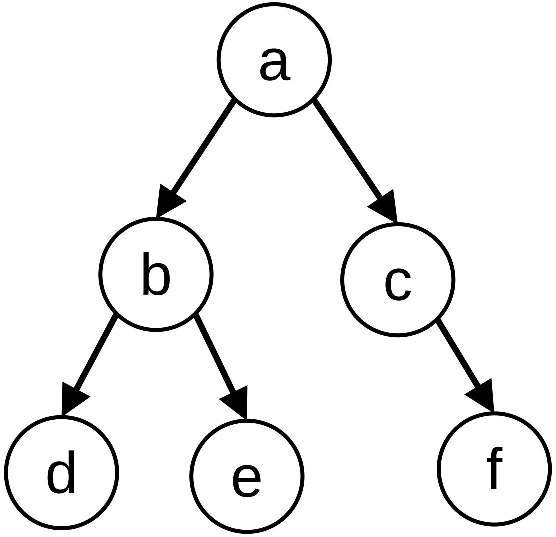
Graphs and Trees

- A powerful abstraction in computing.



Directed **Graph**

Nodes: A B C D
Edges: (A, B) (B, C) (A, C) (C, A) (A, D)



Directed Rooted **Tree**

(*connected acyclic directed* graph)

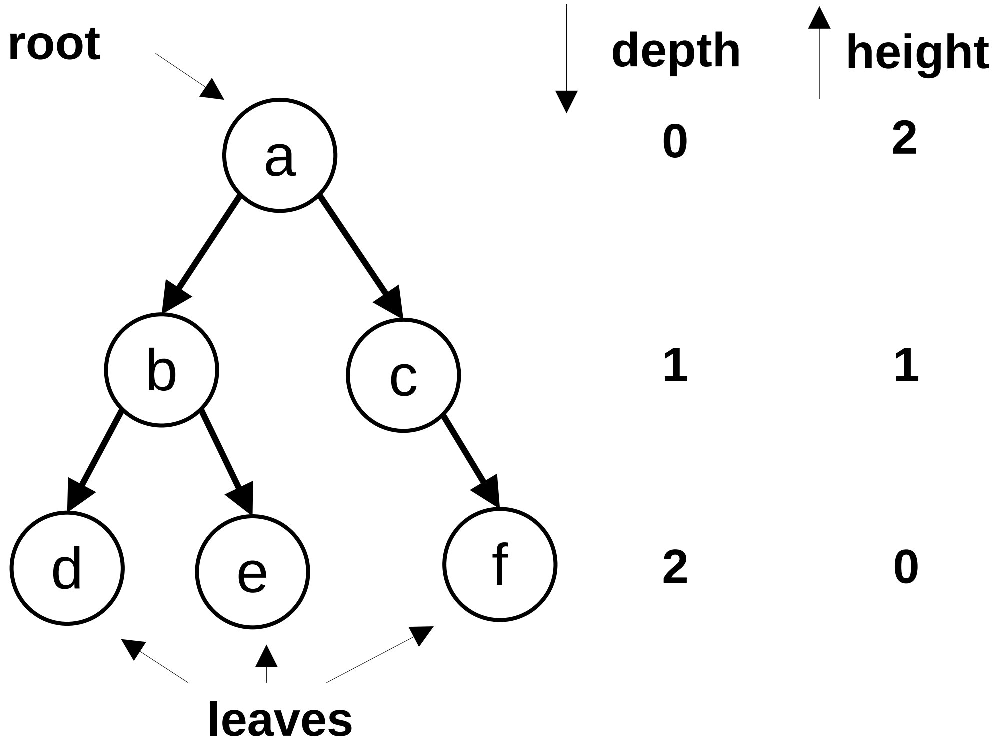
With ordering of children: *Ordered* Tree

Tree Features

b is the **parent** of d and e

d is a **child** of b

b has a **branching factor** (outdegree) of 2 (the number of children)



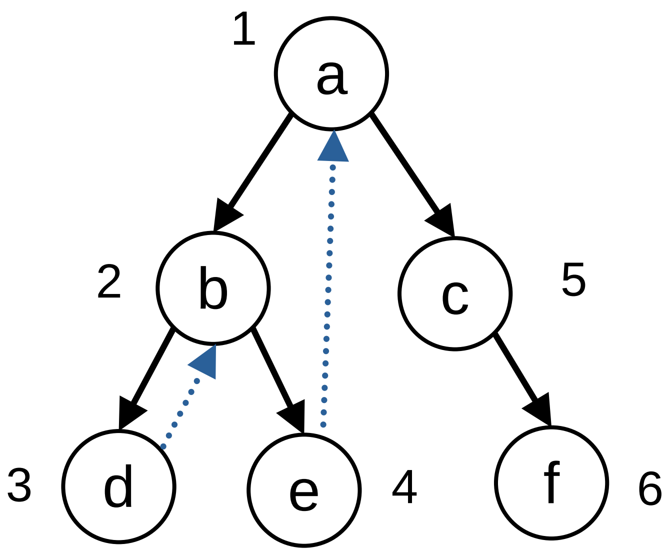
Traversal

- Visiting the elements in a data structure:
  - searching
  - modifying
  - reachability
  - path finding
- Lists / arrays are a form of “linear data structure” that has a natural sequence for traversal.
- Trees and Graphs can be traversed in many ways.

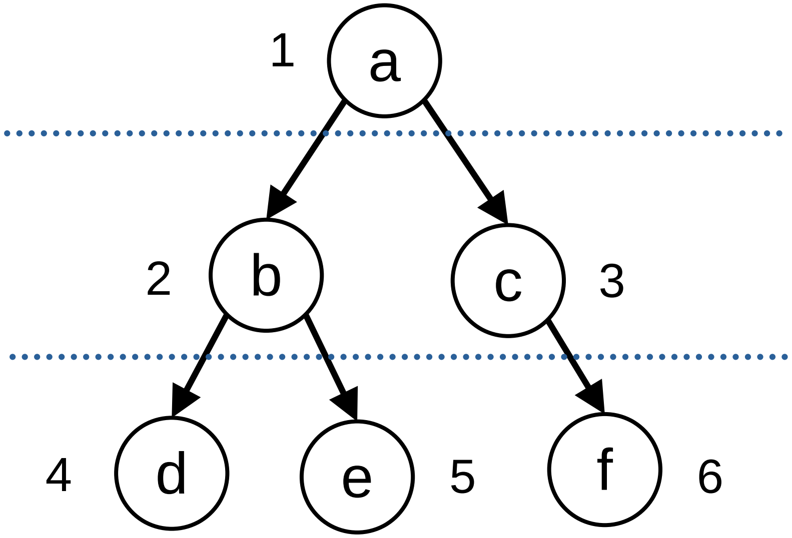
Tree Traversal

- Special case of graph traversal.
- Two common forms:
  - **Depth-First Search (DFS)**
    - Explore as deep as possible along a branch until a leaf is reached.
    - *Backtrack* to another branch (e.g., *sibling* of leaf, or sibling of parent, or …).
  - **Breadth-First Search (BFS)**
    - Starting at root, visit all nodes at given depth before going deeper.

DFS and BFS



Pre-order DFS traversal
a b d e c f



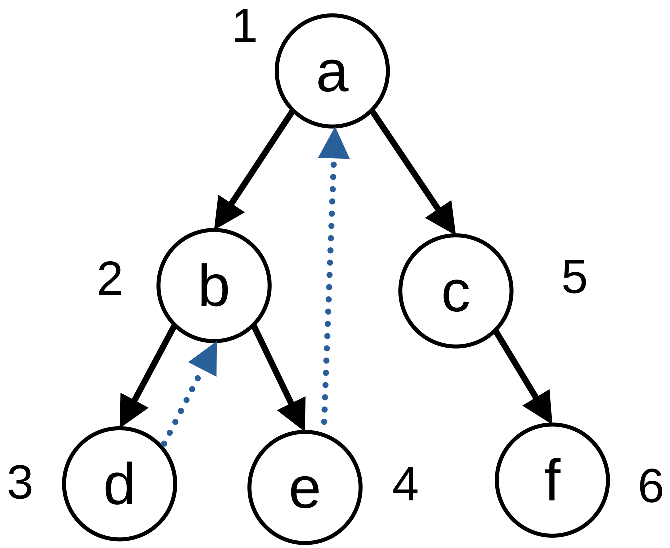
BFS traversal
a b c d e f

Implementing Tree Traversal

- **Depth-First Search (DFS)**
  - Iteratively using a **Stack**: Last-In First-Out (LIFO) data structure
  - Recursively by implicitly using the *call stack*
  - Variations on ordering: post-order, pre-order, in-order
- **Breadth-First Search (BFS)**
  - Iteratively using a **Queue**: First-In First-Out (FIFO) data structure
  - *Corecursively** by passing all sub-trees of same level
  - Only one ordering

* Building (generating) data from a simple “base case”, rather than breaking down (reducing) data until base case reached.

Implementation DFS: Stack



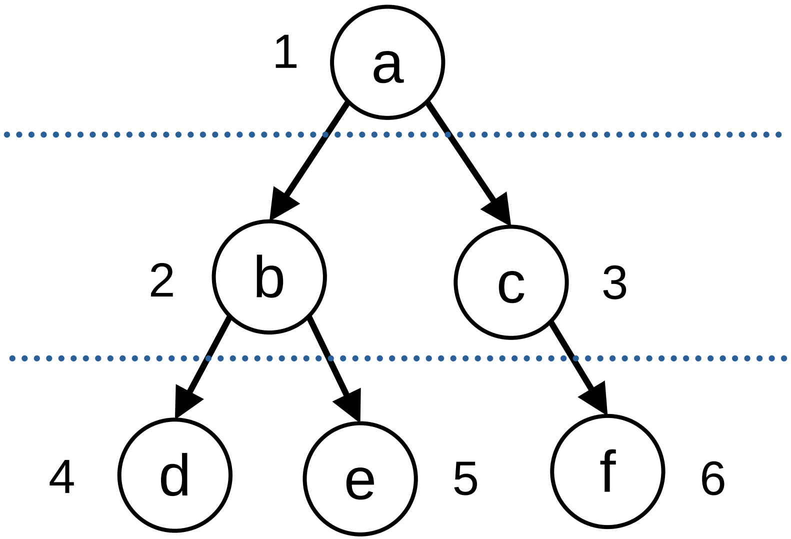
Pre-order DFS traversal
a b d e c f

Stack [ ]: ***push*** onto end, ***pop*** off end

DFS: pop node, push it's children, repeat.

```
0 push a:   [a]
1 pop:      []       a
  push c:   [c]
  push b:   [c b]
2 pop:      [c]      b
  push e:   [c e]
  push d:   [c e d]
3 pop:      [c e]    d
4 pop:      [c]      e
5 pop:      []       c
  push f:   [f]
6 pop:      []       f
```

Implementation BFS: Queue



BFS traversal
a b c d e f

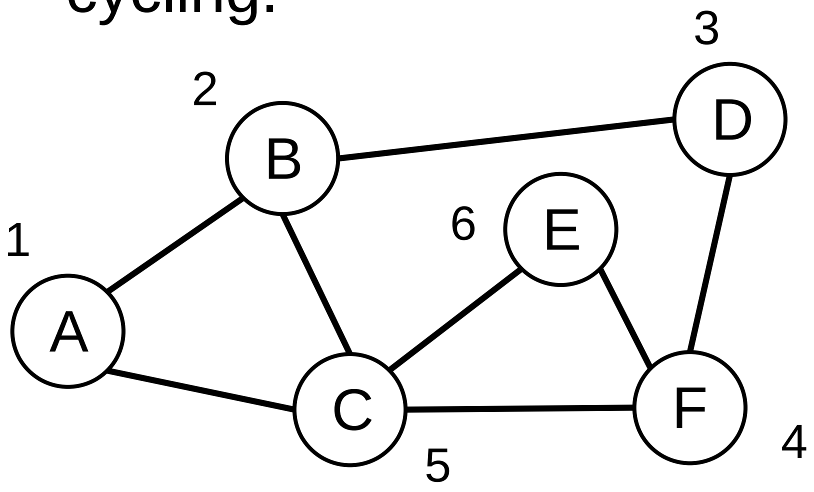
Queue { }: *enqueue* onto back, *dequeue* off front

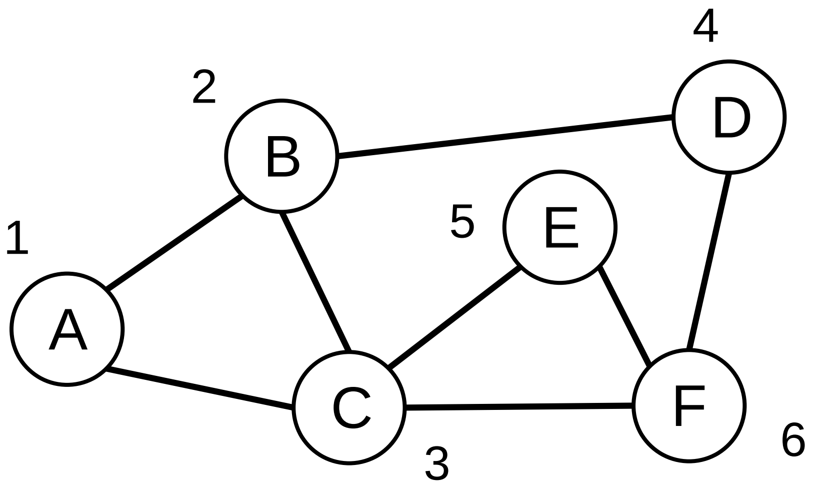
BFS: dequeue node, enqueue it's children, repeat.

```
0 enq a:    {a}
1 deq:      {}       a
  enq b:    {b}
  enq c:    {b c}
2 deq:      {c}      b
  enq d:    {c d}
  enq e:    {c d e}
3 deq:      {d e}    c
  enq f:    {d e f}
4 deq:      {e f}    d
5 deq:      {f}      e
6 deq:      {}       f
```

Graph Traversal

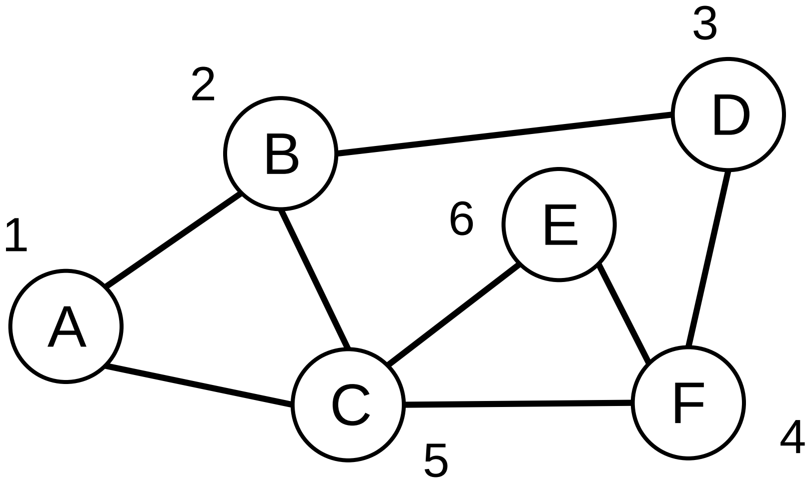
- DFS and BFS generalise from tree traversal.
- Starting node selected based on problem.
- Additionally need to **keep track of “visited”** nodes to avoid cycling.

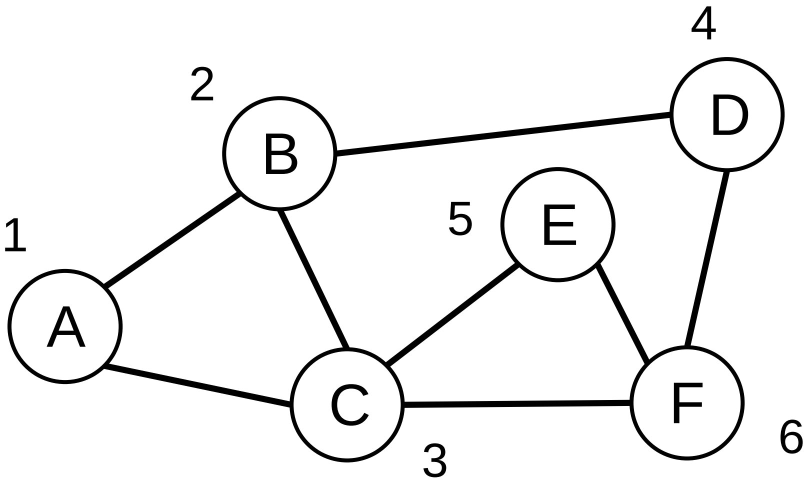




Example: Distance Between Nodes

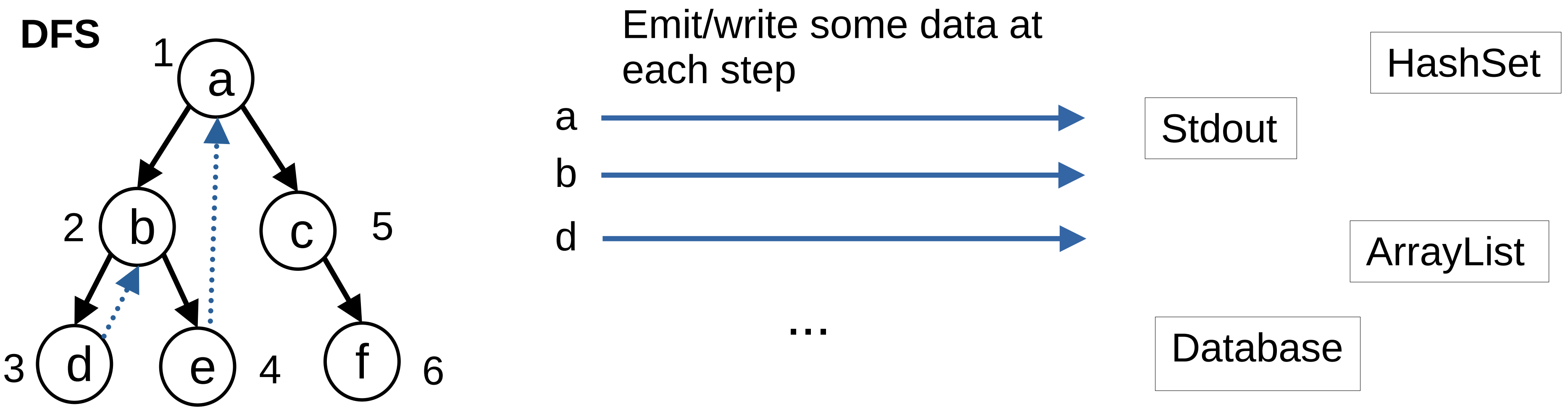
- The *distance* between A and E is the number of edges on a *shortest path* between the two nodes.
- **BFS** can naturally track the distance.
- **DFS** might visit E via a non-shortest *path* – need to revisit nodes





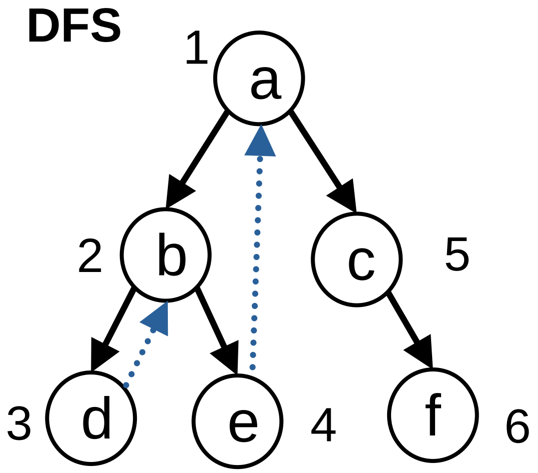
Styles of Using DFS

- Using DFS as skeleton for our code, i.e. we only really care about the traversal pattern.



Styles of Using DFS

- Height/longest path calculation using a single counter

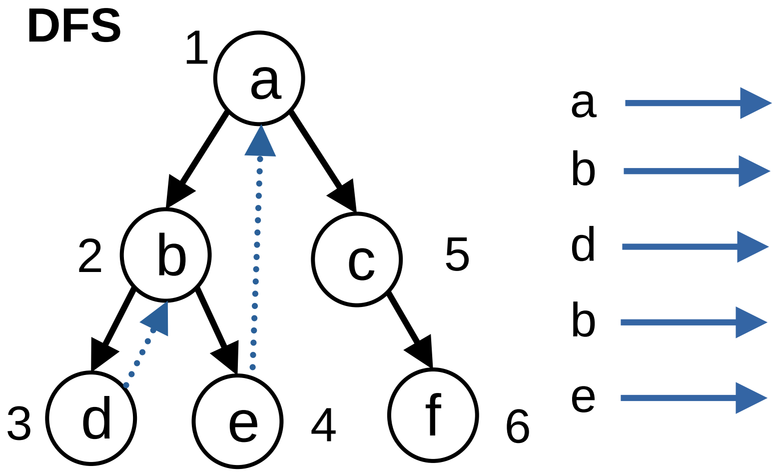


	Action	Current	Best
		0	0
a →	+1	1	0
b →	+1	2	0
d →	+1, update best	3	3
d →	-1 (undo)	2	3
b →	jump to next child	2	3
e →	+1, update best	3	3
e →	-1 (undo)	2	3

Problem: Not all data structures have a clear notion of “undo”, e.g. set

Styles of Using DFS

- Height/longest path calculation using record of history

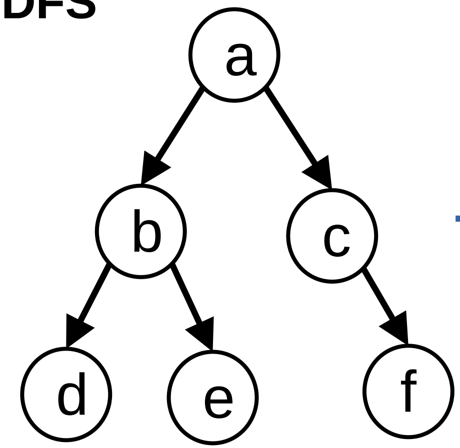


Current	Action	Lengths/history
1		[ ]
2		[ ]
3	Record cur	[3]
2		[3]
3	Record cur	[3, 3]

Styles of Using DFS

- Using DFS to produce well structured data to pass to next stage in a self contained way.

DFS



Array of paths

```
[
  [ a, b, d ],
  [ a, b, e ],
  [ a, c, f ],
]
```

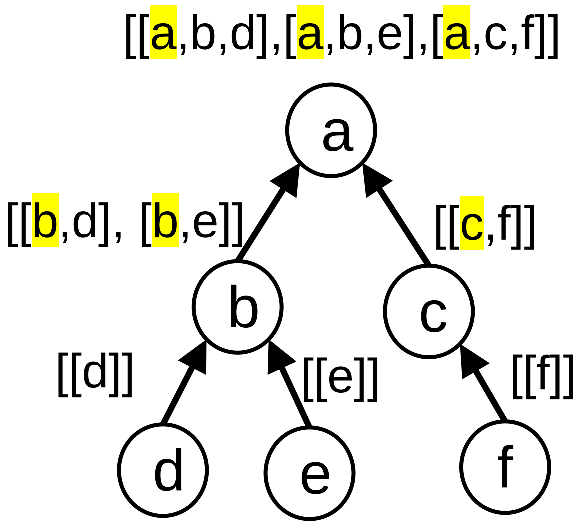
Filter

Statistics

...

Building the data bottom up

- Using DFS to produce well structured data to pass to next stage in a self contained way.

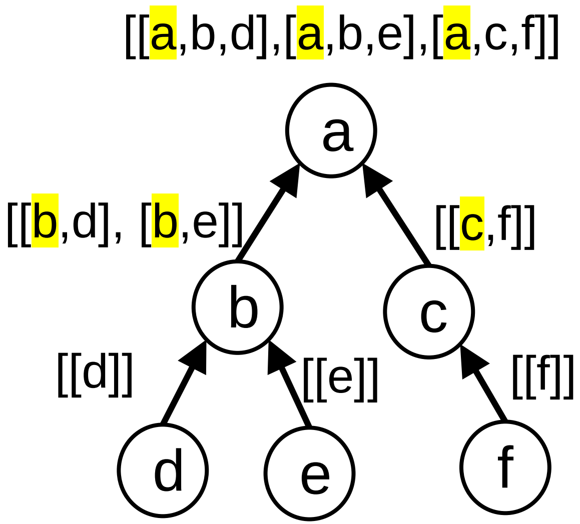


"Concatenation" here is in some sense "combine and flatten"

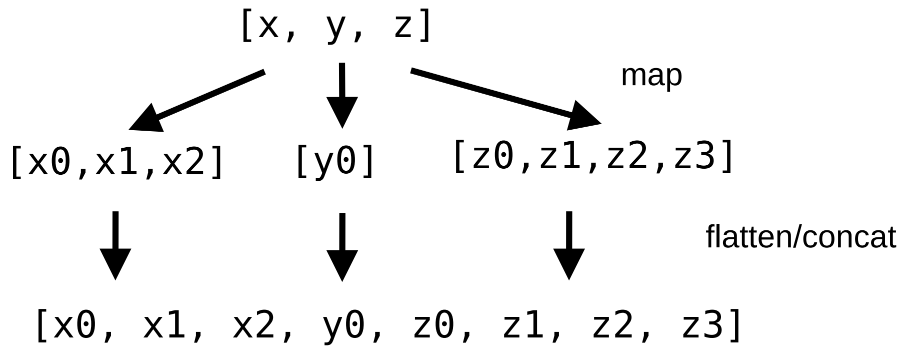
```
[[x0,x1,…]] + [[y0,y1…]]

- combine →
  (combining directly adds one layer of container,
  i.e. we have container of containers of containers)

  [[[x0,x1,…], [y0,y1,…]]]

- flatten →
  (flattening removes that extraneous layer,
  so we get "container of containers" back)

  [[x0,x1,…],[y0,y1,...]]
```

Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.

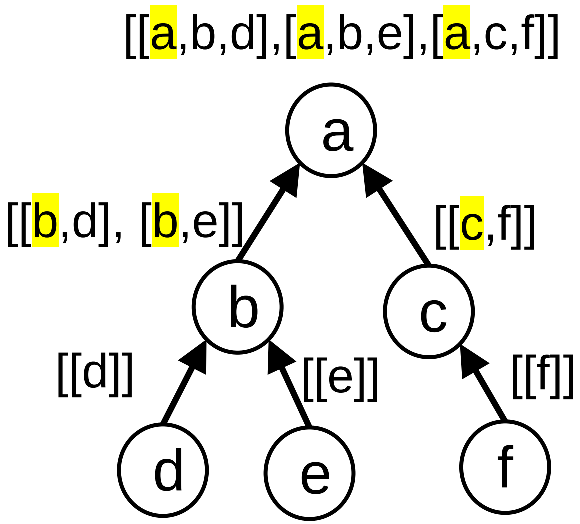


“Flat map” (or “concat map”) is then an extension of that idea

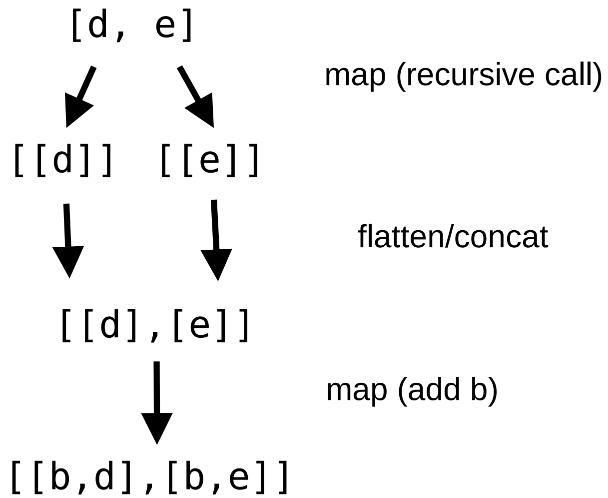


Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.

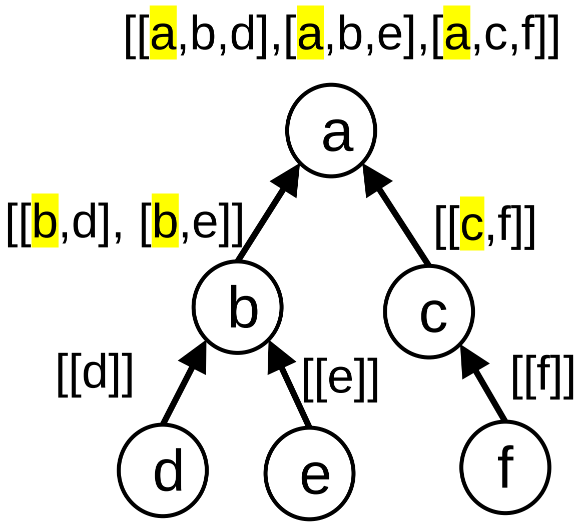


Looking at the bottom left subtree with b as root

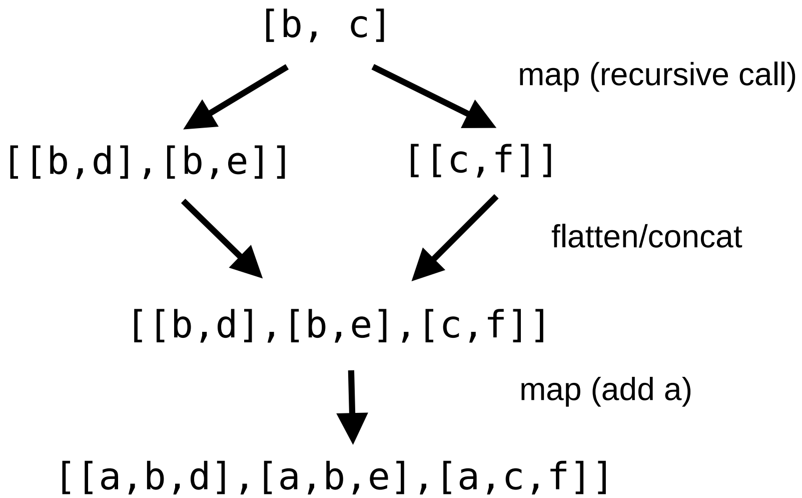


Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.

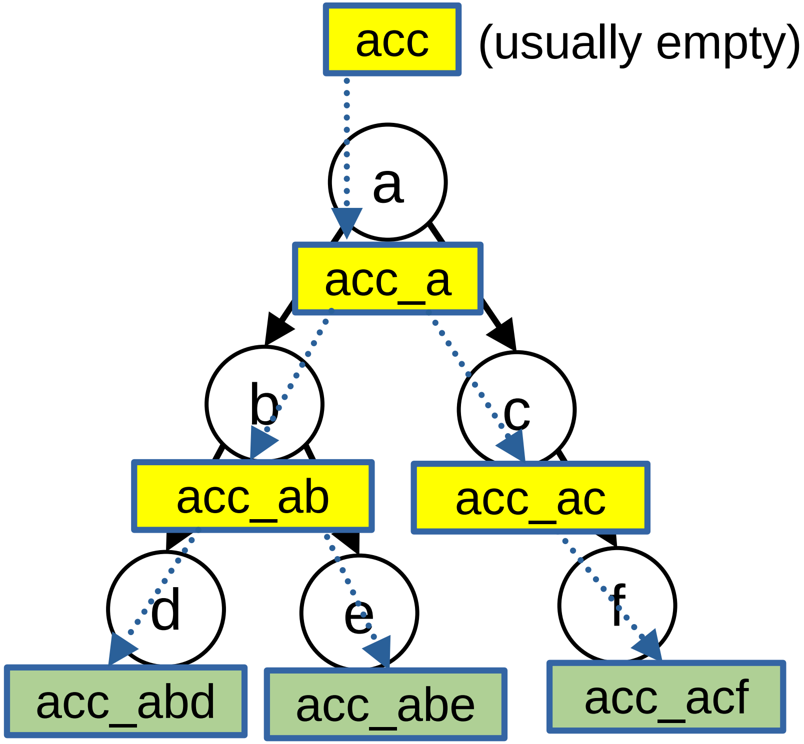


Looking at the entire tree with a as root



Styles of Using DFS

- A more general pattern is an accumulator pattern.



Accumulated value may be:
- Nodes visited
- Path from root so far
- All of above

You can mix accumulator and previous "building bottom up" style by just passing accumulator as argument during recursion