Irregular Data Structures

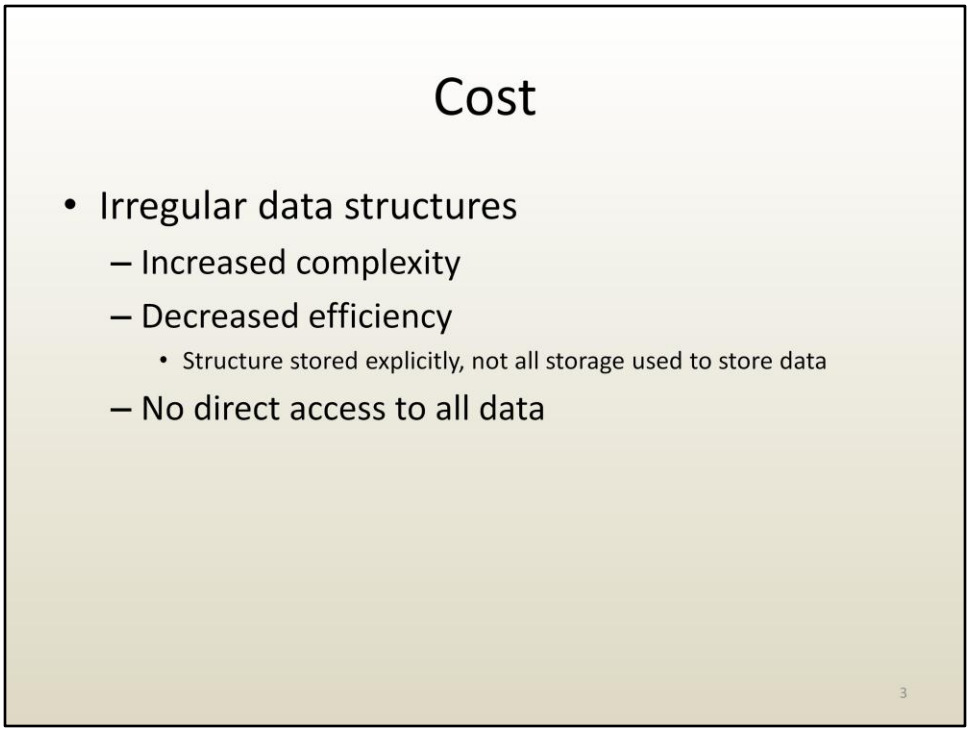
Linked lists, trees, and graphs

Motivation

- Irregular data structures needed to overcome disadvantages of arrays
  - Easy expansion and contraction to keep up with dynamic data size
  - Modeling of irregular data, with complex "neighboring" relationship

2

As mentioned at the end of the "regular data structures" lecture, arrays are not always suitable for a data processing application. As we saw, changes that imply changes to the array size are costly. Moreover, not all data that has to be processed is uniform in size and structure. For example, 1-D arrays allow an element to have two immediate neighbors, elements in 2-D arrays have 8 immediate neighbors and so on, which cannot accommodate scenarios where data elements have a varying number of neighbors.



To overcome these disadvantages, some data processing applications use irregular data structures. Of course, abandoning the regular structure of arrays comes at a cost:

- irregular data structures are more complex to design, implement, use, maintain,
- they are less efficient since some of the memory bytes go towards defining and storing structure, whereas in the case of arrays all bytes go towards the payload,
- and it is also the case that one does not have direct access to all data, like in the case of arrays.

Linked list

- A 1-D sequence data structure
- Not an array
- Each data element is linked to the next
- Link: memory address pointing to a data element
- Link list node: data element + link

- Example
  - credit card transaction amounts in dollars, sorted
  - Links stored explicitly
    - E.g. 32 bit / link
  - Actual address irrelevant here
    - Link shown with arrow
  - Link to first element has to be known (shown in red)
  - Link of last element is null

4

The linked list is a data structure that overcomes the costly size changing operations on 1-D arrays. Like 1-D arrays, a linked list is a 1-D sequence of elements. It is not an array, since the elements are not stored in a contiguous piece of memory, and therefore one cannot access the elements through indexing.

There are explicit links between elements. A link is a memory address. A link says: “the data of interest is at address X”. The actual address X is chosen by the operating system during run time, and it is not important when designing the data structure or when writing the program. What is important is the semantic, the meaning: a pointer to a piece of memory where data “worth pointing to” is stored. In the case of the linked list, the link of one element points to where the next element is stored.

The elements of the list are called “nodes”, since a node stores the data element and the link. Whereas in the case of the array indices were implicit, and were not stored, in the case of linked lists the links are explicit, they are actually stored in the data structure.

In the example, the element might be stored as a floating point number on 32 bits, and the link, which is a memory address, could be stored using 32 bits as well. Consequently, there is a 100% overhead: the amount of memory used to store the link is the same as the amount of memory used to store the number.

Two more important points. Since there is no element preceding the first element, a link to the first element needs to be stored explicitly (red arrow). If the program loses track of this link to the first element, none of the nodes can be accessed, and the program loses access to the entire data. Link mismanagement is another frequent and serious software problem.

The second point is that the last element has no succeeding element, thus its link is invalid. Invalid links are set to 0, or NULL.

Linked list

- Add a new transaction in the amount of $8.12
  - Start at first node (using known red arrow link)
  - Is amount (13.40) smaller than $8.12?
  - No, use node link to go to next node
  - Is amount (12.50) smaller than $8.12?
  - No, use node link to go to next node
  - Is amount (7.45) smaller than $8.12?

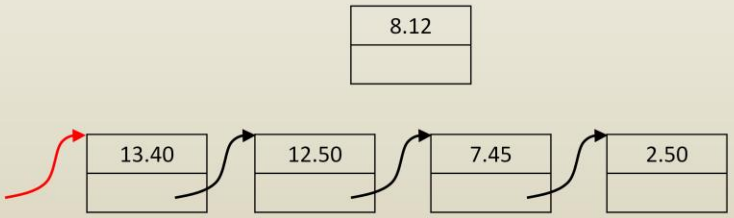
5

Here is how to add a new transaction to the linked list, while keeping the list sorted.

Starting from the first element, the insertion point is found as the “before the node whose transaction amount is less than the amount to be inserted”.

Linked list

- Add a new transaction in the amount of $8.12
  - Yes, insert new node
    - Make new node
    - Set new node amount to $8.12

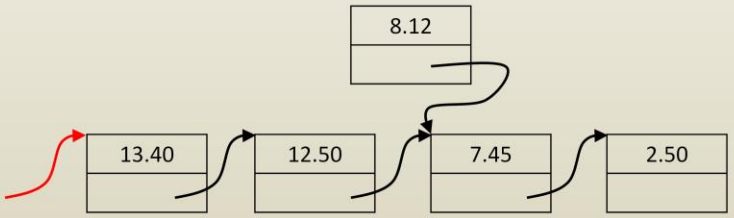


6

Once the insertion point is found, the actual insertion is done with the following steps:
- Make a new node and set the amount to the amount to be inserted

Linked list

- Add a new transaction in the amount of $8.12
  - Yes, insert new node
    - Make new node
    - Set new node amount to $8.12
    - Set new node link to next node

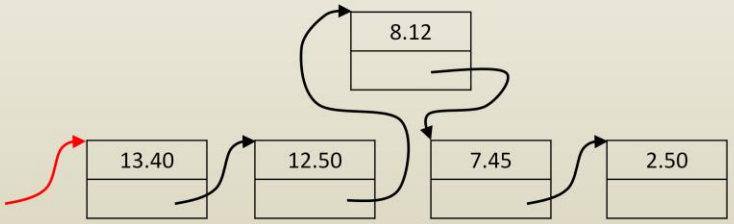


7

- The new node link should point to the node following the insertion point.

Linked list

- Add a new transaction in the amount of $8.12
  - Yes, insert new node
    - Make new node
    - Set new node amount to $8.12
    - Set new node link to next node
    - Set previous node link to new node

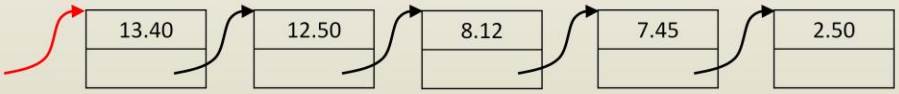


8

- And finally the link of the node preceding the insertion point should be set to point to the new node.

Linked list

- Add a new transaction in the amount of $8.12

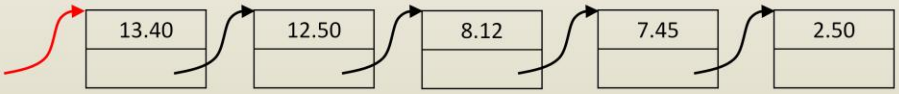


9

Here is the resulting linked list. Note that we did not reallocate the entire linked list, nor did we copy the “old” data to a “new” location.

Linked list

- Delete transaction $12.50
  - Move to node storing $12.50 transaction

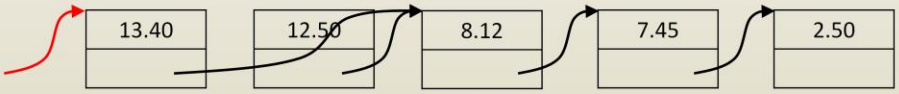


10

Deletion is similar.

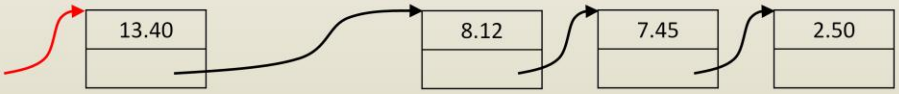
Linked list

- Delete transaction $12.50
  - Move to node storing $12.50 transaction
  - Set link of previous node to next node



Linked list

- Delete transaction $12.50
  - Move to node storing $12.50 transaction
  - Set link of previous node to next node
  - Delete current node

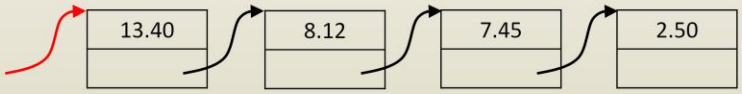


12

It is important to delete the current node, which releases the memory to the pool of available memory such that it can be reused. As you remember from the rationale for deleting the old array, failing to release memory that is not needed anymore creates a memory leak, the program will consume substantially more memory than what is needed, eventually exhausting the amount of available memory, and crashing.

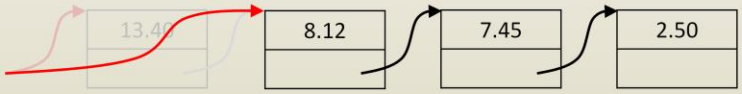
Linked list

- Delete transaction $12.50



Linked list

- Delete transaction $13.40
  - Special case
  - Set red link equal to link of first node
  - Delete first node

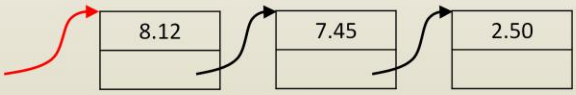


14

If we want to delete the first node, we have to remember to update our link to the first node in the linked list.

Linked list

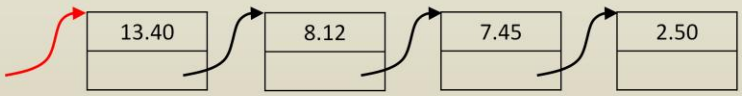
- Delete transaction $13.40
  - Special case
  - Set red link equal to link of first node
  - Delete first node



15

Linked list

- Advantages
  - List grows and shrinks as needed, w/o having to modify entire list
  - Insertion & deletion imply local changes
- Disadvantages
  - You cannot find third transaction directly
    - Have to traverse list
  - Storing link implies overhead



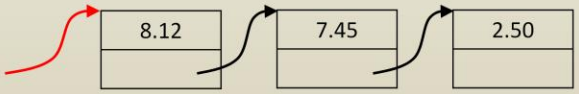
16

Again, here is a summary of the advantages and disadvantages of the linked list, compared to a 1-D array.

iClicker question

When inserting a transaction with value 1.00 in the linked list below, which of the following statements is true:

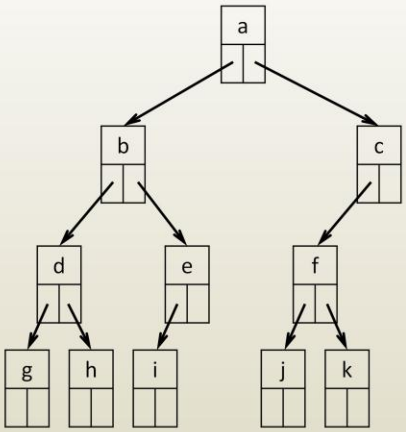
A. The transaction cannot be inserted since there is no transaction of smaller value.
B. The next node link of the new node will be NULL.
C. The insertion point is found when the next node link of the current node is found to be NULL.
D. A, B, and C are true.
E. B and C are true.



17

Binary tree

- Definition
  - A hierarchical data structure
  - A (parent) node links to 0, 1, or 2 (children) nodes
  - The starting node is called root; the root is not the child of any node
  - Nodes with 0 children are called leafs
  - Non-leaf nodes are called internal

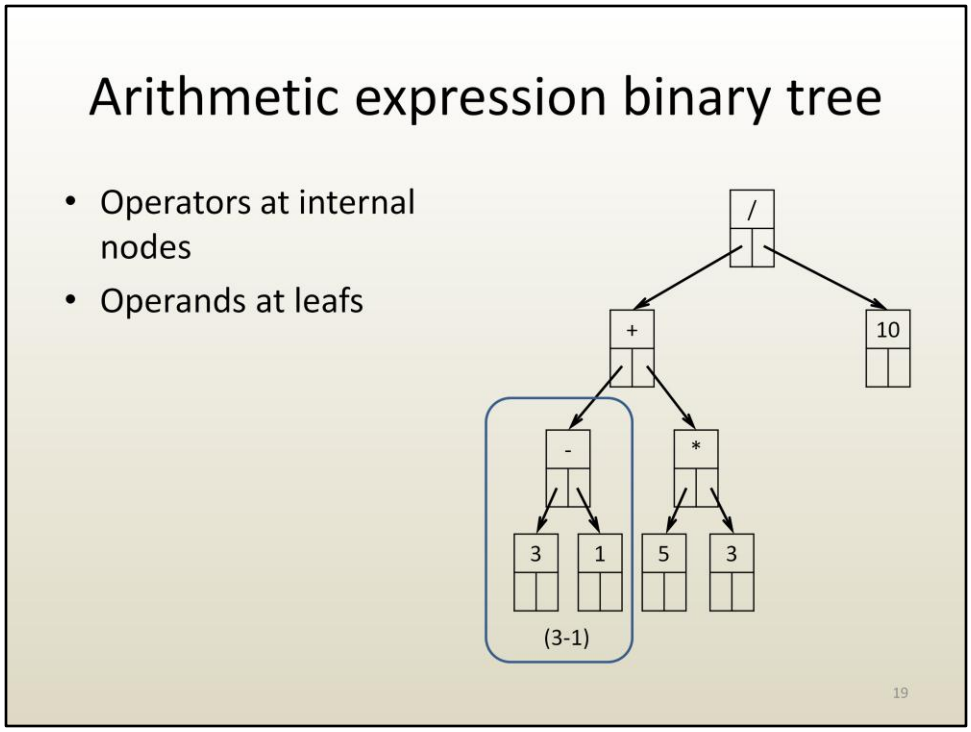


18

Here is a second example of irregular data structure, a binary tree.

Notice that a node can link to more than one node. This tree is called binary because a parent node can have at most 2 children nodes.

a is the root, a is the parent of children nodes b and c, b is the left child of a, c is the right child of a, a-f are internal nodes, g-k are leaf nodes.

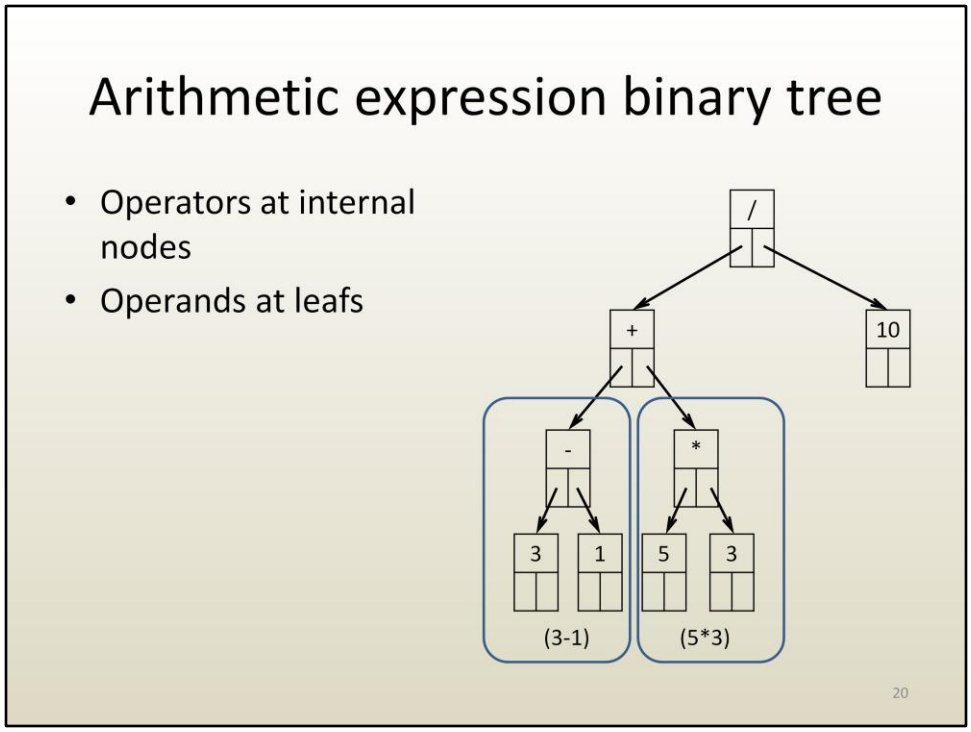


And here is a classic application of binary trees—parsing and evaluating arithmetic expressions.

Internal nodes store operators. The left and right subtrees (i.e. the trees with the left and right children as roots) are the expressions on which the operator is applied.

Leaf nodes store operands (e.g. constants, variables).

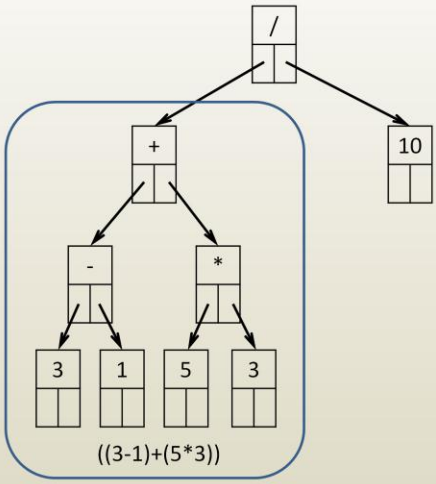
The subtree highlighted corresponds to the subtraction 3-1.



This other subtree corresponds to a multiplication: 5 times 3.

Arithmetic expression binary tree

- Operators at internal nodes
- Operands at leafs

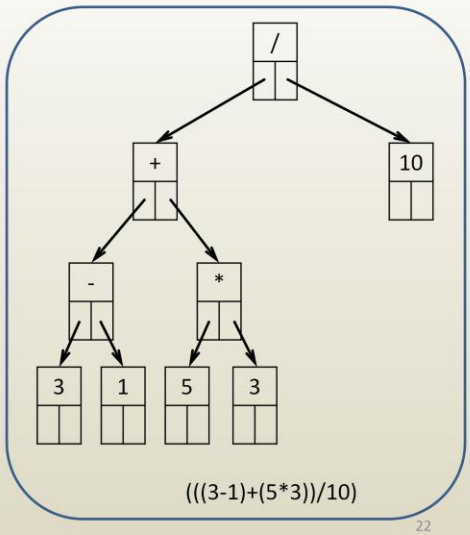


21

The + operator is applied to the two subtrees, resulting in the expression ((3-1)*(5*3)). Notice that here, for simplicity, we ignore operator precedence and use parentheses “all the time”.

Arithmetic expression binary tree

- Operators at internal nodes
- Operands at leafs

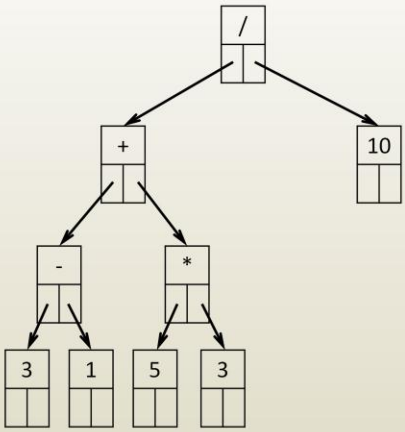


22

The entire binary tree is equivalent to an arithmetic expression.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



23

Given an arithmetic expression binary tree, one can print out the expression to which the tree corresponds by traversing the entire tree.

Traversing or the traversal of a data structure is a systematic approach to visiting all its data elements. In the case of arrays the traversal is straightforward—for example you start at element with index 0, then you increment the index by 1 until you reach the last element. For irregular data structures, such as binary trees, traversals are a little more complicated.

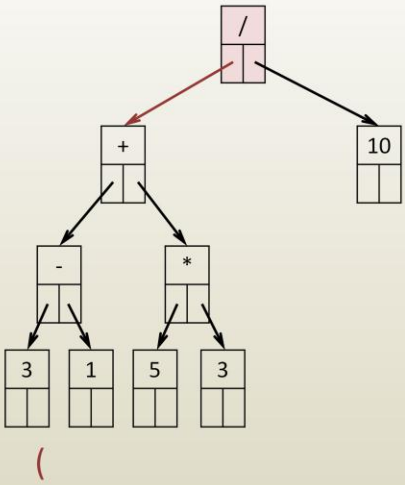
Here are some simple rules for a successful traversal. Successful means visiting all nodes once.

In order to print out the arithmetic expression, the traversal rules are enhanced with the following printout rules.

Let's see these rules at work.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right

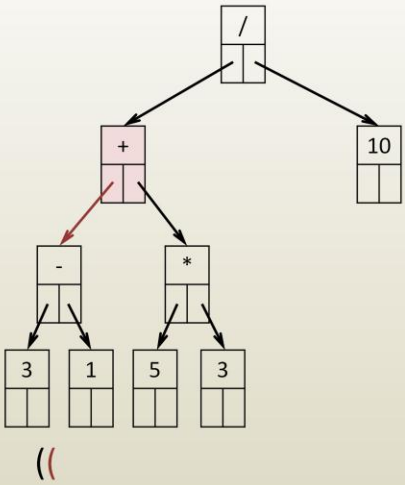


24

We start at the root. We write (before going left.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right

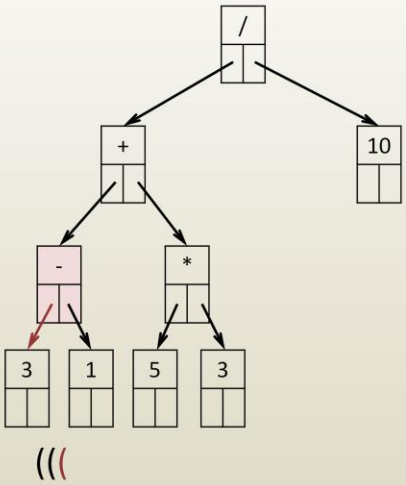


25

Then we go left. We are at +. We write (again before going left.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right

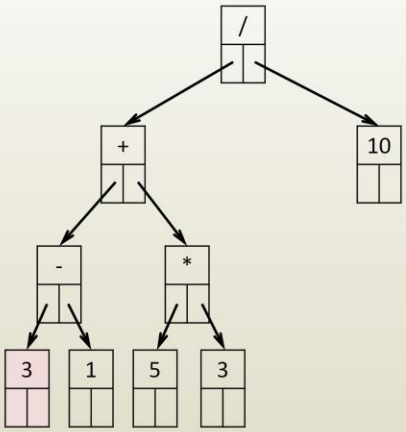


26

One more time.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



(((3

27

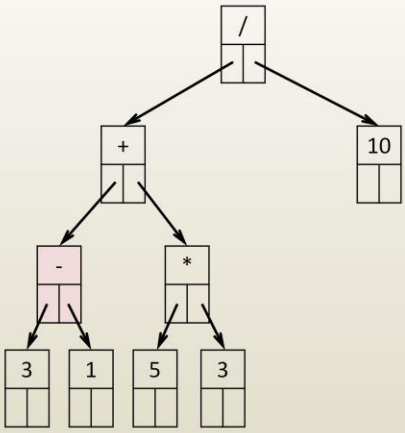
Now we are at the leaf node storing the operand 3. There is no way we can go left, since the current node doesn't have a left child. A leaf node will have both children links invalid, i.e. set to NULL.

So we will try to go right, but before that, we write the current node symbol, according to the second printout rule.

We cannot go right either, since there is no right child. Then we have to go back up.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



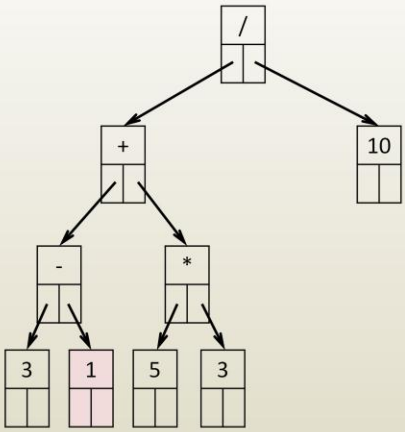
(((3-

28

We are back at -. We have already gone left. Now we write the symbol – before trying to go right. We succeed going right.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



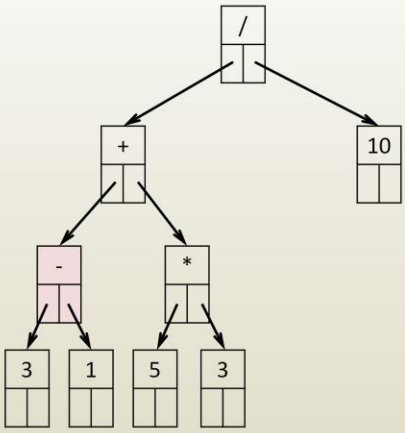
(((3-1

29

Now we are at 1. No children. Like before, we write 1 and go back up.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



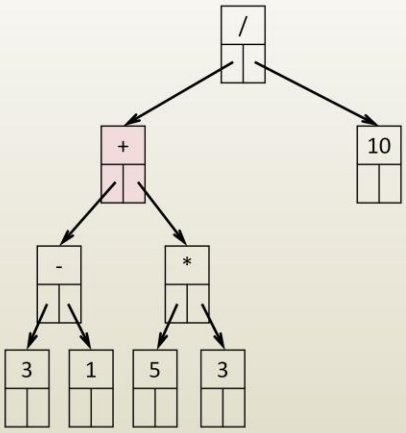
(((3-1)

30

Now were are at – again. We went right, so now we write) and go back up.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



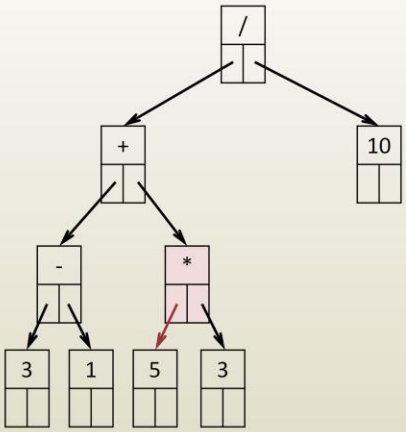
(((3-1)+

31

We are at + now. We write + and go right.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



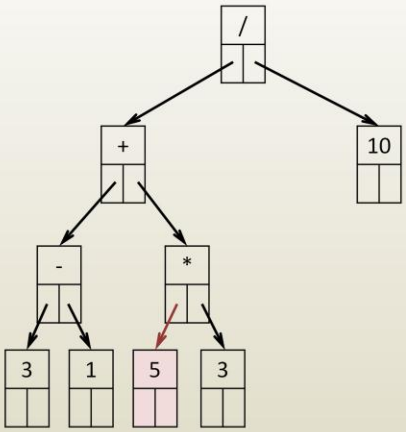
(((3-1)+(

32

We get to *. We can go left so we write (and go left.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



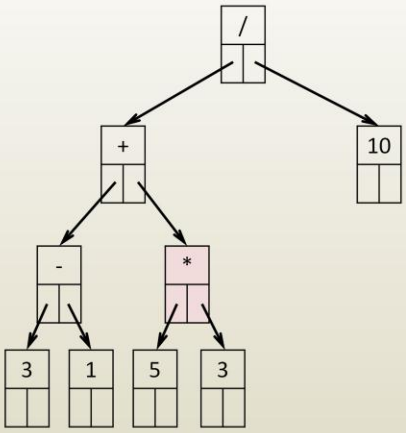
(((3-1)+(5

33

We write the leaf operand 5…

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



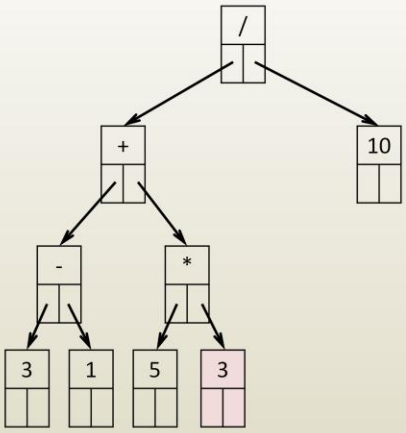
(((3-1)+(5*

34

Then the operand *

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



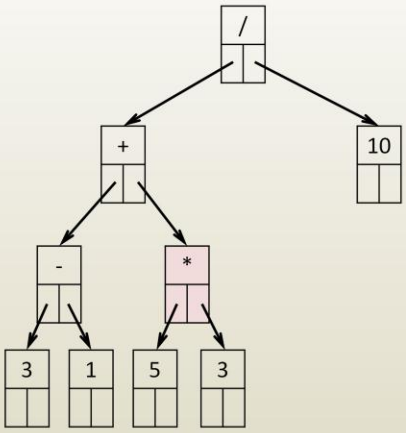
(((3-1)+(5*3

35

And the operand 3.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



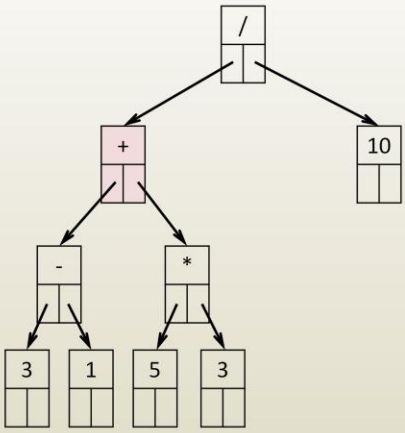
(((3-1)+(5*3)

36

We are back at *, we are done going right, thus we write).

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



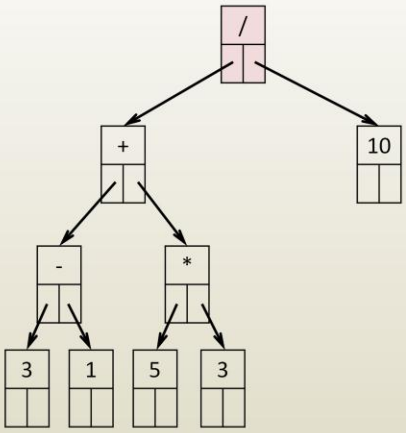
(((3-1)+(5*3))

37

We are at +, done going right, write).

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



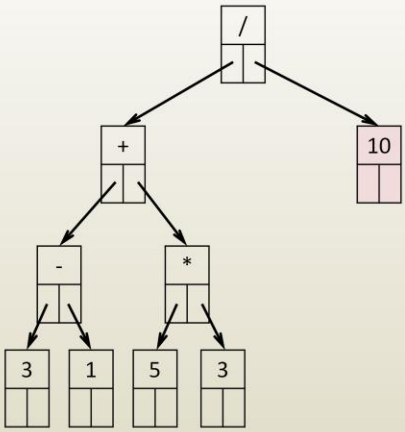
(((3-1)+(5*3))/

38

We are at the root, we write the symbol, and then go right.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



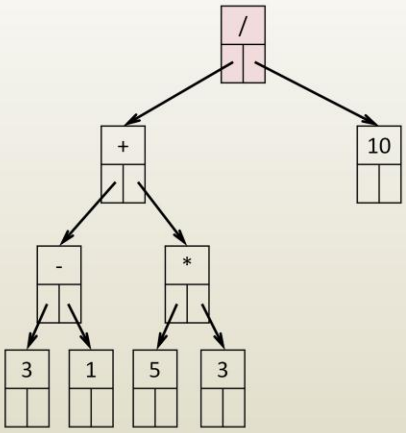
(((3-1)+(5*3))/10

39

We write the operand at the leaf.

Arithmetic expression binary tree

- Arithmetic expression can be recovered by traversing tree
- Traversal: visiting all nodes
- Traversal rules
  - Start at root
  - For every node
    - go left until dead end
    - then go right until dead end
    - then go back up
- Printout rules
  - Write "(" before going left
  - Write current node symbol before going right
  - Write ")" after having gone right



(((3-1)+(5*3))/10)

40

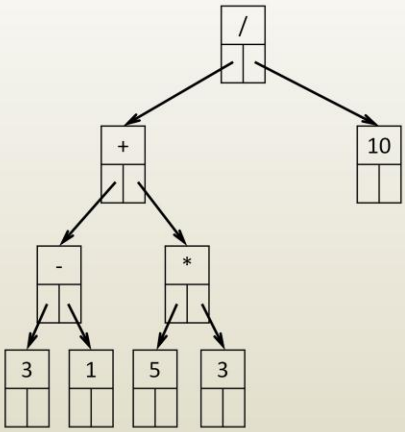
We are back to the root. We are done going right, we write). We are done with the root, we want to go up, there is no more up, we are done.

We printed out the entire expression.

Evaluating the binary tree is even easier.

Arithmetic expression binary tree

- Arithmetic expression can be evaluated by traversing tree
- Evaluation rules
  - if leaf, return operand
  - valLeft = Evaluate left
  - valRight = Evaluate right
  - return valLeft operator valRight



41

We are at root.

Is it a leaf? No, evaluate left i.e. +.

Is it a leaf? No, evaluate left, i.e. -.

Is it a leaf? No, evalue left, i.e. 3.

Is it a leaf? Yes, return 3.

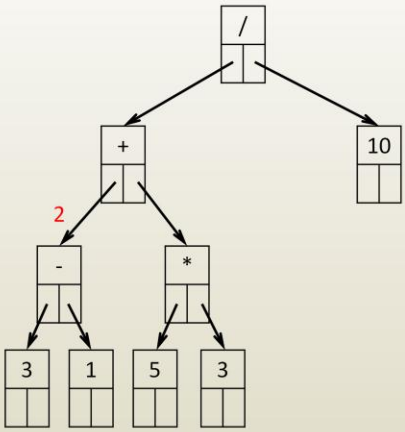
We are back at -. Evaluate right, i.e. 1.

Is it a leaf? Yes, return 1.

We are back at -. Return 3-1=2.

Arithmetic expression binary tree

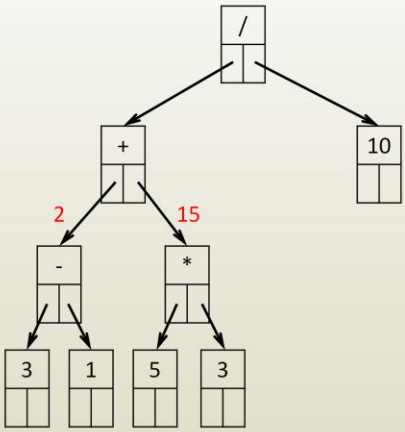
- Arithmetic expression can be evaluated by traversing tree
- Evaluation rules
  - if leaf, return operand
  - valLeft = Evaluate left
  - valRight = Evaluate right
  - return valLeft operator valRight



42

Arithmetic expression binary tree

- Arithmetic expression can be evaluated by traversing tree
- Evaluation rules
  - if leaf, return operand
  - valLeft = Evaluate left
  - valRight = Evaluate right
  - return valLeft operator valRight

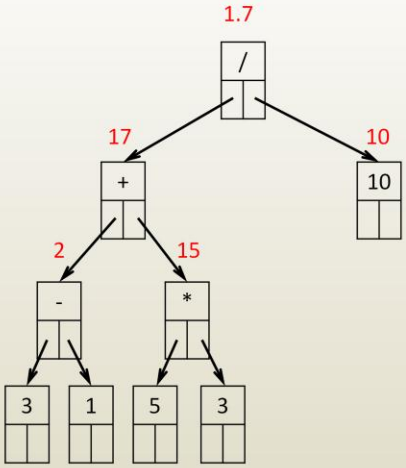


43

The subtree starting at * is evaluated similarly to 15.

Arithmetic expression binary tree

- Arithmetic expression can be evaluated by traversing tree
- Evaluation rules
  - if leaf, return operand
  - valLeft = Evaluate left
  - valRight = Evaluate right
  - return valLeft operator valRight



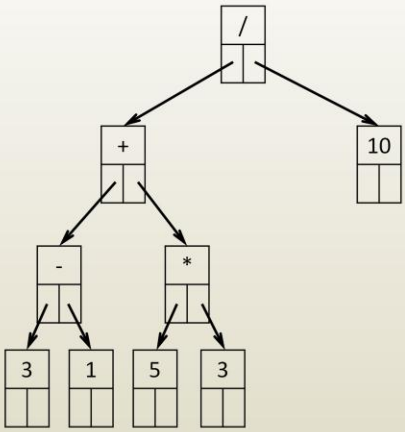
44

And the final expression value is 1.7.

iClicker question

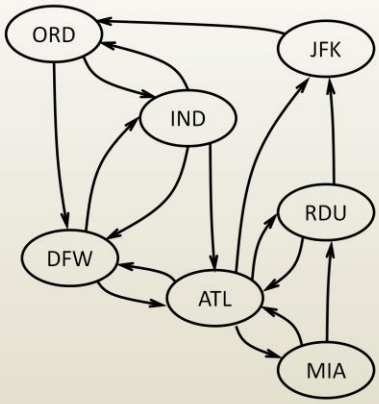
- Which traversal called COUNT counts the number of leafs in a binary tree.

A. If leaf, return 1. If not leaf, return COUNT(left child) + COUNT(right child)
B. If leaf, return 1. If not leaf return 0. COUNT(left child). COUNT(right child).
C. If leaf, return operand. If not leaf, return COUNT(left child) + COUNT(right child)
D. None of the above.
E. All of the above.



Graphs

- Graphs
  - Nodes (also called vertices) connected by links, called edges
  - Nodes have a variable number of incident edges
  - Great flexibility
- Example: airline routes

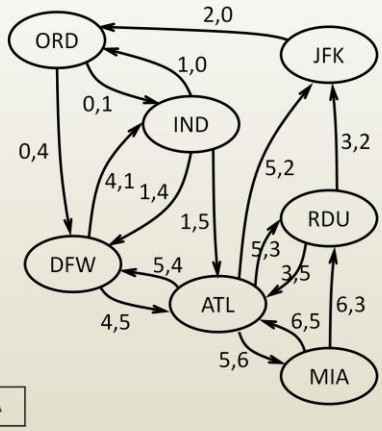


46

Here is a third example of irregular data structures, graphs. Trees are more flexible than linked lists and graphs are more flexible than trees.

Graph data structure encodings

- "List of edges"
  - Array of nodes & array of edges
  - Edges pair of node indices
  - Origin node first
  - Destination node second



0	1	2	3	4	5	6
ORD	IND	JFK	RDU	DFW	ATL	MIA

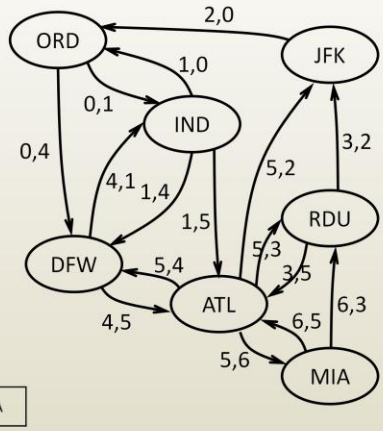
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0,4	2,0	3,2	6,3	5,6	4,5	6,5	5,4	4,1	1,4	1,5	5,3	3,5	5,2	1,0	0,1

47

One way of encoding a graph is by specifying an array of nodes and an array of edges.

Graph data structure encodings

- "List of edges"
  - Used only for sparse graphs (i.e. a small number of edges)
  - Difficult to find whether there is an edge between two nodes (requires traversal of edge list)

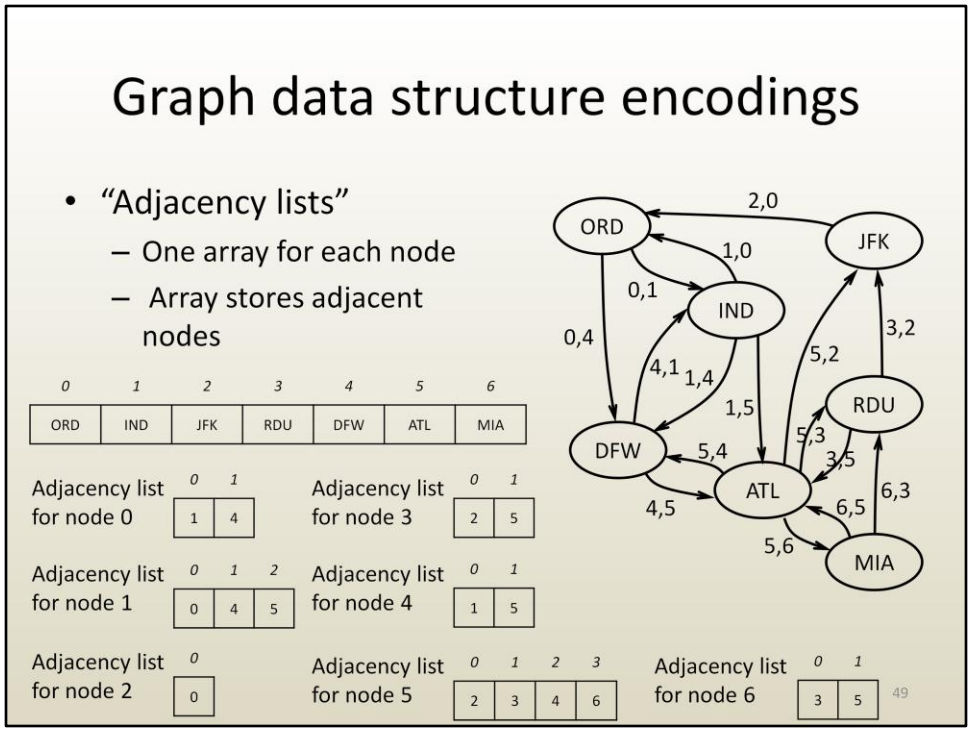


0	1	2	3	4	5	6
ORD	IND	JFK	RDU	DFW	ATL	MIA

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

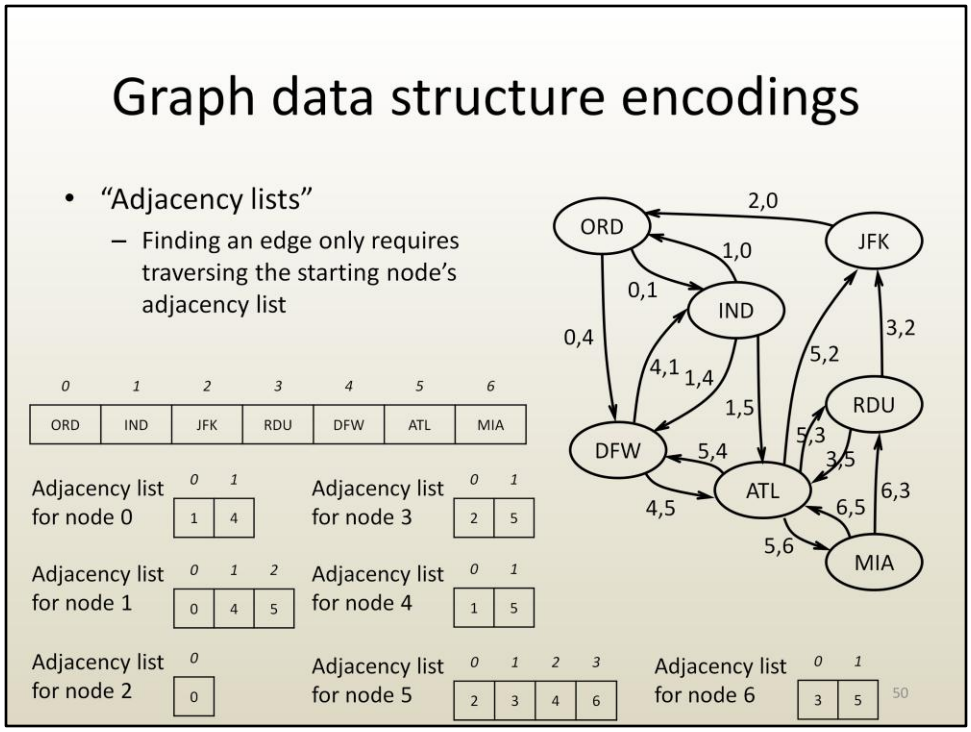
48

The list of edges representation is compact but it is not easy to find whether there is a node between two given nodes, or to enumerate all the edges from a given node.



A more powerful encoding of the graph is by using adjacency lists. Adjacency lists could be encoded as linked lists or as 1-D arrays. Here we chose 1-D arrays.

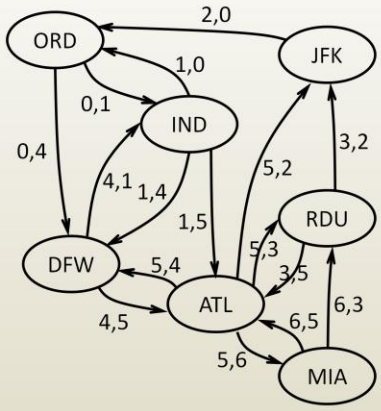
There is one adjacency list for each node. The adjacency list provides the nodes to which there is an edge from the current node.



Finding whether one can fly directly from IND and DFW only requires traversing the adjacency list of IND to search for DFW.

Graph data structure encodings

- "Adjacency matrix"
  - A 2-D matrix
  - Row corresponds to start node
  - Column corresponds to end node
  - 0 if no edge, 1 if edge



	0	1	2	3	4	5	6
	ORD	IND	JFK	RDU	DFW	ATL	MIA

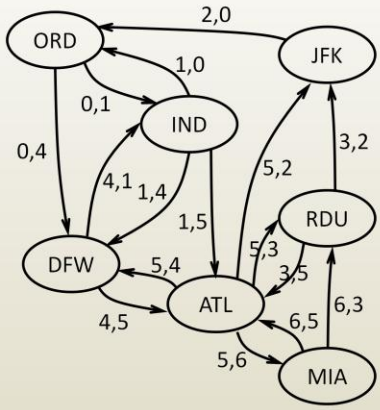
	0	1	2	3	4	5	6
0	0	1	0	0	1	0	0
1	1	0	0	0	1	1	0
2	1	0	0	0	0	0	0
3	0	0	1	0	0	1	0
4	0	1	0	0	0	1	0
5	0	0	1	1	1	0	1
6	0	0	0	1	0	1	0

51

A third possible way of encoding graphs is to use what is called an adjacency matrix. The adjacency matrix is a square 2-D array A, where element A[i][j] is 1 if there is an edge from node i to node j.

Graph data structure encodings

- "Adjacency matrix"
  - An edge is found in constant time
    - Is there an edge between ATL and ORD?
    - A[5][0] is 0 so the answer is no
  - Storage quadratic in number of nodes
    - Inefficient for sparse graphs



	0	1	2	3	4	5	6
	ORD	IND	JFK	RDU	DFW	ATL	MIA

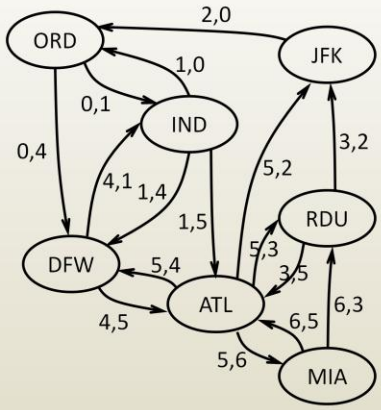
	0	1	2	3	4	5	6
0	0	1	0	0	1	0	0
1	1	0	0	0	1	1	0
2	1	0	0	0	0	0	0
3	0	0	1	0	0	1	0
4	0	1	0	0	0	1	0
5	0	0	1	1	1	0	1
6	0	0	0	1	0	1	0

52

The adjacency matrix allows answering the question "is there an edge from node i to node j" in constant time, by simply checking whether element A[i][j] is 1 or 0.

Directed graphs

- So far we talked about directed graphs
  - an edge started from one node and ended at another
  - the edge could only be traversed in one direction

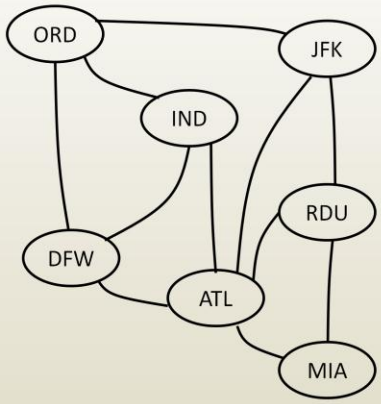


53

The adjacency matrix allows answering the question “is there an edge from node i to node j” in constant time, by simply checking whether element A[i][j] is 1 or 0.

Undirected graphs

- In an undirected graph edges can be traversed in either direction.



54

Here an edge AB means that one can fly directly from airport A to airport B and from airport B to airport A.