Presentation for use with the textbook Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Ch 02 Data Structures

1

What is Bitwise Structure?

- The smallest type in Java (or C++) is of 8 bits (char).
- Sometimes we need only a single bit.
- For instance, storing the status of the lights in 8 rooms:
  - We need to define an array of at least 8 chars. If the light of room 3 is turned on the value of the third char is 1, otherwise 0.
  - Total array of 64 bits.

EXPENSIVE in place and time!!!

2

What is Bitwise Structure?

- It is better to define only 8 bits since a bit can also store the values 0 or 1.
- But the problem is that there is no data type which is 1 bit long (char is the longer with 1 byte).
- Solution: define a char (8 bits) but refer to each bit separately.
- **Bitwise** operators, introduced by many languages, provide one of its more powerful tools for using and manipulating memory. They give the language the real power of a "low-level language".

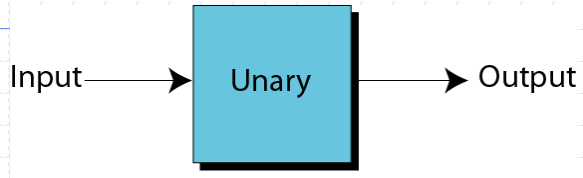
3

What is Bitwise Structure?

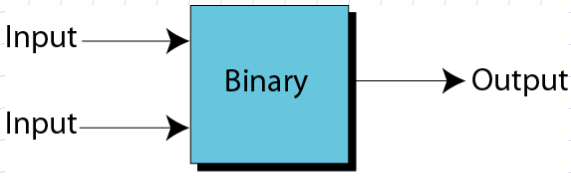
- Accessing bits directly is fast and efficient, especially if you are writing a real-time application.
- A single bit cannot be accessed directly, since it has no address of its own.
- The language introduces the **bitwise** operators, which help in manipulating a single bit of a byte.
- **bitwise** operators may be used on integral types only.
- https://www.programiz.com/java-programming/bitwise-operators

4

Unary and binary operations



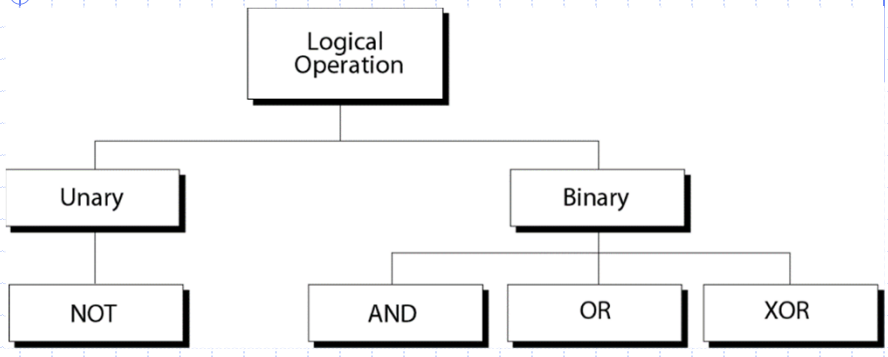
a. Unary operator



b. Binary operator

5

Logical operations



6

Truth tables of logical operations

NOT

x	NOT x
0	1
1	0

AND

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

7

Bitwise Operators in Java

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	1's compliment
<<	Shift left
>>	Shift right

All these operators can be suffixed with =
For instance a &= b; is the same as a = a & b;

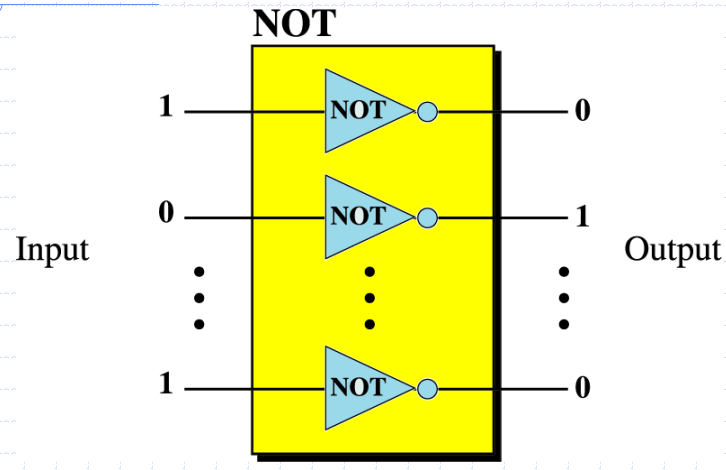
8

Bitwise Operators – truth table

a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

9

NOT operator



10

Use the NOT operator on one byte

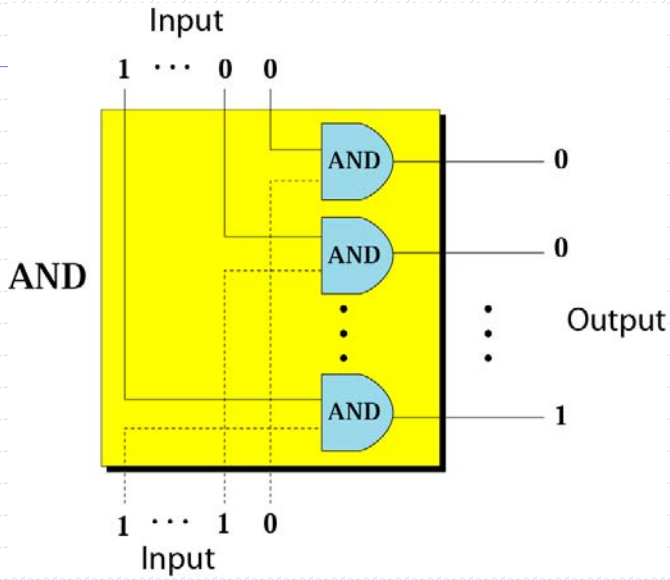
Example

Target $x = 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0$ *NOT*

Result $\sim x = 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1$

11

AND operator



12

Use the AND operator on two bytes

Example

Target $x = 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0$ *AND*
$y = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1$

Result $x\&y = 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0$

13

Figure 4-8

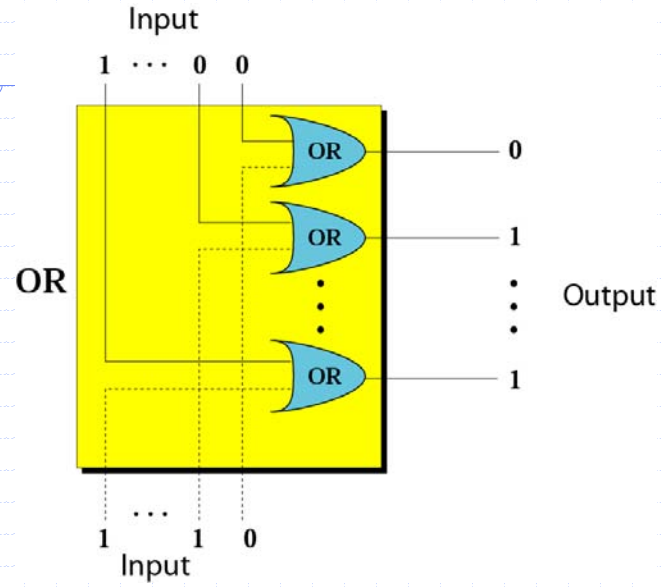
Inherent rule of the AND operator

(0) AND (X) → (0)

(X) AND (0) → (0)

14

OR operator



15

Use the OR operator on two bytes

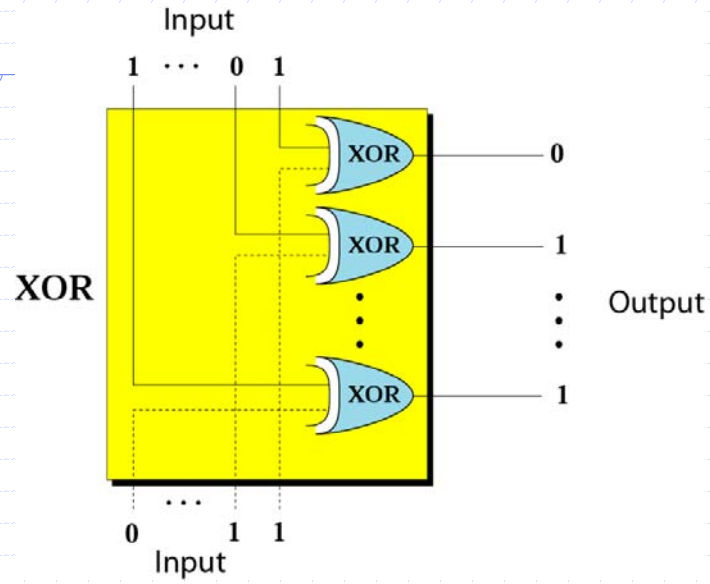
Example

Target x = **1 0 0 1 1 0 0 0** *OR*
y = **0 0 1 1 0 1 0 1**

Result $x|y$ = **1 0 1 1 1 1 0 1**

16

XOR operator



17

Use the XOR operator on two bytes

Example

Target $x = 1\ 0\ 0\ 1\ 1\ 0\ 0\ 0$ *XOR*

$y = 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1$

Result $x \hat{} y = 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1$

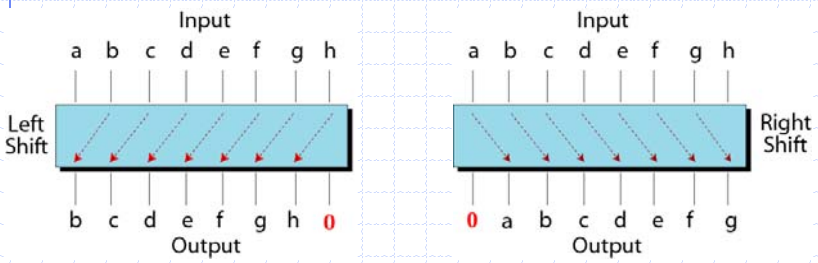
18

Exercise A-1.11

Given an array, A, of n positive integers, each of which appears in A exactly twice, except for one integer, x, describe an $O(n)$-time method for finding x using only a single variable besides A.

19

Shift operations



20

Examples

Divide or multiply a number by a power of 2 using shift operations.

Let x be byte 00111011 which represents 59.
Then ($x >> 1$) is 00011101, which is 29 = 59/2.

($x << 1$) is 01110110, which is 118 = 59*2.

($x >> 2$) is 00001110, which is 14 = 59/4.

($x << 2$) is 11101100, which is 236 = 59*4.

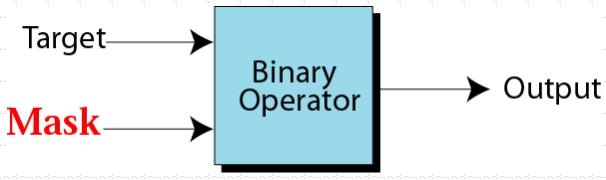
21

Setting Bits

- How can we set a bit on or off?
- Manipulations on bits are enabled by mask and bitwise operators.
- Bitwise OR of anything with 1 results in 1.
- Bitwise AND of anything with 0 results in 0.

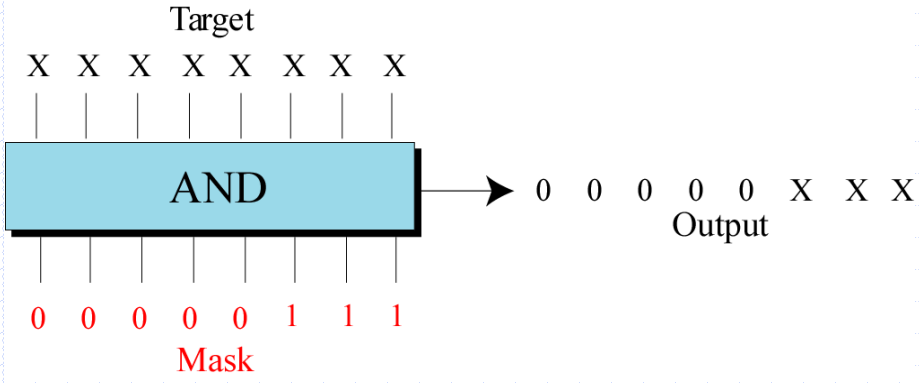
22

Mask: a bit pattern with known values



23

Example of unsetting specific bits



Use a mask to unset (clear) the 5 leftmost bits of a pattern.

24

Test the mask with the byte 10100110.

Example

The mask is 00000111.

Target	**1 0 1 0 0 1 1 0**	***AND***
Mask	**0 0 0 0 0 1 1 1**	

Result	**0 0 0 0 0 1 1 0**	

25

Getting Bits

- How can we know if a bit is on or off?
- Manipulations on bits are enabled by mask and bitwise operators.
- Bitwise AND of anything with 1 results in the same value.

26

Getting Bits

- For instance, how can we check if the light in room #3 is turned on or off?

```
int lights = 0x27;
int mask = 0x1;
mask <<= 2;
if(lights & mask)
  puts("turned on");
else
  puts("turned off");
```

lights: ...00100111

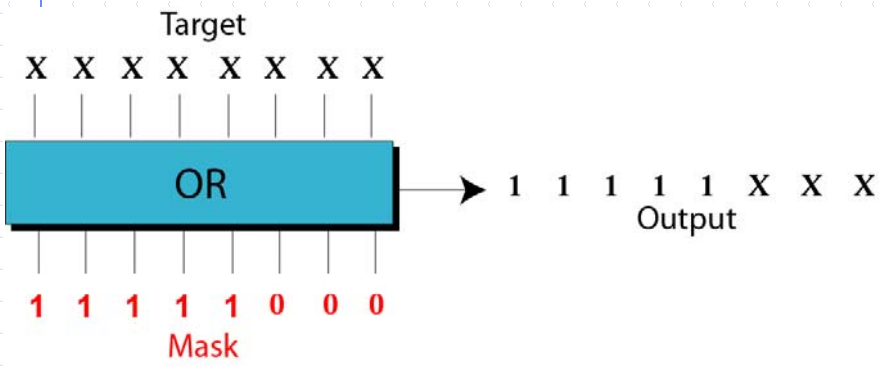
mask: ...00000001

mask: ...00000100

lights & mask: ...00000100

27

Example of setting specific bits



Use a mask to set the 5 leftmost bits to be 1.

28

Test the mask with the pattern 10100110.

Example

The mask is 11111000.

Target	**1 0 1 0 0 1 1 0**	*OR*
Mask	**1 1 1 1 1 0 0 0**	

Result	**1 1 1 1 1 1 1 0**	

29

Setting Bits

- For instance, how can we turn on the light in room #3?

```
int lights = 0;
int mask = 1;
mask <<= 2;
lights |= mask;
```

lights: ...00000000

mask: ...00000001

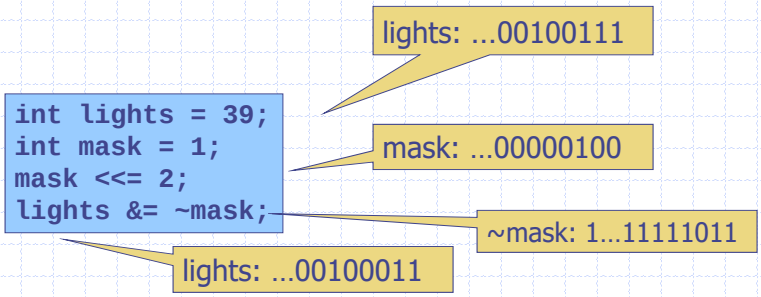
mask: ...00000100

lights: ...00000100

30

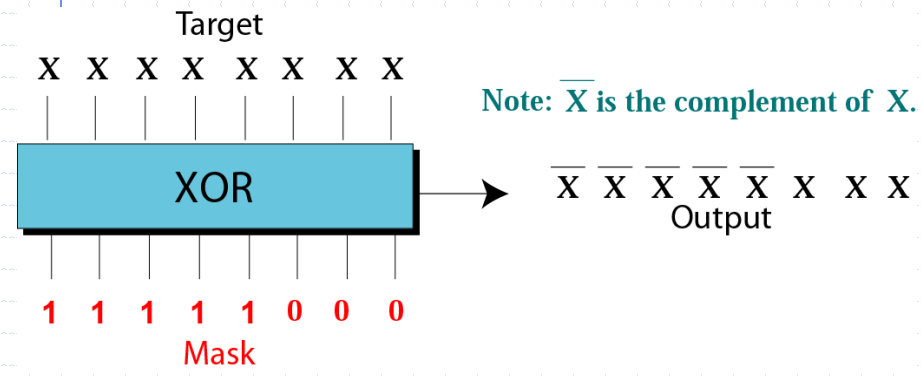
Setting Bits

- For instance, how can we turn off the light in room #3?



31

Example of flipping specific bits



Use a mask to flip the 5 leftmost bits of a byte.

32

Test the mask with the pattern 10100110.

Example

Target	**1 0 1 0 0 1 1 0**	*XOR*
Mask	**1 1 1 1 1 0 0 0**	

Result	**0 1 0 1 1 1 1 0**	

33

Possible Quiz Questions

Give the results of the following bitwise operations:

```
11010011
&
10001100
----------
```

```
11010011
|
10001100
----------
```

```
11010011
^
10001100
----------
```

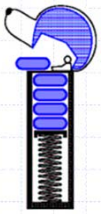
```
~11010011
----------
```

```
11010011>>3
----------
```

```
11010011<<3
----------
```

34

Stacks



- Java provides an inbuilt object type called **Stack**.
  - public Stack()
- Insertions and deletions follow the last-in first-out scheme (LIFO)
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(e): inserts an element, e
  - pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - top(): returns the last inserted element without removing it
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored

35

Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

36

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language supporting recursion
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

37

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    return null
  else
    t ← t – 1
    return S[t + 1]
```

38

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then either grow the array or causes the array to double its size.

```
Algorithm push(o)
  if t = S.length – 1 then
    // double S.length
    …
  t ← t + 1
  S[t] ← o
```

39

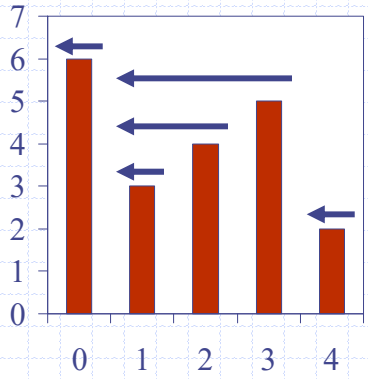
Performance

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Qualifications
  - Trying to push a new element into a full stack causes an implementation-specific exception or
  - Pushing an item on a full stack causes the underlying array to double in size, which implies each operation runs in $O(1)$ **amortized** time.

40

Computing Spans (not in book)

- Using a stack as an auxiliary data structure in an algorithm
- Given an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ before and including $X[i]$, such that $X[j] \leq X[i]$. In other words, the span at i is the distance from i to the first element before i and larger than $X[i]$.
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

41

Quadratic Algorithm

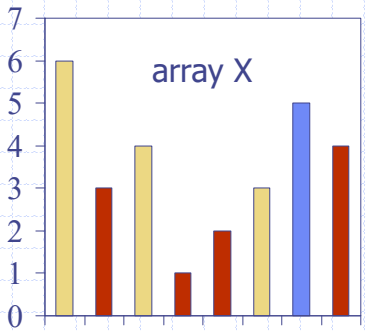
```
Algorithm spans1(X, n)                          # of operations
    Input array X of n integers
    Output array S of spans of X
    S ← new array of n integers                        n
    for i ← 0 to n − 1 do                              n
        s ← i − 1                                      n
        while s >= 0 ∧ X[s] ≤ X[i]          1 + 2 + …+ (n − 1)
            s ← s − 1                       1 + 2 + …+ (n − 1)
        S[i] ← i − s                                   n
    return S                                           1
```

- Algorithm *spans1* runs in $O(n^2)$ time

42

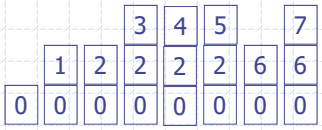
Computing Spans with a Stack

- We keep in a stack the indices of the elements larger than the current.
- We scan the array from left to right
  - Let i be the current index
  - While stack is not empty, we pop indices from the stack until we find index j such that $X[j] > X[i]$
  - If stack is empty, we set $S[i] \leftarrow i + 1$ otherwise
  - we set $S[i] \leftarrow i - j$
  - We push i onto the stack



X:	6	3	4	1	2	3	5	4
S:	1	1	2	1	2	3	6	1

Stack:



43

Computing Spans with a Stack

- We keep in a stack the indices of the elements larger than the current.
- We scan the array from left to right
  - Let i be the current index
  - While stack is not empty, we pop indices from the stack until we find index j such that $X[j] > X[i]$
  - If stack is empty, we set $S[i] \leftarrow i + 1$ otherwise
  - we set $S[i] \leftarrow i - j$
  - We push i onto the stack

```
Algorithm spans2(X, n)
    S ← new array of n integers
    A ← new empty stack
    for i ← 0 to n − 1 do
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] )
            A.pop()
        if A.isEmpty() then
            S[i] ← i + 1
        else
            S[i] ← i − A.top()
        A.push(i)
    return S
```

$A.top() = j$ s.t. $X[j] > X[i]$

44

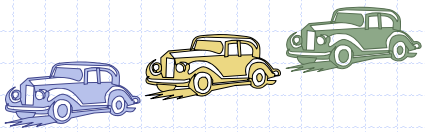
Analysis: Linear Time

- Each index of the array
  - is pushed into the stack exactly one
  - is popped from the stack at most once
- The body of the while-loop is executed at most n times
- Algorithm *spans2* runs in $O(n)$ time
- The body of For-loop has O(1) amortized cost.

```
Algorithm spans2(X, n)                          #
  S ← new array of n integers                   n
  A ← new empty stack                           1
  for i ← 0 to n − 1 do                         n
    while (¬A.isEmpty() ∧
           X[A.top()] ≤ X[i] )                  n
      A.pop()                                   n
    if A.isEmpty() then                         n
      S[i] ← i + 1                              n
    else
      S[i] ← i − A.top()                        n
    A.push(i)                                   n
  return S                                      1
```

45

Queues



- In a Queue, insertions and deletions follow the first-in first-out scheme (FIFO)
- Insertions are at the "rear" or "end" of the queue and removals are at the "front" of the queue
- Main queue operations:
  - enqueue(e): inserts an element, e, at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue
- Java.util.Queue is an interface:

  Queue queueA = new LinkedList();

  enqueue = add, dequeue = remove
- Auxiliary queue operations:
  - first(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue signals an error or returns null

46

Example

Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size()	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	*9*	(7, 3, 5)	

47

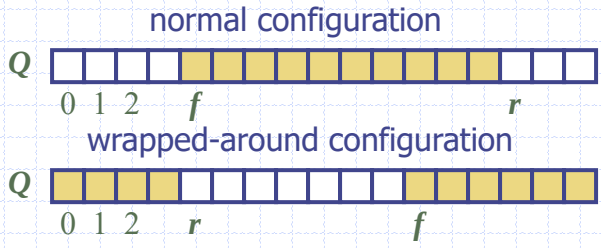
Application: Buffered Output

- The Internet is designed to route information in discrete packets, which are at most 1500 bytes in length.
- Any time a video stream is transmitted on the Internet, it must be subdivided into packets and these packets must each be individually routed to their destination.
- Because of vagaries and errors, the time it takes for these packets to arrive at their destination can be highly variable.
- Thus, we need a way of "smoothing out" these variations

48

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



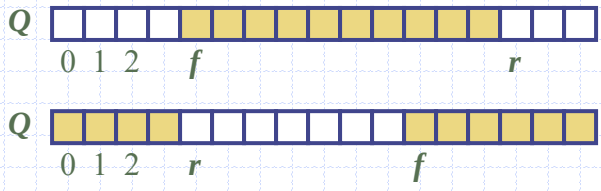
49

Queue Operations

- We use the modulo operator (remainder of division)

```
Algorithm size()
  return sz

Algorithm isEmpty()
  return (sz == 0)
```

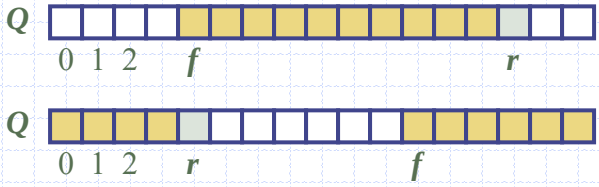


50

Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- One could also grow the underlying array by a factor of 2

```
Algorithm enqueue(o)
  if sz = N then
    signal queue full error
  else
    r ← (f + sz) mod N
    Q[r] ← o
    sz ← (sz + 1)
```



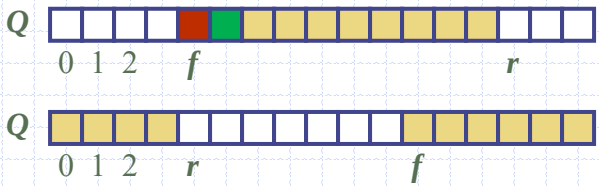
51

51

Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty
- One could alternatively signal an error

```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    o ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz – 1)
    return o
```



52

Index-Based Lists

- An index-based list supports the following operations:

get(r): Return the element of S with index r; an error condition occurs if $r < 0$ or $r > n - 1$.

set(r, e): Replace with e the element at index r and return it; an error condition occurs if $r < 0$ or $r > n - 1$.

add(r, e): Insert a new element e into S to have index r; an error condition occurs if $r < 0$ or $r > n$.

remove(r): Remove from S the element at index r; an error condition occurs if $r < 0$ or $r > n - 1$.

53

Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

54

54

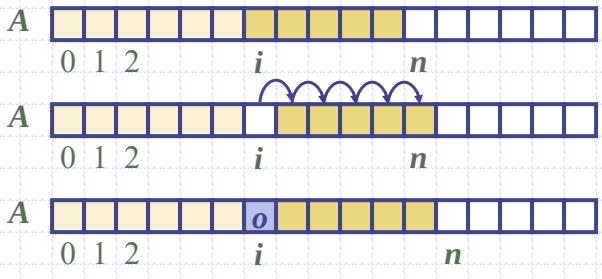
Array-based Lists

- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).

55

Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time

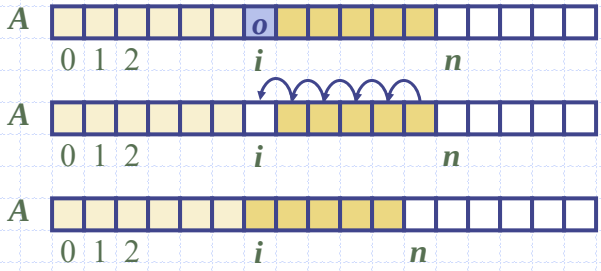


56

56

Element Removal

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



57

57

Pseudo-code

- Algorithms for insertion and removal:

Algorithm add(r, e):
  if $n = N$ **then**
    return "Array is full."
  if $r < n$ **then**
    for $i \leftarrow n - 1, n - 2, \ldots, r$ **do**
      $A[i + 1] \leftarrow A[i]$ // make room for the new element
  $A[r] \leftarrow e$
  $n \leftarrow n + 1$

Algorithm remove(r):
  $e \leftarrow A[r]$ // e is a temporary variable
  if $r < n - 1$ **then**
    for $i \leftarrow r, r + 1, \ldots, n - 2$ **do**
      $A[i] \leftarrow A[i + 1]$ // fill in for the removed element
  $n \leftarrow n - 1$
  return e

58

Performance

- In an array-based implementation of a dynamic list:
  - The space used by the data structure is $O(n)$
  - Indexing the element at i takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time in the worst case
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

59

Linked Lists

- Linked lists store elements at "nodes" or "positions".
- Access methods:

first( ): Returns the position of the first element of L (or null if empty).

last( ): Returns the position of the last element of L (or null if empty).

before(p): Returns the position of L immediately before position p (or null if p is the first position).

after(p): Returns the position of L immediately after position p (or null if p is the last position).

isEmpty( ): Returns true if list L does not contain any elements.

size( ): Returns the number of elements in list L.

60

60

Linked Lists

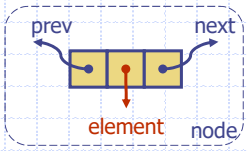
- Update methods:

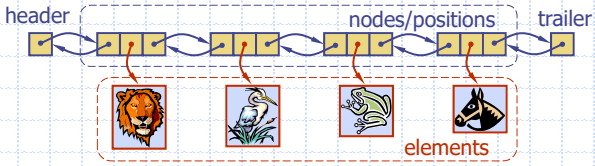
insertBefore(p, e): Insert a new element e into S before position p in S.

insertAfter(p, e): Insert a new element e into S after position p in S.

remove(p): Remove from S the element at position p.

- Implementation:
  - The most natural way to implement a positional list is with a doubly-linked list.

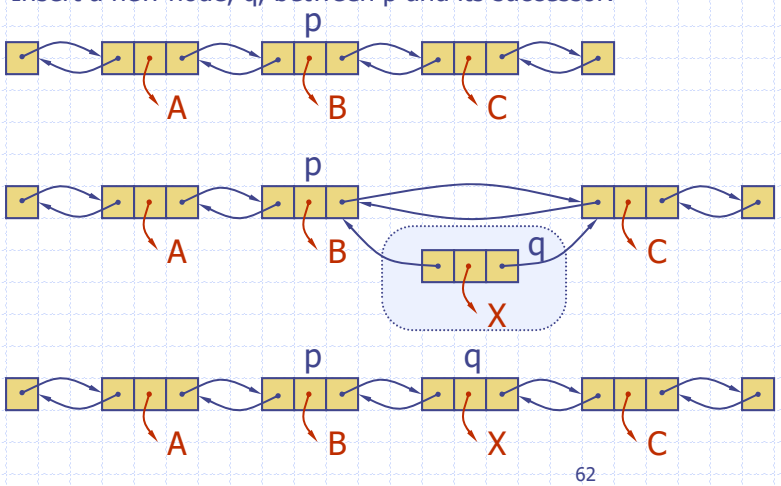




61

Insertion

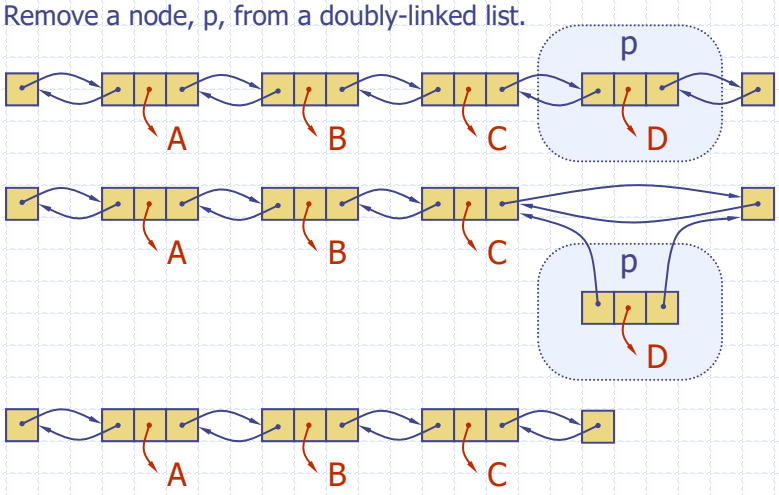
- Insert a new node, q, between p and its successor.



62

Deletion

- Remove a node, p, from a doubly-linked list.



63

63

Pseudo-code

- Algorithms for insertion and deletion in a linked list:

Algorithm insertAfter(p, e):
Create a new node v
v.element $\leftarrow e$
v.prev $\leftarrow p$ // link v to its predecessor
v.next $\leftarrow p$.next // link v to its successor
(p.next).prev $\leftarrow v$ // link p's old successor to v
p.next $\leftarrow v$ // link p to its new successor, v
return v // the position for the element e

Algorithm remove(p):
$t \leftarrow p$.element // a temporary variable to hold the return value
(p.prev).next $\leftarrow p$.next // linking out p
(p.next).prev $\leftarrow p$.prev
p.prev $\leftarrow$ **null** // invalidating the position p
p.next $\leftarrow$ **null**
return t

64

64

Performance

- A linked list can perform all of the access and update operations for a positional list in constant time.

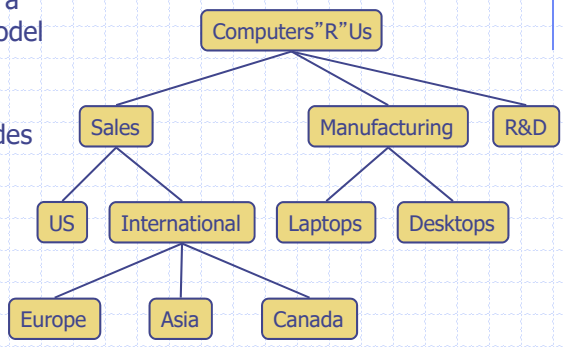
Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

65

65

What is a Tree

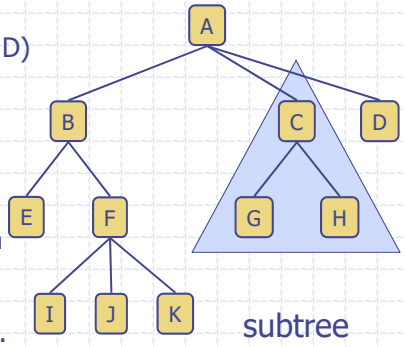
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- **Applications**:
  - Organization charts
  - File systems
  - Programming environments



66

Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



67

Tree Operations

- Accessor methods:

root(): Return the root of the tree.

parent(v): Return the parent of node v; an error occurs if v is root.

children(v): Return a set containing the children of node v.

- Query methods:

isInternal(v): Test whether node v is internal.

isExternal(v): Test whether node v is external.

isRoot(v): Test whether node v is the root.

- Generic methods:

size(): Return the number of nodes in the tree.

elements(): Return a set containing all the elements stored at nodes of the tree.

positions(): Return a set containing all the nodes of the tree.

swapElements(v, w): Swap the elements stored at the nodes v and w.

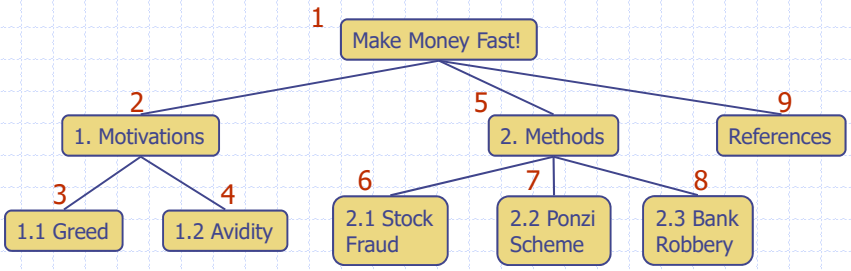
replaceElement(v, e): Replace with e and return the element stored at node v.

68

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(*v*)
  visit(*v*)
  for each child *w* of *v*
    preorder (*w*)

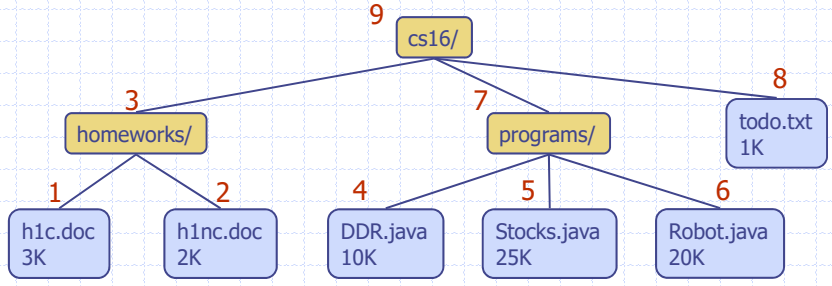


69

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(*v*)
  for each child *w* of *v*
    postOrder (*w*)
  visit(*v*)

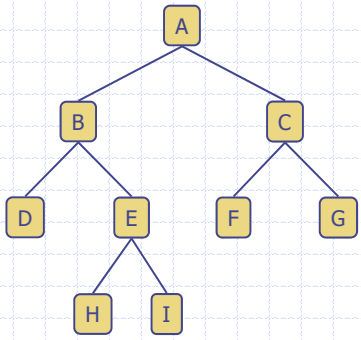


70

Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

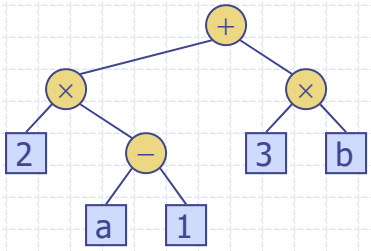
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



71

Arithmetic Expression Tree

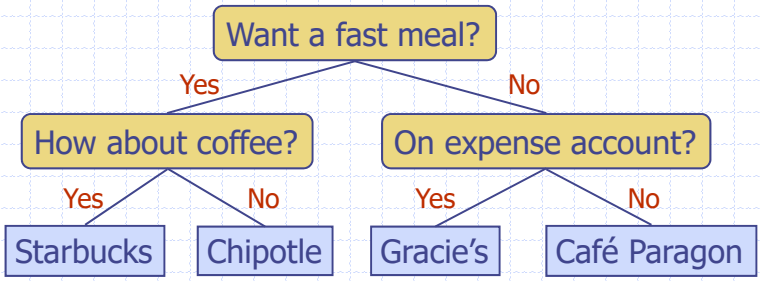
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$
- Post-order traversal:
  - 2 a 1 – x 3 b x +
  - Using a queue to store the list and a stack to compute the value of the expression



72

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



73

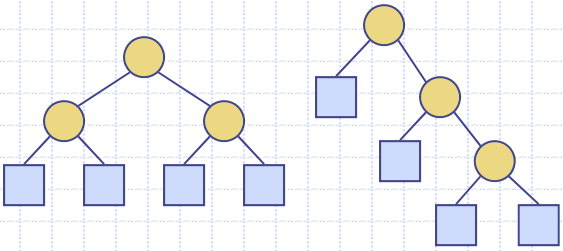
73

Properties of Proper Binary Trees

- Notation
  - n: number of nodes
  - e: number of external nodes
  - i: number of internal nodes
  - h: height

- Properties:
  - $n = e + i$ (*)
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$ (*)
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$ (*)
  - $h \geq \log_2 e$ (*)
  - $h \geq \log_2 (n + 1) - 1$

(*): true for any binary tree



74

Binary Tree Operations

- A binary tree extends the Tree operations, i.e., it inherits all the methods of Tree.
- Additional methods:
  - position leftChild(v)
  - position rightChild(v)
  - position sibling(v)
- The above methods return null when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the binary tree

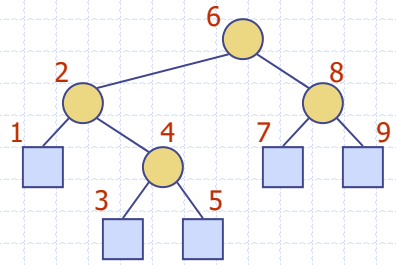
75

75

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

Algorithm *inOrder*(*v*)
  if *left* (*v*) ≠ **null**
    inOrder (*left* (*v*))
  visit(*v*)
  if *right*(*v*) ≠ **null**
    inOrder (*right* (*v*))

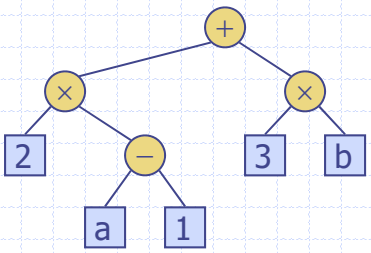


76

76

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree




```
Algorithm inOrder(v)
    if left (v) ≠ null || right(v) ≠ null
        print("(")
    if left (v) ≠ null
        inOrder (left(v))
    print(v.element ())
    if right(v) ≠ null
        inOrder (right(v))
    if left (v) ≠ null || right(v) ≠ null
        print (")")
```

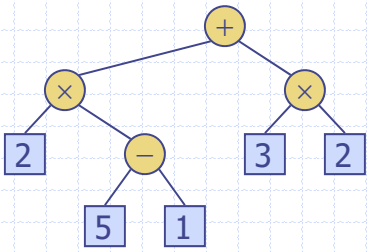
((2 × (a − 1)) + (3 × b))

77

77

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



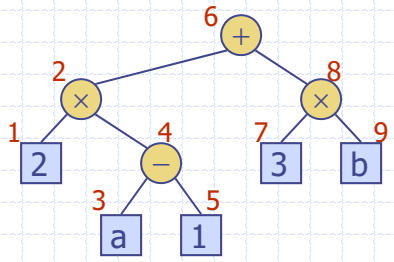

```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(left(v))
        y ← evalExpr(right(v))
        return v.element(x, y)
```

78

78

Inorder Traversal

- Application: draw a binary tree from x(v) and y(v):
  - x(v) = inorder rank of v
  - y(v) = depth of v
- Example:
- v: 2, x, a, –, 1, +, 3, x, b
- x(v): 1, 2, 3, 4, 5, 6, 7, 8, 9
- y(v): 2, 1, 3, 2, 3, 0, 2, 1, 2



Algorithm ?

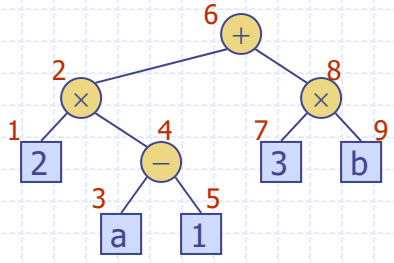
1. Sort v by inorder rank
2. Call create(1, n, 0)

```
create(f, t, d):
1. If (f = t) return v[f]
2. Pick j in [f..t] such that y[j] = d
3. Return (v[j], create(f, j-1, d+1),
                 create(j+1, t, d+1))
        // as (root, leftChild, rightChild)
```

79

Possible Quiz Question

- Draw a binary tree from x(v) and y(v):
  - x(v) = inorder rank of v
  - y(v) = postorder rank of v
- Example:
- v: 2, x, a, –, 1, +, 3, x, b
- x(v): 1, 2, 3, 4, 5, 6, 7, 8, 9
- y(v): 1, 5, 2, 4, 3, 9, 6, 8, 7

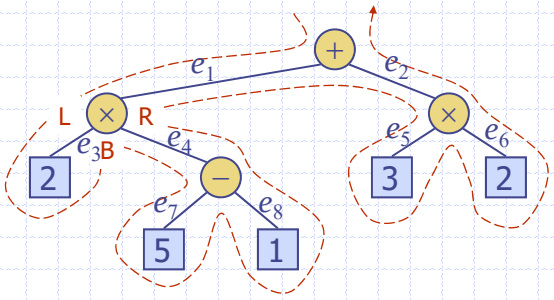


Algorithm ?

80

Euler Tour Traversal

- Generic traversal of a tree
- Travel each edge exactly twice.



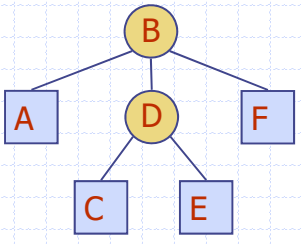
Euler Tour = $e_1 e_3 e_3 e_4 e_7 e_7 e_8 e_8 e_4 e_1 e_2 e_5 e_5 e_6 e_6 e_2$

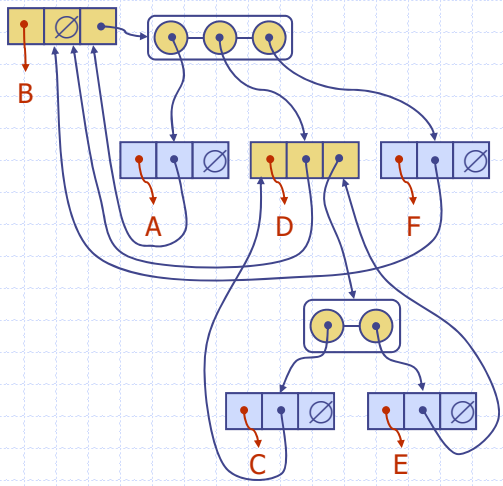
81

81

Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

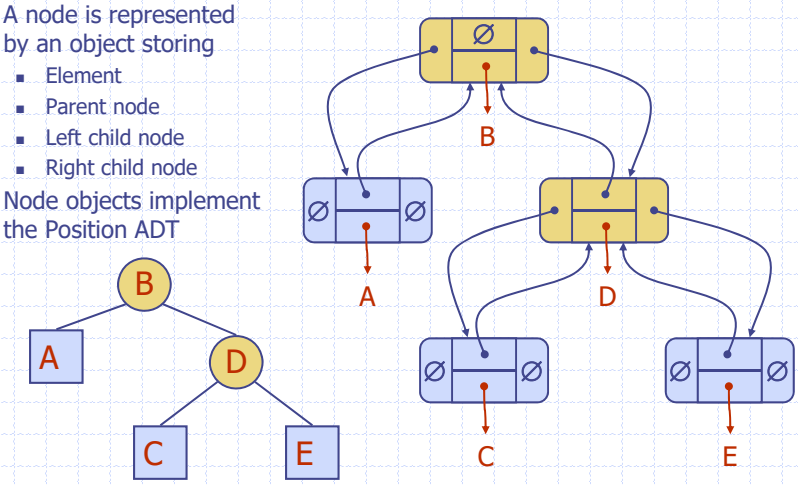




82

Linked Structure for Binary Trees

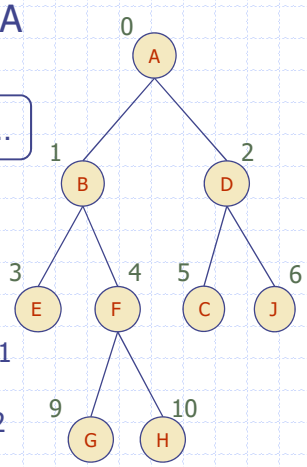
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



83

Array-Based Representation of Binary Trees

- Nodes are stored in an array A



Node v is stored at A[pos(v)]

- pos(root) = 0
- if node is the left child of parent(node),
  pos(node) = 2 · pos(parent(node)) + 1
- if node is the right child of parent(node),
  pos(node) = 2· pos(parent(node)) + 2

84

Exercise A-1.12

Given an array, A, of $n - 2$ unique integers in the range from 1 to n, describe an $O(n)$-time method for finding the two integers in the range from 1 to n that are not in A. You may use only $O(1)$ space in addition to the space used by A.

$$1+2+\ldots+n = n(n+1)/2$$
$$1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6$$

Let the missing numbers be x and y, then

$\text{sum}(A) + x + y = n(n+1)/2$

$\text{sum}(A^2) + x^2 + y^2 = n(n+1)(2n+1)/6$

85

Exercise A-1.4

An evil king has a cellar containing n bottles of expensive wine, and his guards have just caught a spy trying to poison the king's wine. Fortunately, the guards caught the spy after he succeeded in poisoning only one bottle. Unfortunately, they don't know which one. To make matters worse, the poison the spy used was very deadly; just one drop diluted even a billion to one will still kill someone. Even so, the poison works slowly; it takes a full month for the person to die. Design a scheme that allows the evil king to determine exactly which one of his wine bottles was poisoned in just one month's time while using a least number of taste testers and expending at most $O(\log n)$ of his taste testers.

86

Exercise A-1.5

Suppose you are given a set of small boxes, numbered 1 to n, identical in every respect except that each of the first i contain a pearl whereas the remaining $n-i$ are empty. You also have two magic wands that can each test whether a box is empty or not in a single touch, except that a wand disappears if you test it on an empty box. Show that, without knowing the value of i, you can use the two wands to determine all the boxes containing pearls using at most $o(n)$ wand touches. Express, as a function of n, the asymptotic number of wand touches needed.

87

Exercise A-1.6

Repeat the previous problem assuming that you now have k magic wands, with $k > 2$ and $k < \log n$. Express, as a function of n and k, the asymptotic number of wand touches needed to identify all the magic boxes containing pearls.

88