SEARCHING

There may be different situations where searching is performed

SEQUENTIAL SEARCH

Given $a_1, \ldots, a_i, \ldots, a_n$ (numbers / characters: objects to be searched)

Find: x

(naïve or brute force search or straight search)

It is in fact one loop:

```
FOR i = 1 TO n
    IF x = a_i THEN STOP
```

O(n)

Note: this kind of search is very simple (primitive),

but what is if a_i is a matrix?

records of a file?

The comparison is more complicated here. But in principle it is very simple.

RANDOM SEARCH

Introduce some sort of probability.

Given $a_1, \ldots, a_i, \ldots, a_n$ to be searched.

Find: x

Coin: probability element

Flip a coin
- If head: search from 1 to n.
- If tail: search from n to 1.

Two sequential searches are combined, applied together.

Assume: $x = a_i$ (i^{th} position)

The coin is fair: the head and the tail occur with equal probabilities.

If head: i comparisons to find x.

If tail: $n - i + 1$ comparisons to find x.

$\frac{1}{2} i + \frac{1}{2} (n - i + 1) = (n + 1) / 2$ better than n. (Average the two case.)

In average we need $(n + 1) / 2$ comparisons rather than n.

The order of the elements is irrelevant (no need for pre-sorting) in

- Sequential search,
- Randomized search.

BINARY SEARCH

It is very quick and used almost everywhere.

The elements to be searched are sorted.

Given $a_1 \leq \ldots \leq a_i \leq \ldots \leq a_n$

Find: x

Idea: guess the number I am thinking at

Ex. 2 3 7 | 8 9 | 10

Find: 9

- Half the sequence.
- Compare the last element with x.

$a_1, \ldots, a_n$

x

low: leftmost element in the half = 1

high: rightmost element in the half = n

```
REPEAT
        mid = (low + high) DIV 2
        IF low > high THEN mid = 0
        ELSE
        IF a(mid) < x THEN low = mid + 1
        ELSE high = mid – 1
UNTIL x = a(mid)
```

$\frac{n}{2^0}$

$\frac{n}{2^1}$

.

.

.

$\frac{n}{2^{\lceil \log n \rceil}}$

O(log n) , very fast.

Note: n finite

If n infinite then binary search: div 2^m

It may not happen in practice, just in theory.

Note:

1. Will the search work when all elements are given at once (at the same time)?

2. Will the search work when all elements are given on line (one by one)?

	1.	2.
Sequential search	YES	YES
Randomized search	YES	NO
Binary search	YES	NO

In every case we assume that we have just one processor to do the job.

PARALLEL BINARY SEARCH

It speeds up the binary search.

P processors with shared memory (PRAM parallel RAM).

CREW: Concurrent Read Exclusive Write

Given the elements $a_1, \ldots, a_j, \ldots, a_n$ sorted.

Find: x

Divide the sequence into p + 1 parts:

$$a_1, \ldots, \underline{a_{i1}} \mid a_{i1+1}, \ldots, \underline{a_{i2}} \mid \ldots, a_{ij} \mid \ldots, \underline{a_n}$$

x is compared with the boundary elements (or the leftmost or the rightmost)

in parallel: processor j compares x with the j^{th} boundary

processor j sets a variable cj: 0, if $x > a_{ij}$

1, otherwise

Thus: $\exists s$: $c_s = 0 \wedge c_{s+1} = 1$ (to locate part)

Repeat recursively until $x = a_{ij}$

Complexity:

BS	PBS
$\frac{n}{2^0}$	$\frac{n}{(p+1)^0}$
.	$\frac{n}{(p+1)^2}$
.	
.	
$\frac{n}{2^{\lfloor \log n \rfloor}}$	$\frac{n}{(p+1)^n}$ $\Big\} = 1$

$O(\log_{p+1} n)$ better than $O(\log n)$, if $p > 1$

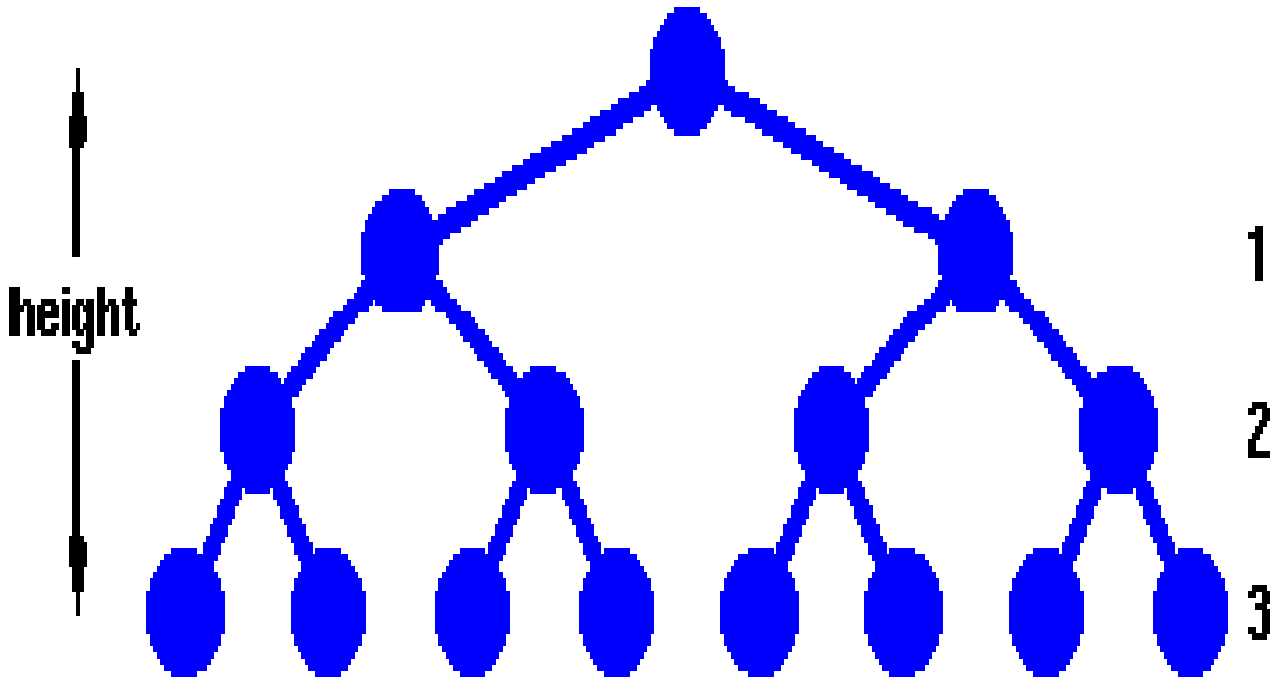
Note: PBS doesn't online

PBS works offline

Binary Search Tree

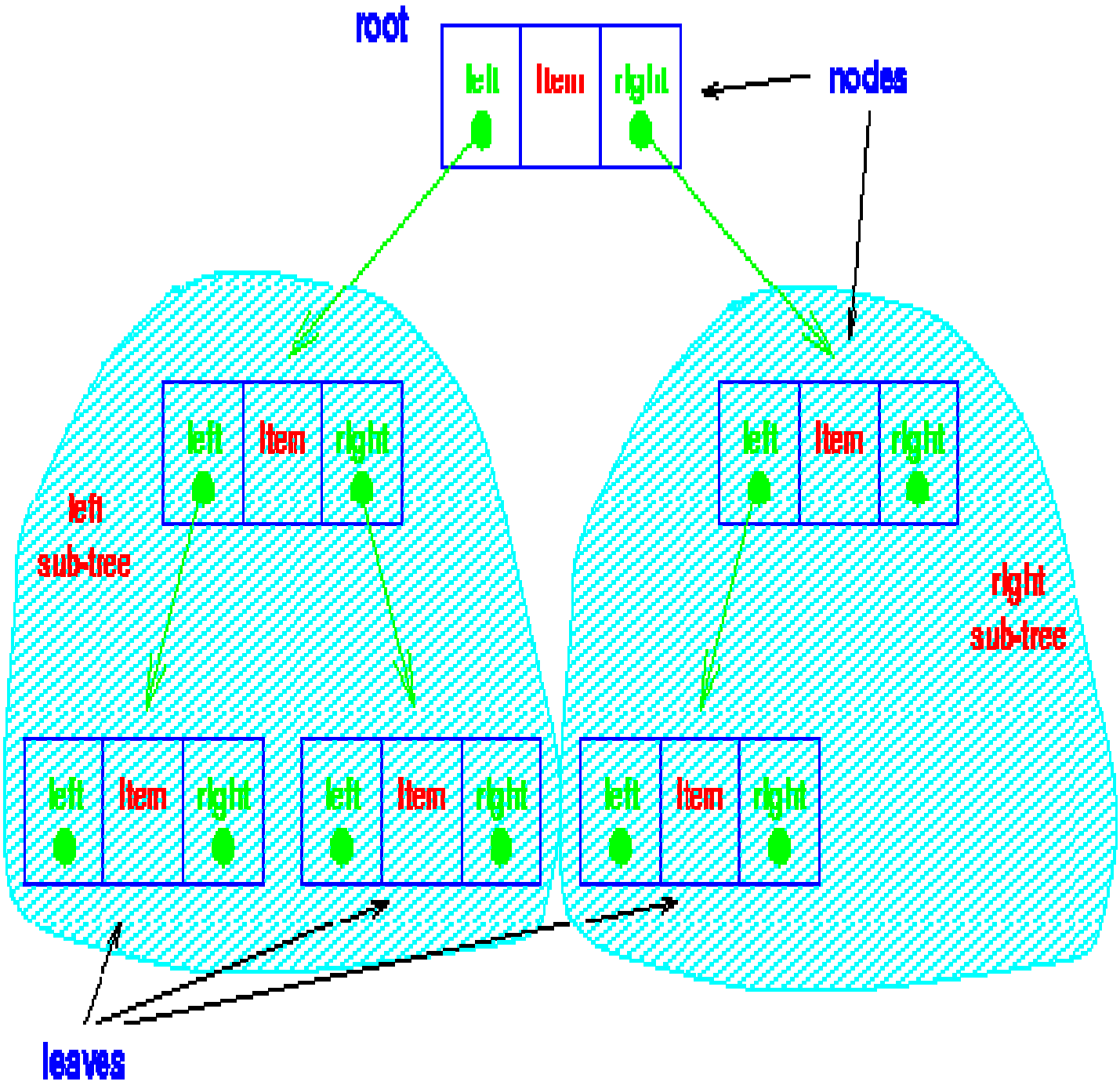
Binary Search tree is a binary tree in which each internal node *x* stores an element such that the element stored in the left subtree of *x* are less than or equal to *x* and elements stored in the right subtree of *x* are greater than or equal to *x*. This is called binary-search-tree property.

The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in $\Theta(\lg n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations takes (n) worst-case time.



The height of the Binary Search Tree equals the number of links from the root node to the deepest node.

Binary Search Tree can be implemented as a linked data structure in which each node is an object with three pointer fields. The three pointer fields left, right and *p* point to the nodes corresponding to the left child, right child and the parent respectively NIL in any pointer field signifies that there exists no corresponding child or parent. The root node is the only node in the BTS structure with NIL in its *p* field.


root
left
item
right
nodes
left
item
right
left
sub-tree
left
item
right
right
sub-tree
left
item
right
left
item
right
left
item
right
leaves

Inorder Tree Walk

During this type of walk, we visit the root of a subtree between the left subtree visit and right subtree visit.

```
INORDER-TREE-WALK
(x)
If x ≠NIL then
   INORDER-TREE-WALK
(left[x])
   print key[x]
   INORDER-TREE-WALK
(right[x])
```

It takes $\Theta(n)$ time to walk a tree of n nodes. Note that the Binary Search Tree property allows us to print out all the elements in the Binary Search Tree in sorted order.

Preorder Tree Walk

In which we visit the root node before the nodes in either subtree.

```
PREORDER-TREE-WALK
(x)
If x not equal NIL then
   PRINT key[x]
   PREORDER-TREE-
WALK (left[x])
   PREORDER-TREE-
WALK (right[x])
```

Postorder Tree Walk

In which we visit the root node after the nodes in its subtrees.

```
POSTORDER-TREE-
WALk (x)
If x not equal NIL then
   POSTORDER-TREE-
WALK (left[x])
   PREORDER-TREE-
WALK (right[x])
   PRINT key [x]
```

It takes O(*n*) time to walk (inorder, preorder and pastorder) a tree of *n* nodes.

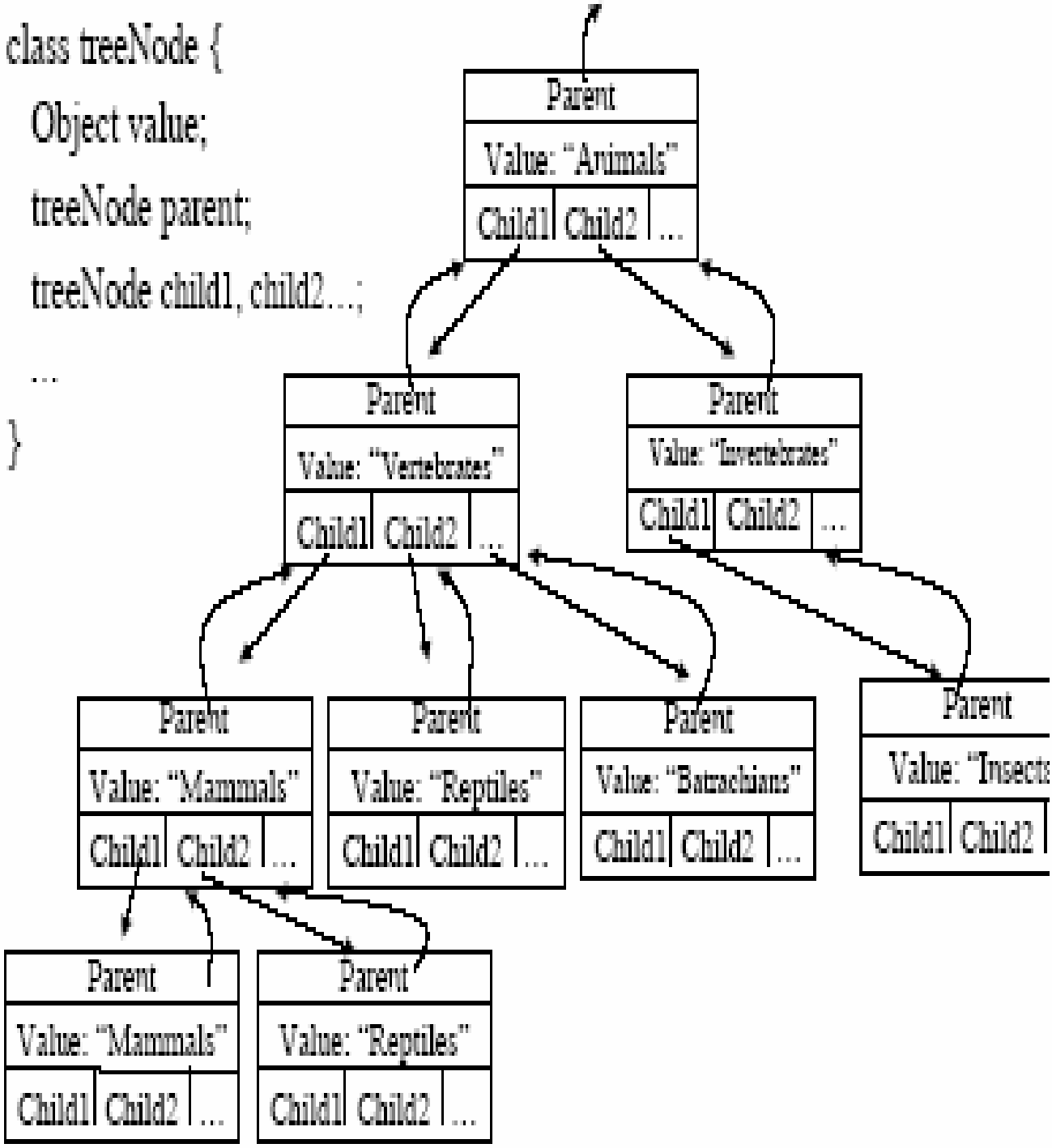
- **inorder: walk the left child, visit the root, walk the right child**

- **preorder: visit the root, walk the left child, walk the right child**

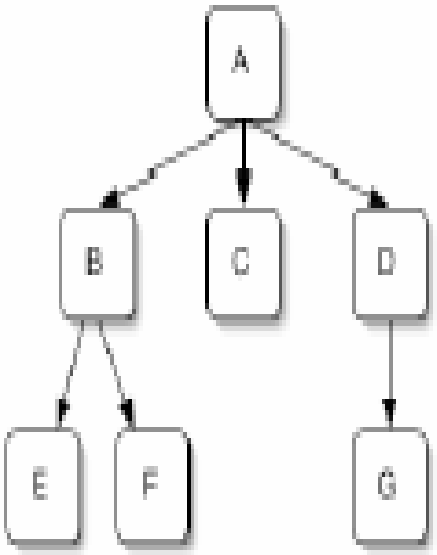
- **postorder: walk the left child, walk the right child, visit the root**

Tree data structure

```
class treeNode {
   Object value;
   treeNode parent;
   treeNode child1, child2…;
   …
}
```



Tree vocabulary



Root: A (only node with parent==null)

Children(B) = E, F

Siblings(X) = {Nodes with the same parent as X, excluding X}

Siblings(B) = {C, D}, **Siblings**(A) = {}

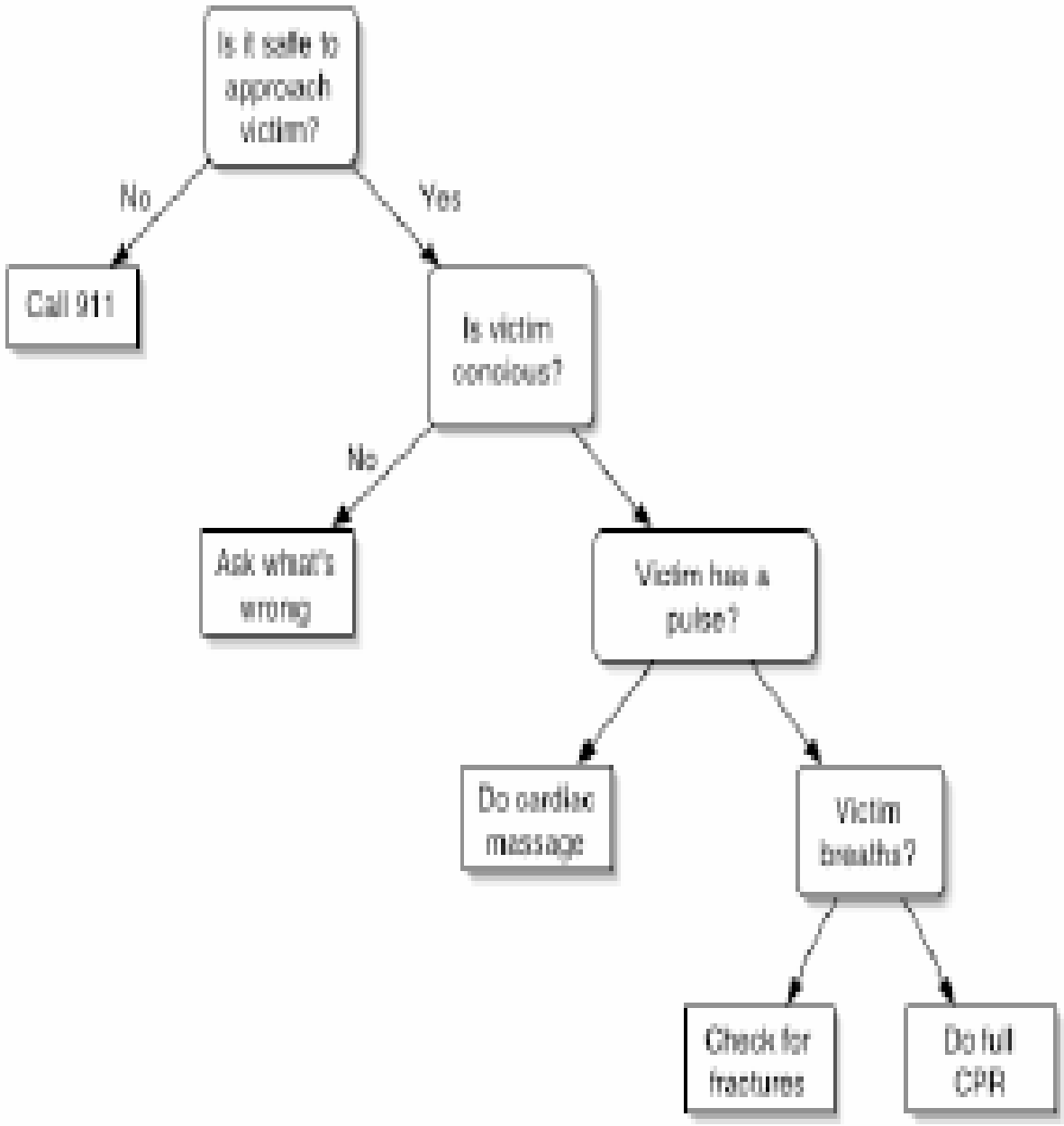
Descendants(X) = {Nodes below X}

Descendants(A) = {B,C,D,E,F,G}

Ancestors(X) = {Nodes between X and the root}

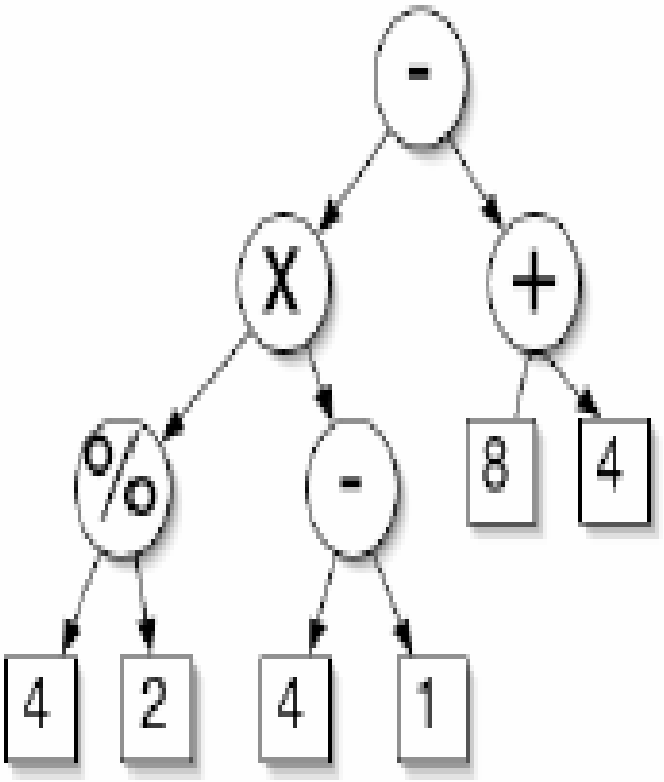
Ancestors(E) = {B, A}

Application of binary trees: Decision trees



Application of binary trees: Representation of math. expressions

- ((4 % 2) x (4 - 1)) - (8 + 4))



Binary Search and Inverted Files

Original Documents:

1. The use of computers in library and information science education.

2. Education of librarians in faculties of education, library and information science, and communications compared.

3. Computers in science: a case study of computers and computer use by scientists.

Step #1 Extract Terms		Step #2 Sort Terms		Step #3a Remove Duplicates	
Term	Doc#	Term	Doc#	Term	Doc#
use	1	case	3	case	3
computers	1	communications	2	communications	2
library	1	compared	2	compared	2
information	1	computer	3	computer	3
science	1	computers	1	computers	1;3
education	1	computers	3	education	1; 2
education	2	education	2	faculties	2
librarians	2	education	1	information	1;2
faculties	2	education	2	librarians	2
education	2	faculties	2	library	1:2
library	2	information	1	science	1;2;3
information	2	information	2	scientists	3
science	2	librarians	2	study	3
communications	2	library	2	use	1;3
compared	3	library	1		
computers	3	science	3		
science	3	science	2		
case	3	science	1		
study	3	scientists	3		
computer	3	study	3		
use	3	use	1		
scientists	3	use	3		

Postfix Evaluation

Postfix Evaluation :

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows :

- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be at least two operands in the stack.
    - If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate topStack(Operator)temp. Let the result of this operation be retVal.

Pop the stack and Push retVal into the stack.

Repeat this step till all the characters are scanned.

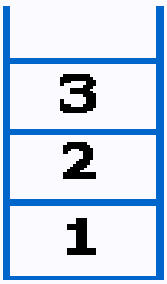
- After all characters are scanned, we will have only one element in the stack. Return topStack.

Example :

Let us see how the above algorithm will be imlemented using an example.

Postfix String : 123*+4-

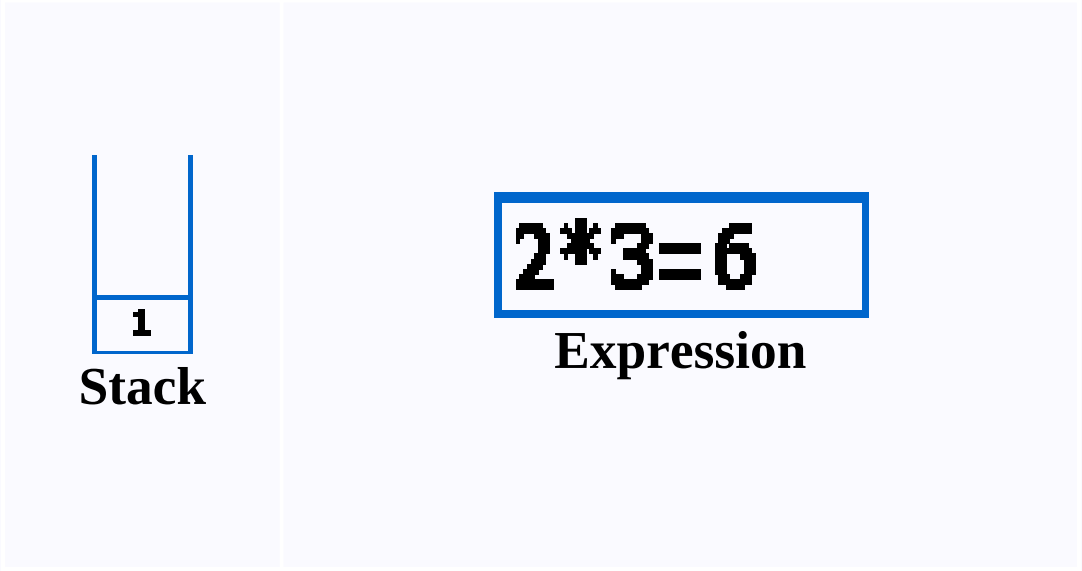
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.



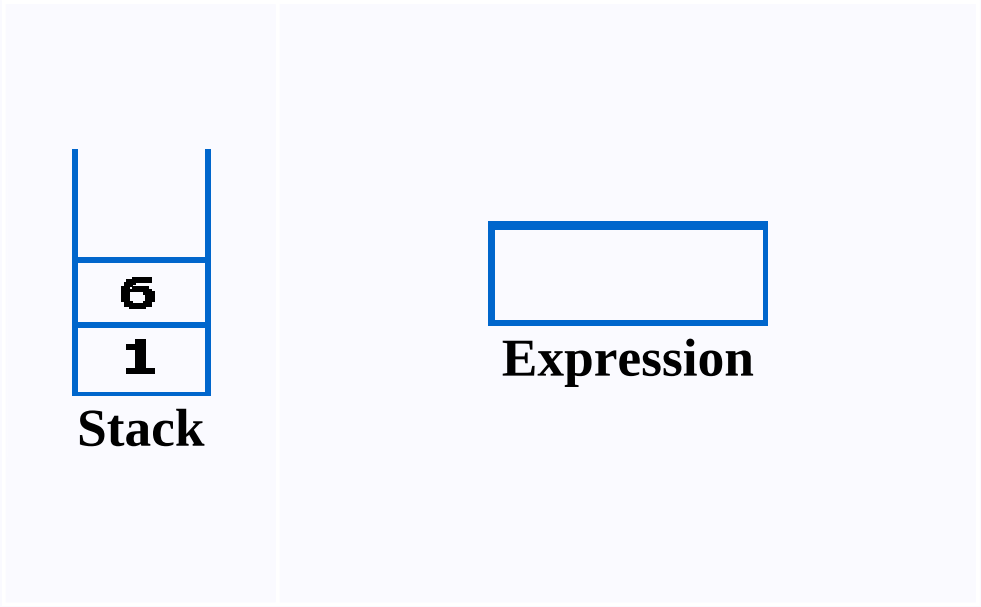
Stack

Expression

Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.

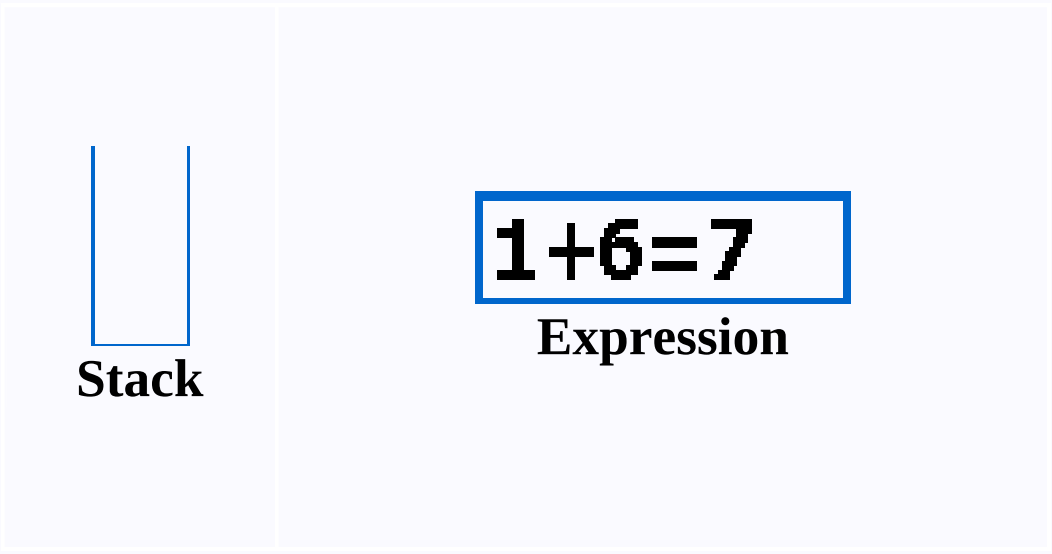


The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.

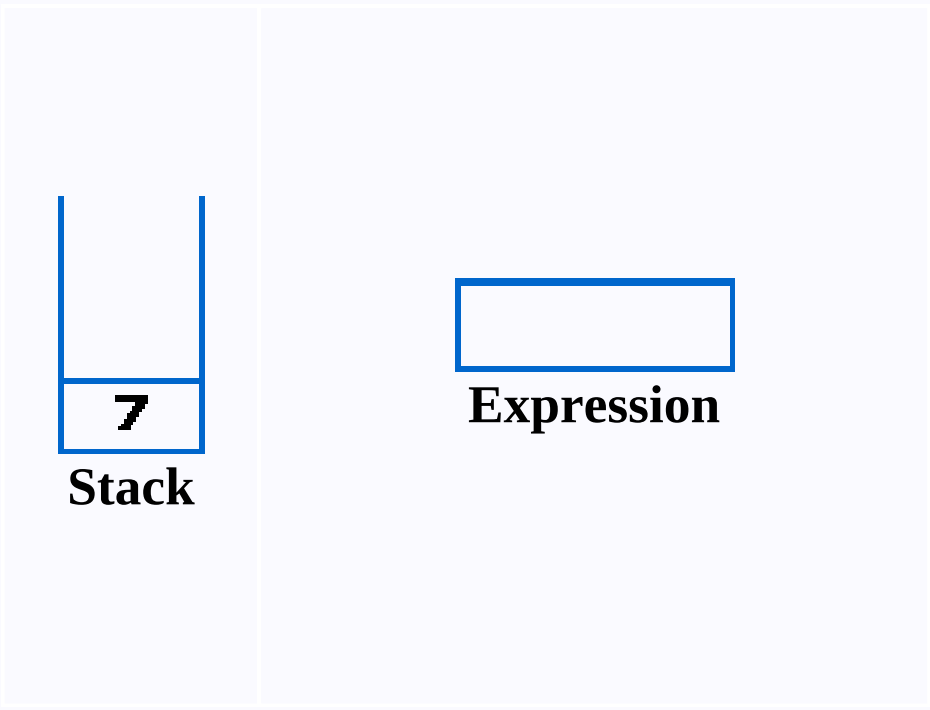


Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the

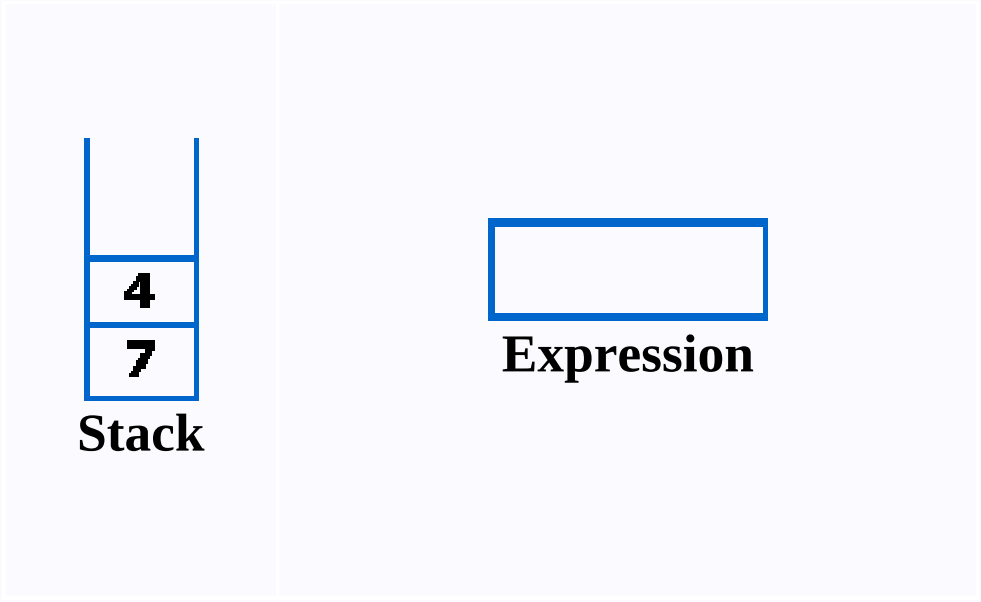
two operands. The second operand will be the first element that is popped.



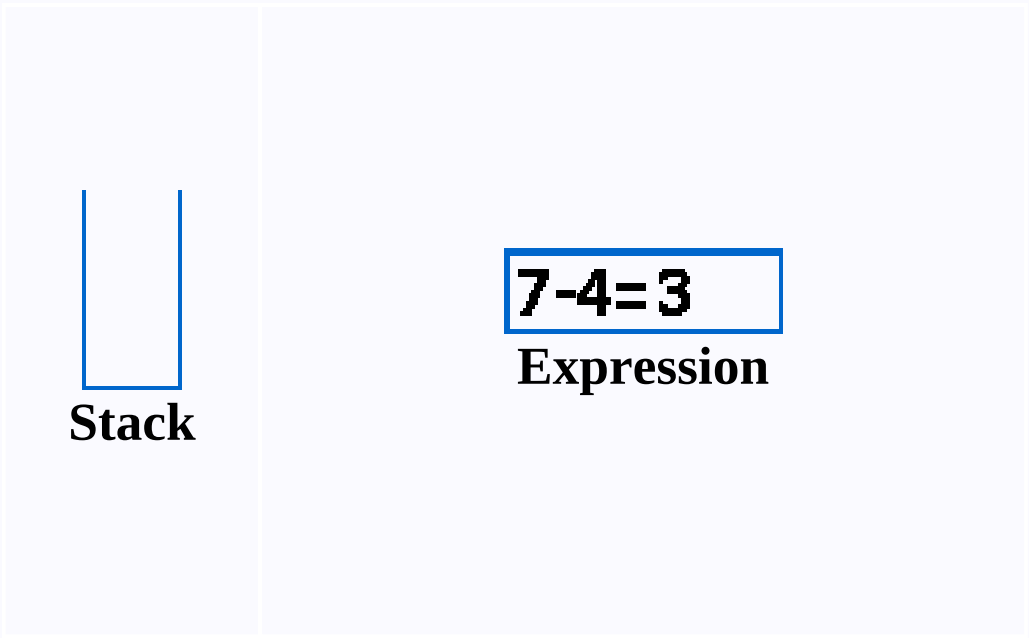
The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.



Next character scanned is "4", which is added to the stack.

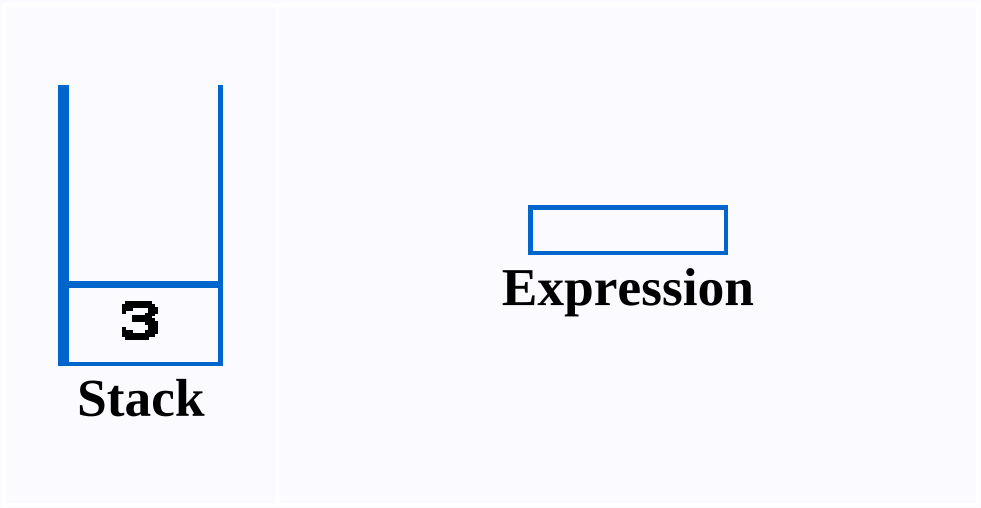


Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into

the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3