Trees

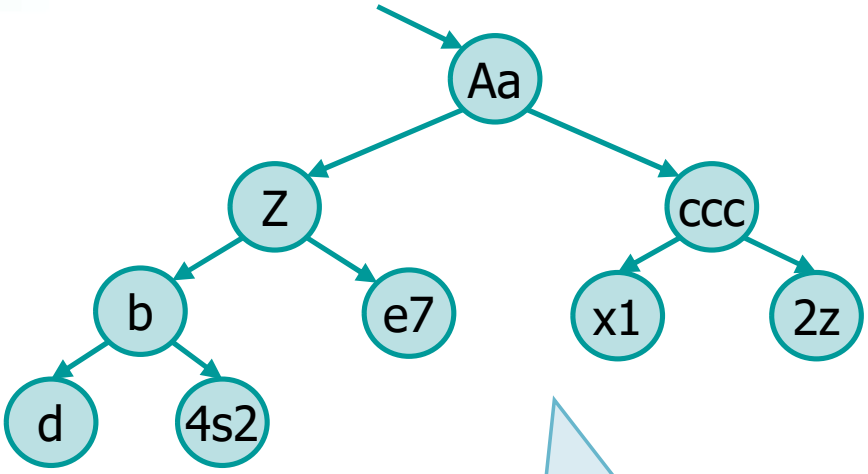
Binary Trees

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

insertion, deletion and visit are the basic operations to deal with a data structure

Visits

- A tree traversal or a tree visit lists the nodes according to a strategy
- Three strategies are used

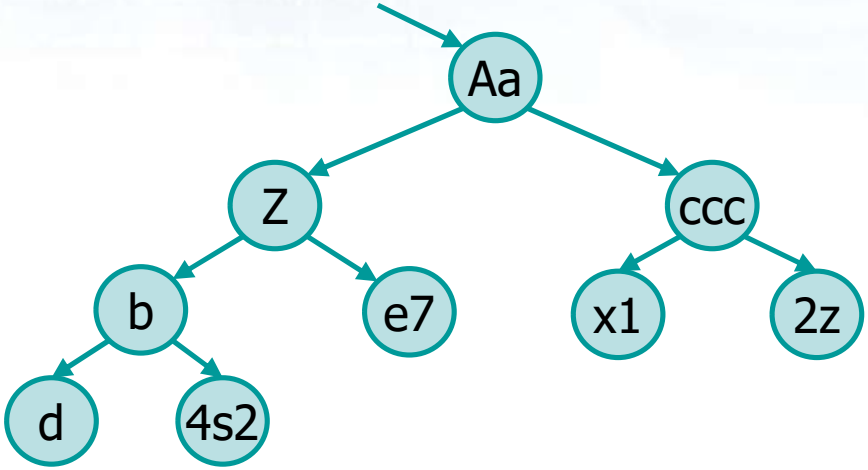


We suppose the key is a string

  - Pre-order
    - **Root**, Left child (l), Right child (r)
  - In-order
    - Left child (l), **Root**, Right child (r)
  - Post-order
    - Left child (l), Right child (r), **Root**

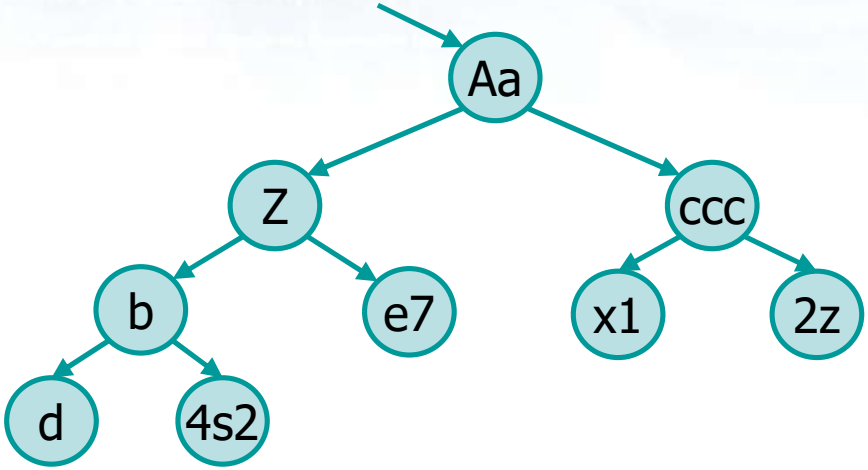
EXAMPLES BELOW

Pre-order



Aa Z b d 4s2 e7 ccc x1 2z

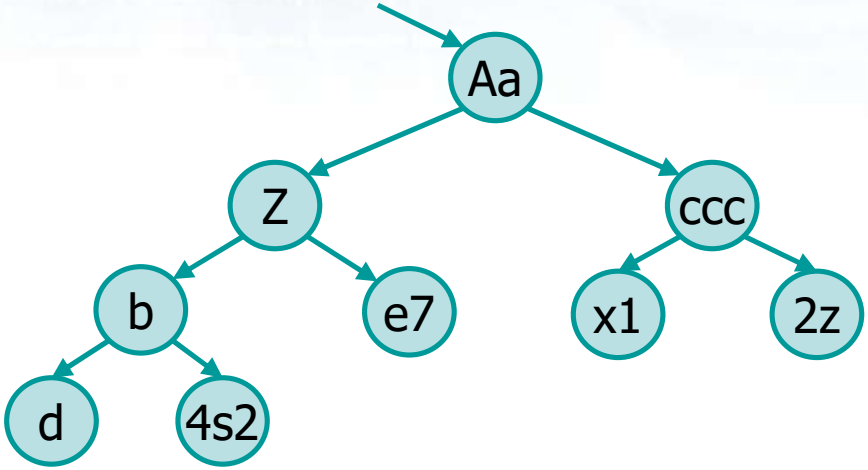
Pre-order



Aa Z b d 4s2 e7 ccc x1 2z

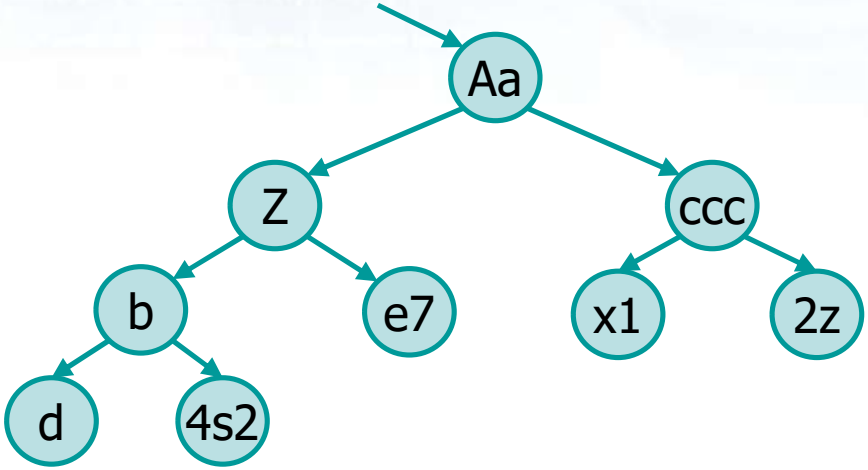
```
void preorder_r (
   node_t *root
) {
   if (root == NULL)
      return;
   print ("%s ", root->key);
   preorder_r (root->l);
   preorder_r (root->r);
   return;
}
```

In-order



d b 4s2 Z e7 Aa x1 ccc 2z

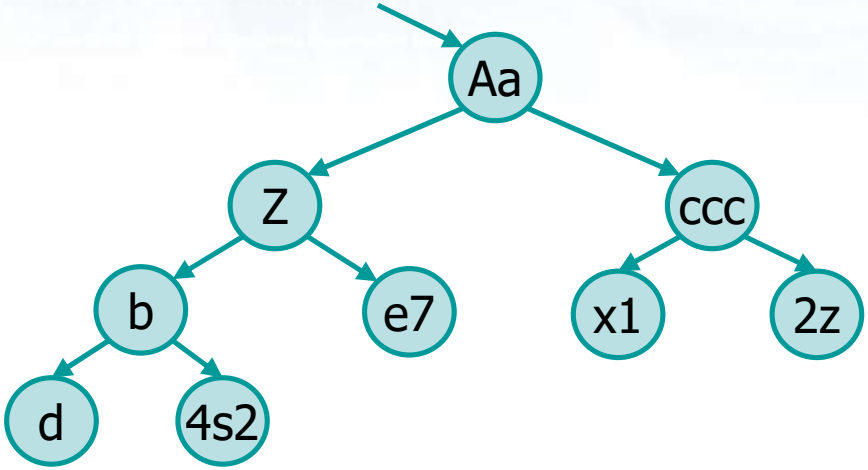
In-order



d b 4s2 Z e7 Aa x1 ccc 2z

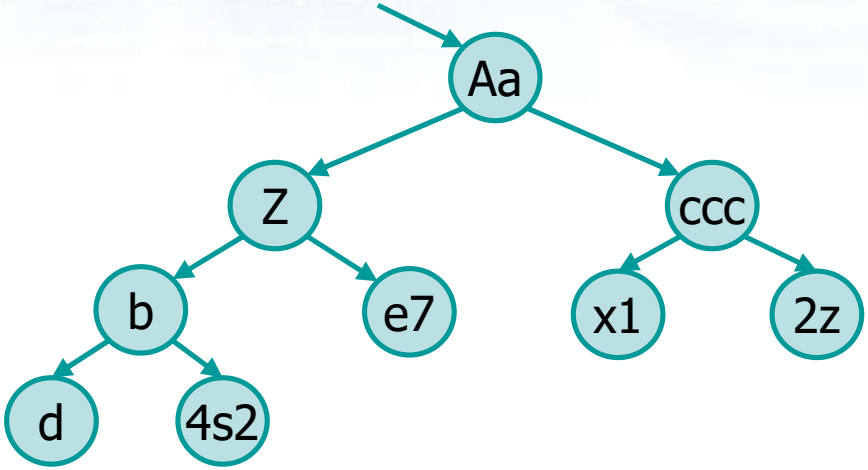
```
void inorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  inorder_r (root->l);
  print ("%s ", root->key);
  inorder_r (root->r);
  return;
}
```

Post-order



d 4s2 b e7 Z x1 2z ccc Aa

Post-order



d 4s2 b e7 Z x1 2z ccc Aa

```
void postorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  postorder_r (root->l);
  postorder_r (root->r);
  print ("%s ", root->key);
  return;
}
```

Comparison

```
void preorder_r (
  node_t *root
) {
  if (root == NULL)
    return;
  print ("%s ", root->key);
  preorder_r (root->l);
  preorder_r (root->r);
  return;
}
```

```
void inorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  inorder_r (root->l);
  print ("%s ", root->key);
  inorder_r (root->r);
  return;
}
```

reasoning:
I go to the left until null
I print and call the right
then I still go to the left
until null, etc...

```
void postorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  postorder_r (root->l);
  postorder_r (root->r);
  print ("%s ", root->key);
  return;
}
```

Complexity Analysis

- Case 1
  - Complete tree
    - Already analyzed for the first example of section u04s01

$$T(n) = O(n)$$

```
void inorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  inorder_r (root->l);
  print ("%s ", root->key);
  inorder_r (root->r);
  return;
}
```

Divide and conquer problem	
Number of subproblems	$a = 2$
Reduction factor	$b = {}^{n}/_{\hat{n}} = 2$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + a \cdot T\left({}^{n}/_{b}\right) + C(n)$$

$$T(n) = \Theta(1)$$

$n > 1$

$n \leq 1$

Complexity Analysis

- Case 2
  - Completely unbalanced tree
    - Already analyzed for the factorial example of section u04s01

$$T(n) = O(n)$$

```
void inorder_r (
  node_t *root
){
  if (root == NULL)
    return;
  inorder_r (root->l);
  print ("%s ", root->key);
  inorder_r (root->r);
  return;
}
```

Divide and conquer problem	
Number of subproblems	$a = 1$
Reduction value	$k_i = 1$
Division cost	$D(n) = \Theta(1)$
Recombination cost	$C(n) = \Theta(1)$

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n - k_i) + C(n)$$

$$T(n) = \Theta(1) \quad \begin{matrix} n > 1 \\ n \leq 1 \end{matrix}$$

Parameter Computation

- Compute the number of nodes of a binary tree

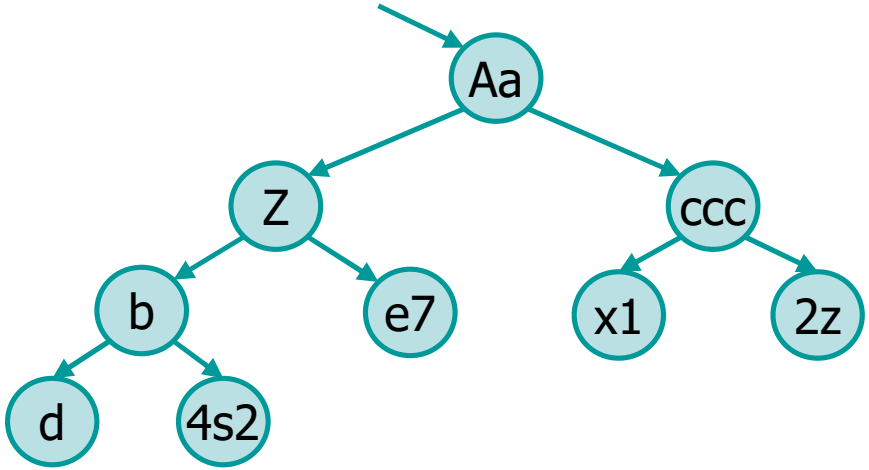
Number of nodes

Root and Sentinel (or nothing for a termination condition checking on NULL)

```
int count (node_t *root) {
 int l, r;

 if (root == NULL)
   return 0;

 l = count (root->l);
 r = count (root->r);
 return (l+r+1);
}
```



Parameter Computation

- Compute the height of a binary tree

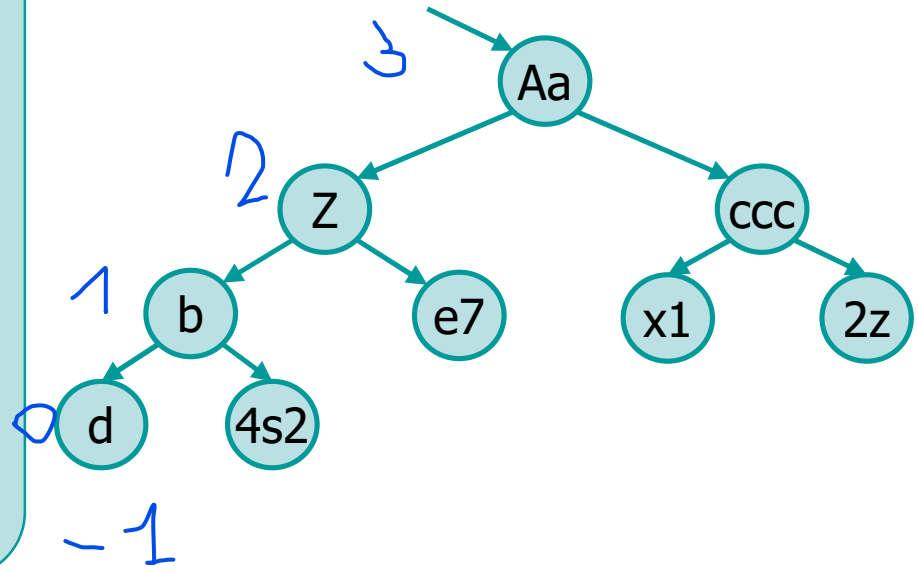
```
int height (node_t *root) {
  int u, v;

  if (root == NULL)
    return -1;

  u = height (root->l);
  v = height (root->r);

  if (u>v)
    return (u+1);
  else
    return (v+1);
}
```

because 4 consequent nodes have height=3

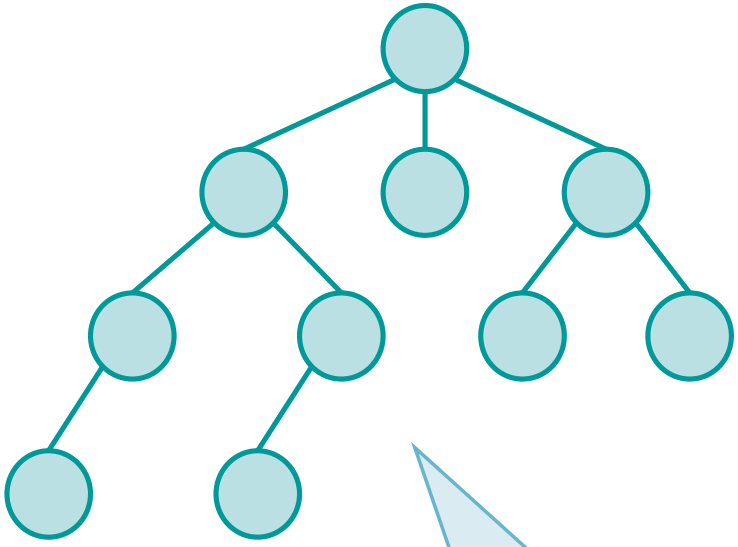


Exercise

- Given an n-ary tree extend the previous functions to compute its
  - Number of nodes
  - Height

Node structure

```
typedef struct node_s node_t;
struct node_s {
  int key;
  ...
  int degree;
  node_t **children;
};
```



Tree of
10 nodes
and height 3

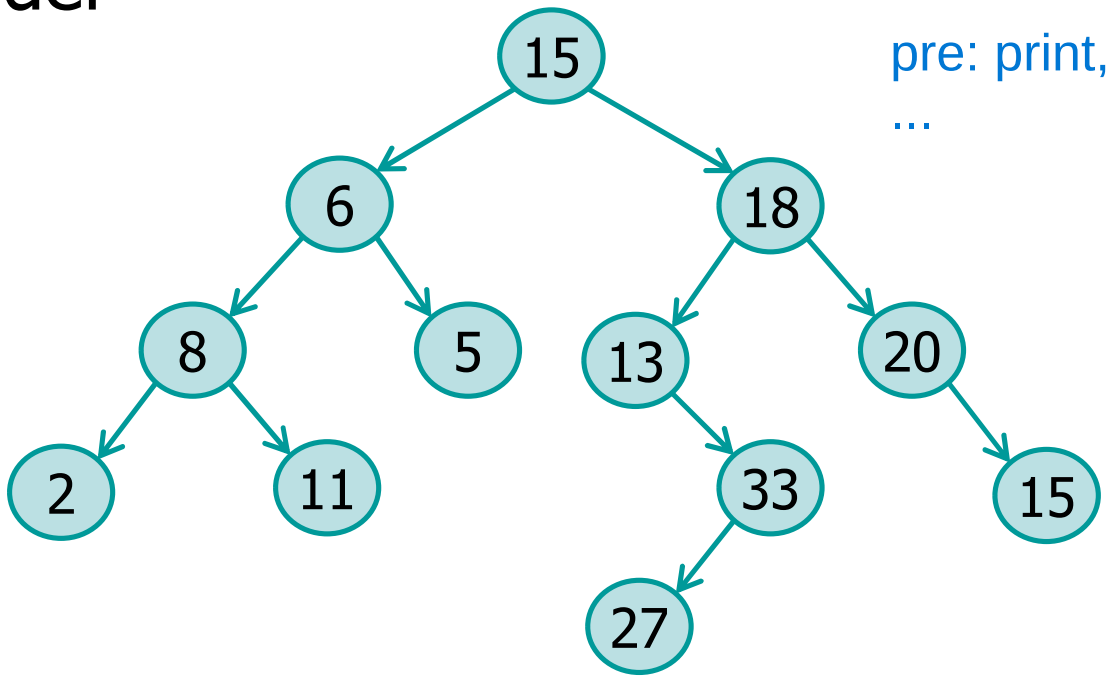
Solution

```
int count (node_t *root) {
  int i, c;
  if (root == NULL)
    return 0;
  for (c=0, i=0; i<root->degree; i++) {
    c = c + count (root->children[i]);
  }
  return (c+1);
}
```

```
int height (node_t *root) {
  int i, tmp, max=-1;
  if (root == NULL)
    return -1;
  for (i=0; i<root->degree; i++) {
    tmp = height (root->children[i]);
    if (tmp > max)
      max = tmp;
  }
  return (max+1);
}
```

Exercise

- Given the following tree visit it in pre, in, and post-order

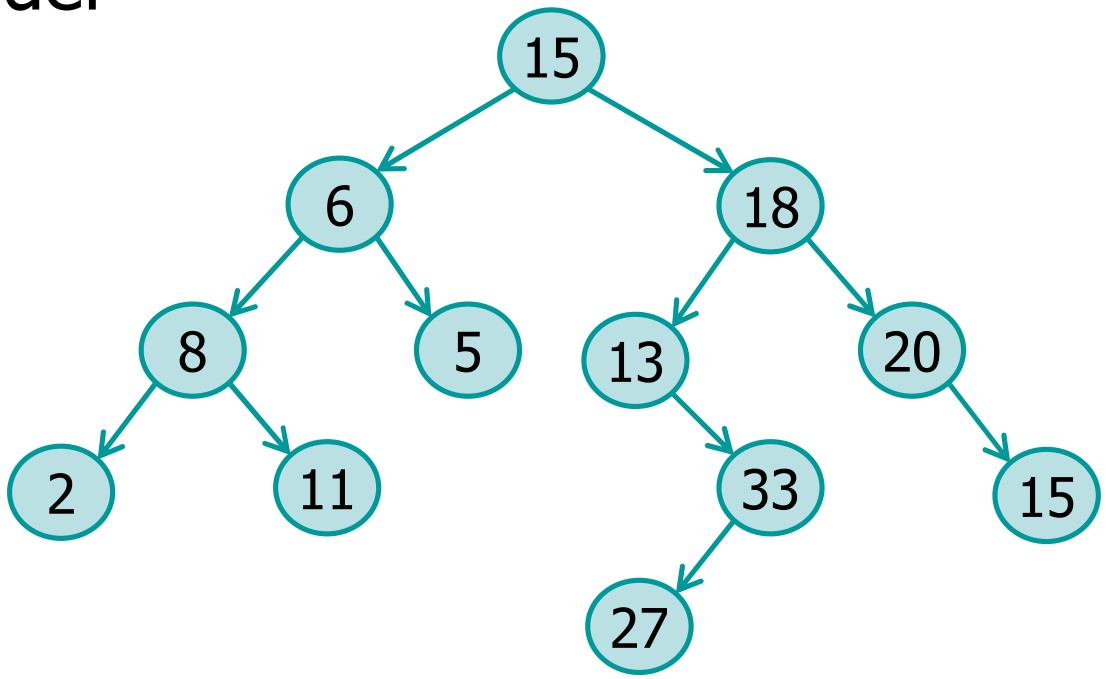


pre: print, left, right
...

pre: 15-6-8-2-11-5-18-13-33-27-20-15
in: 2-8-11-6-5-15-13-27-33-18-20-15
post: 2-11-8-5-6-27-33-13-15-20-18-15

Solution

- Given the following tree visit it in pre, in, and post-order



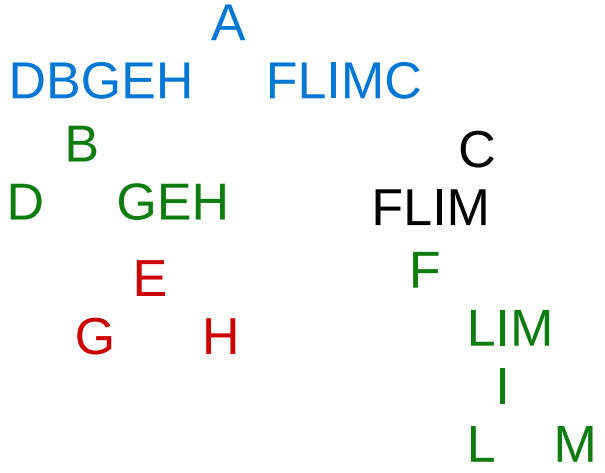
- Pre-order : 15 6 8 2 11 5 18 13 33 27 20 15
- In-order : 2 8 11 6 5 15 13 27 33 18 20 15
- Post-order: 2 11 8 5 6 27 33 13 15 20 18 15

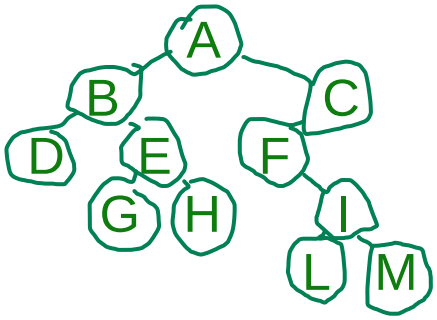
Exam: 29 January 2018

Exercise

- Consider a binary tree with 11 nodes
- Draw it considering that its pre, in and post-order visits return the following sequences
  - Pre-order: A B D E G H C F I L M
  - In-order: D B G E H A F L I M C
  - Post-order: D G H E B L M I F C A

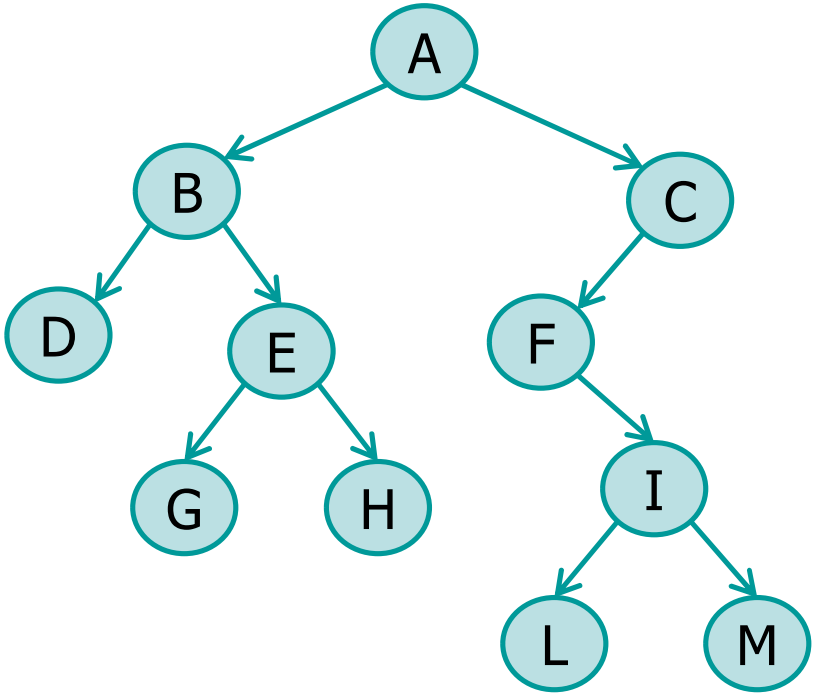
2 visits are enough
third is to check





Solution

- Pre-order: A B D E G H C F I L M
- In-order: D B G E H A F L I M C
- Post-order: D G H E B L M I F C A



Application: Expressions

passing from low to high level language

- Given an algebraic expression (brackets to change operator priority), it is possibile to build the corresponding tree according to the simplified grammar

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```

Termination Condition

Recursion

Example

- Using the following grammar

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```

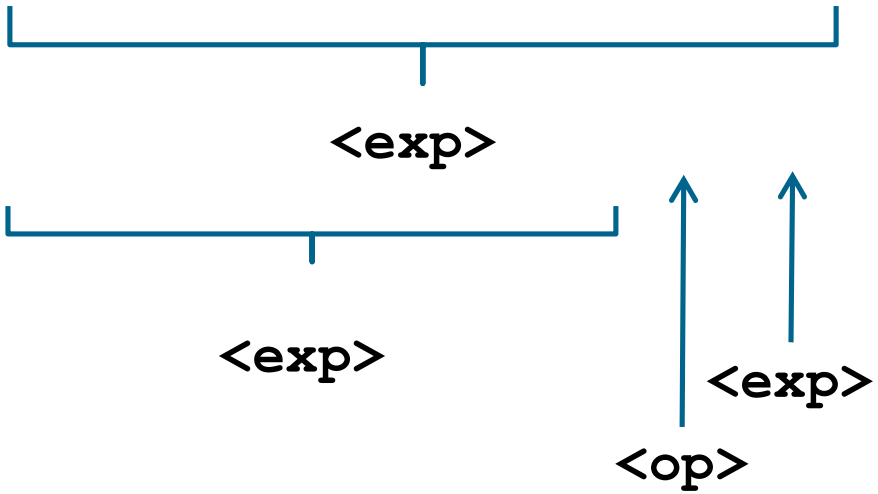
parse the following equation

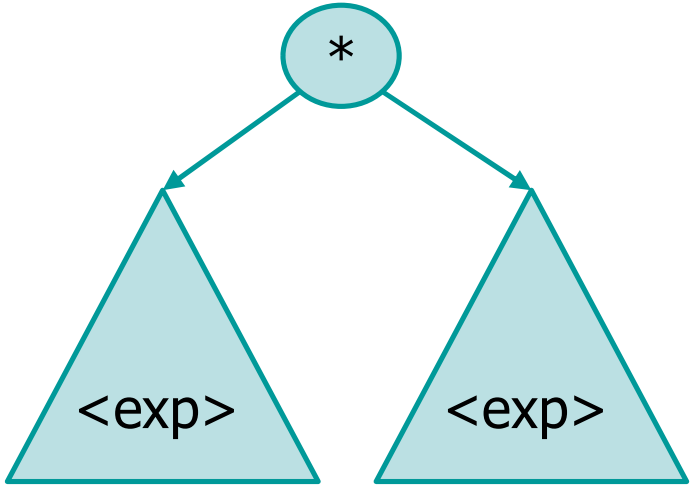
```
[(A+B)*(C-D)] * E
```

Solution: Step 1

[(A + B) * (C – D)] * E

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```



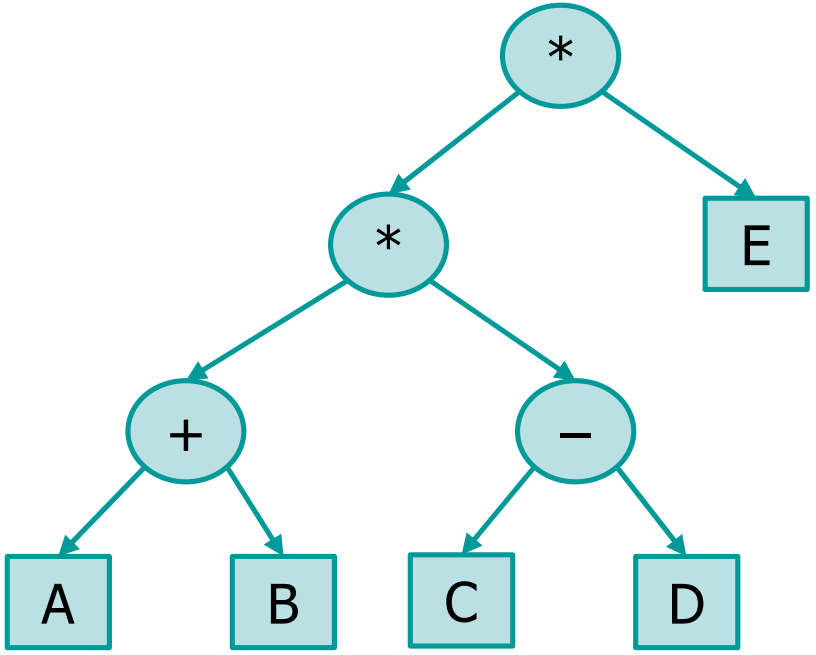


Solution

[(A + B) * (C – D)] * E

```
<exp> = <operand> | <exp> <op> <exp>
<operand> = A .. Z
<op>  = + | * | - | /
```

hence in order

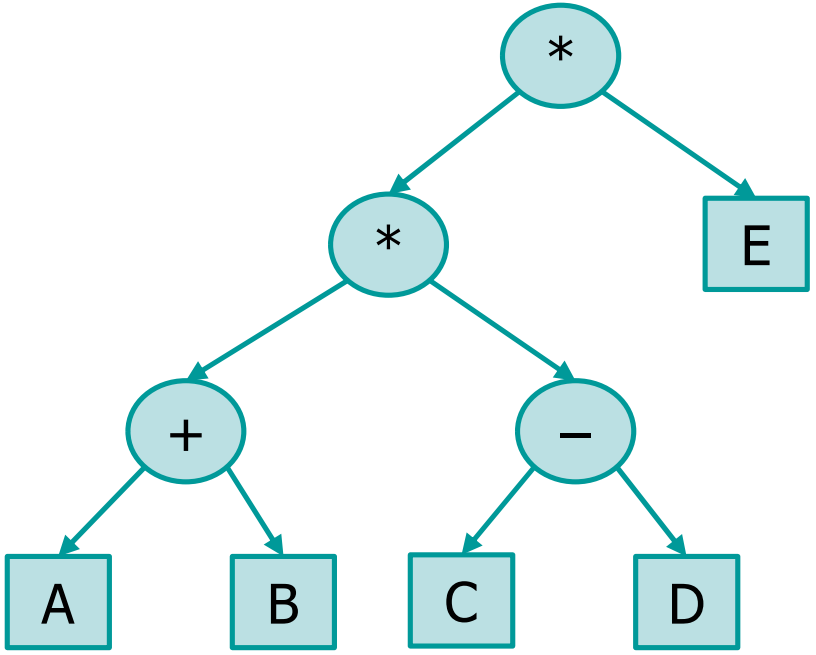


Example

[(A + B) * (C – D)] * E

- A pre-order visits returns the expression in the rarely used **prefix** form (**Polish Notation**)
  - * * + A B – C D E

Brakets are no more needed

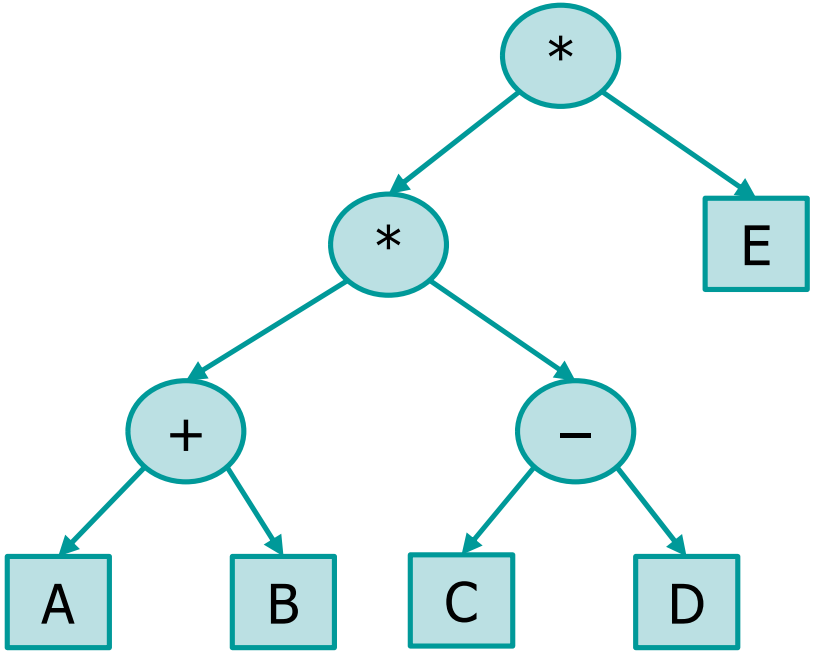


Example

[(A + B) * (C – D)] * E

- A post-order visits returns the expression in **postfix** form (**Reverse Polish Notation**)
  - A B + C D - * E *

Brakets are no more needed



Exam: 29 January 2018

Exercise

- Convert the following expressions from in-fix to pre-fix and post-fix notations

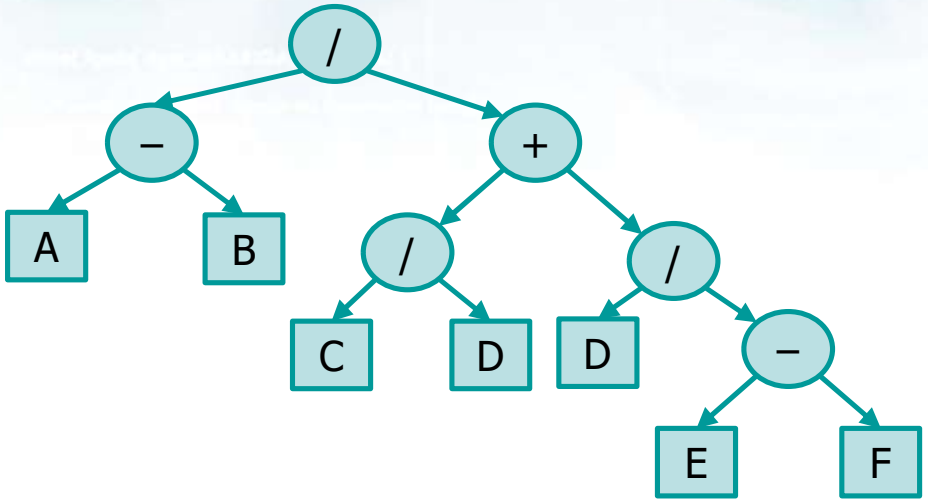
(A – B) / { (C / D) + [(D / (E – F)] }

(A – B) / { (C / D) + [(D / (E – F)) * G] }

solution: convert this into binary tree and visit it with the required technique

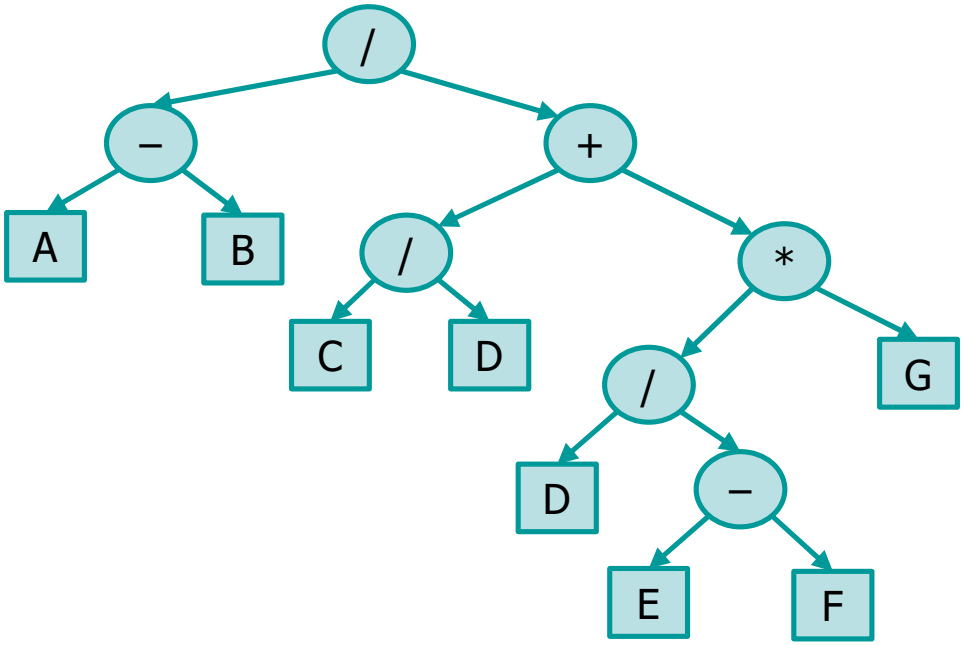
Exam: 29 January 2018

Solution



(A – B) / { (C / D) + [(D / (E – F)] }

Prefix: / – A B + / C D / D – E F
Postfix: A B – C D / D E F – / + /



(A – B) / { (C / D) + [(D / (E – F)) * G] }

Prefix: / – A B + / C D * / D – E F G
Postfix: A B – C D / D E F – / G * + /

A parser for the prefix form

EXTRA EXERCISES

- The following grammar specifies the prefix form (Polish notation)

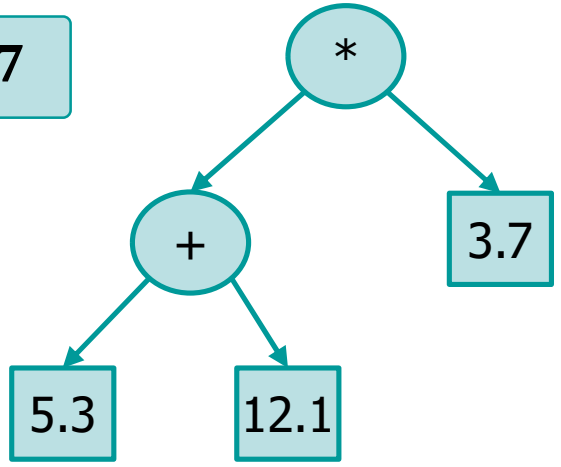
```
<exp> = <operand> | <op> <exp> <exp>
<operand> = float
<op>  = + | * | - | /
```

- Example

Input string (no parenthesis)

Output sequence

$(5.3 + 12.1) * 3.7$ ⇨ $* \; + \; 5.3 \;\; 12.1 \;\; 3.7$



- Write a recursive program to transform an in-fix equation into a prefix one

Implementation

```
int main(int argc, char *argv[]) {
  float result;
  int pos=0;

  if (argc < 2) {
    fprintf(stderr, "Error: missing parameter.\n");
    fprintf(stderr, "Run as: %s prefix_expression\n",
      argv[0]);
    return 1;
  }

  result = eval_r(argv[1], &pos);
  fprintf(stdout, "Result = %.2f\n", result);
  return EXIT_SUCCESS;
}
```

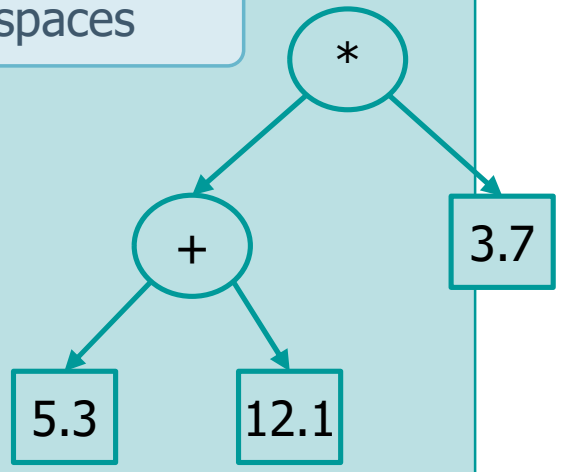
Implementation

Expression

```
float eval_r (char *expr, int *pos_ptr) {
  float left, right, result;
  char operator;
  int k = *pos_ptr;
  while (isspace(expr[k])) {
    k++;
  }
  if (expr[k]=='+' || expr[k]=='*' ||
      expr[k]=='-' || expr[k]=='/') {
    operator = expr[k++];
    left = eval_r(expr, &k);
    right = eval_r(expr, &k);
    switch (operator) {
      case '+': result = left+right; break;
      case '*': result = left*right; break;
      case '-': result = left-right; break;
      case '/': result = left/right; break;
    }
```

Parsing index

Skip spaces

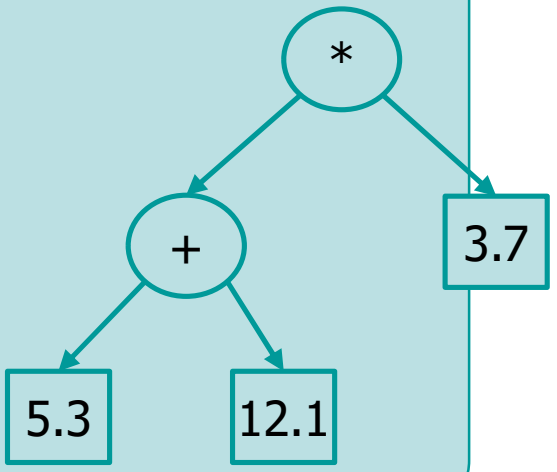


Implementation

```
  } else {
    sscanf(&expr[k], "%f", &result);
    while (isdigit(expr[k]) || expr[k]=='.') {
      k++;
    }
  }

  *pos_ptr = k;
  return result;
}
```

Terminal case:
A real value



A parser for the postfix form

- The following grammar specifies the postfix form (Reverse Polish notation)

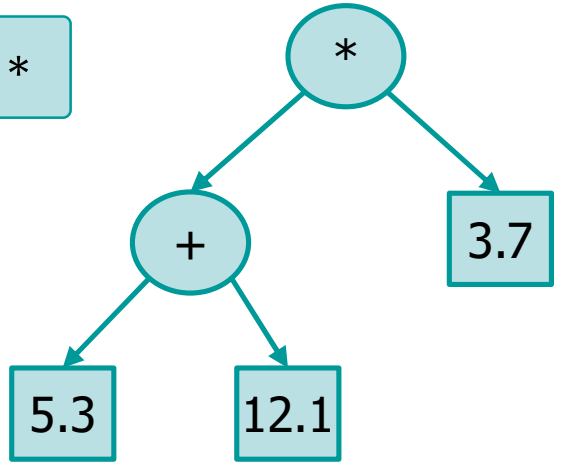
```
<exp> = <operand> | <exp> <exp> <op>
<operand> = float
<op>  = + | * | - | /
```

- Example

Input string (no parenthesis)

Output sequence

$(5.3 + 12.1) * 3.7 \Rightarrow 5.3 \ 12.1 \ + \ 3.7 \ *$



- Write a recursive program to transform an in-fix equation into a prefix one

Implementation

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "util.h"
#include "stackPublic.h"

int main(int argc, char *argv[]) {
  float result;
  int left, right, length, k=0;
  stack_t *sp=NULL;
  char *expr;

  util_check_m(argc>=2, "missing parameter.");
  expr = argv[1];
  length = strlen(expr);
  sp = stack_init(length);
```

ADT for utility functions

ADT for the stack

Implementation

```
while (k < length) {
  if (isdigit(expr[k])) {
    sscanf(&expr[k], "%f", &result);
    stack_push(sp, (void *)result);
    while (isdigit(expr[k]) || expr[k]=='.') {
      k++;
    }
} else if (expr[k]=='+' || expr[k]=='*' ||
    expr[k]=='-' || expr[k]=='/') {
    stack_pop(sp, (void **)&right);
    stack_pop(sp, (void **)&left);
    switch (expr[k]) {
      case '+': result = left+right; break;
      case '*': result = left*right; break;
      case '-': result = left-right; break;
      case '/': result = left/right; break;
    }
    stack_push(sp, (void *)result);
  }
  k++;
}
```

Skip float

Implementation

```
  stack_pop(sp, (void **)&result);
  fprintf(stdout, "Result = %ld\n", result);
  stack_dispose(sp, NULL);

  return EXIT_SUCCESS;
}
```