# F28HS2 Hardware-Software Interface

Lecture 5: Programming in C - 5

# Recursive structures

- cannot directly define recursive `structs`
- e.g. linked list

```
struct list{ int value;
             struct list next;};
struct list l;
```

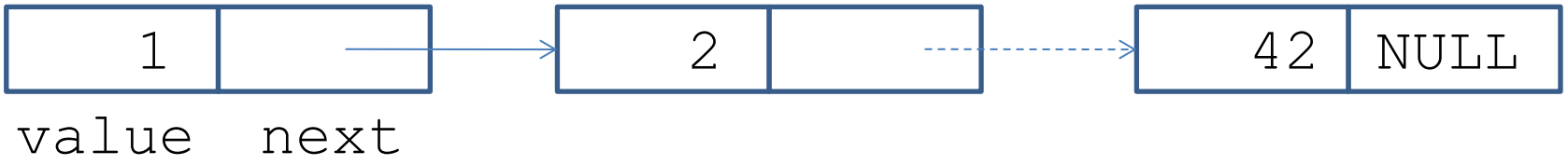
- `l` is allocated space for a `struct list`
  - space for an `int`
  - space for a `struct list`
    - space for an `int`
    - space for a `struct list`…

# Recursive structures

- define recursive structures with pointers
- end chains with `NULL`

e.g. linked list

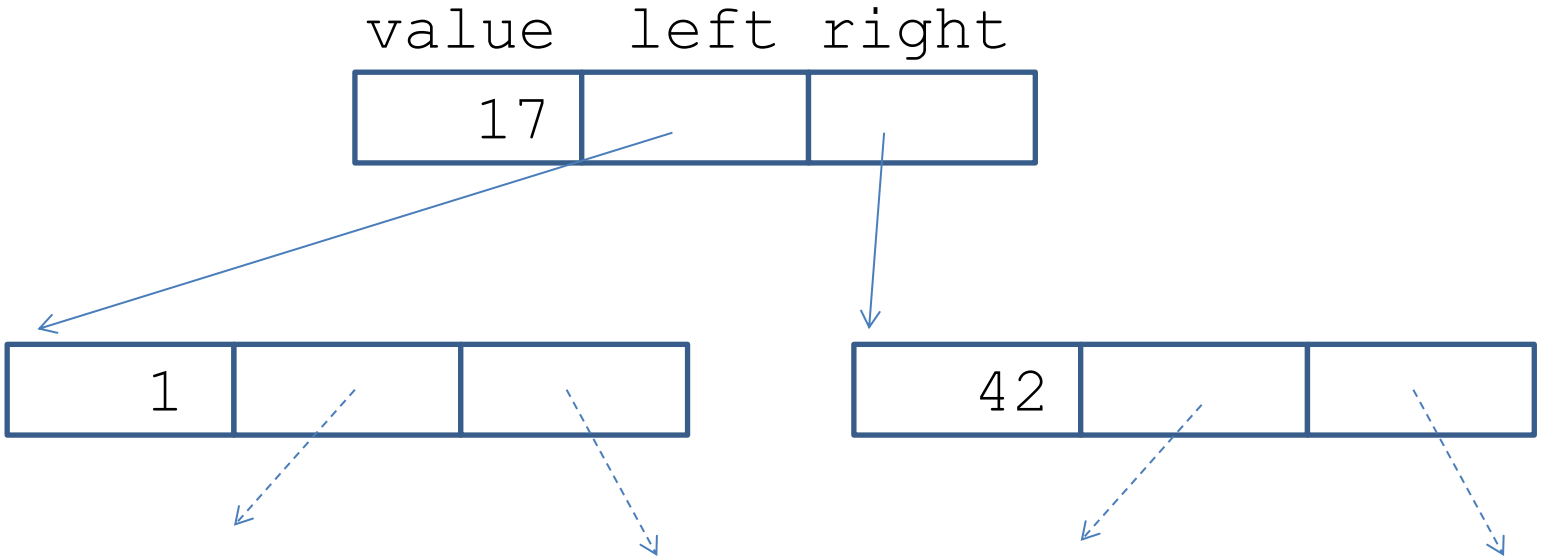
```
struct  list{int value;
            struct list * next;};
```



# Recursive structures

e.g. binary tree

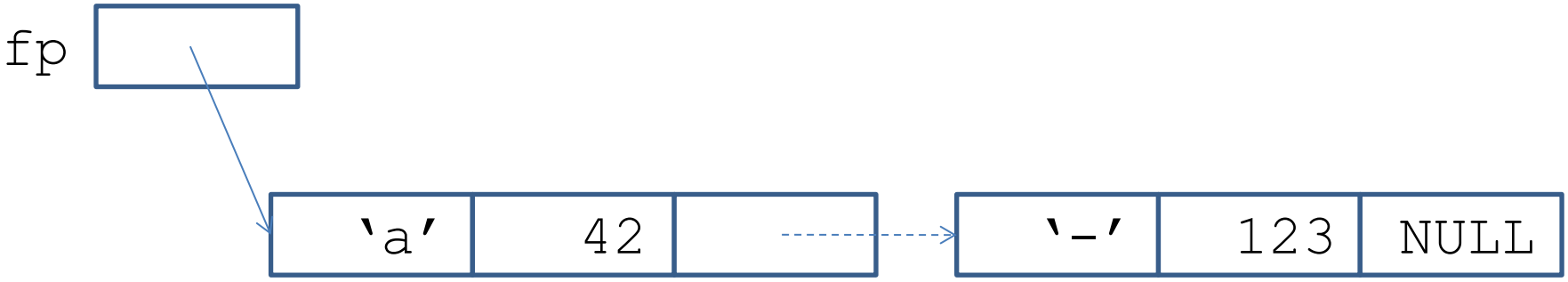
```
struct node {int value;
             struct node * left;
             struct node * right;};
```



# Example: character frequency 3

- allocating stack space for more pointers then needed
- replace stack with linked list

```
struct freq {int ch;int count;
             struct freq * next;};
struct freq * fp;
```



# Example: character frequency 3

- allocate first entry

```
incFreq(int ch)
{   struct freq * f;
    if(fp==NULL)
    {   fp = (struct freq *)
                    malloc(sizeof(struct freq));
        fp ->ch = ch;
        fp->count = 1;
        fp->next = NULL;
        return;
    }
```

# Example: character frequency 3

- if character found then increment count
- if not at end then try next entry

```
f = fp;
  while(1)
   if(f->ch==ch)
   {   f->count = f->count+1;
       return;
   }
   else
   if(f->next!=NULL)
    f = f->next;
```

# Example: character frequency 3

- at end so add new entry

```
    else
    { f->next = (struct freq *)
                  malloc(sizeof(struct freq));
      f->next->ch = ch;
      f->next->count = 1;
      return;
    }
}
```

# Example: character frequency 3

```
showFreq()
{   struct freq * f;
    f = fp;
    while(f!=NULL)
    {   printf("%c : %d\n, ",f->ch,f->count);
        f = f->next;
    }
    printf("\n");
}
```

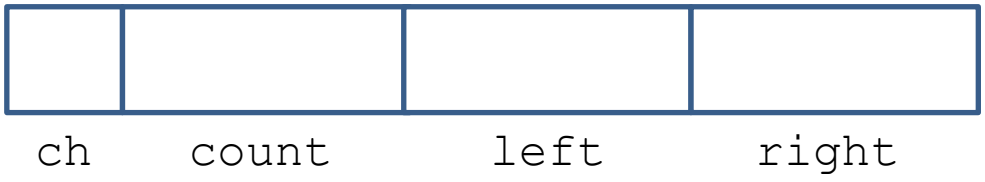
# Macro: text substitution

`#define` *name text*

- replace *name* with *text* throughout file
- *text* can include spaces
- very useful for `struct` types

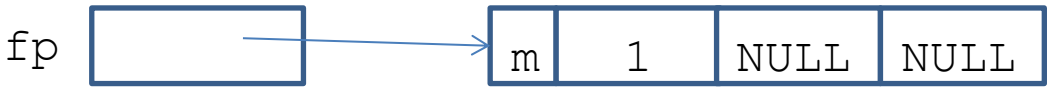
# Example: character frequency binary tree

```
#include <stdio.h>
#include <stdlib.h>

#define FREQ struct freq *

struct freq
  {int ch;int count;FREQ left;FREQ right;};

FREQ fp;
```

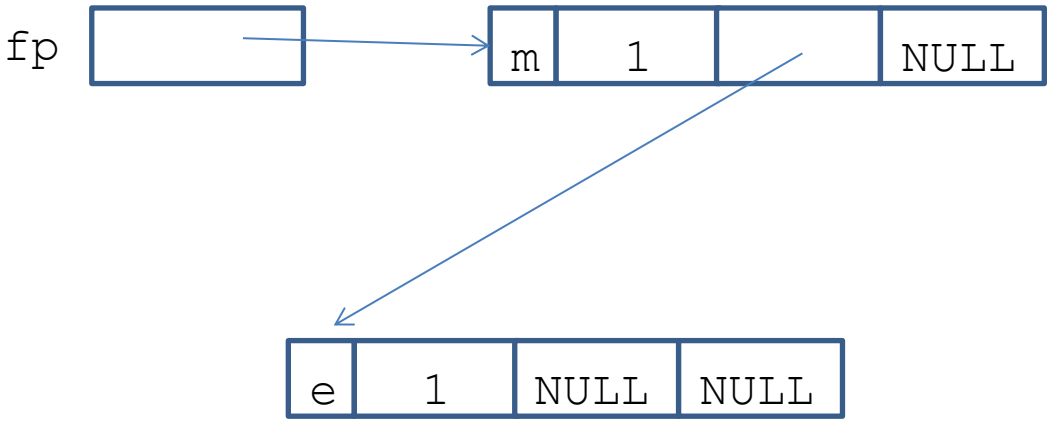


# Example: character frequency binary tree

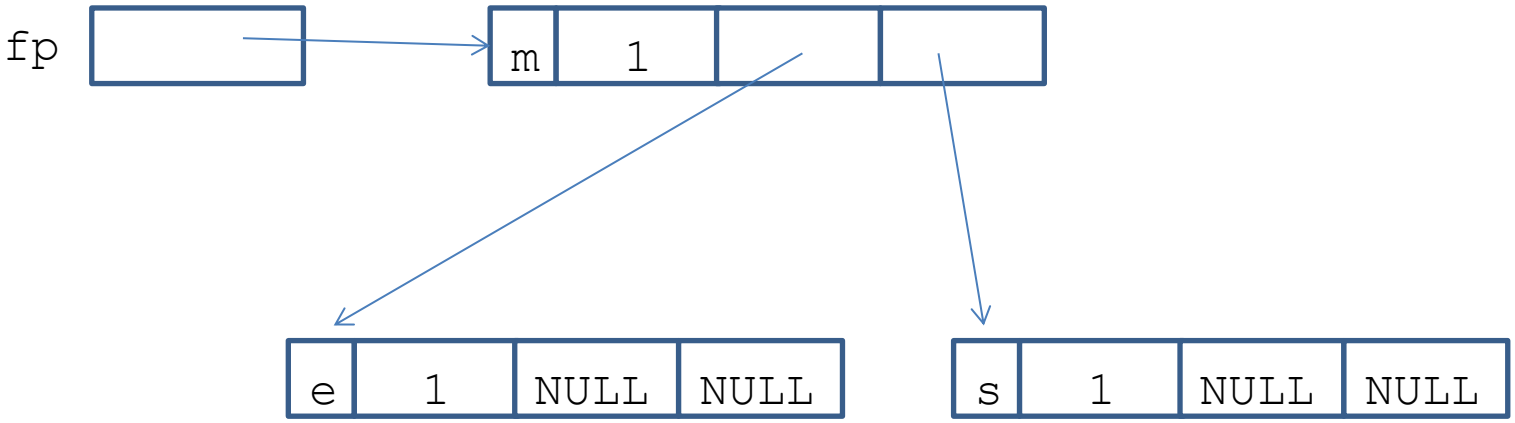
# Example: character frequency binary tree



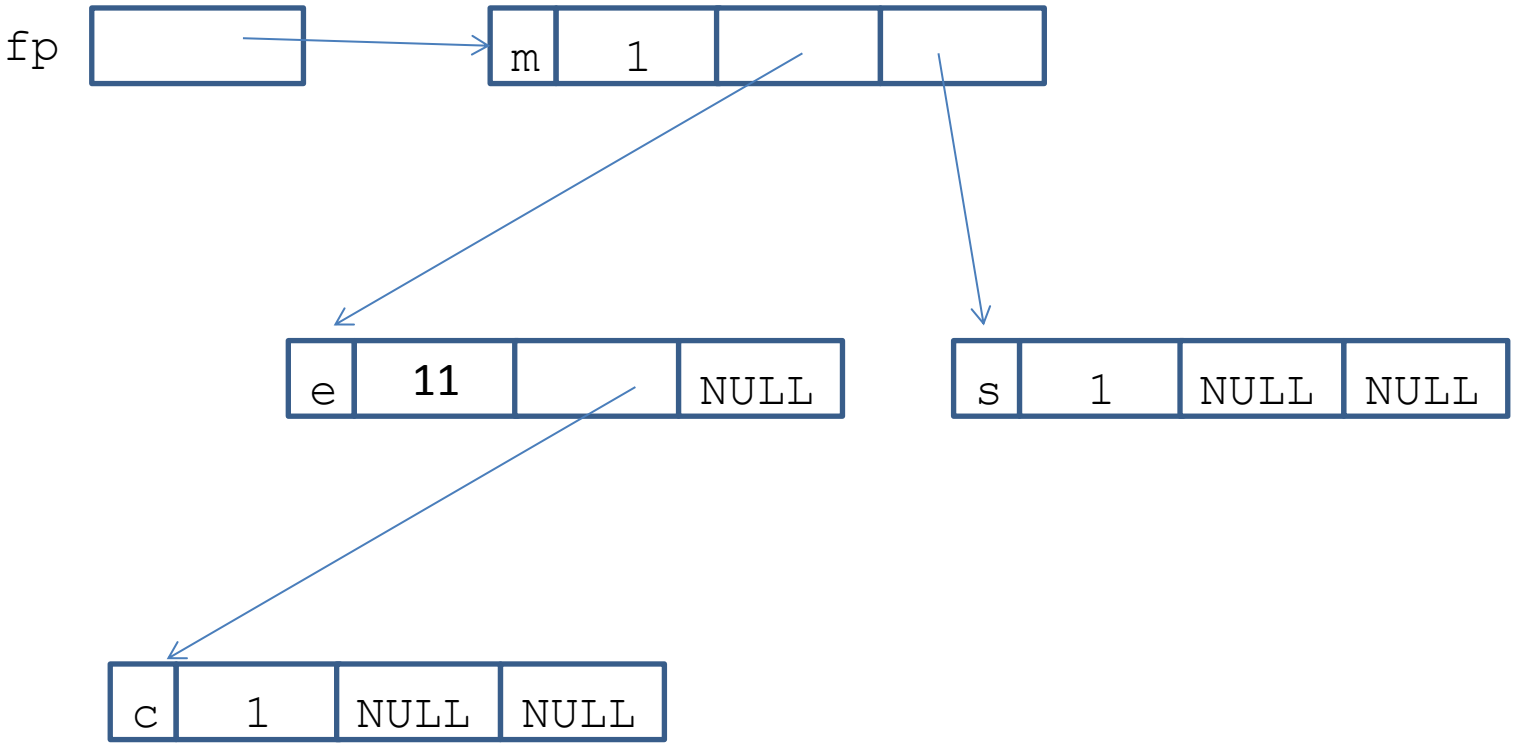
# Example: character frequency binary tree



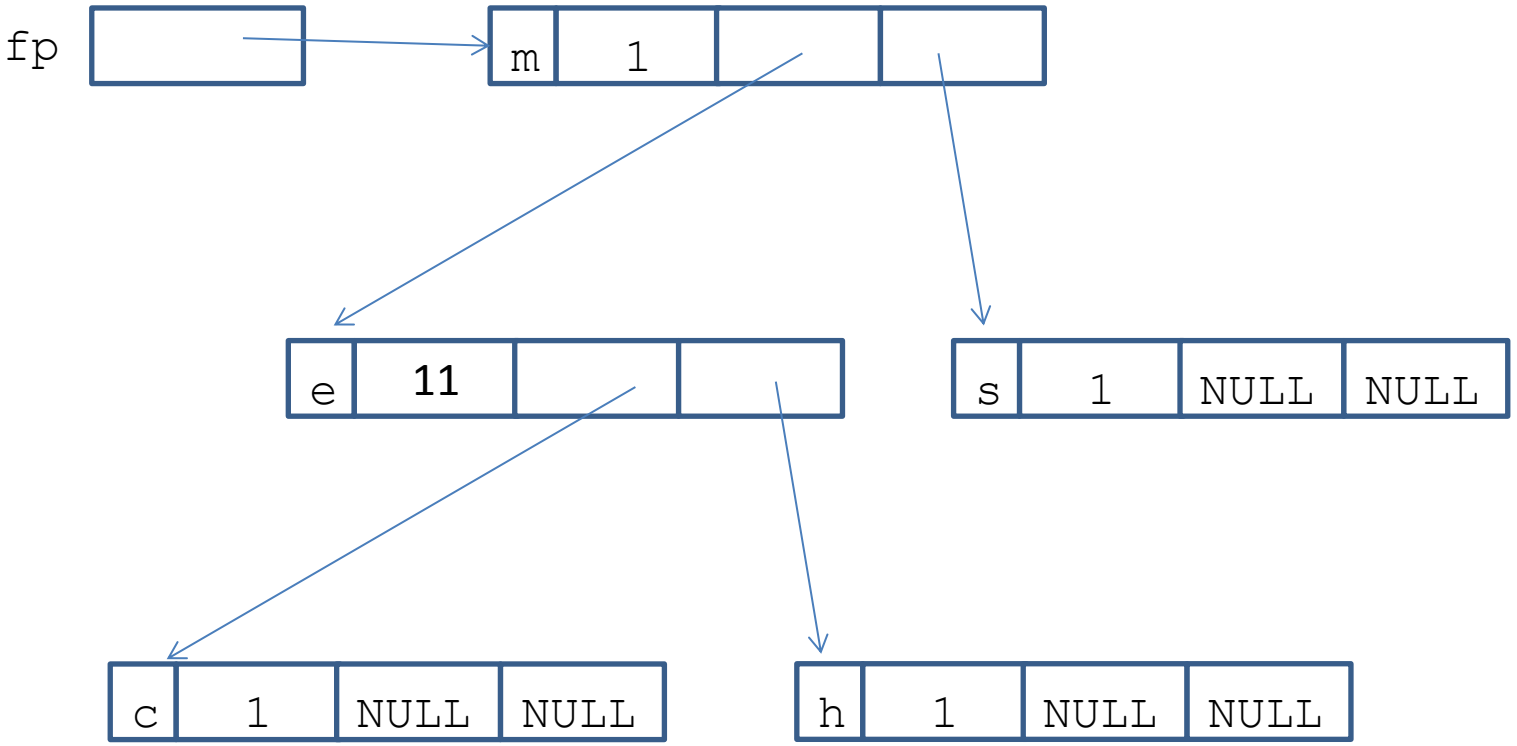
# Example: character frequency binary tree



# Example: character frequency binary tree



# Example: character frequency binary tree



# Example: character frequency binary tree

```
FREQ newFreq(char ch)
{   FREQ f;
    f = (struct freq *)
          malloc(sizeof(struct freq));
    f->ch = ch;
    f->count = 1;
    f->left = NULL;
    f->right = NULL;
    return f;
}
```

# Example: character frequency binary tree

- if root empty then add first node
- if found node then increment count

```
incFreq(int ch)
{   FREQ f;
    if(fp==NULL)
    {   fp = newFreq(ch); return; }
    f = fp;
    while(1)
     if(f->ch==ch)
     {   f->count = f->count+1; return; }
     else
```

# Example: character frequency binary tree

- if empty left then add new to left else try left branch
- if empty right then add new to right else try right branch

```
    if(f->ch>ch)
     if(f->left==NULL)
     {  f->left = newFreq(ch); return; }
     else f = f->left;
    else
    if(f->right==NULL)
    {  f->right = newFreq(ch); return; }
    else f = f->right;
}
```

# Example: character frequency binary tree

- print left branch
- print character & count
- print right branch

```
showFreq(FREQ f)
{   if(f==NULL)
     return;
    showFreq(f->left);
    printf("%c:%d\n",f->ch,f->count);
    showFreq(f->right);
}
```

# Example: character frequency binary tree

```
main(int argc,char ** argv)
{  int ch;
   FILE * fin;
   if((fin=fopen(argv[1],"r"))==NULL)
   {  printf("can't open %s\n",argv[1]); exit(0); }
   fp = NULL;
   ch = getc(fin);
   while(ch!=EOF)
   {  incFreq(ch); ch = getc(fin); }
   fclose(fin);
   showFreq(fp);
}
```

NB tree not balanced...

# Multi-dimensional arrays

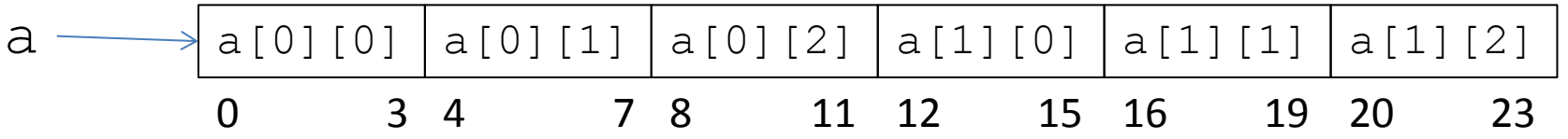
*type identifier*[$int_1$][$int_2$];

- allocates space for $int_1$ rows of $int_2$ columns * size for *type*
- e.g `int a[2][3];`

| | | |
|---|---|---|
| a[0][0] | a[0][1] | a[0][2] |
| a[1][0] | a[1][1] | a[1][2] |

# Multi-dimensional arrays

- allocated:
  - as $int_1$ * $int_2$ continuous locations for *type*
  - by row



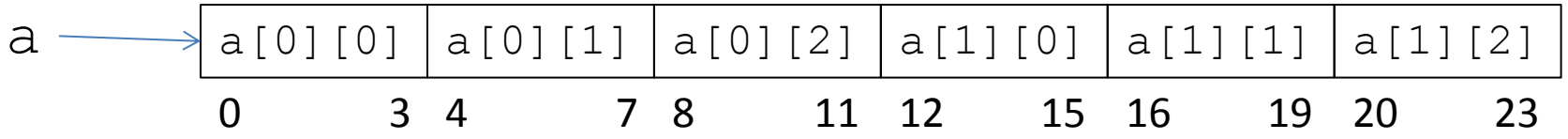
# Multi-dimensional access

- in expression

*exp* `[`*exp*$_1$`]` `[`*exp*$_2$`]` ➔

`*` (*exp*+(*exp*$_1$`*`*int*$_2$ + *exp*$_2$)`*`size for *type*)

- i.e. start at *exp* and skip *exp*$_1$ rows of *int*$_2$ columns and then skip *exp*$_2$ columns




- e.g. `a[1,1]` == a+(1*3+1)*4 == 16

# Example: Matrix multiplication

- integer matrices

- matrix held in file as:

```
matrix1: row 1/col 1 ...
matrix1: row 1/col n
...
matrix1: row m/col 1 ...
matrix1: row m/col n
```

# Example: Matrix multiplication

- use 2D array representation

```
#include <stdio.h>

#define MAXR 1000
#define MAXC 1000
```

- must specify maximum dimension sizes

# Example: Matrix multiplication

```
readMatrix(FILE * fin, int M[][MAXC],int m,int n)
{  int i,j;

   for(i=0;i<m;i++)
    for(j=0;j<n;j++)
     fscanf(fin,"%d",&(M[i][j]));
}
```

- must specify size of column for 2D array parameter
- `&(M[i][j])` for address of element i/j in `fscanf`

# Example: Matrix multiplication

```
writeMatrix(FILE * fout,int M[][MAXC],int m,int n)
{   int i,j;

    for(i=0;i<m;i++)
    {   for(j=0;j<n;j++)
         fprintf(fout,"%d ",M[i][j]);
       putc('\n',fout);
    }
}
```

# Example: Matrix multiplication

```
matrixProd(int M1[][MAXC],int M2[][MAXC],
           int M3[][MAXC],int m,int n)
{  int i,j,k;

   for(i=0;i<m;i++)
    for(j=0;j<m;j++)
    {  M3[i][j]=0;
       for(k=0;k<n;k++)
        M3[i][j] =
         M3[i][j]+M1[i][k]*M2[k][j];
    }
}
```

# Example: Matrix multiplication

- e.g. multiply m*n matrix by n*m matrix
- file holds:

```
m n
matrix 1: row 1/col 1 ... matrix1: row 1/col n
...
matrix 1: row m/col 1 ... matrix1: row m/col n
matrix 2: row 1/col 1 ... matrix2: row 1/col m
...
matrix 2: row n/col 1 ... matrix2: row n/col m
```

# Example: Matrix multiplication

```
main(int argc,chr ** argv)
{   FILE * fin;
    int m1[MAXR][MAXC];
    int m2[MAXR][MAXC];
    int m3[MAXR][MAXC];
    int m,n;

    fin = fopen(argv[1],
                "r");
    fscanf(fin,
           "%d %d",&m,&n);
    readMatrix(fin,m1,m,n);
    readMatrix(fin,m2,n,m);
    fclose(fin);

    writeMatrix(stdout,m1,m,n);
    putchar('\n');
    writeMatrix(stdout,m2,n,m);
    putchar('\n');
    matrixProd(m1,m2,m3,m,n);
    writeMatrix(stdout,m3,m,m);
}
```

# Example: Matrix multiplication

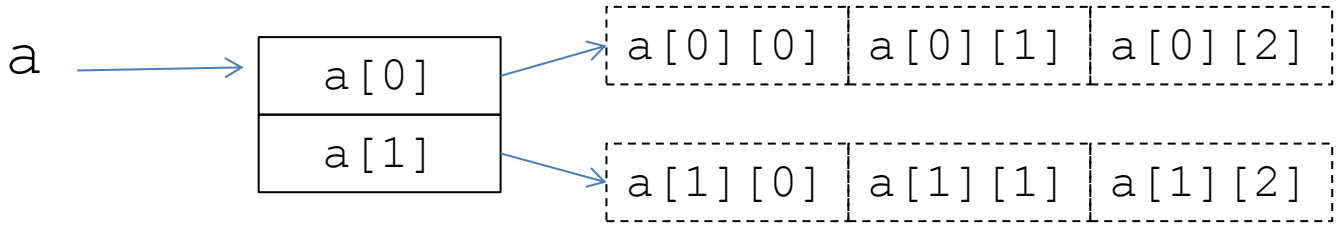
```
$ matrix1 m55.data
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25


1  0  0  0  0
0  1  0  0  0
0  0  1  0  0
0  0  0  1  0
0  0  0  0  1
```

```
1  2  3  4  5
6  7  8  9  10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

# Pointers and multi-dim arrays

*type* `*` *name* `[`*int*`]`

- allocates array of *int* pointers to type
- e.g. `int * a[2];`




- must use `malloc` to allocate 2nd dimension
- arrays in 2nd dimension can be any sizes
  - need not all be same size!

# Pointers and multi-dim arrays

*type* `**` *name*`;`

- allocates pointer to pointer to *type*
- must use `malloc` to allocate 1st and 2nd dimension
- 1st and 2nd dimension can be any size!

- can access all three forms as *identifier*`[`$exp_1$`][`$exp_2$`]`

# Example: Matrix Multiplication 2

- disadvantages of 2D array representation
- have to allocate worst case space
- use array of arrays

```
int * makeVector(int n)
{  return (int *)malloc(n*sizeof(int));  }
```

- returns array of `n ints`

# Example: Matrix Multiplication 2

```
int ** makeMatrix(int m,int n)
{   int ** newm =
     (int **)malloc(m*sizeof(int *));
    int i;
    for(i=0;i<m;i++)
     newm[i] = makeVector(n);
    return newm;
}
```

- returns array of `m` arrays of `n` `int`s

# Example: Matrix Multiplication 2

- in other functions, only need to change “matrix” type from `[][]` to `**`

```
readMatrix(FILE * fin, int ** M,int m,int n)
...
writeMatrix(FILE * fout,int ** M,int
...
matrixProd(int ** M1,int ** M2,
           int ** M3,int m,int n)
...
```

# Example: Matrix Multiplication 2

```
main(int argc,char ** argv)
char ** argv;
{   FILE * fin;
    int ** m1;
    int ** m2;
    int ** m3;
    ...
    m1 = makeMatrix(m,n);
    m2 = makeMatrix(n,m);
    m3 = makeMatrix(m,m);
    ... }
```

# Union type

- to construct pointers to different types

`union` *name* `{` $type_1$ $name_1$`;` `...`

$type_N$ $name_N$`;}`;

- definition
- `union` *name* is the name of a new type
- can take different types of value at different times
- can be:
  - $type_1$ accessed through $name_1$
  - $type_2$ accessed through $name_2$ etc

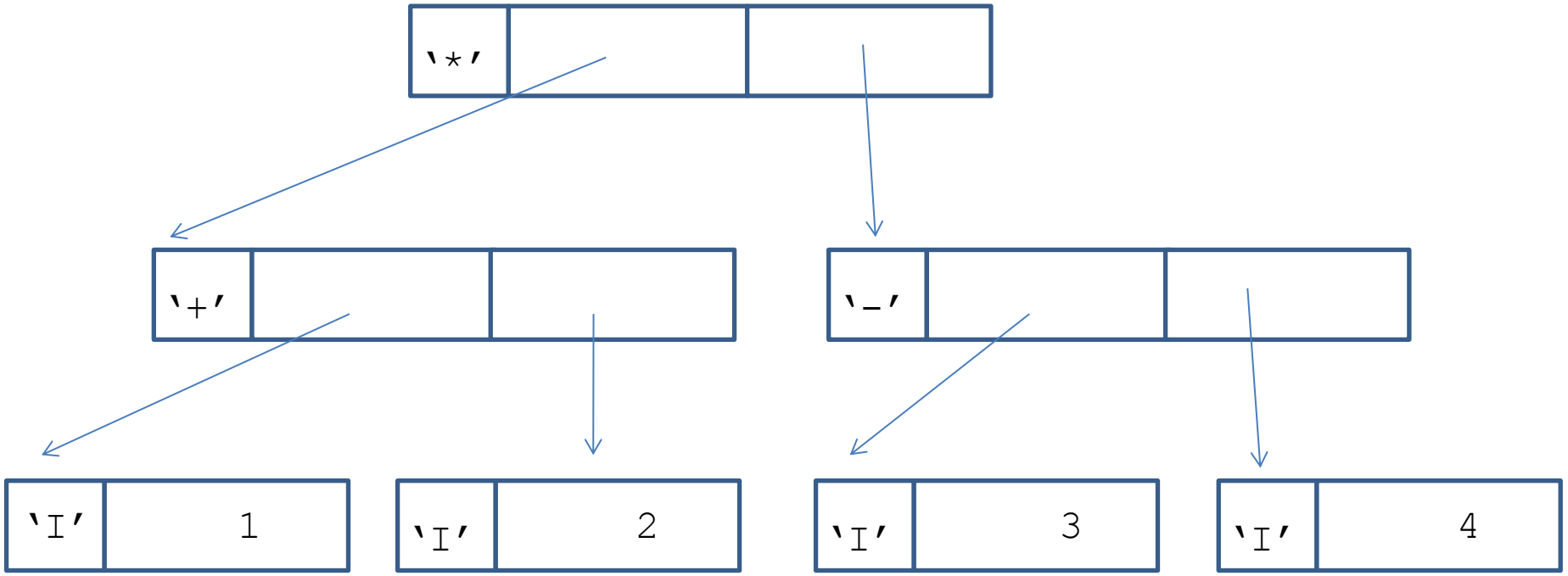
# Union type

`union` $name_1$ $name_2$`;`

- declaration
- $name_2$ is a variable of type `union` $name_1$
- space is allocated for the largest $type_l$
- other types are aligned within this space
- NB no way to tell at run time which type is intended
  - use an explicit tag

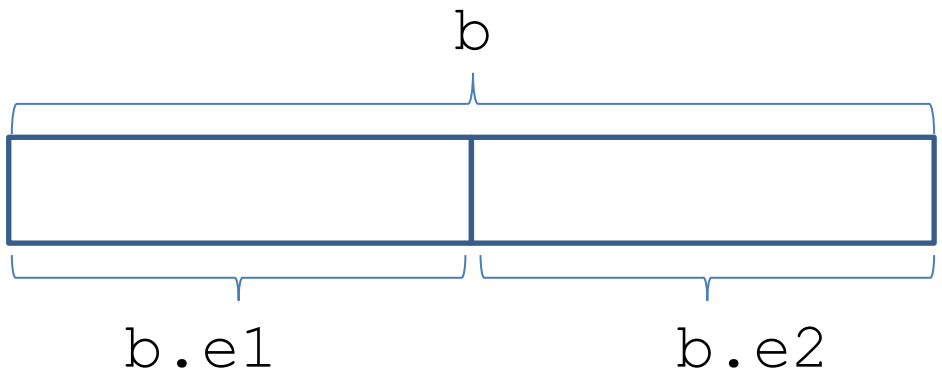
# Example: syntax tree

- e.g. `(1+2)*(3-4)`



# Example: syntax tree

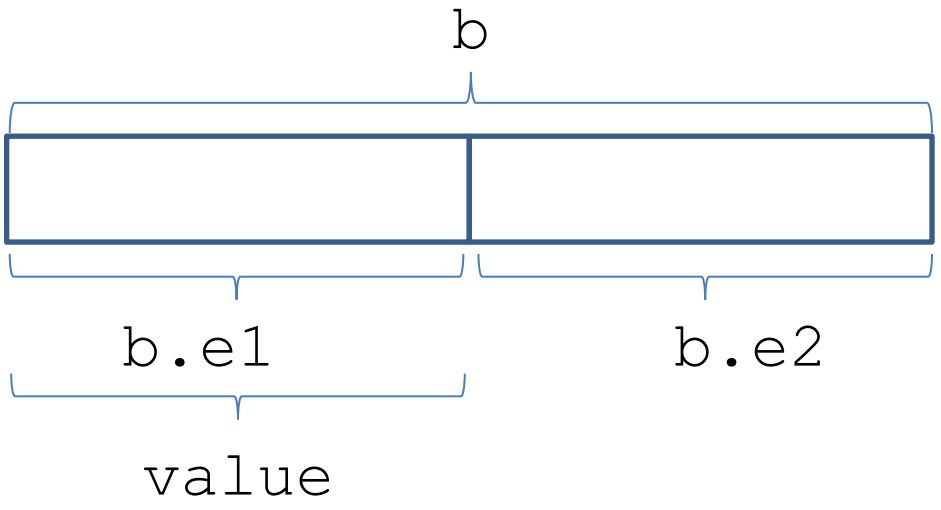
```
struct binop{struct exp * e1,
             struct exp * e2;};
```



# Example: syntax tree

```
union node{ int value; struct binop b; } ;
```

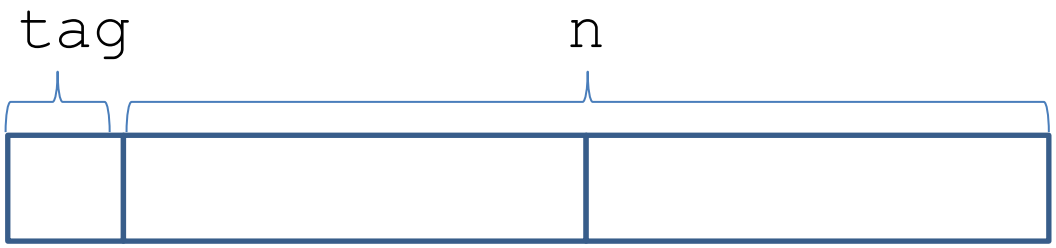
- `node` is:
  - leaf `int value`, or…
  - branch `struct binop b`
- `value` and `b.e1` are same 4 bytes



# Example: syntax tree

- can’t tell what arbitrary `union` represents
- use an explicit tag field

```
struct exp {char tag; union node n;};
```

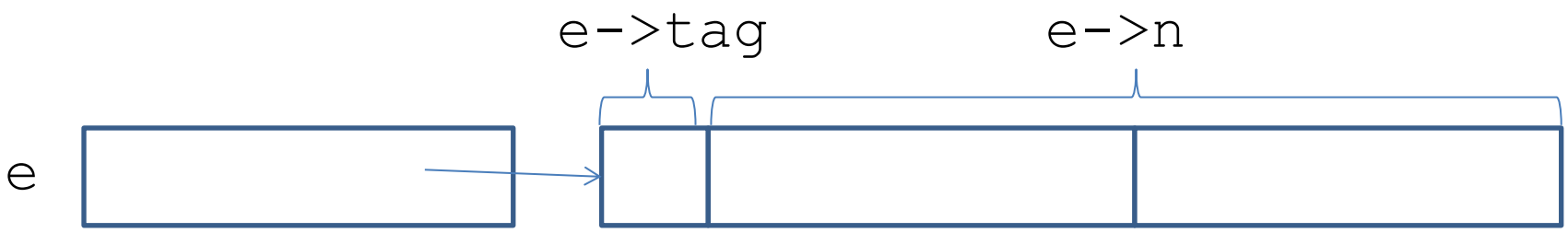


# Example: syntax tree

- `tag == 'I'` ==> leaf for integer
- `tag == 'op'` ==> branch for operator
- NB this is our convention
  - no necessary connection between `tag` and `node` contents

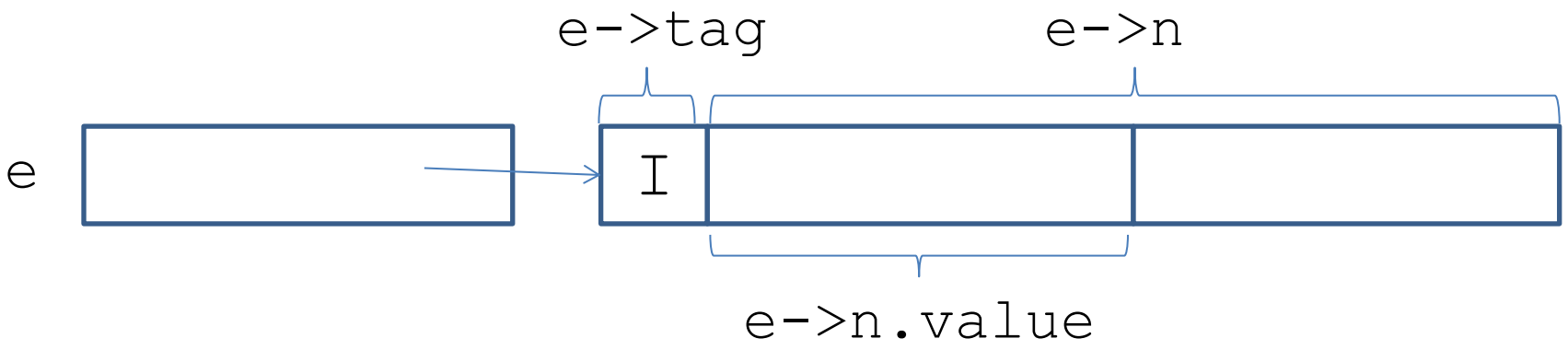
# Example: syntax tree

```
struct * exp e;
e = (struct * exp)
      malloc(sizeof(struct exp));
```



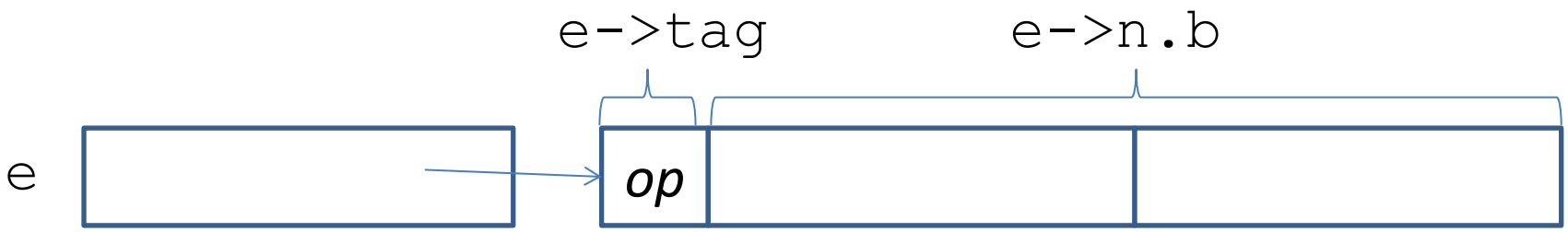
# Example: syntax tree

```
struct * exp e;
e = (struct * exp)
      malloc(sizeof(struct exp));
```



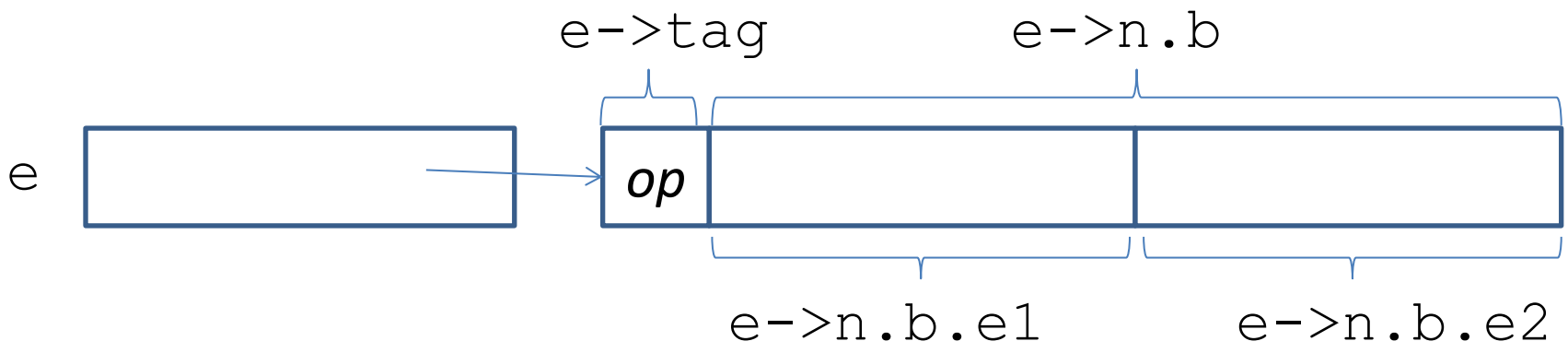
# Example: syntax tree

```
struct * exp e;
e = (struct * exp)
      malloc(sizeof(struct exp));
```



# Example: syntax tree

```
struct * exp e;
e = (struct * exp)
      malloc(sizeof(struct exp));
```



# Example: syntax tree

```
struct exp * newInt(int i)
{   struct exp * e;
    e = (struct exp *)
          malloc(sizeof(struct exp));
    e->tag = 'I';
    e->n.value = i;
    return e;
}
```

# Example: syntax tree

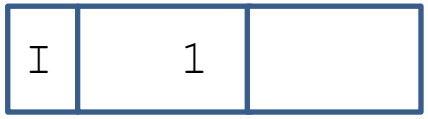
```
struct exp * newBinop
             (char op,struct exp * e1,
                      struct exp * e2)
{   struct exp * e;
    e = (struct exp *)
          malloc(sizeof(struct exp));
    e->tag = op;
    e->n.b.e1 = e1;
    e->n.b.e2 = e2;
    return e;
}
```

# Example: syntax tree

```
main(int argc,char ** argv)
{  struct exp * e;
   e = newBinop
        ('*',newBinop
             ('+', newInt(1), newInt(2)),
           newBinop
             ('-', newInt(3), newInt(4)));
   printTree(e,0);
}
```

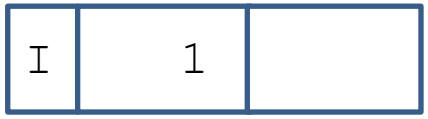
# Example: syntax tree

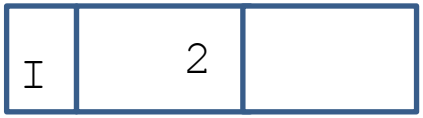
```
newBinop
  ('*',
   newBinop
      ('+',
       newInt(1),
       newInt(2)),
   newBinop
      ('-',
       newInt(3),
       newInt(4)));
```



# Example: syntax tree

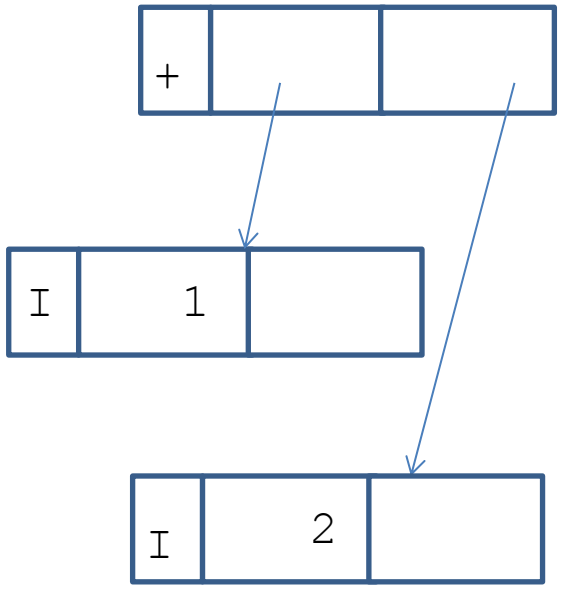
```
newBinop
  ('*',
   newBinop
      ('+',
       newInt(1),
       newInt(2)),
   newBinop
      ('-',
       newInt(3),
       newInt(4)));
```



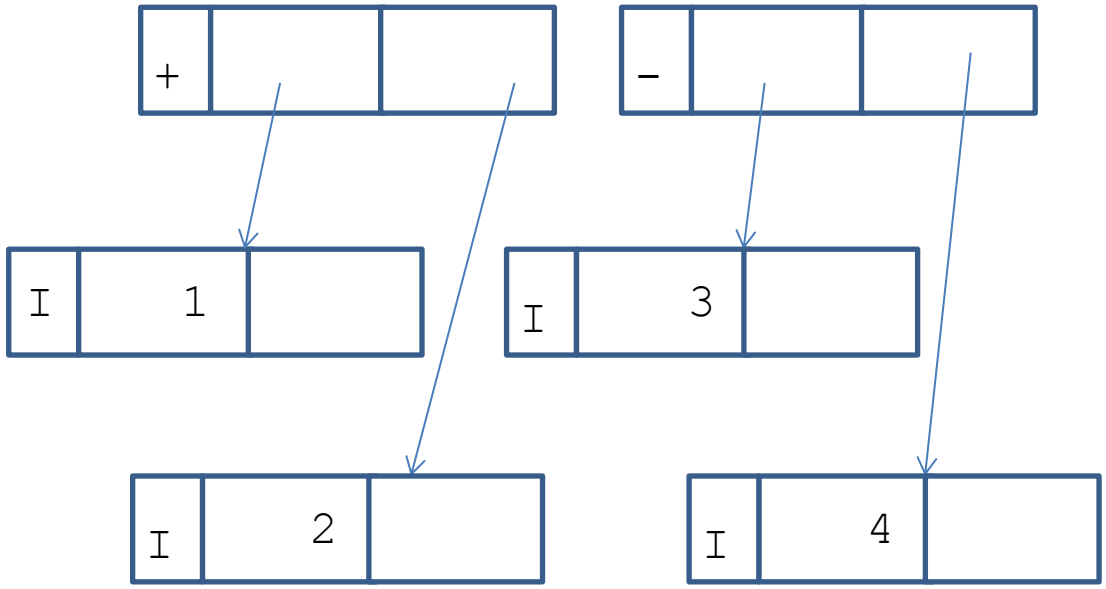
# Example: syntax tree

```
newBinop
  ('*',
   newBinop
      ('+',
       newInt(1),
       newInt(2)),
   newBinop
      ('-',
       newInt(3),
       newInt(4)));
```



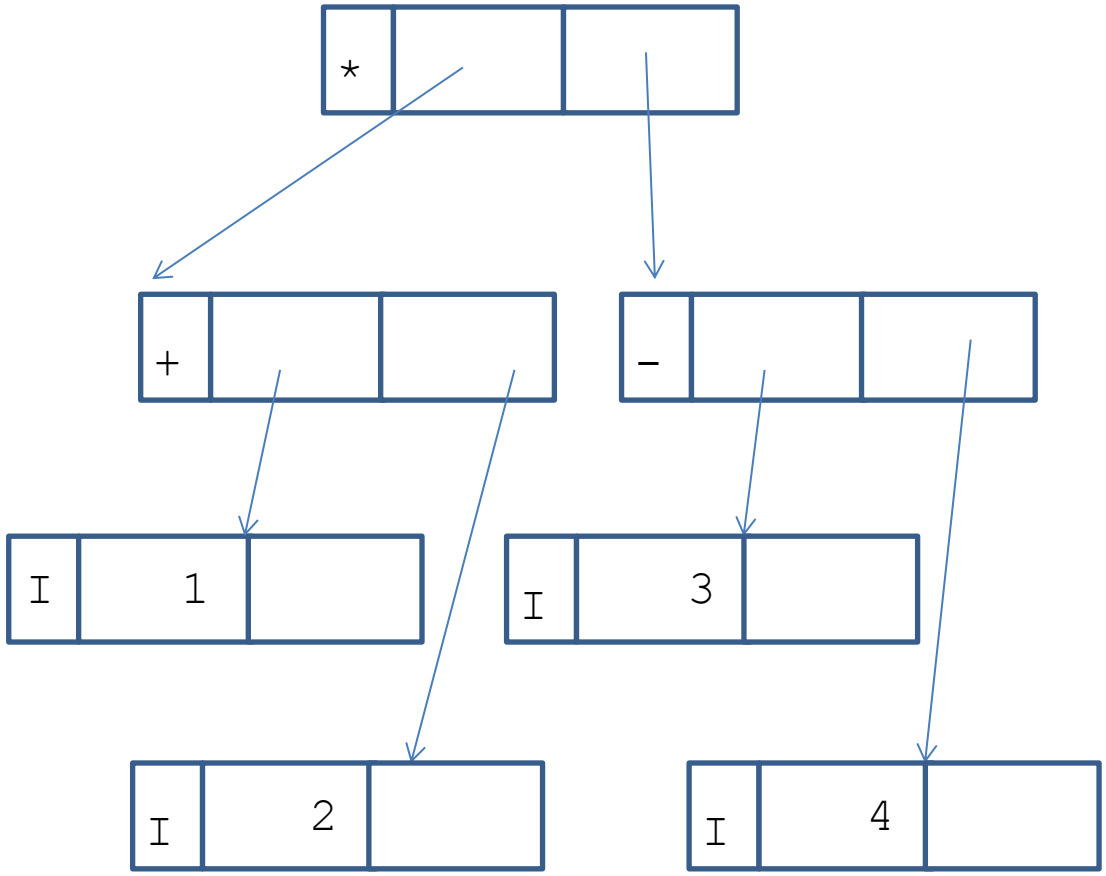
# Example: syntax tree

```
newBinop
  ('*',
   newBinop
      ('+',
       newInt(1),
       newInt(2)),
   newBinop
      ('-',
       newInt(3),
       newInt(4)));
```



# Example: syntax tree

```
newBinop
  ('*',
   newBinop
      ('+',
       newInt(1),
       newInt(2)),
   newBinop
      ('-',
       newInt(3),
       newInt(4)));
```



# Example: syntax tree

- display tree left to right
- like a folder tree
- keep track of node *depth*
- at each node
  - *depth* spaces
  - print tag
  - print left node at *depth*+1
  - print right node at *depth*+1

# Example: syntax tree

```
printTree(struct exp * e, int depth)
{  int i;
   for(i=0;i<depth;i++)printf(" ");
   switch(e->tag)
   {  case 'I': printf("%d\n",e->n.value);
                return;
      default: printf("%c\n",e->tag);
               printTree(e->n.b.e1,depth+1);
               printTree(e->n.b.e2,depth+1);
   }
}
```

# Example: syntax tree

```
$ exp
*
  +
    1
    2
  -
    3
    4
```