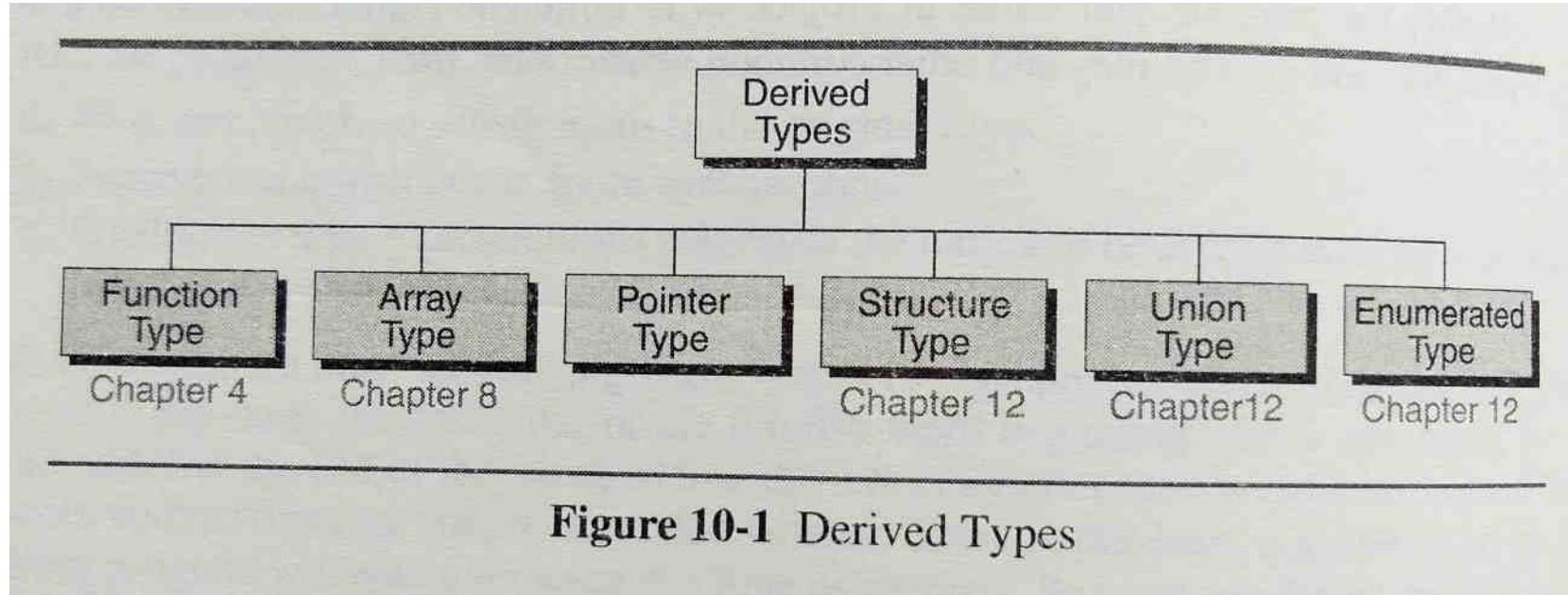


Pointers

Pointers



Introduction

- Pointer constants
- Pointer variables
- Accessing variables through pointers
- Pointer declaration and definition
- Initialization of pointer variables
- Pointers and functions
- Pointers to pointers

Pointers - Concepts

- Every computer has addressable memory locations
- Data Manipulation
- 1) Indirect Approach: Identifiers
- We use memory location addresses symbolically
 - We assign **identifiers** to data and then manipulate their contents through the identifiers
- 2) Direct Approach: Pointers
- Uses data addresses directly as well with ease and flexibility of symbolic names

Pointers

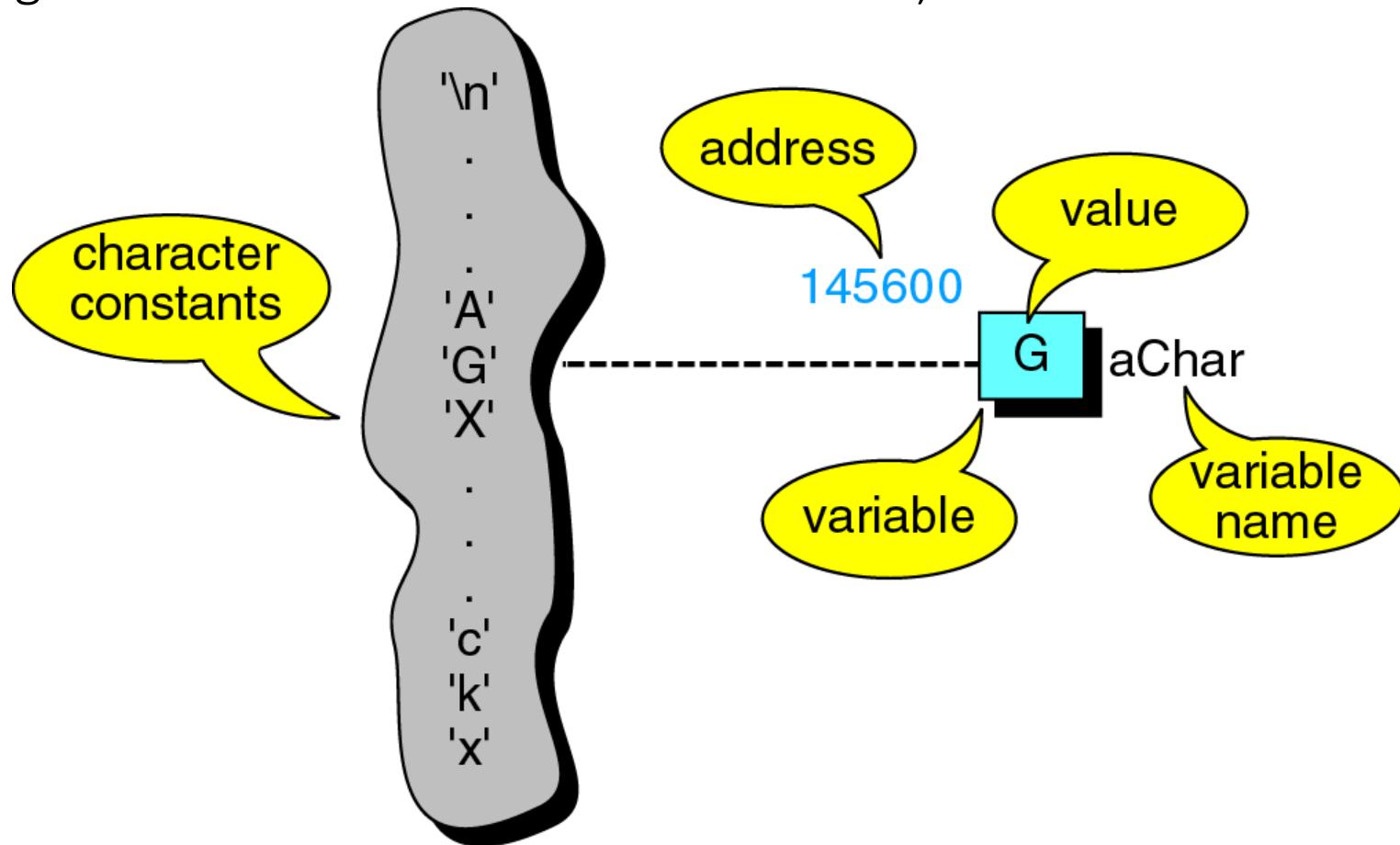
- A pointer is a derived data type : a data type built from one of the standard types
- Its value is any of the addresses available in the computer for storing and accessing data

Pointer Constants

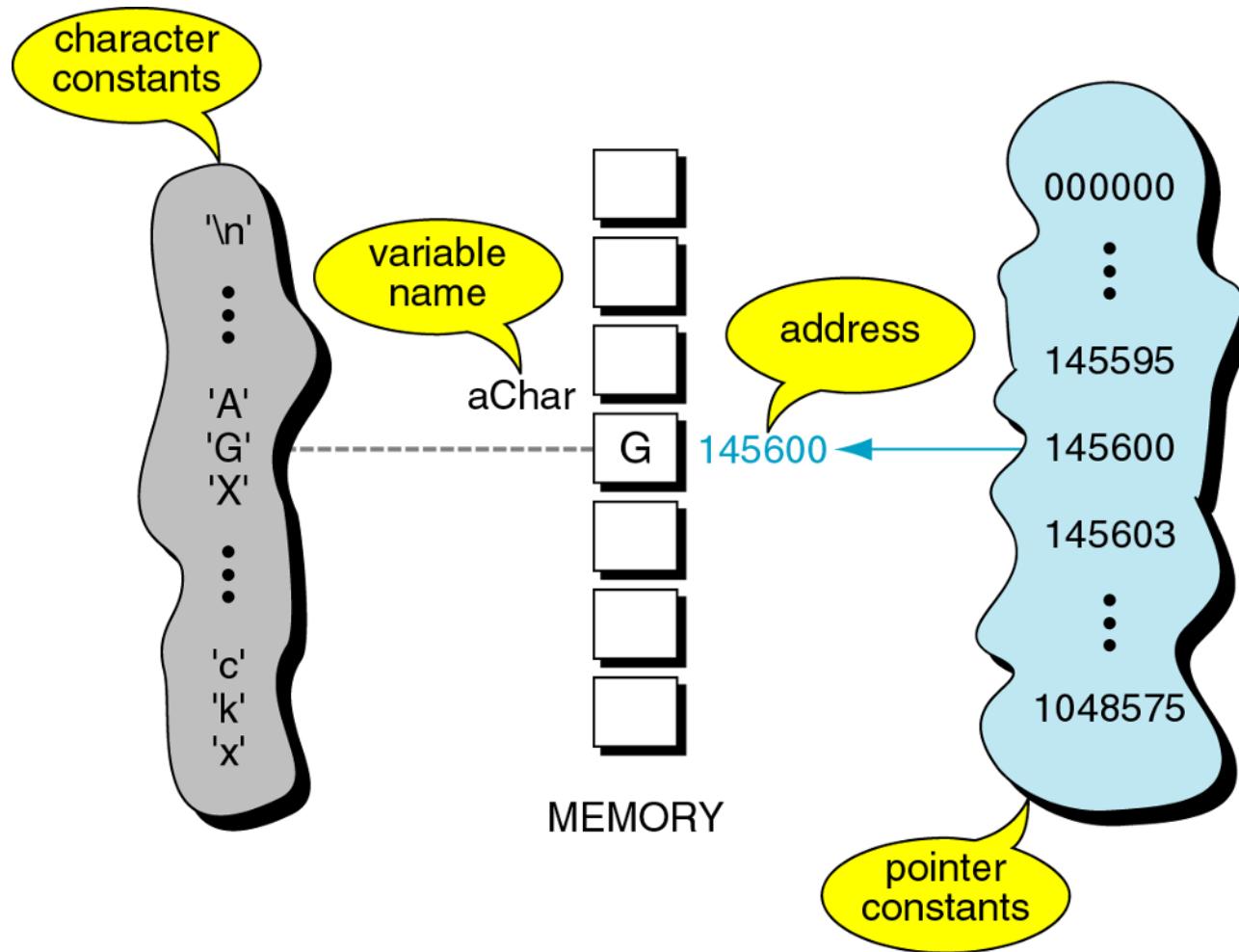
- To understand pointers, we begin with **pointer constants**
- First, compare **character constants and pointer constants**
- Character constant – we can have character constants from a universe of all characters
 - For most computers, it is known as **ASCII**
- A character constant can become a value and can be stored in a variable
- `char aChar = 'G';`

Pointer constants

(fig 10.2 Character constants & variables)



Pointer constants (fig 10.3)



Character Constants and Pointer Constants

- Like character constants, pointer constants cannot be changed
- The address for variable aChar is drawn from the set of pointer constants for our computer
- Although addresses within a computer cannot change (remains constant for the duration of the run)
 - but the address of a variable will change for each run of the program
 - Thus it is necessary to refer pointer variables symbolically

Pointer values

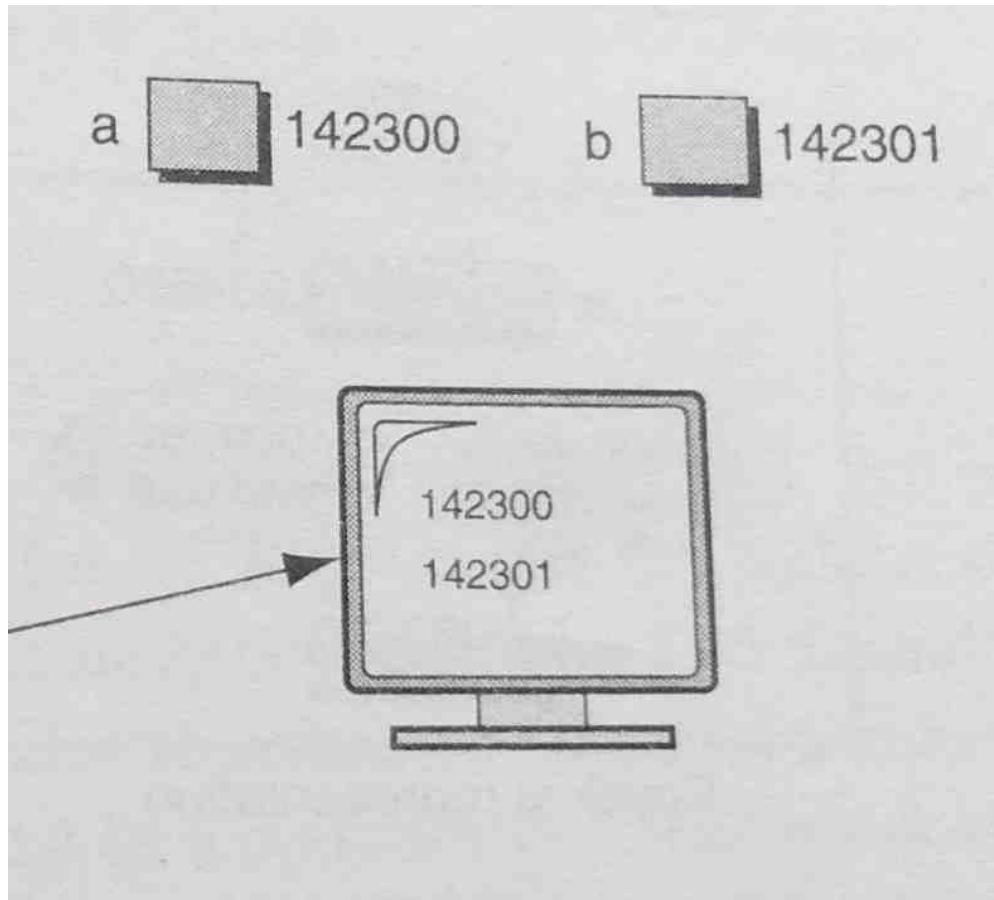
Pointer Constants

- Pointer constants
- an address in memory
- Drawn from the set of addresses for a computer
- Exist by themselves
- Cannot change them; only use them
- How to save this address?
- We have already done it by scanf with address operator &
- WAP to print addresses as pointers

Pointer values (Fig 10.4 – Print Character Addresses)

```
int main (void)
{
    // Local Declarations
    char a;
    char b;
    // Statements
    printf ("%p\n %p\n", &a, &b);
    return 0;
} // main
```

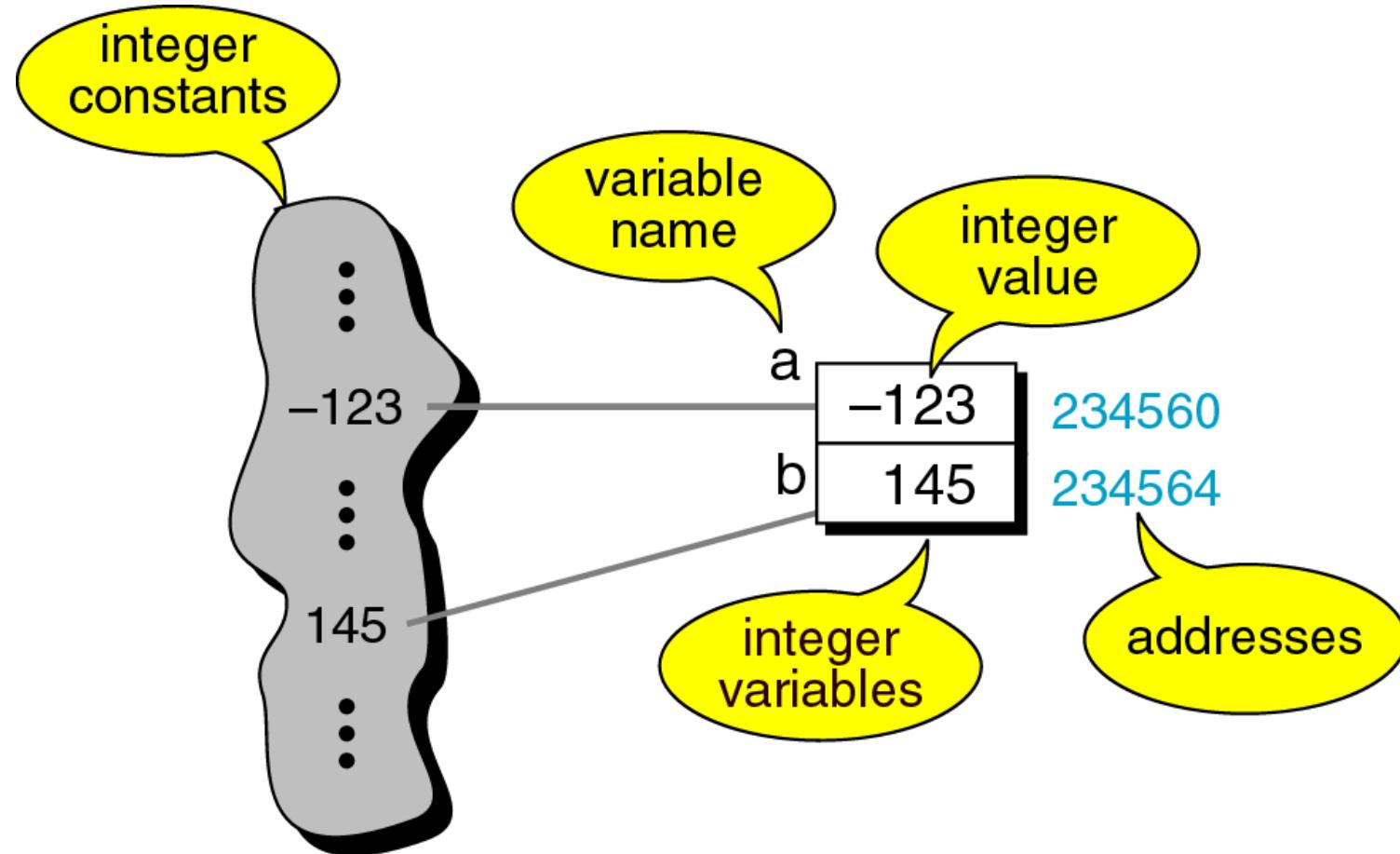
Pointer values (Fig 10.4 contd)



Integer Constants and variables

- In most computers integers occupy either 2 or 4 bytes
- Assume we are working on a system with 4-byte integers
- This means that each integer occupies 4 memory locations
- Which one is then used to **find the address of the variable?**
 - The address of a variable is the address of the first byte occupied by that variable
 - For characters, there is only one byte, so its location is the address
 - For integers, the address is the first byte of 4

Pointer values (fig 10.5 Integer constants & variables)



Pointer variables

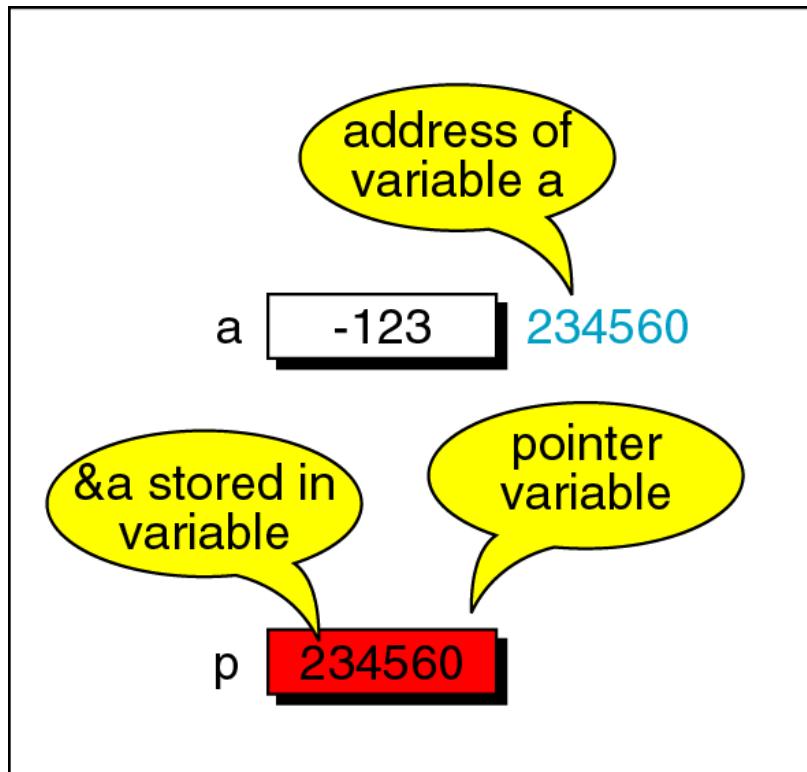
Pointer variables

- we have a pointer constant and a pointer value
- So, we can have a **pointer variable**
- To store the address of a variable into another variable which is called a pointer variable

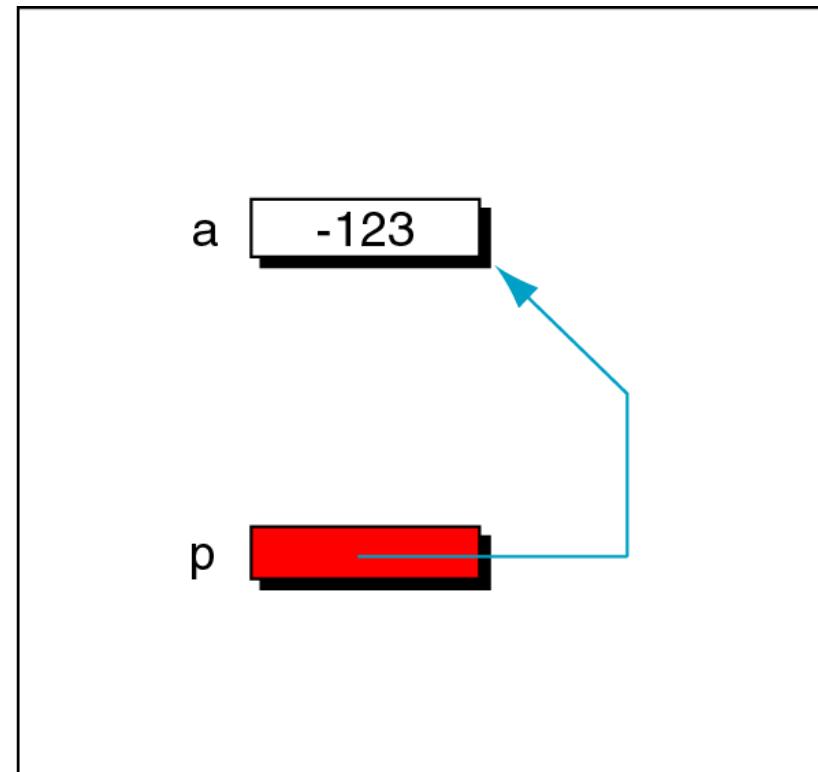
Pointer variables

- Distinguish between a pointer variable and its value
- There is an integer variable whose name and location are constant, the value may change as the program executes
- There is also a pointer which has a name and a location, both of which are constants

Pointer variables (fig 10.6)

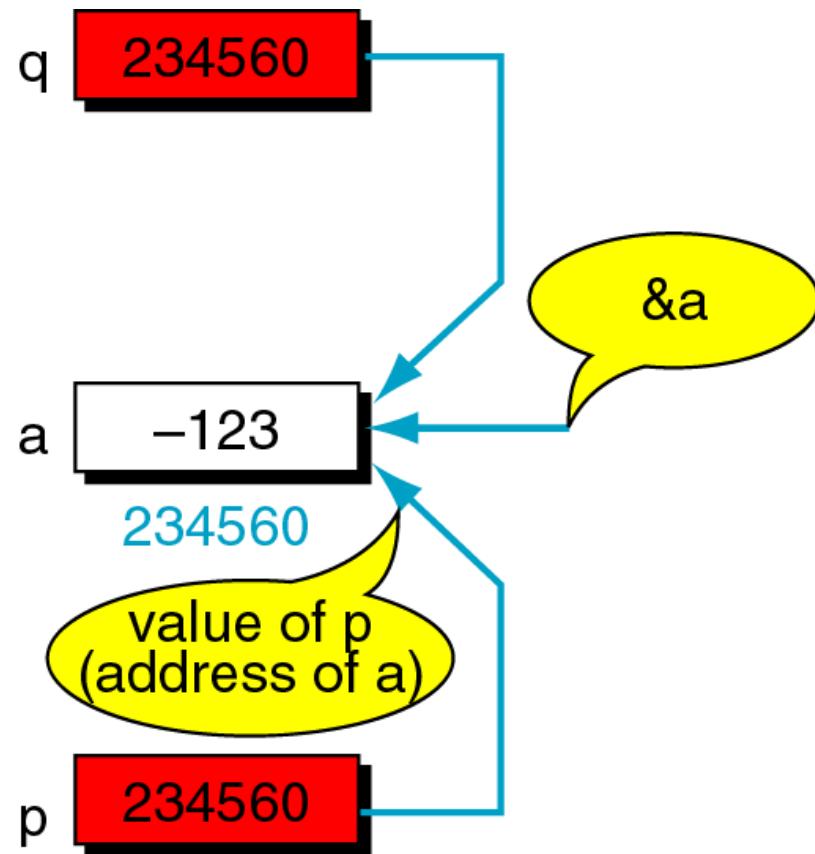


Physical representation



Logical representation

Pointer variables (fig 10.7 Multiple pointers to a variable)



Pointer variables

- What if we have a pointer variable, but not want to point to anywhere?
- What is its value then?
- C provides a special null pointer constant NULL defined in standard input-output <stdio.h> library

Accessing variables through pointers

The indirection operator *

- We have a variable and a pointer to the variable
- How can we use the pointer?
- C has indirection operator *
- When we dereference a pointer, we are using its value to reference (address) another variable
- The indirection operator is a unary operator whose operand must be a pointer value
- To access the variable a through the pointer p, we write *p

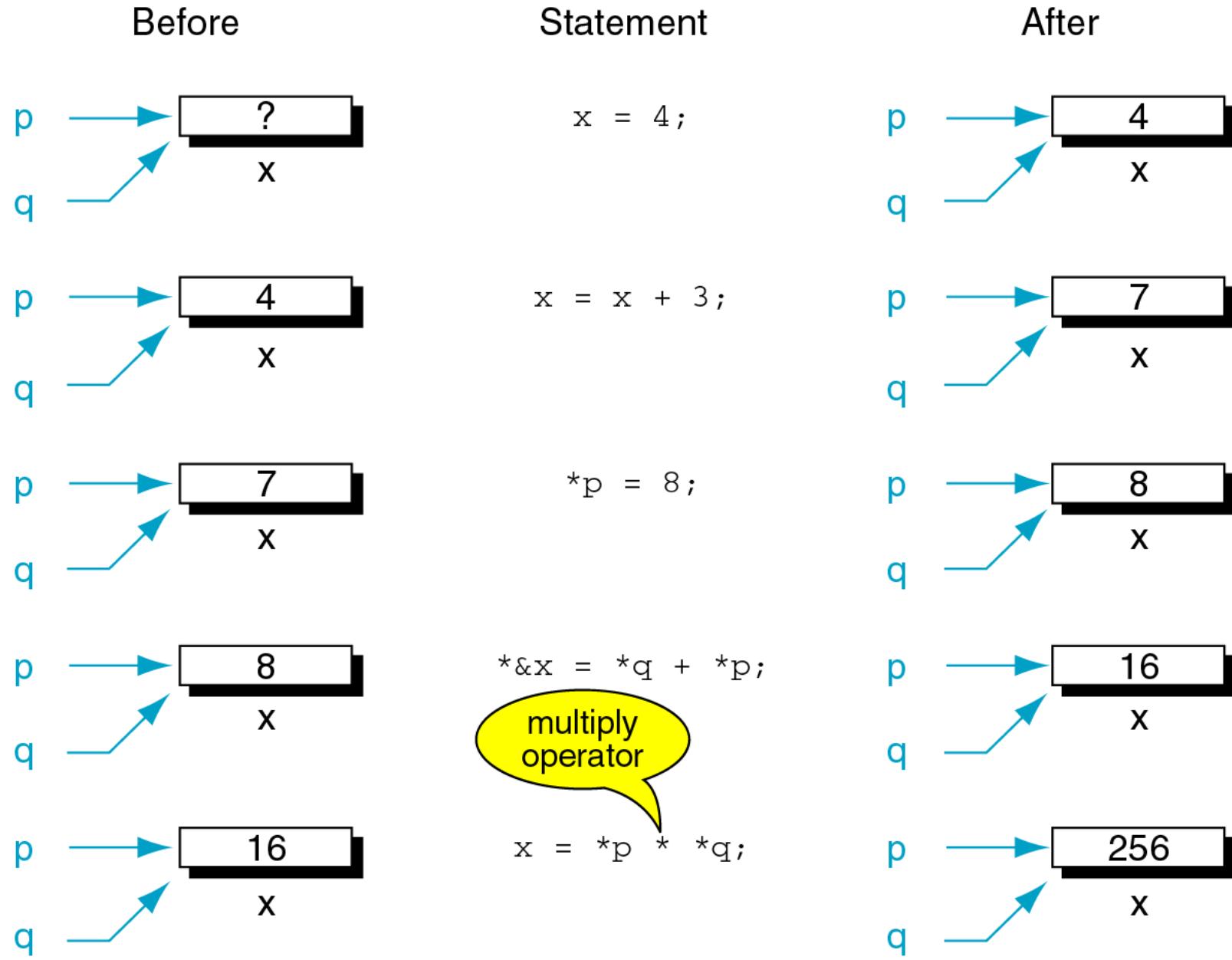
The indirection operator

- int a=1 and int *p=&a;
- Add 1 to variable a using pointers and other means
- a++; a= a+1; *p = *p + 1; (*p)++;
- Any one of this statement will do it , assuming that the pointer p is properly initialized as p = &a;
- (*p)++ need parenthesis because ++ has more priority than *
- The parenthesis force dereference to occur and then addition
- Without parenthesis, we add to the pointer first, which would change the address

The indirection operator *

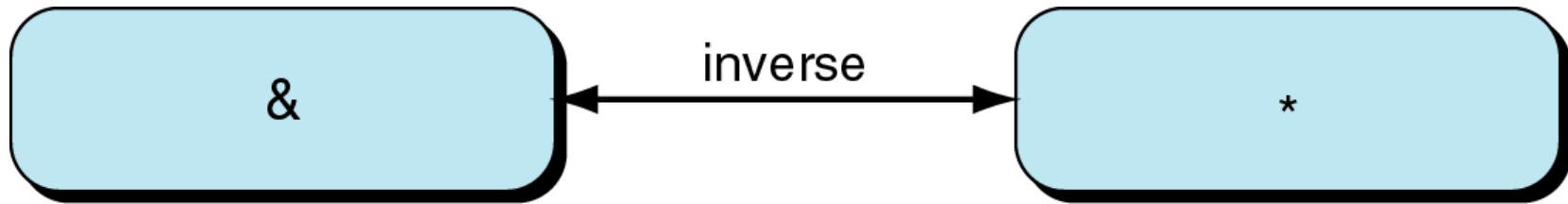
- Assume that the variable x is pointed to by 2 pointers p and q
- So x, *p, *q – allow the variable to inspect when used in RHS of the assignment operator
- When used in LHS, they change the value of x

Figure 10-8



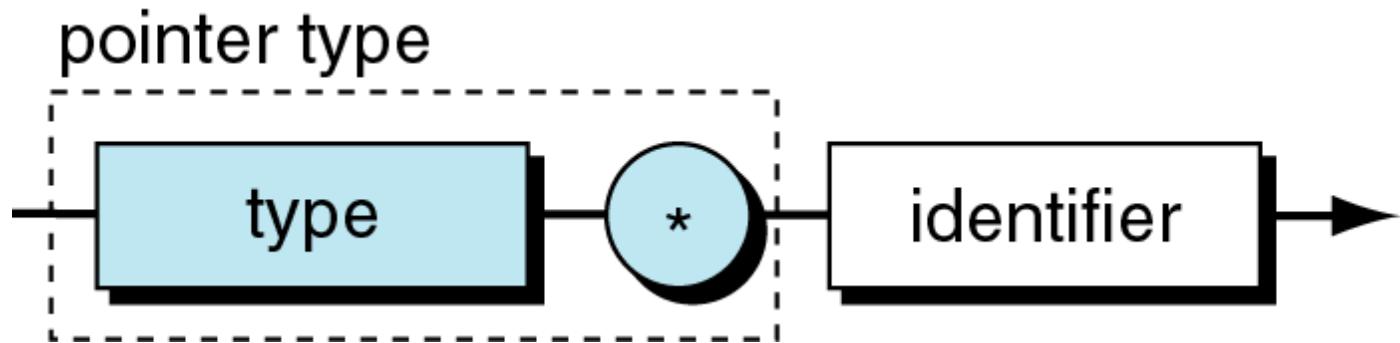
Accessing variables through pointers

(fig 10.9 Address & Indirection Operators)

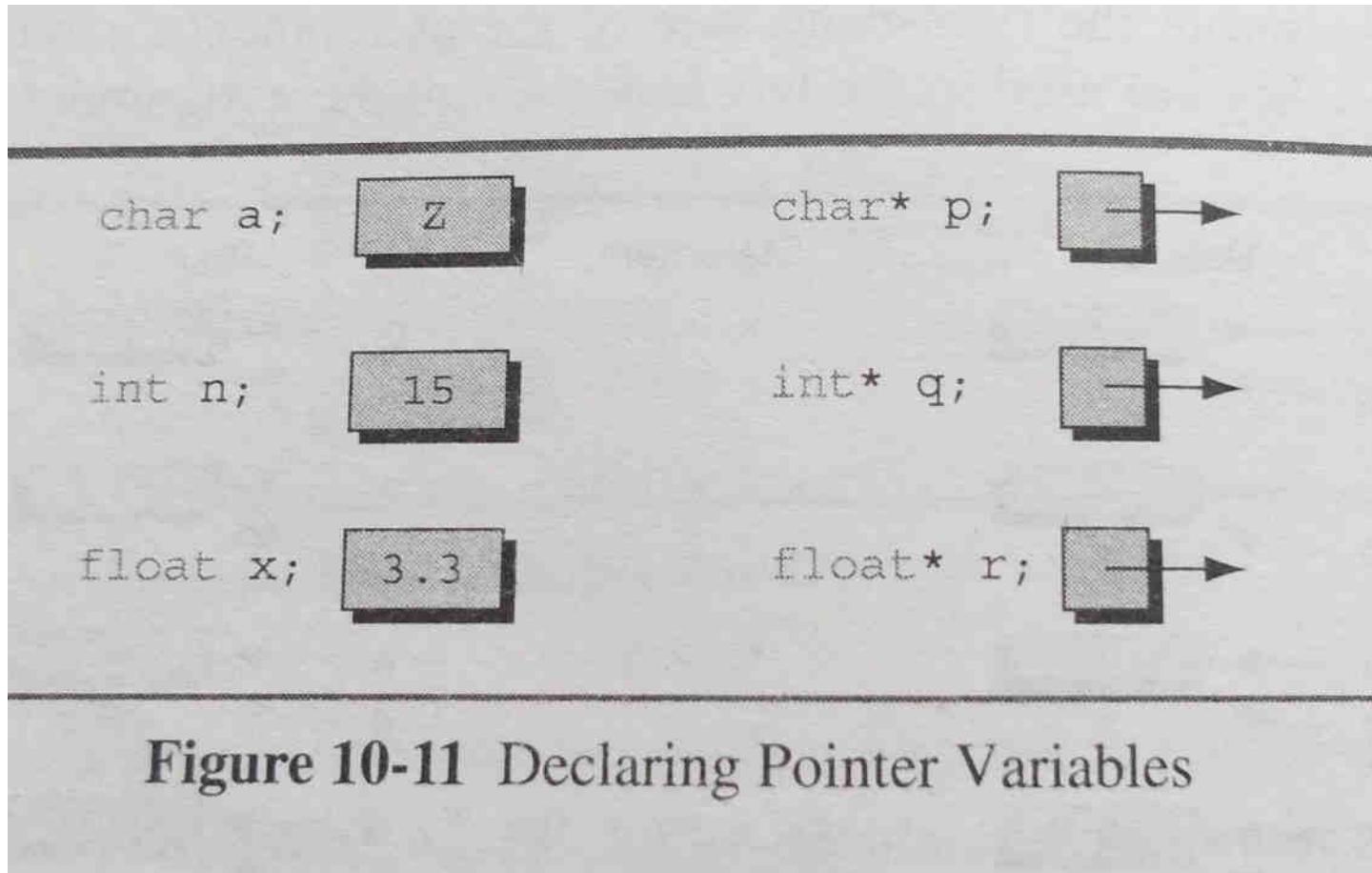


Pointer declaration and definition

Figure 10-10 Pointer Variable Declaration



Pointer declaration & definition



Problem

- WAP to define an integer variable a
- Define a pointer to integer and assign a's address
- Print a and its address
- Print the pointer value containing the address of a

Pointer declaration & definition

(Prog 10.1 Demonstrate use of pointers)

```
int main (void) {  
    int a;  
    int *p;  
    a = 14;  
    p = &a;  
    printf("%d %p\n", a, &a);  
    printf("%p %d %d\n", p, *p, a);  
    return 0;  
}
```

Pointer declaration & definition (Prog. 10.1 contd)

Results:

14 00135760

00135760 14 14

- Declaration versus Redirection

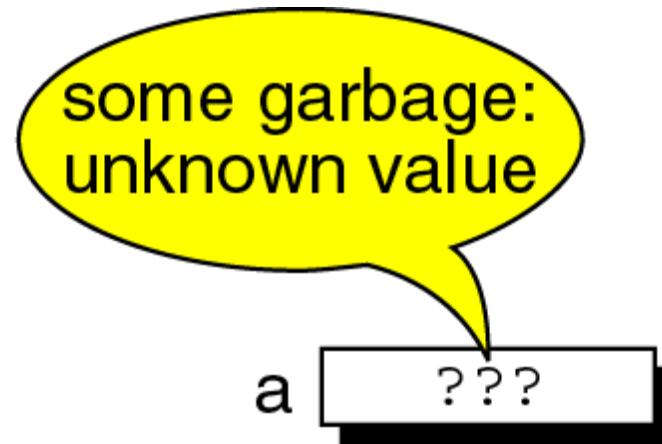
Initialization of pointer variables

Initialization of pointer variables

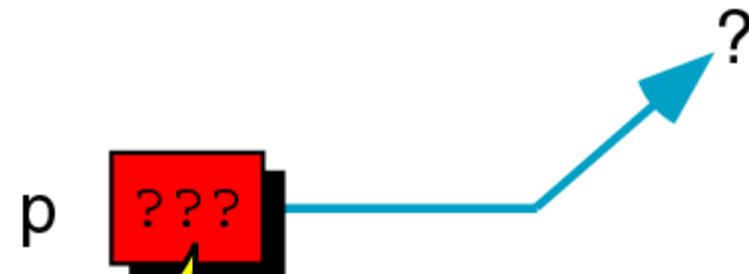
- C language does not initialize variables
 - The initialization has to be done by the OS loader
 - So that when we start our program uninitialized variables have garbage values
 - Although OS clears memory when it loads a program cannot be considered especially with programs that runs on multiple systems

Figure 10-12 Uninitialized variables and Pointers

```
int a;
```



```
int *p;
```

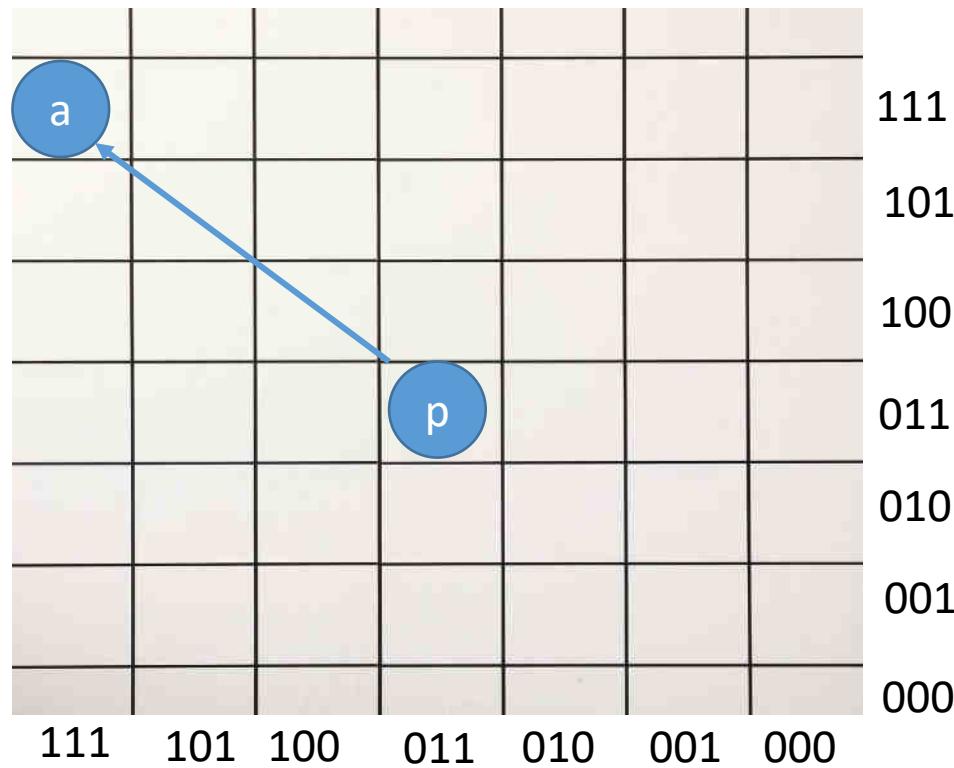
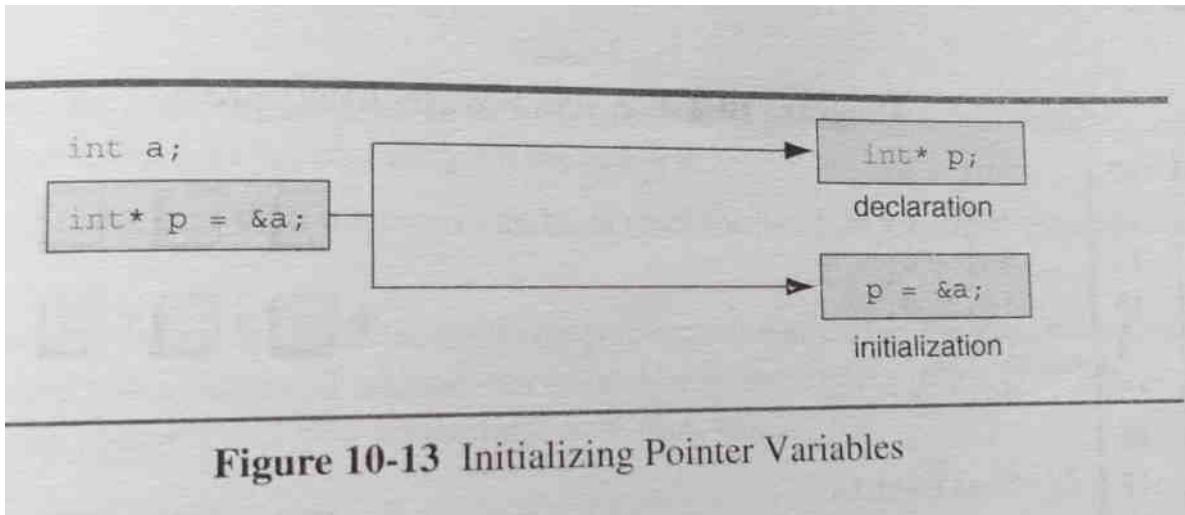


Initialization of pointer variables

- Initialization of pointer variables in C language:
- In the same way, C language does not initialize pointer variables
- So, when the program starts, uninitialized pointer variables will have garbage address
- So assign a valid memory address to the pointer
- Ex:

```
int a;  
int *p= &a;           // p has valid address  
*p = 90;             // a is assigned 90
```

Initialization of Pointer Variables



```
int main(void)
{
int a;
int *p = &a;
a=15;

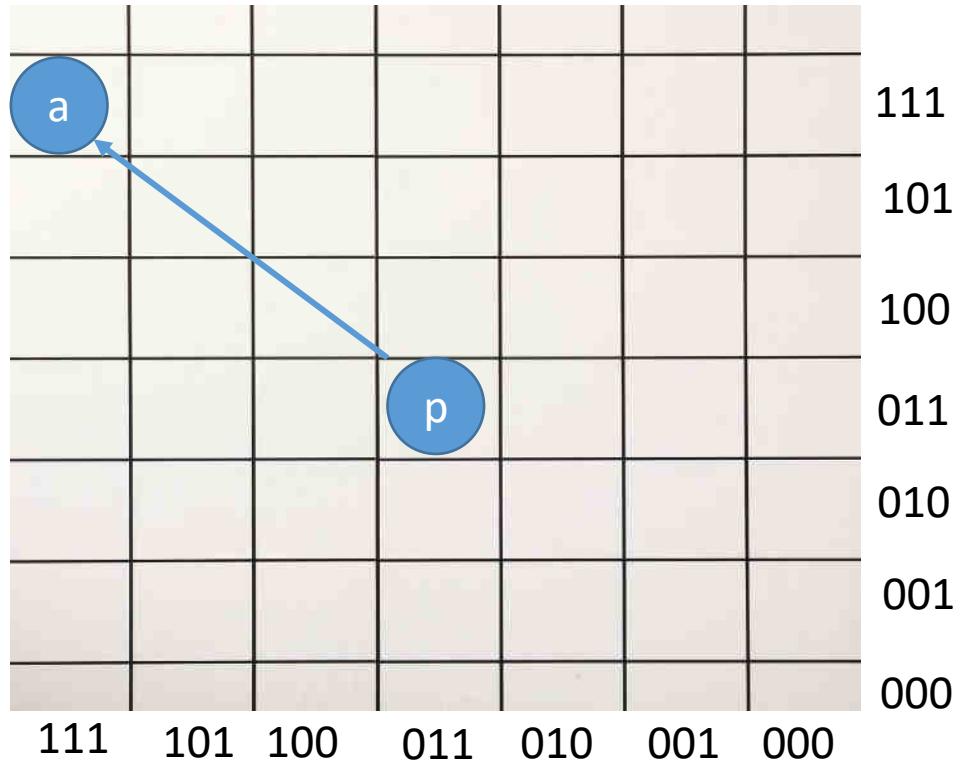
printf("Address of a: %p\n", p);

printf("Address of ptr_a: %p\n", &p);

printf("Address of a: %p\n", &a);

printf("value of a: %d\t%d\n", a,*p);

return 0;
}
```



Initialization of pointer variables

- How to set pointer to null during definition or during execution?

```
int *p = NULL;
```

- What happens when you dereference the pointer when it is null (or null pointer) ?
 - When we dereference a null pointer, we are using address zero
 - A valid address in the computer
 - Depending on OS, this can be the physical address zero or can be the first address location in our program area
 - In some systems
 - A Runtime error, NULL is not a valid address

Change Variables (prog 10.2)

```
int main (void) {
```

```
    int a, b, c;
```

```
    int *p, *q, *r;
```

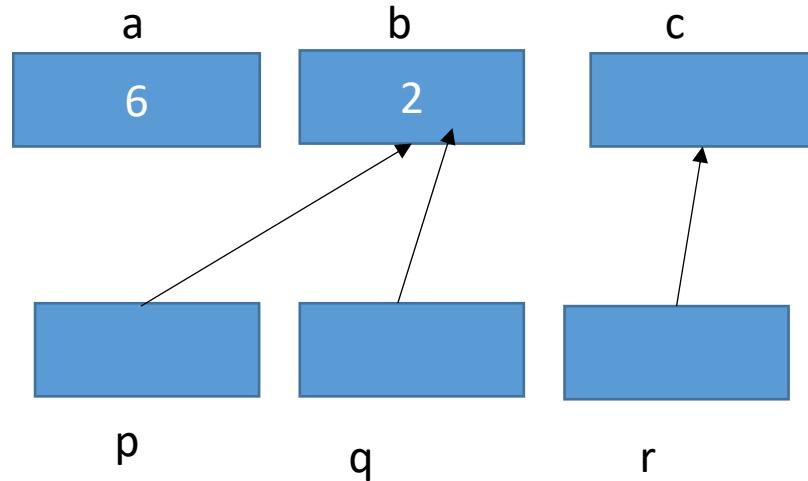
```
a = 6;
```

```
b = 2;
```

```
p = &b;
```

```
q = p;
```

```
r = &c;
```



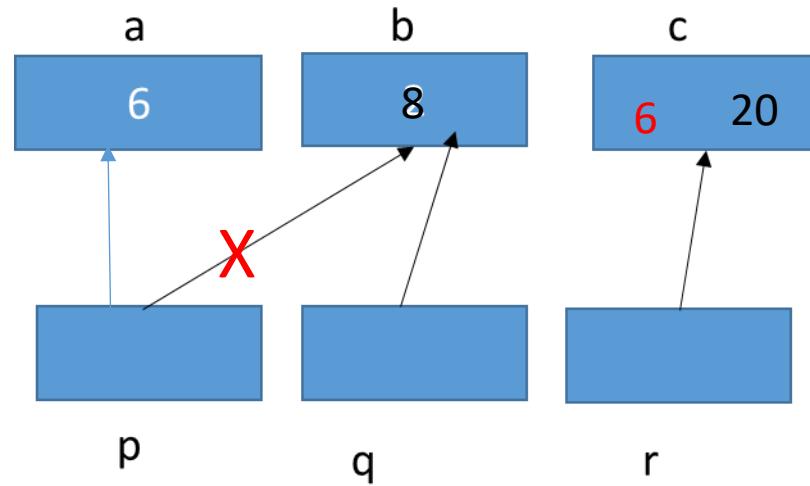
a,b,c

Change Variables (prog 10.2 contd)

```
p = &a;  
*q = 8;  
*r = *p;  
*r = a + *q + *&c;
```

```
printf("%d %d %d\n", a, b, c);  
printf("%d %d %d\n", *p, *q, *r);  
return 0;
```

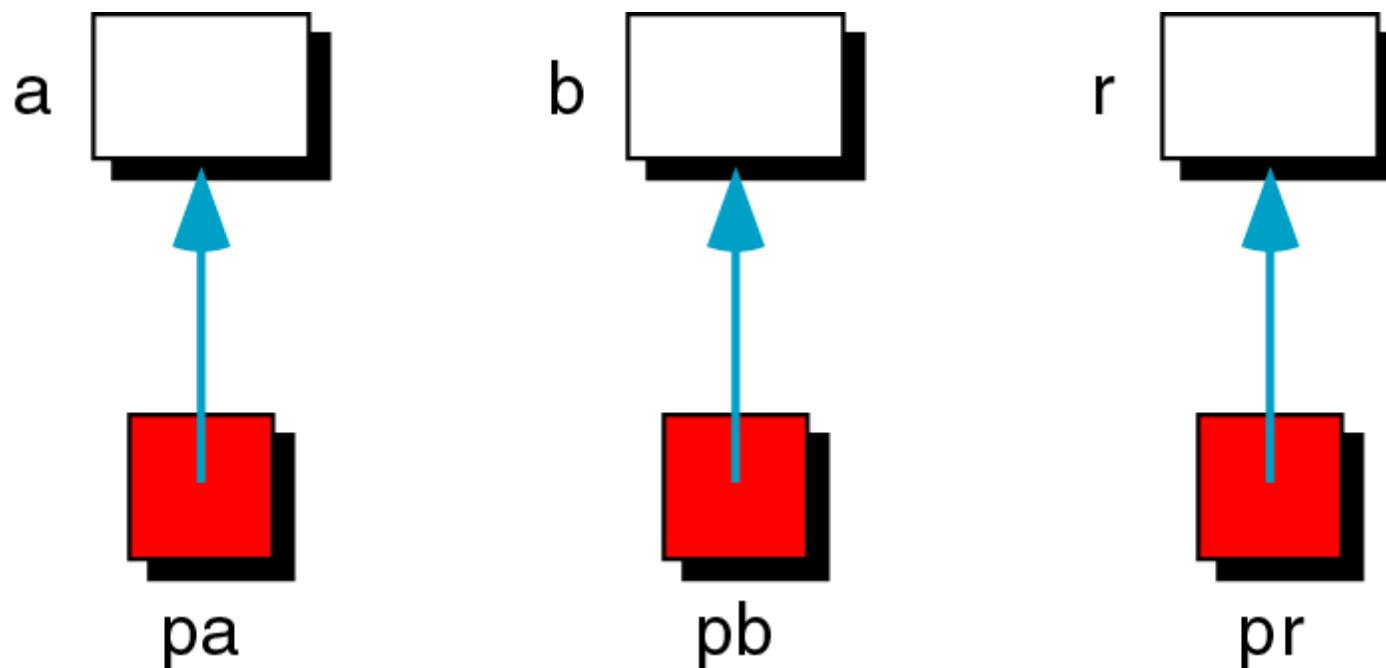
```
}
```



Change Variables (prog 10.2 contd)

```
Results:  
6 8 20  
6 8 20
```

Figure 10-14 Add two numbers using pointers



Add two numbers using pointers (prog 10.3)

```
int main (void) {  
    int a, b, r;  
    int *pa = &a;  
    int *pb = &b;  
    int *pr = &r;
```

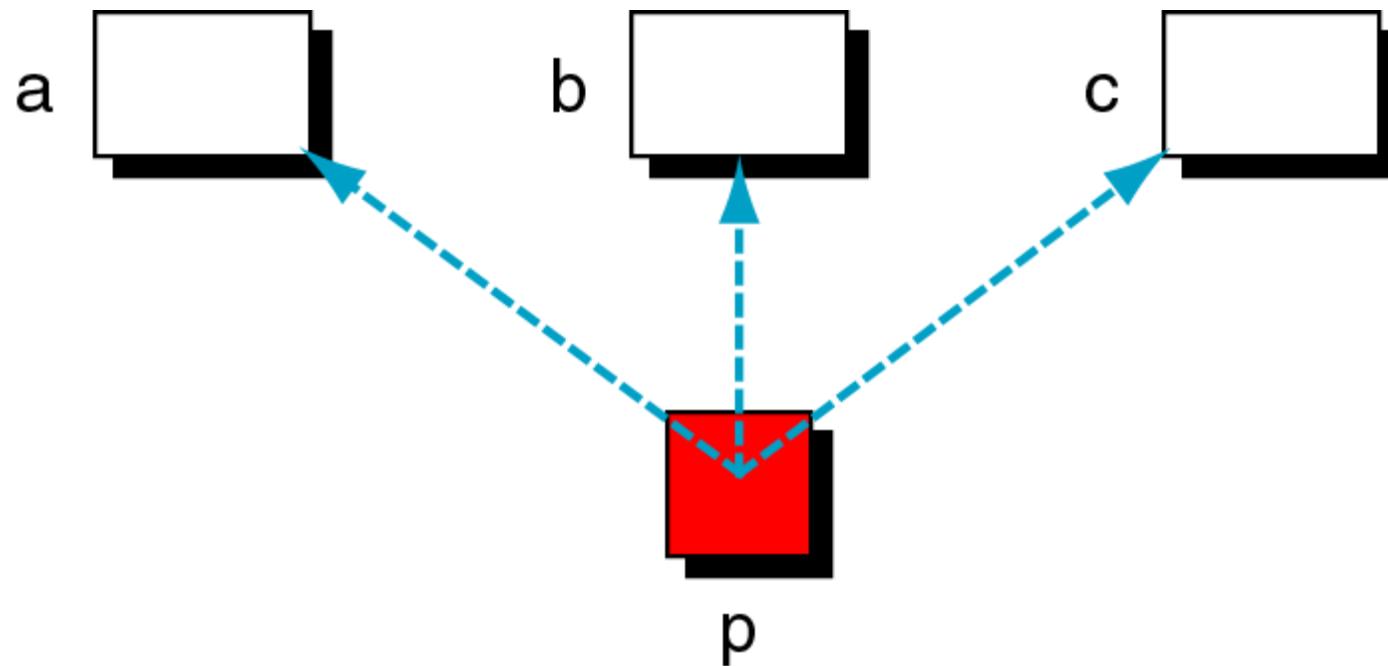
```
printf("Enter the first number : ");
```

- - -

Add two numbers using pointers (prog 10.3)

```
int main (void) {  
    int a, b, r;  
    int *pa = &a;  
    int *pb = &b;  
    int *pr = &r;  
  
    printf("Enter the first number : ");  
    scanf("%d", pa);  
    printf("Enter the second number : ");  
    scanf("%d", pb);  
    *pr = *pa + *pb;  
    printf("\nThe sum is : %d\n", *pr);  
    return 0;  
}
```

Figure 10-15 Demonstrate pointer flexibility



Using one pointer for many variables (prog 10.4)

```
int main (void) {
```

```
    int a, b, c;
```

```
    int *p;
```

```
    printf("Enter three numbers : ");
```

```
    scanf("%d %d %d", &a, &b, &c);
```

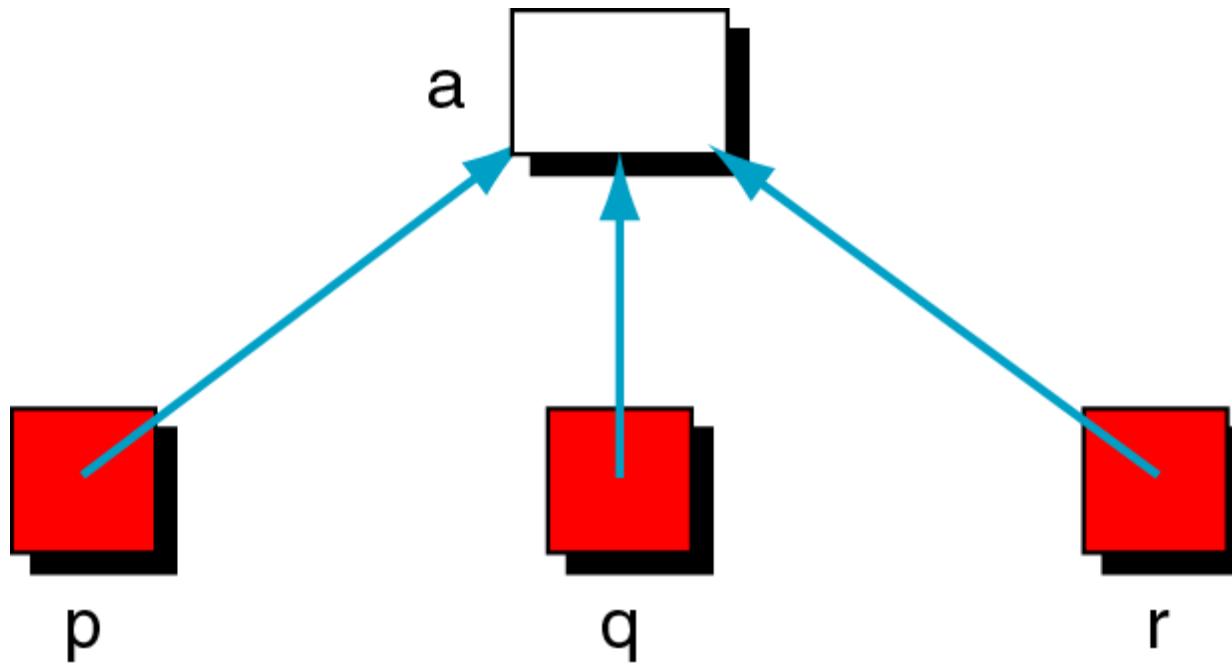
- - - -

Using one pointer for many variables (prog 10.4)

```
int main (void) {
    int a, b, c;
    int *p;

    printf("Enter three numbers : ");
    scanf("%d %d %d", &a, &b, &c);
    p = &a;
    printf("%d\n", *p);
    p = &b;
    printf("%d\n", *p);
    p = &c;
    printf("%d\n", *p);
    return 0;
}
```

Figure 10-16 One variable with many pointers



Using a variable with many pointers (prog 10.5)

```
int main (void) {
    int a;
    int *p = &a;
    int *q = &a;
    int *r = &a;

    printf("Enter a number : ");
    scanf("%d", &a);
    printf("%d\n", *p);
    printf("%d\n", *q);
    printf("%d\n", *r);
    return 0;
}
```

Pointers and Functions

Pointers for Inter-function communication

Passing addresses (fig 10.17)

```
void exchange (int x, int y);

int main (void)
{
    int a = 5;
    int b = 7;
    exchange (a, b);
    printf ("%d %d\n", a, b);
    return 0;
} // main
```

Pointers for Inter-function communication

Passing addresses (fig 10.17 contd)

```
void exchange (int x, int y)
{
    int temp;

    temp = x;
    x    = y;
    y    = temp;
    return;
} // exchange
```

Pointers and Functions

- Functions receiving pointer values as arguments
 - Ex1: exchange program without pointers
 - Ex2: exchange program with pointers

Pointers and Functions

- Most useful application of pointers is in functions
- How does C function operate?
 - C uses pass by value concept
 - This means that the only direct way to send something back from a function is through return value
- How to simulate pass by address?
 - We can simulate the pass by reference by passing an address and using it to refer back to data in the calling program
 - When we pass by address, we are actually passing a pointer to a variable

Pointers and Functions

- pass pointers to the values
- Given a pointer to a variable anywhere in our memory space
 - whether it is local to a function or main() or a global variable
 - we can change the content of the variable

Pointers and Functions

- It is important to understand that C still uses pass by value
 - Now the value is the address of the variable we need to change
 - We are only simulating pass by reference

Pointers and Functions

- If we want a called function to have access to a variable in the calling function send the address of that variable to the called function and use the indirection operator & to access it
- To send back more than one value from a function, use pointers
- By passing the address of variables defined in calling function, we can store data directly in the calling function rather than using return

Pointers for Inter-function communication

Passing addresses (fig 10.18 the correct way)

```
void exchange (int*, int*);  
  
int main (void)  
{  
    int a = 5;  
    int b = 7;  
  
    exchange (&a, &b);  
    printf ("%d %d\n", a, b);  
    return 0;  
} // main
```

Pointers for Inter-function communication

Passing addresses (fig 10.18 the correct way - contd)

```
void exchange (int* px, int* py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
    return;
} // exchange
```

Functions Returning Pointers

Pointers and Functions

- Functions returning a pointer to the calling function
 - Ex: program to find the smaller of two numbers by taking address of two numbers as input and returning address of smaller number as output

Pointers for Inter-function communication

Functions returning pointers (fig 10.19)

```
int* smaller (int* p1, int* p2);  
  
int main (void)  
{  
    ...  
    int a;  
    int b;  
    int* p;  
    ...  
    scanf ( "%d %d", &a, &b );  
    p = smaller (&a, &b);  
    ...  
}
```

Pointers for Inter-function communication

Functions returning pointers (fig 10.19 contd)

```
int* smaller (int* px, int* py)
{
    return (*px < *py ? px : py);
} // smaller
```

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

```
int *smaller(int *px, int *py)
```

```
{
```

```
    return (*px < *py? px:py )
```

```
}
```

```
int main()
```

```
{
```

```
    int a,b, *p;
```

```
/* Input two numbers from user */
```

```
printf("Enter any two numbers: ");
```

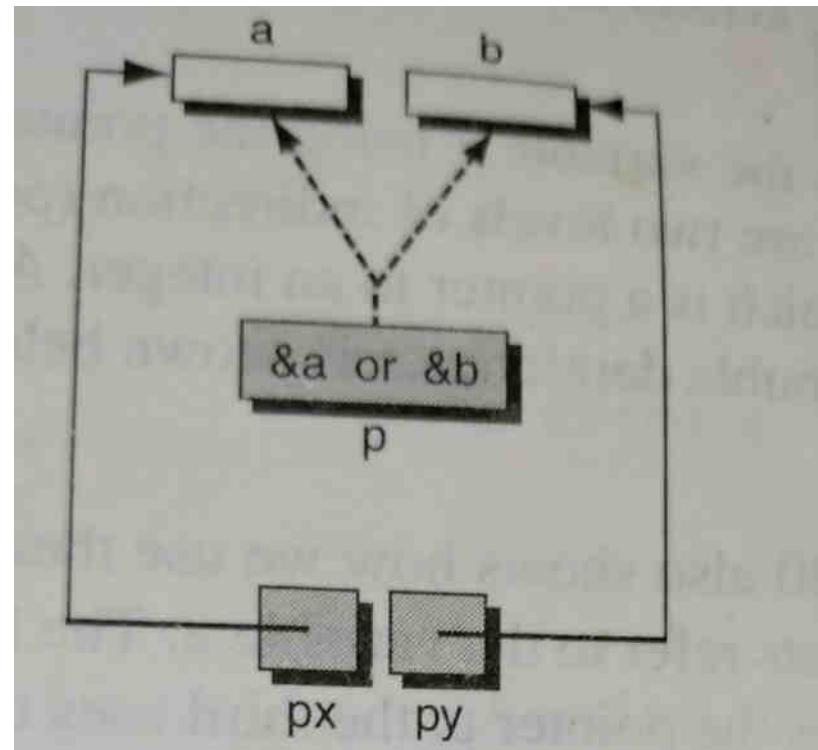
```
scanf("%d%d", &a, &b);
```

```
p = smaller(&a, &b); // Call minimum function
```

```
printf("Minimum = %d", *p);
```

```
return 0;
```

```
}
```



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

```
int *smaller(int *px, int *py)
{
    int *ptr;
    if(*px > *py )
        ptr=px;
    else ptr=py;
    return ptr;
}
```

```
int main()
{
```

```
    int a,b, *p;
```

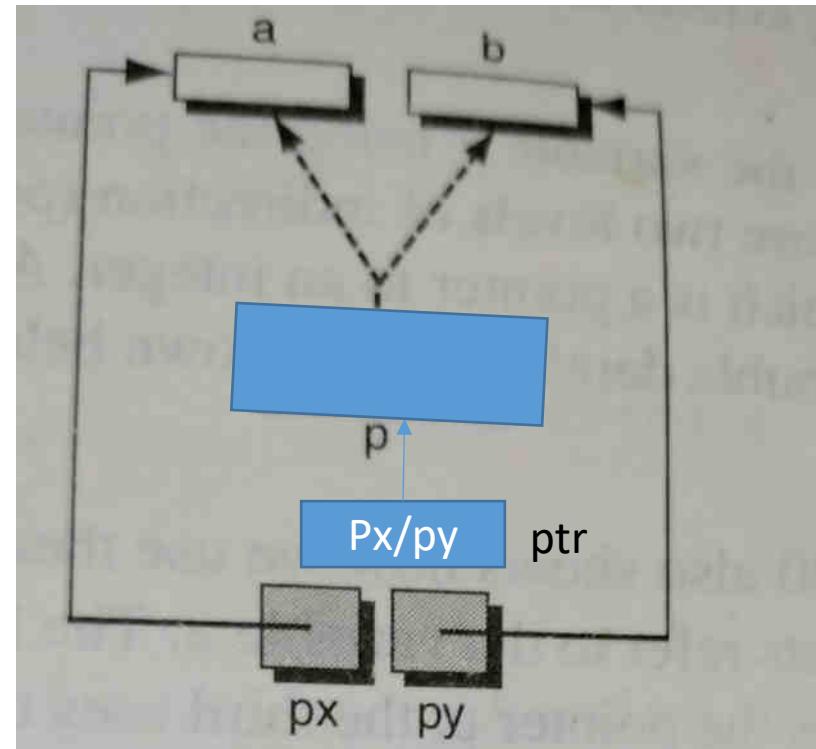
```
/* Input two numbers from user */
printf("Enter any two numbers: ");
scanf("%d%d", &a, &b);
```

```
p = smaller(&a, &b); // Call minimum function
```

```
printf("Minimum = %d", *p);
```

```
return 0;
```

```
}
```



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

```
int *smaller(int *px, int *py)
```

```
{
```

```
    int ptr;
```

```
    if(*px > *py )
```

```
        ptr= *px;
```

```
    else ptr= *py;
```

```
    return &ptr; X
```

```
}
```

```
int main()
```

```
{
```

```
    int a,b, *p;
```

```
/* Input two numbers from user */
```

```
printf("Enter any two numbers: ");
```

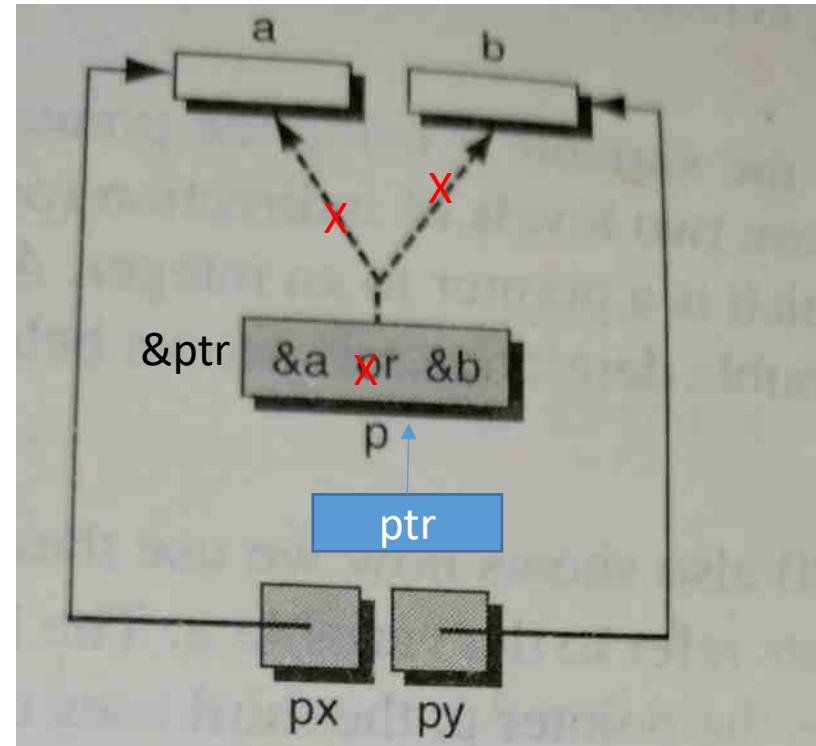
```
scanf("%d%d", &a, &b);
```

```
p = smaller(&a, &b); // Call minimum function
```

```
printf("Minimum = %d", *p);
```

```
return 0;
```

```
}
```



Functions Returning Pointers – Points to note

- It is a serious error to return a pointer to a local variable
- When we return a pointer, it must point to data in the calling function or a higher level function
- It is an error to return a pointer to a local variable in the called function because
 - When the function terminates, its memory may be used by other parts of the program
 - Especially evident in large programs

Pointers to Pointers

Pointers to Pointers

- All pointers have been pointing directly to data
- Advanced data structures require
- Pointers that point to other pointers
- Ex: we can have a pointer pointing to a pointer to an integer
- This two-level indirection is shown next

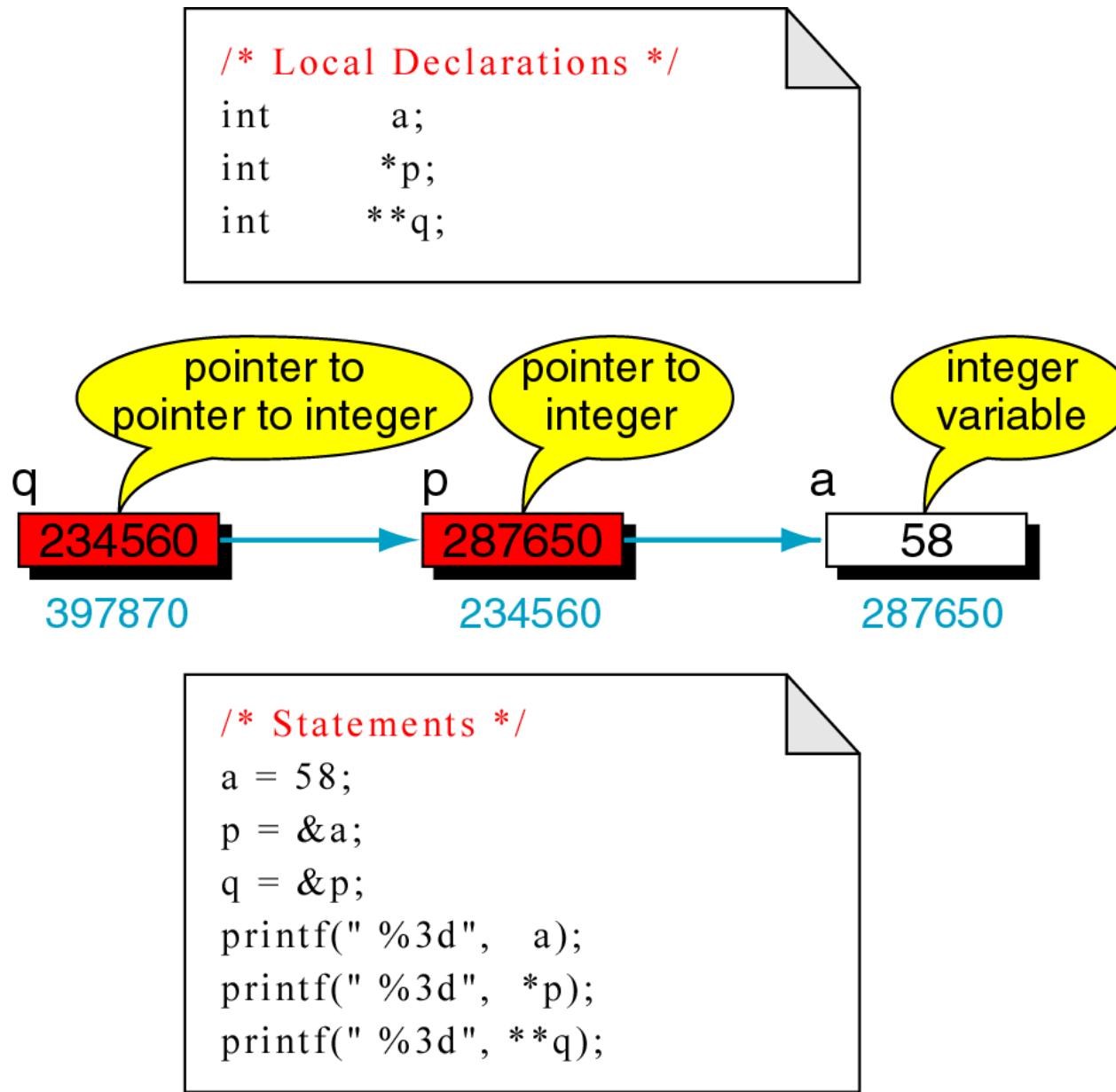
Note: Although many levels of indirection can be used but practically not more than two levels are needed

Pointers to Pointers (fig 10.20)

```
// Local Declarations
int  a;
int* p;
int** q;
```

```
// Statements
a = 58;
p = &a;
q = &p;
printf(" %3d", a);
printf(" %3d", *p);
printf(" %3d", **q);
```

Figure 10-20 Pointers to pointers



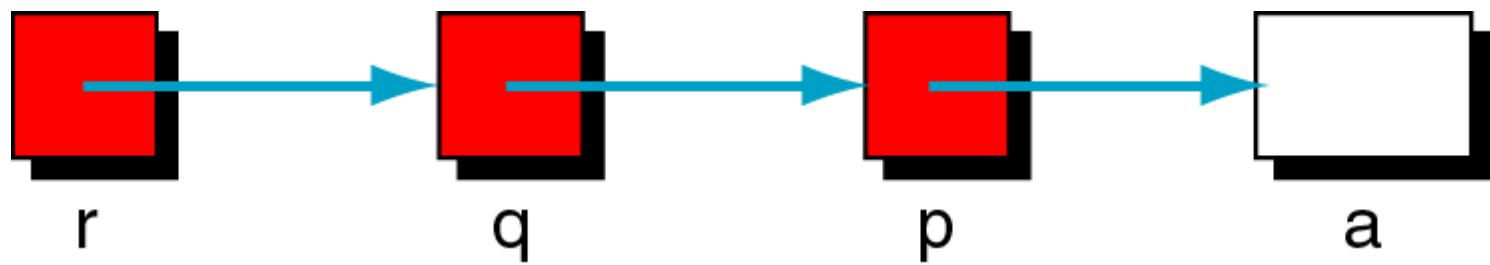
Pointers to Pointers

- Each level of pointer indirection requires a separate indirection operator when it is dereferenced
- Ex: To refer to a using the pointer p, we have to dereference it once as
 - `*p`
- To refer to a using the pointer q, we need to dereference it twice to get to the integer a as there are 2 levels of indirection (pointers) involved
- By first dereference, we reference p, which is a pointer to an integer

Pointers to Pointers

- q is a pointer to a pointer to an integer
- `**q`
- Program:
- WAP to print the value of a by 4 means:
 - Directly using a
 - Using pointer p
 - Using pointer q
 - Using pointer r

Figure 10-21 Using Pointers to Pointers



Pointers to Pointers (fig 10.6)

```
int main (void)
{
    // Local Declarations
    int      a;
    int*     p;
    int**    q;
    int***   r;

    // Statements
    p = &a;
    q = &p;
    r = &q;
```

Pointers to Pointers (fig 10.6 contd)

```
printf("Enter a number: ");
scanf ("%d", &a);
printf("The number is : %d\n", a);

printf("\nEnter a number: ");
scanf ("%d", p);
printf("The number is : %d\n", *a);

printf("\nEnter a number: ");
scanf ("%d", *q);
printf("The number is : %d\n", *a);

printf("\nEnter a number: ");
scanf ("%d", **r);
printf("The number is : %d\n", *a);
```

Chapter 11

Pointer Applications

Pointer Applications

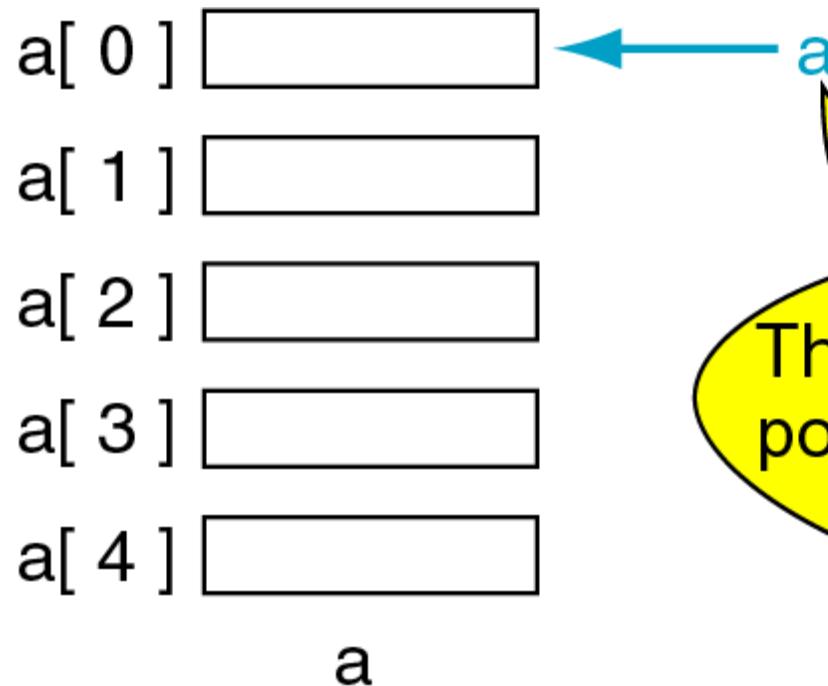
1. Arrays
2. Dynamic Memory

Pointer Applications

- **Arrays and Pointers**
- The name of an array is a pointer constant to the first element
- Since the array name is a pointer constant to the first element
 - its value cannot be changed
 - the address of the first element and the name of the array both represent the same location in memory
 - we can use the array name anywhere we can use a pointer, as long as it is used as an rvalue
 - we can use it with the indirection operator

```
int main()
{
    int a[10]={2,4,6,8,22};
    int *ptr;
    printf("%d",*a);
//    a++; //invalid
    ptr=a+1;
    printf("%3d",*ptr);
    ptr++;
    printf("%3d%3d",*ptr,*(a+2));
    return 0;
}
```

Figure 11-1 Pointer to Arrays



Arrays and Pointers

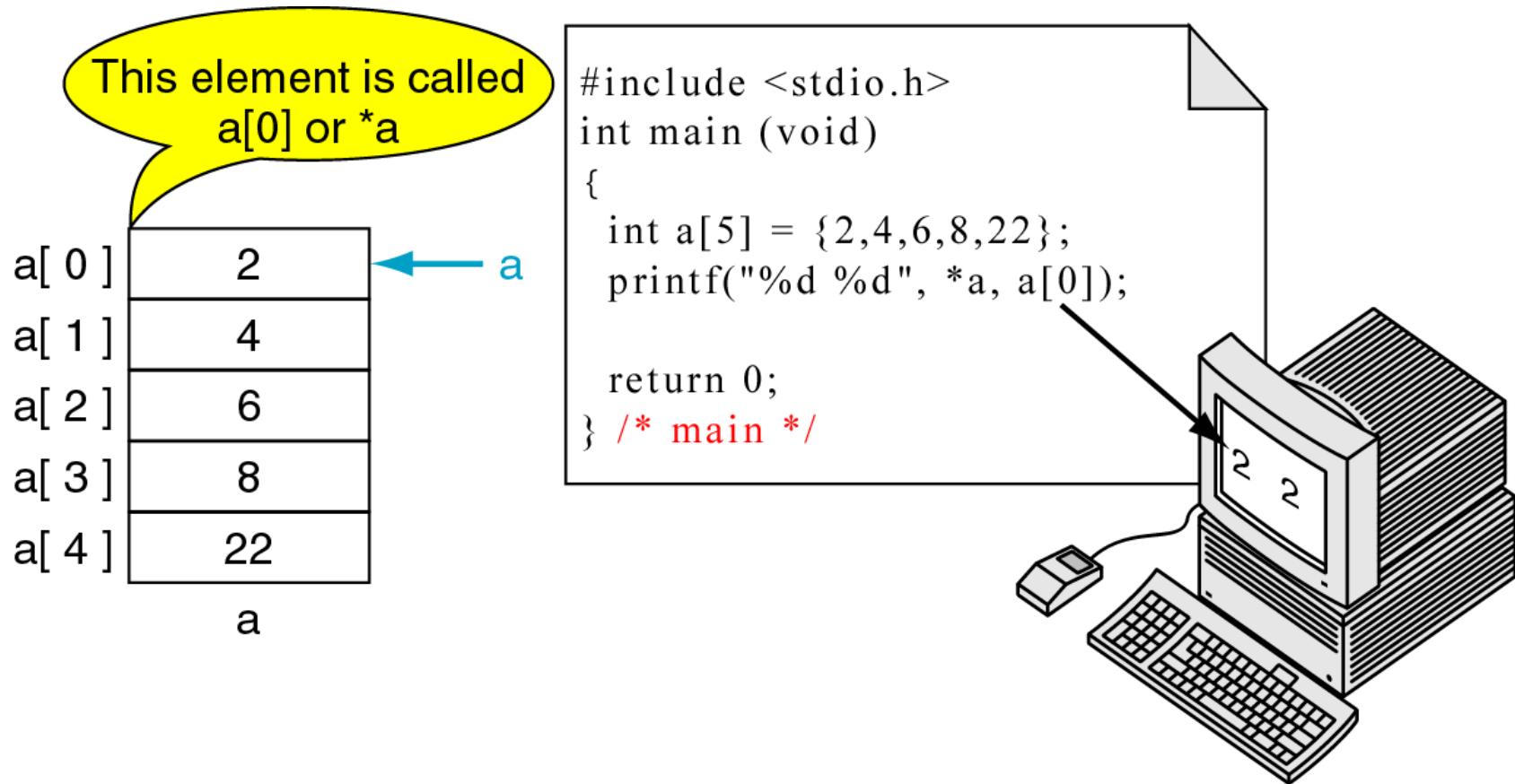
- When we dereference an array name . . .
- we are dereferencing the first element of the array & not the whole array
- a is same as &a[0]
- Program: demonstrate that array name is a pointer constant
- Print the address of the first element of the array and the array name

```
int a[5];
printf("%p %p", &a[0], a);
```

Dereference of Array name

- Variation – prove that the array name is a pointer constant to the first element of the array
- Program: Demonstrate by printing the value in the first element of the array using both a pointer and an index

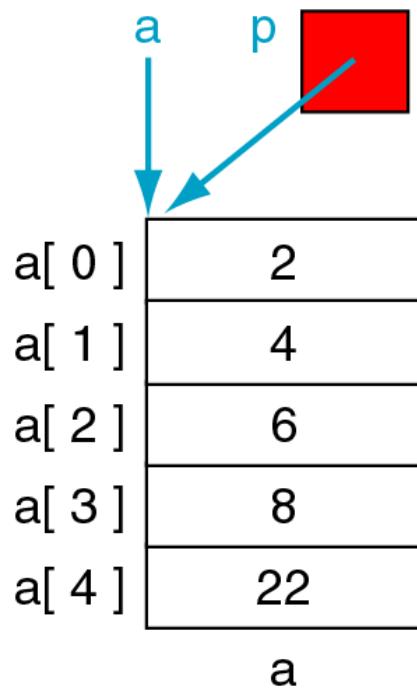
Figure 11-2 Dereference of Array name



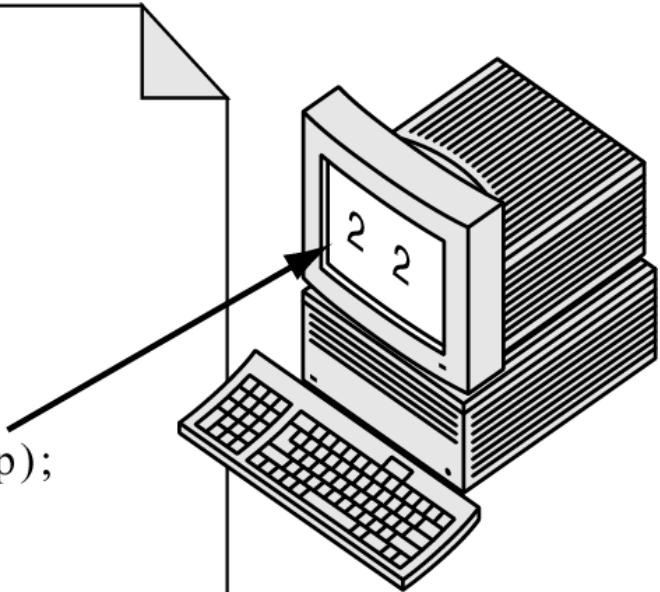
Array names as pointers

- If the name of an array is a pointer, show that we can store this pointer in a pointer variable and use it in the same way we use the array name
- Define and initialize the array
- Define a pointer and initialize it to point to the first element of the array by assigning the array name
- Print the first element in the array using both index and pointer notations
- Note: the array name is unqualified. There is no address operator or index

Figure 11-3 Array names as pointers



```
#include <stdio.h>
int main (void)
{
    int a[5] = {2, 4, 6, 8, 22};
    int *p = a;
    int i = 0;
    ...
    printf("%d %d\n", a[i], *p);
    ...
    return 0;
} /* main */
```

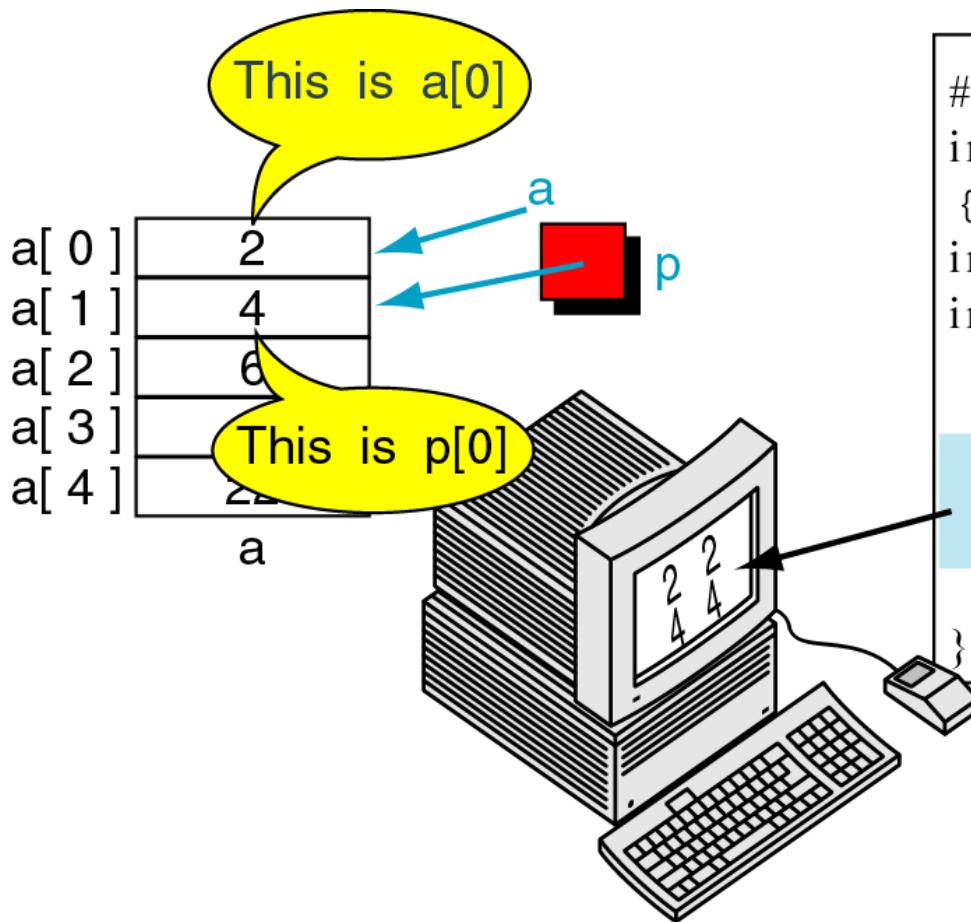


```
int main()
{
    int a[]={2,4,6,8,22};
    int i=0;
    int *ptr=a;
    for(i=0;i<5;i++)
    {
        printf("%d\t%d\n",a[i],*ptr);
        ptr++;
    }
    return 0;
}
```

Array and a pointer

- Program: demonstrate the use of multiple names for an array to reference different locations at the same time
 - 1) by an array name
 - 2) create a pointer to integer and set it to the second element of the array a[1]
- This pointer can be used as an array name and can be indexed
- Print the first 2 elements using array name as well as by pointer
- Note: negative index

Figure 11-4 Multiple Array Pointers



```
#include <stdio.h>
int main (void)
{
    int a[5] = {2, 4, 6, 8, 22};
    int*p;
    ...
    p = &a[1];
    printf("%d %d", a[0], p[-1]);
    printf("%d %d", a[1], p[0]);
    ...
} /* main */
```

- To access an array, any pointer to the first element can be used instead of the name of the array

```
int main()
{
    int a[]={2,4,6,8,22};
    int i=0;
    int *ptr=&a[1];
    printf("%d\t%d\n",a[1],ptr[0]);
    printf("%d\t%d\n",a[0],ptr[-1]);
    printf("%d\t%d\n",a[2],ptr[1]);
    printf("%d\t%d\n",a[3],*(ptr+2));
    printf("%d\t%d\n",a[1],*(ptr--));
    printf("%d\t%d\n",a[0],*ptr);
    return 0;
}
```

Pointer Arithmetic and Arrays

Pointer Arithmetic and Arrays

- Moving through an array:
 1. indexing
 2. Pointer Arithmetic
- **Pointer Arithmetic:** offers a set of arithmetic operators for manipulating the addresses in pointers
- Powerful for moving element by element in an array (search sequentially)

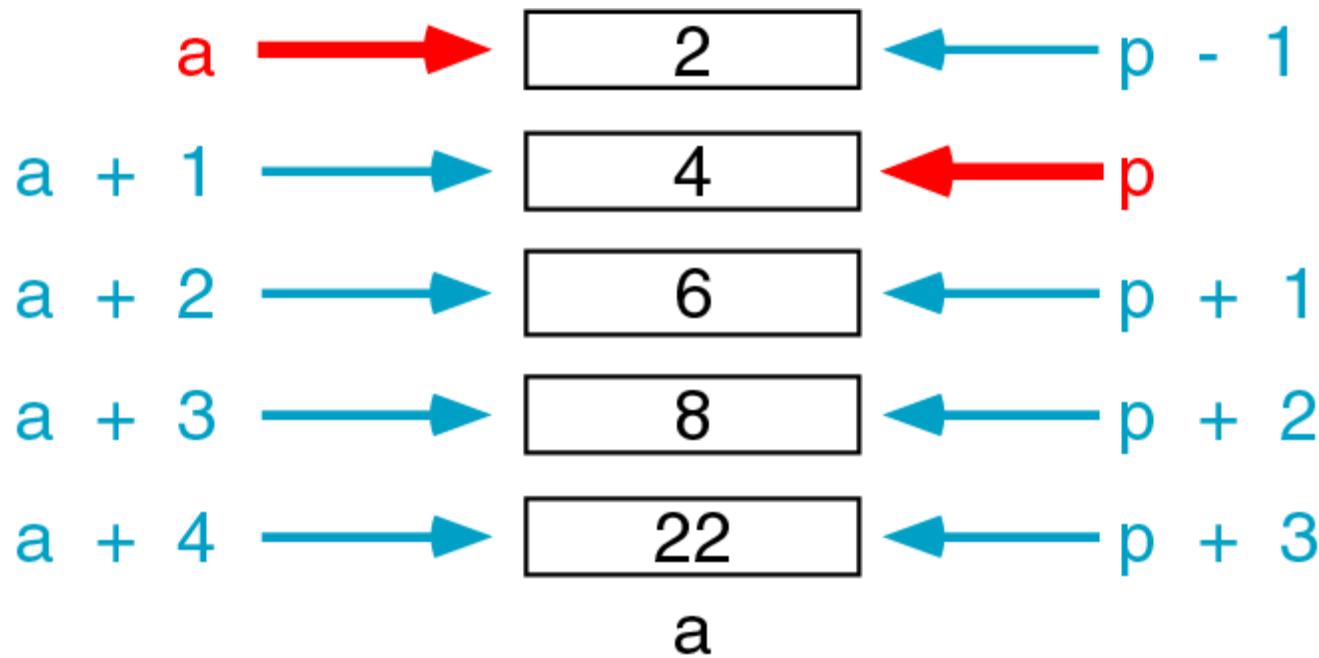
Pointers and One-Dimensional Arrays

- If we have an array, a , then
 - a is a constant pointing to the first element
 - $a+1$ is a constant to the second element
- If p is a pointer pointing to $a[1]$, then
 - $p-1$ is a pointer to the previous (first) element
 - $p + 1$ is a pointer to the next (third) element
- Generalizing:
- Given pointer p , $p + n$ or $p-n$ is a pointer to the value n elements away

Pointers and One-Dimensional Arrays

- As long as `a` or `p` are pointing to one of the elements of the array
 - we can add or subtract to get the address of other elements of the array

Figure 11-5 Pointer Arithmetic



- Different from normal arithmetic
- Adding an integer n to a pointer value gives a new value that corresponds to an index location, n elements away
- So n is an offset from the original pointer

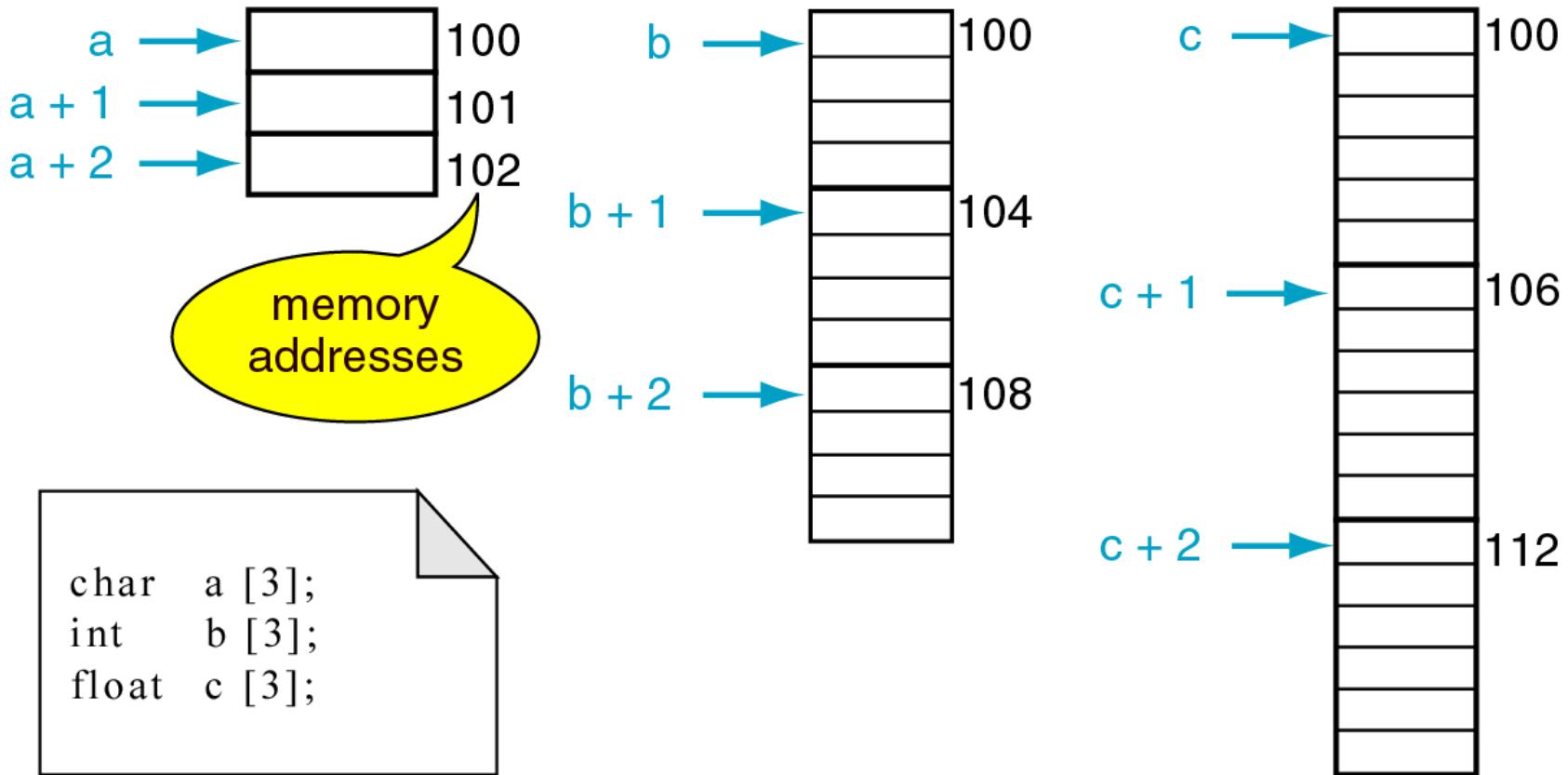
Pointers and One-Dimensional Arrays

- To determine the new value, C must know the size of one element
- The size of element is determined by the type of the pointer
- This is the reason why pointers of different types cannot be assigned to each other

Pointers and One-Dimensional Arrays

- If offset is 1:
 - C will add or subtract one element size from the current pointer value
 - access is more efficient than corresponding index notation
 - If offset is more than 1:
 - C must compute the offset by multiplying the offset by the size of one array element and adding it to the pointer value as below:
 - address = pointer + (offset * size of element)
 - Depending on the hardware the multiplication can make it less efficient than simply adding 1
 - So pointer arithmetic is not efficient compared to indexing
- 1/9/2022 $a + n$ becomes $a + n * (\text{sizeof(one element)})$ 21

Figure 11-6 Pointer Arithmetic and different types



- Pointer arithmetic on different sized elements
1/9/2022

Pointer arithmetic on different sized elements

- **a+1means different things:**
- **char takes 1 byte:**
 - Adding 1 moves to the next memory address (101)
- **Integer taking 4 bytes:**
 - Adding 1 to array pointer b moves to 4 bytes in memory (104)
- **Float taking 4 bytes:**
 - Adding 1 to array pointer c moves to 4 bytes in memory (104)

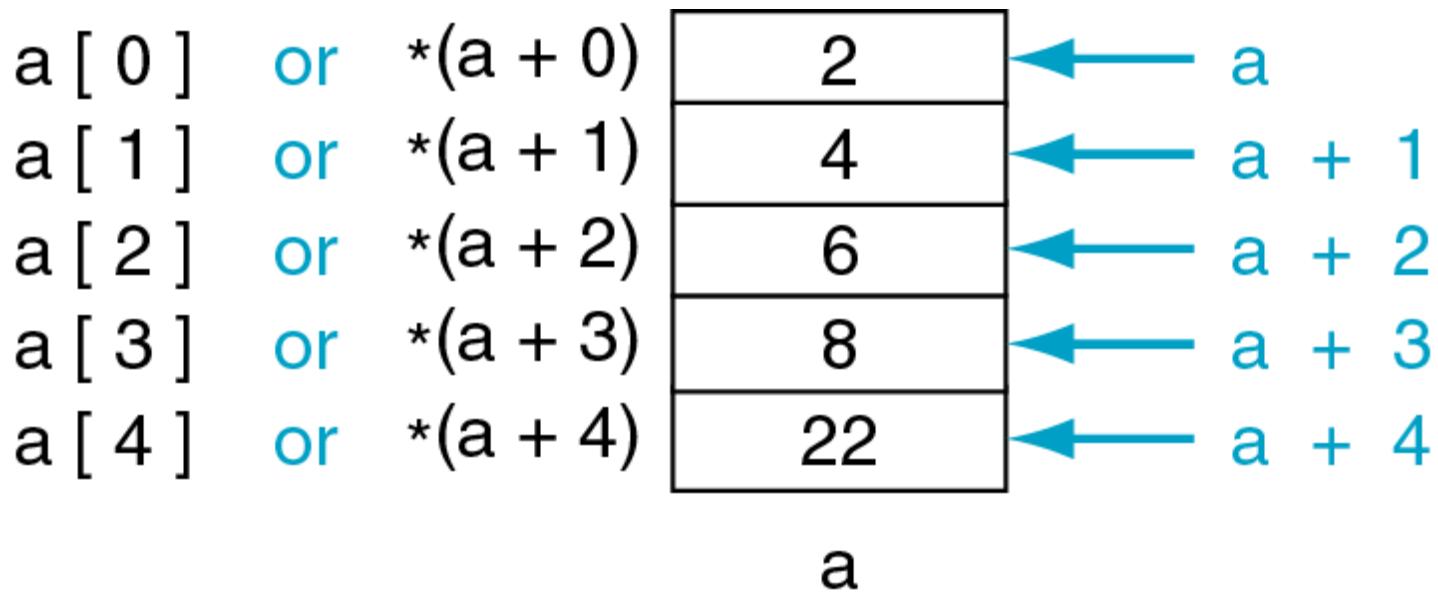
Note: size is compiler dependent

Pointers and One-Dimensional Arrays

- We have seen how to get the address of an array element using a pointer and an offset
- How can we use that value?
 - 1) we can assign it to another pointer

`p = arrayName + 5;`
 - 2) use with indirection operator to access or change the value of the element we are pointing to

Figure 11-7 Dereferencing array pointers



$a[n]$ and $*(a+n)$ are identical expressions

Program to find the smallest number

- Find the smallest number among 5 integers stored in an array
- pSm is set to the first element of the array
- pWalk is working pointer pointing to the second element
- Working pointer advances through the remaining elements
- Compares current element with element pointed to by pSm & assigning smaller to pSm

Figure 11-8 Find smallest

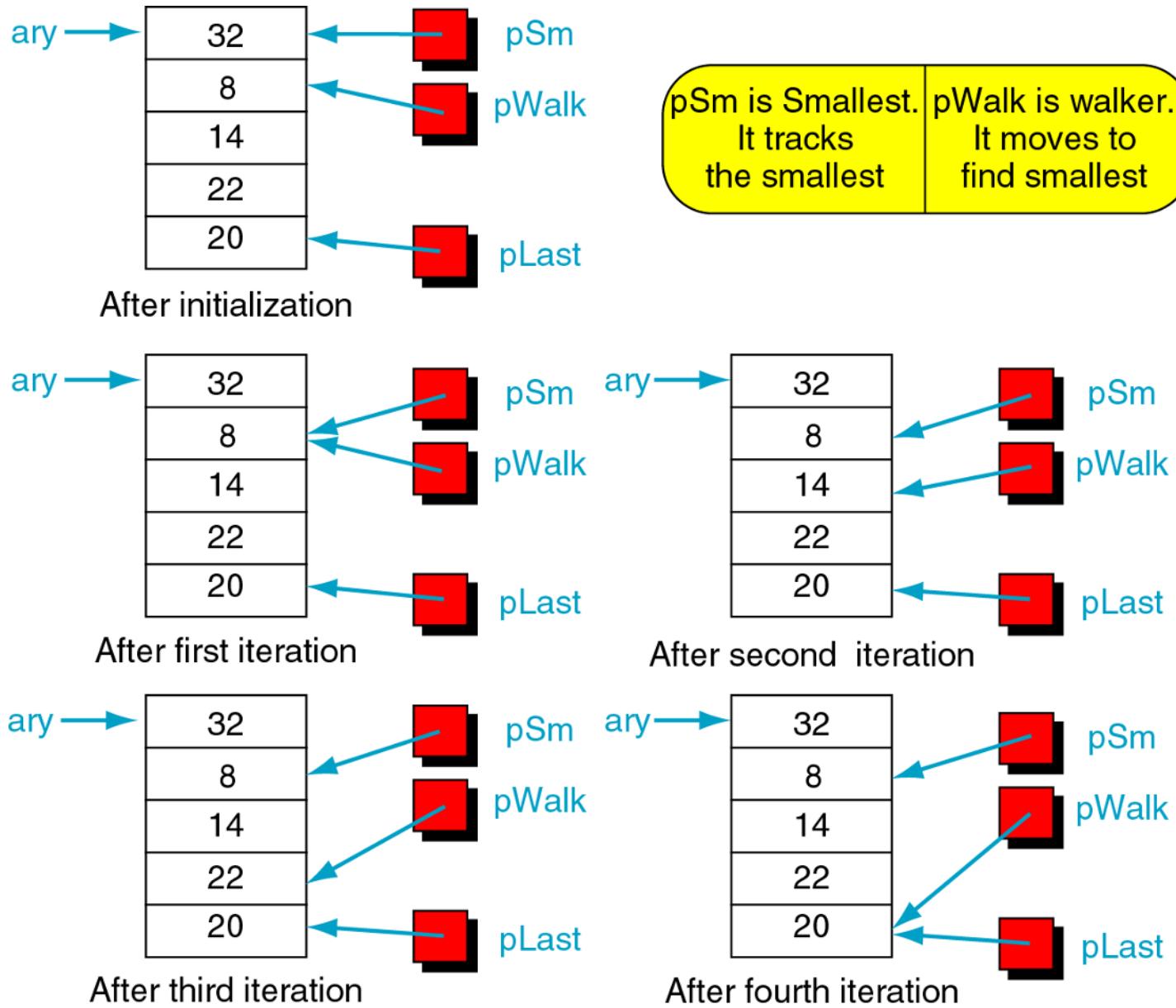


Figure 11-8 Find smallest

```
pLast = ary + arySize - 1;  
for (pSm = ary, pWalk = ary + 1;  
     pWalk <= pLast;  
     pWalk++)  
    if (*pWalk < *pSm)  
        pSm = pWalk;
```

```
#include <stdio.h>
#define arrzy_size 5
int main()
{
    int a[]={32,8,14,22,20};
    int *psm,*plast,*pwalk;
    plast=a+arrzy_size-1;psm=a;
    for(pwalk=a+1;pwalk<=plast;pwalk++)
        if(*pwalk<*psm)
            psm=pwalk;
    printf("the smallest element %d",*psm);
    return 0;
}
```

Arithmetic Operations on Pointers

- **Arithmetic operations involving pointers**
- Addition can be used when one operand is a pointer and the other is an integer
- Subtraction can be used only when both operands are pointers or when one operand is a pointer and the other is an index integer
- Note: result is meaningful only if the two pointers are associated with the same array structure
- Also, pointer with postfix and unary increment and decrement operators are valid
- Ex: $p+5$, $5+p$, $p-5$, p_1-p_2 , $p++$, $--p$ are valid

Pointers and other operators

- Relational operators involving pointers
- Allowed only if both operands are pointers of the same type
- Ex: pointer relational expns
- $P1 \geq p2$, $p1 \neq p2$
- If($\text{ptr} \neq \text{NULL}$) same as if(ptr)
- If ($\text{ptr} == \text{NULL}$) same as if($!\text{ptr}$)

Using Pointer Arithmetic

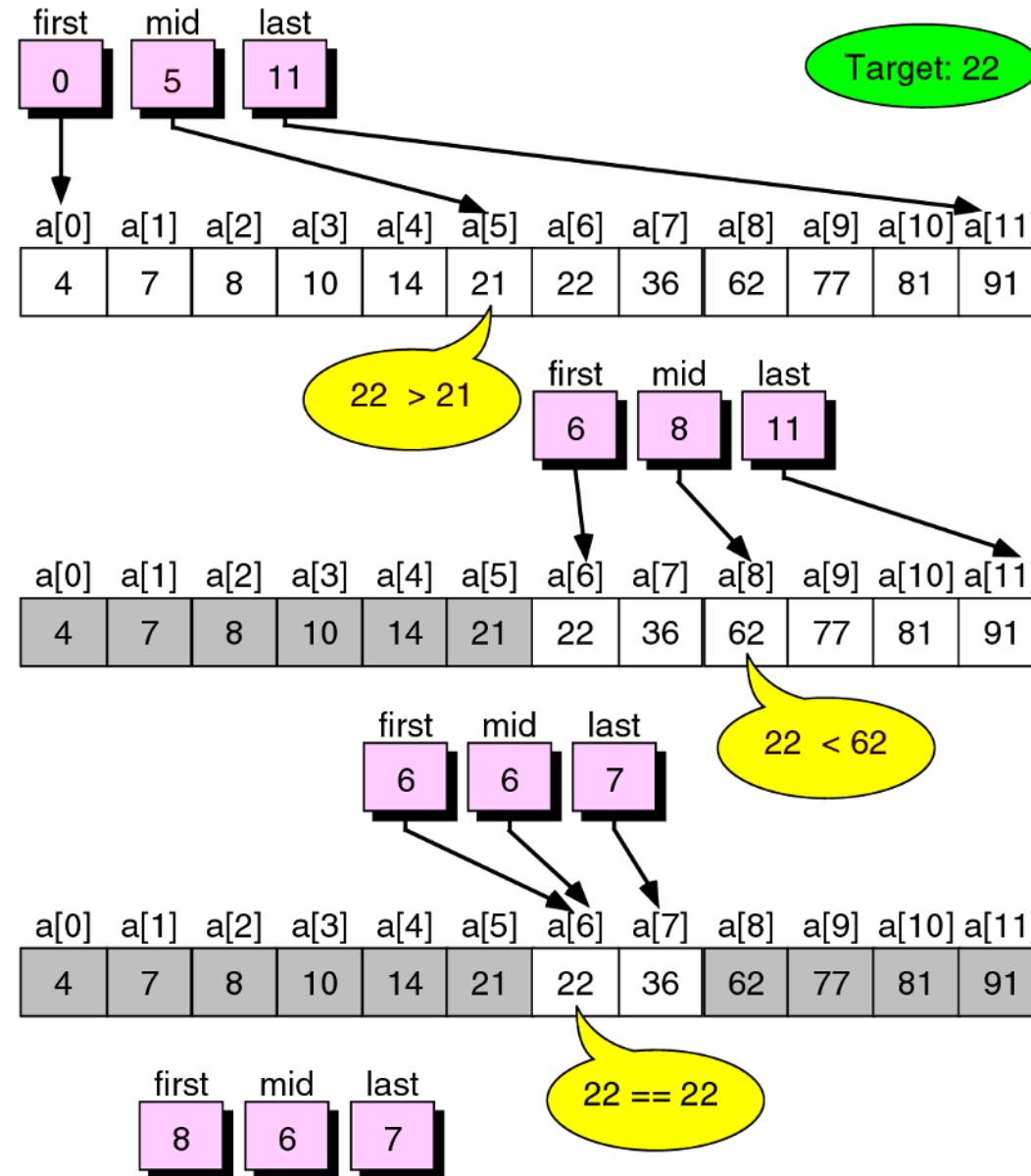
Using pointer arithmetic

- Program: demonstrate moving through array using pointers forward and backward

Program 11-1 Print Array with Pointers

```
#define MAX_SIZE 10
int main(void){
    int arr[] = {1, 2, --, 10};
    int *pWalk;
    int *pEnd;
    //print array forward
    for(pWalk = arr, pEnd = arr + MAX_SIZE;
        pWalk < pEnd; pWalk++)
        printf("%3d", *pWalk);
    printf("\n");
    //print array backward
    for(pWalk = pEnd - 1; pWalk >= arr; pWalk--)
        printf("%3d", *pWalk);
    printf("\n");
    return 0;
}
```

Figure 8-26 binary search example



Background on 2D arrays

```
#define MAX_ROWS 3
#define MAX_COLS 4

void print_square(int x[]) {
    int col;
    for (col = 0; col < MAX_COLS; col++)
        printf("%d\t", x[col] * x[col]);
    printf("\n");
}

void main(void) {
    int row;
    int num[MAX_ROWS][MAX_COLS] = { { 0, 1, 2, 3 },
                                    { 4, 5, 6, 7 },
                                    { 8, 9, 10, 11 } };

    for (row = 0; row < MAX_ROWS; row++)
        print_square(num[row]);
}
```

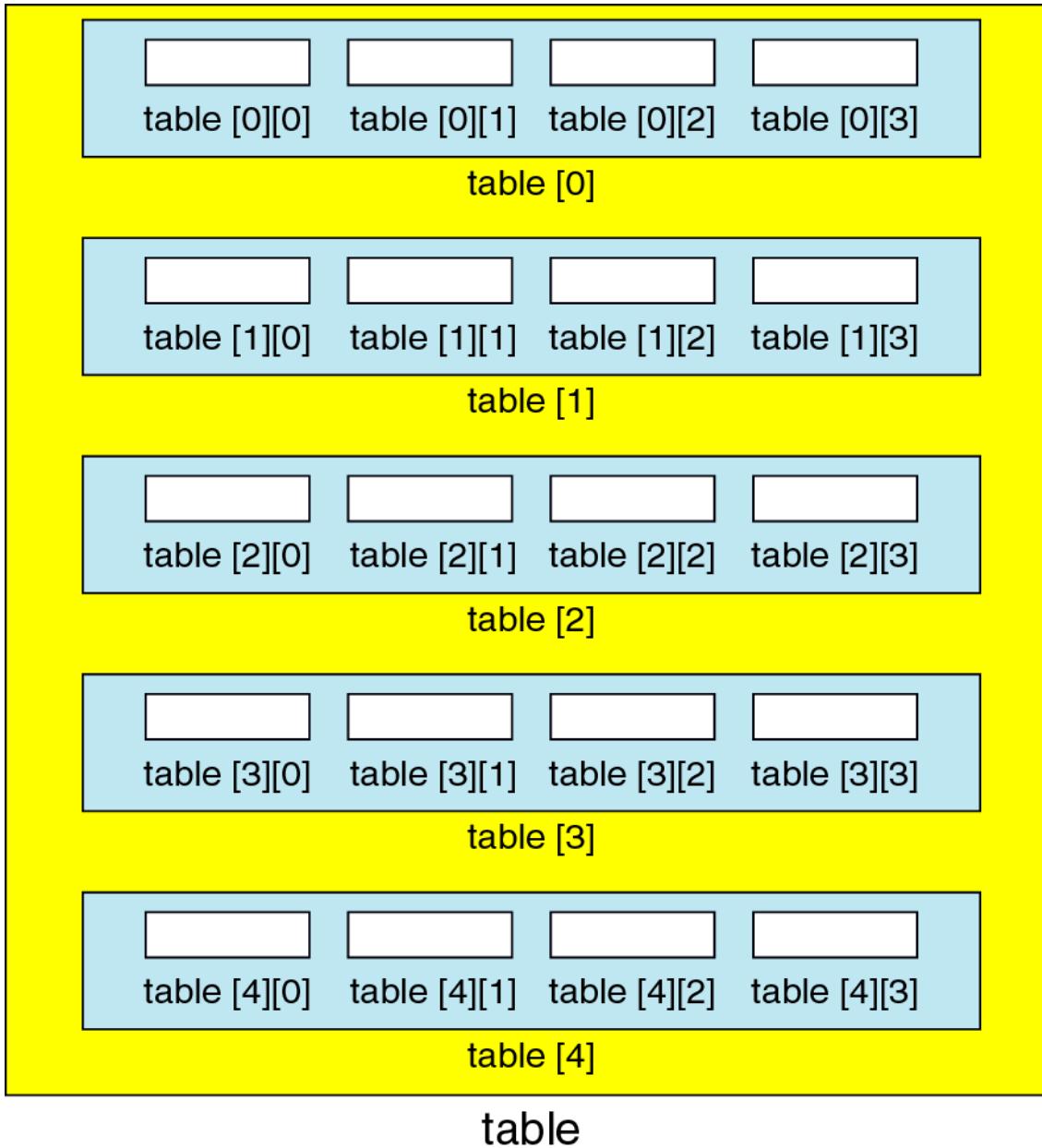
```
#include<stdio.h>
#include<conio.h>

#define MAX_ROWS 3
#define MAX_COLS 4

float print_square(int x[][MAX_ROWS]) {
    int row,col,sum=0;
    for (row = 0; row < MAX_ROWS; row++)
        for (col = 0; col < MAX_COLS; col++)
            sum+=x[row][col];
    return ((float)sum/(MAX_ROWS*MAX_COLS));
}

void main(void) {
    float avg;
    int table[MAX_ROWS][MAX_COLS] = { { 0, 1, 2, 3 },
                                      { 4, 5, 6, 7 },
                                      { 8, 9, 10, 11 } };
    avg=print_square(table);
    printf("%3f",avg);
    return 0;
}
```

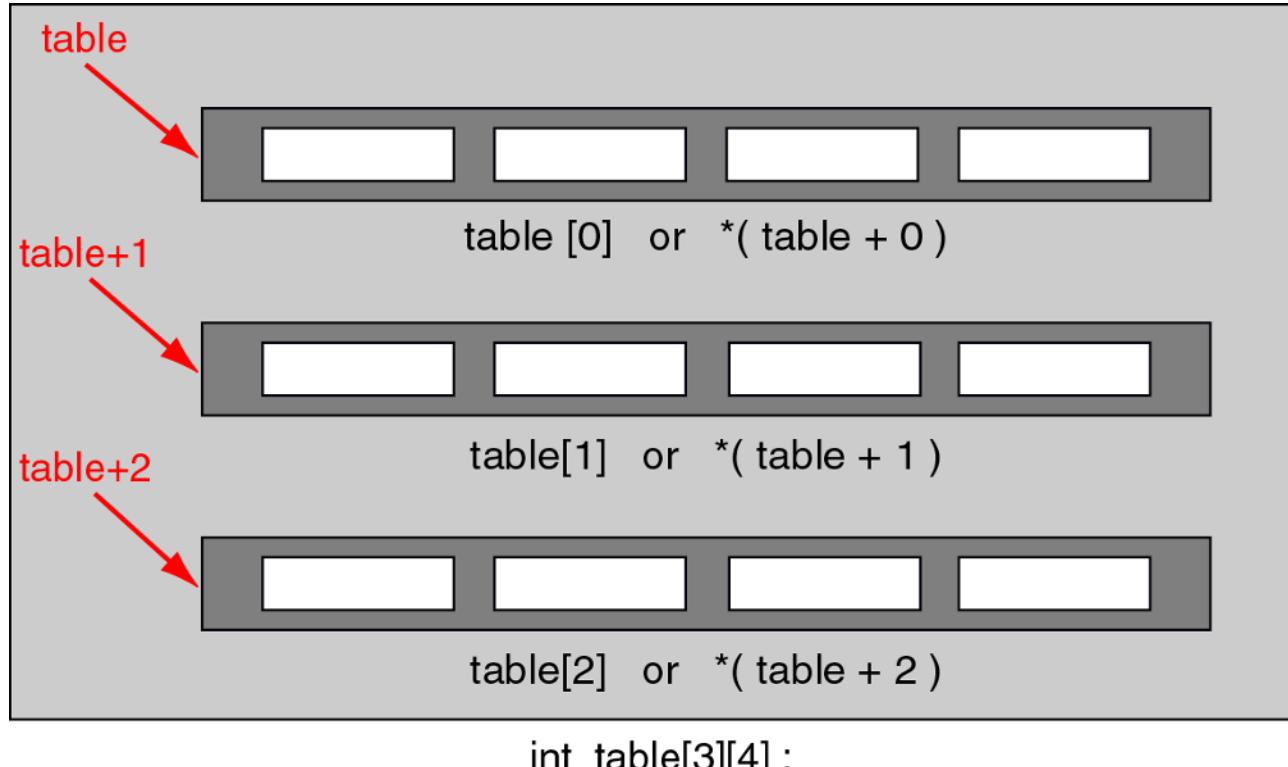
Pointers to 2D arrays



Pointers to 2D arrays

- Just as in 1D array, the name of the array is a pointer constant to the first element of the array
- Here, the first element is another array
- Suppose, we have an array of integers
- For a 2D array, when we dereference the array name, we get an array of integers (we do not get one integer)
- So, dereferencing of the array name of a 2D array is a pointer to a 1D array

Figure 10-9 Pointers to 2D arrays



```
for (i = 0; i < 3; i++)
{
    for (j = 0; j < 4; j++)
        printf("%6d", *(*(table + i) + j));
    printf("\n");
} /* for i */
```

Print table

table

0	1	2
0	0001	0002
1	1	2
2	3	4
		5

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

#define MAX_ROWS 3
#define MAX_COLS 4
void main(void) {
    int i;
    int table[MAX_ROWS][MAX_COLS] = { { 0, 1, 2, 3 },
                                      { 4, 5, 6, 7 }, { 8, 9, 10, 11 } };
    int *ptr=table;
    for(i=0;i<MAX_ROWS*MAX_COLS;i++)
        printf("%3d",*(ptr+i));
    return 0;
}
```

Pointers to 2D arrays

- In the figure, each element is shown in both index and pointer notation
 - `table[0]`: refers to an array of 4 integer values
 - `*(table+0)` : equivalent pointer notation is by dereferencing the array name
- Program: demonstrate pointer manipulation with 2D array by printing the table
 - To refer to a row: dereference the array pointer which gives us a pointer to a row
 - Given a pointer to a row, to refer to an element, dereference the row pointer
 - So it leads to double dereference as `*(*table)`

Pointers to 2D arrays

- The double dereference `*(*(table))` : refers only to the first element of the first row
- To step through all the elements, we need 2 offsets:
 - One for the row
 - Another for the element within the row
- i and j: loop counters for offsets
- Same logic as printing 2D array using indexes
 - `table [i][j]`: index notation
 - `*(*(table+i)+j)` : pointer notation
- Note: With multi-dimensional arrays, the pointer arithmetic is not efficient over indexing
- As it is complex, index notation is preferred

Passing an array to a function

- If array name is a pointer to the first element, we can pass the array name to a function
- When we pass the array name, no need to use the address operator
- The array name is a pointer constant, so the array name is already the address of the first element in the array
- Ex: `compute(list)` // calling program
- 1) `int compute(int ary[])` // 3 different syntax for called program

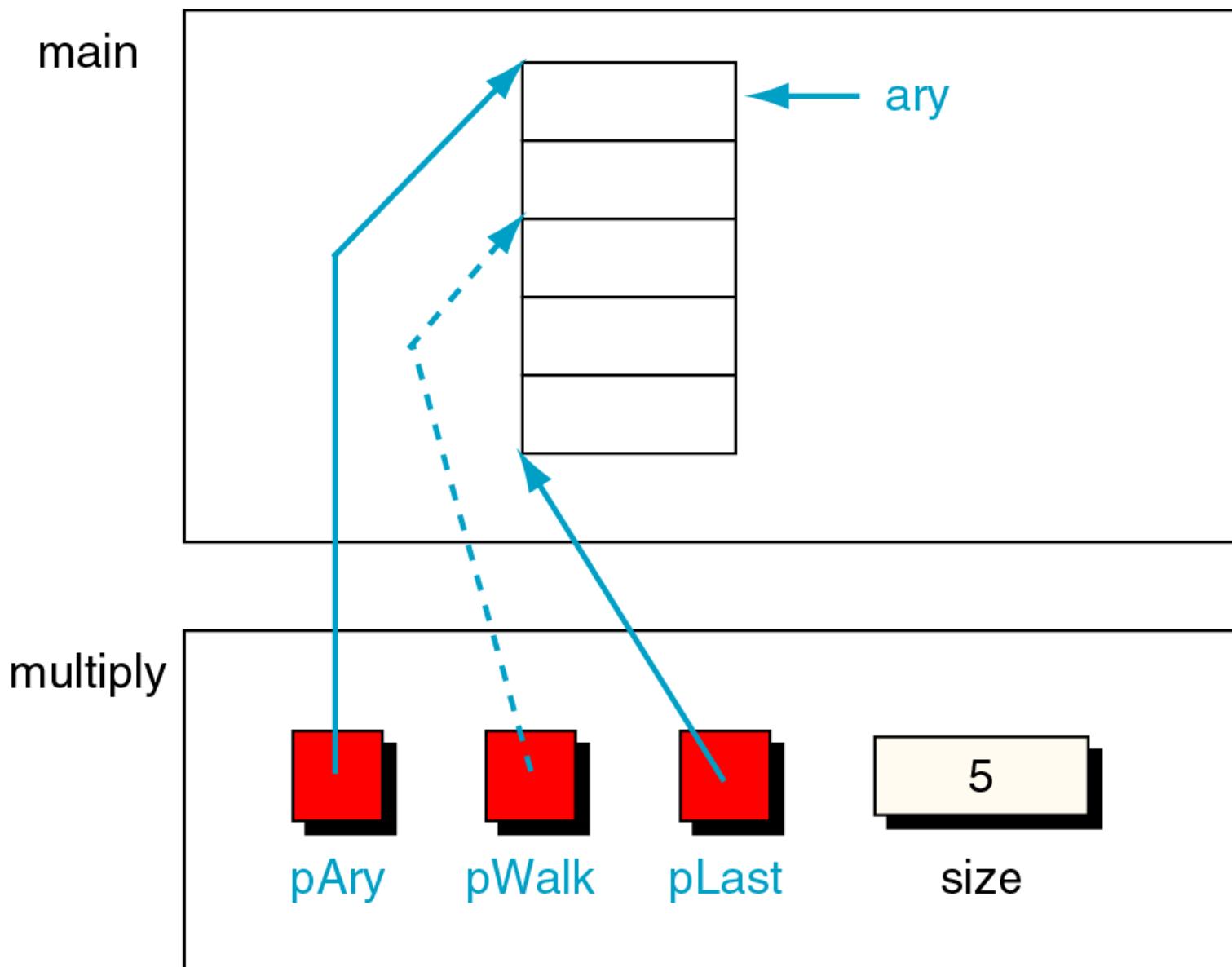
Passing an array to a function

- 2) int compute(int *arySalary) //needs documentation that array is passed
- 3) for a 3D array int compute(int bigary[][][5]) //the compiler needs to know the size of dimension after the first to calculate the offset for pointer arithmetic

Passing an array to a function

- Program: call a function to multiply each element of a 1D array by 2
- The variables are shown next

Figure 10-10 Variables for multiply array elements by 2



```
void multiply(int *pAny, int size);
int main(void) {
    int any [SIZE];
    int *pLast;
    int *pWalk;
    pLast = any + SIZE - 1;
    for (pWalk = any; pWalk <= pLast; pWalk++) {
        printf("Enter an integer:");
        scanf("%d", pWalk);
    }
    multiply (any, SIZE);
    printf("Doubled value is: \n");
    // statements for printing the array
}
```

5

```
void multiply(int *pAny, int size){  
    int *pWalk;  
    int *pLast;  
    pLast = pAny + size - 1;  
    for (pWalk = pAny; pWalk <= pLast; pWalk++)  
        *pWalk = *pWalk * 2;  
}
```

6

Memory Allocation Functions

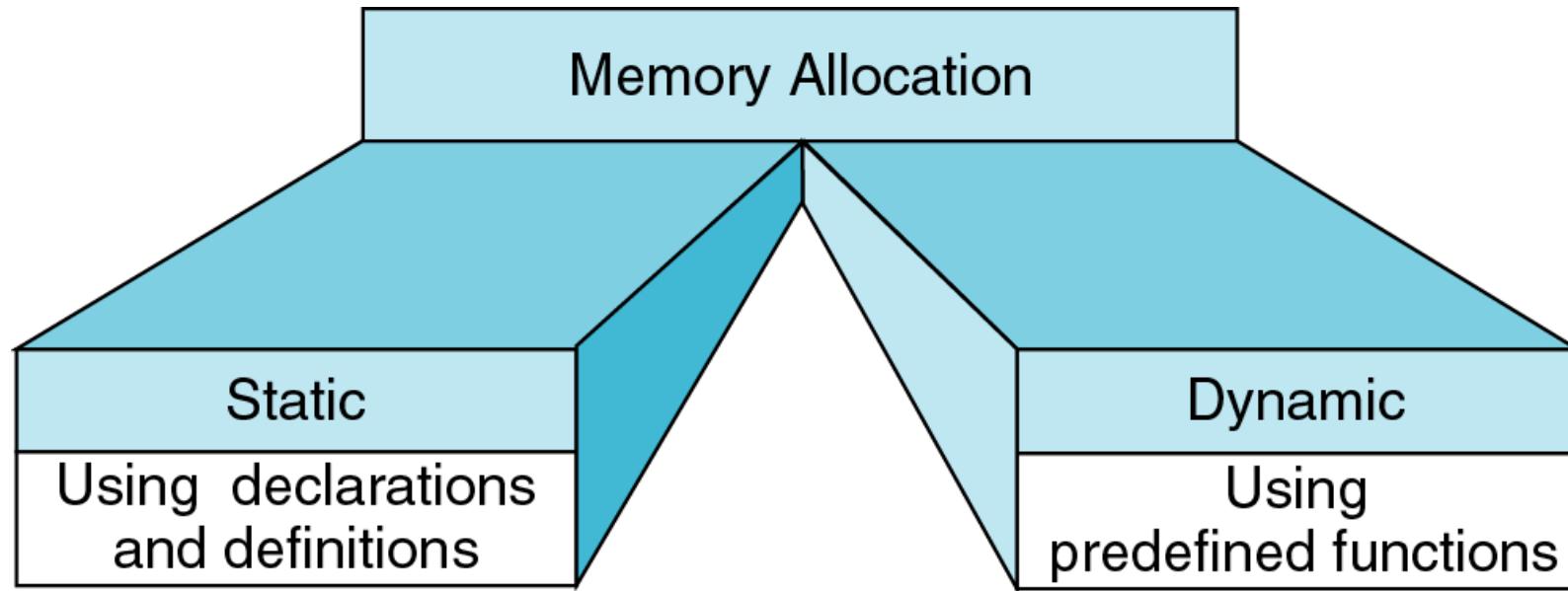
Memory Allocation Functions

- Two choices to reserve memory locations for an object:
- 1. Static memory allocation
- Requires that the declaration and definitions of memory be fully specified in the source program
- The number of bytes reserved cannot be changed during run time
- Works fine if the data requirements are exactly known
- This is what we have done so far
 - Note: earlier PLs (FORTRAN, COBOL) had this limitation. The data structure need to be fully specified during compile time

Memory Allocation Functions

- 2. Dynamic memory allocation
- Uses predefined functions to allocate and release memory for data while the program is running
- Postpones the data definition to run time
- Can be used either with standard data types or with derived data types

Figure 10-13 Memory Allocation



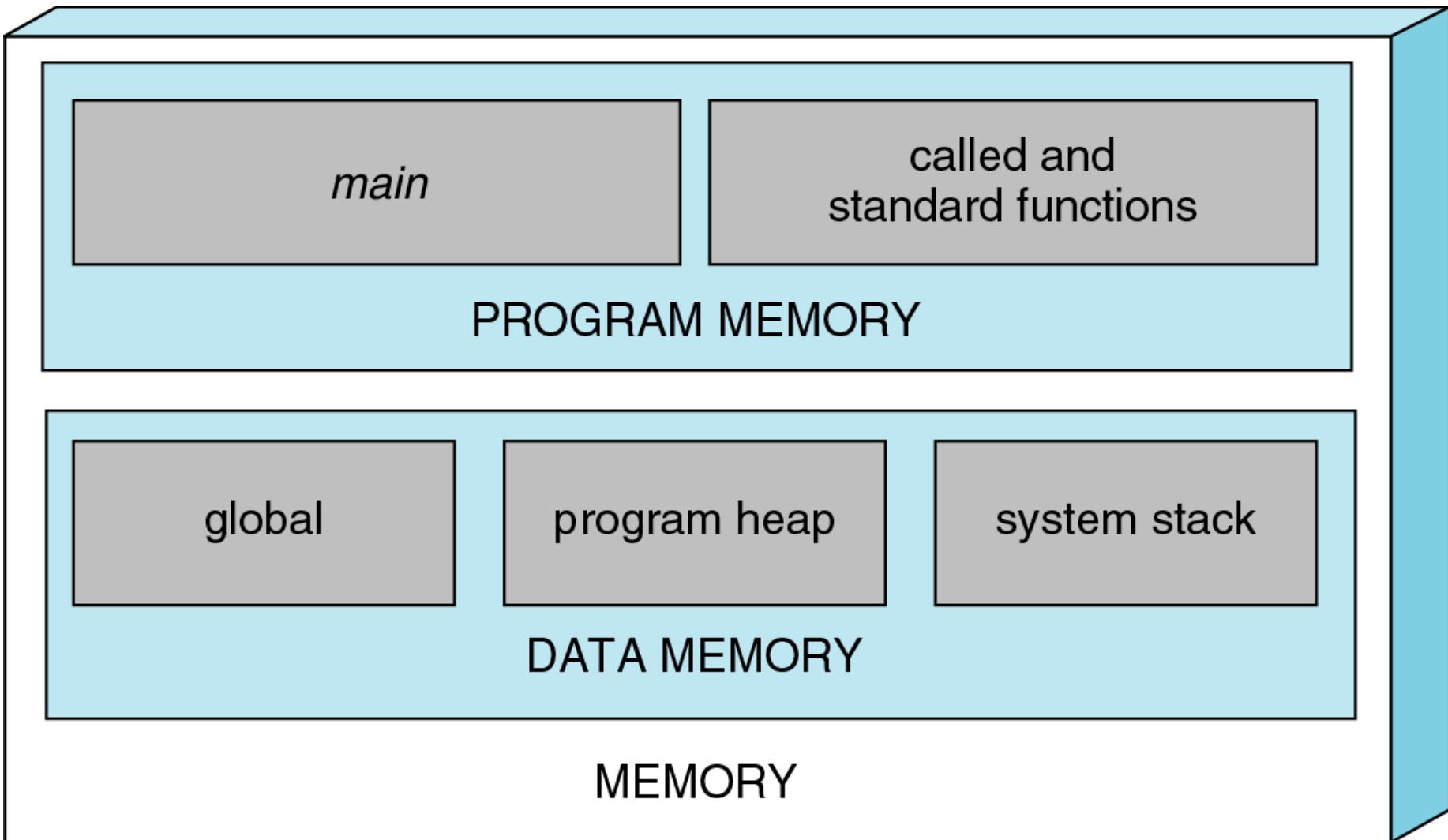
```
char carr[]="D Section";
char *cptr=" Students";
char c[10]="are good";
char *cptr1=c;
char char1[4][9]={{“hello”},{“students”},{“are you”},{“there”}}
```

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char c1[4][9]={"hello","students","are you","there"};
    printf("%s\n",c1[0]);
    printf("%u\n",sizeof(c1[0]));
    printf("%s",*(c1+1));
}
```

Understanding dynamic Memory Allocation

- Conceptually memory is divided into program memory and data memory

Figure 10-15 A conceptual View of memory



- Note: implementation of memory is left to the designers who design the system
- Stack and heap can share the same pool of memory
- So the above diagram is only conceptual

Understanding dynamic Memory Allocation

- program memory:
- Consists of the memory used for main and all called functions
- data memory:
- Consists of permanent definitions such as global data and constants, local definitions and dynamic data memory
- Note: Exact ways of handling these needs will be decided by the OS and compiler

Understanding dynamic Memory Allocation

- program memory:
- Consists of the memory used for main and all called functions
- main() must be in memory all the time
- Each called function must be in memory while it is active
- Note: any function has to be in memory when it is running

Understanding dynamic Memory Allocation

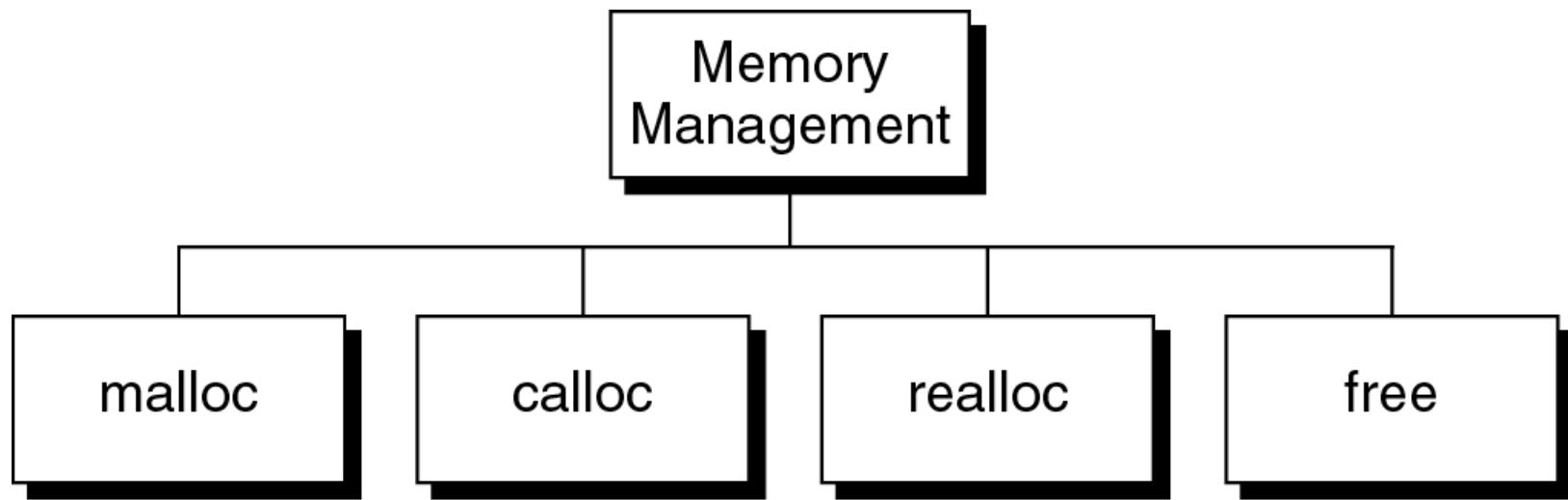
- **Memory allocation : stack**
- Although the program code for a function may be in memory at all times
 - The local variables for the function are available only when it is active
- In recursion, more than one version of the function can be active at a time
 - Hence multiple copies of the local variables are allocated although only one copy of the function is present
- The memory facilities for these capabilities is known as **stack**

Understanding dynamic Memory Allocation

- **Memory allocation : heap**
- Heap is unused memory allocated to the program
- and available to be assigned during its execution
- The memory pool from which memory is allocated when requested by the memory allocation functions

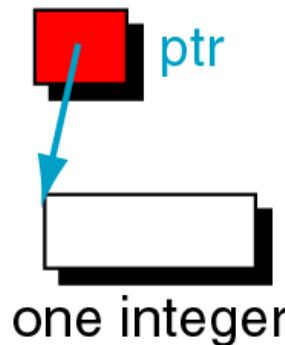
Figure 10-14 Memory Management Functions

```
int arr[200]
```



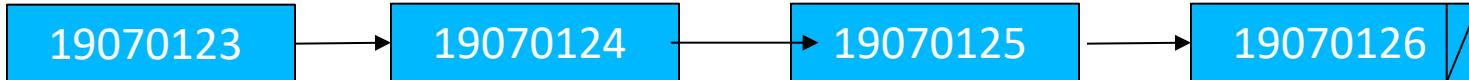
Note: all memory management functions are defined in <stdlib.h>

Figure 10-16 malloc (memory allocation)



```
if (!(ptr = (int *)malloc(sizeof(int))))  
    /* No memory available */  
    exit (100) ;  
    /* Memory available */  
    ...
```

- Note: Allocates a block of memory that contains the number of bytes specified as the parameter
- Returns a void pointer to the first byte of the allocated memory
- The allocated memory is uninitialized (contains garbage)
- To allocate an integer in the heap
 * ptr = (int *) malloc (sizeof (int));
- The pointer returned by malloc is shown to be casted as an integer
- For portability use sizeof()

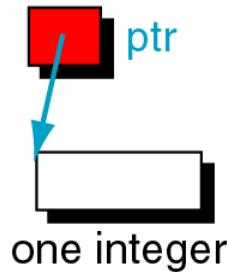


malloc

- **Successful call**
- malloc returns the address of the first byte in the memory allocated
- **Unsuccessful call**
- malloc returns a NULL pointer
- Overflow – attempt to allocate memory from the heap when memory is insufficient
- Note: Refer to the memory allocated in the heap only through a pointer
1/9/2022 does not have its identifier

malloc

- Allocating one integer object
- If the memory allocated if successful, ptr contains a value
- Else, no memory then exit with error code 100
- malloc cannot be called with zero size (results are unpredictable, may return a NULL pointer or some other value)



```
if (!(ptr = (int *)malloc(sizeof(int))))  
    /* No memory available */  
    exit (100) ;  
    /* Memory available */  
...
```

- Note: the program has to check for memory overflow
- If it does not, the program produces invalid results

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr= (int *) malloc(sizeof(int));
    if (ptr==NULL) exit(100);
    scanf("%d",ptr);
    printf("%d",*ptr);
}
```

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr=(int *) malloc(2*sizeof(int));
    if (ptr==NULL) exit(100);
    scanf("%d",ptr);
    printf("%d\n",*ptr);
    ptr++;
    *ptr=2456;
    printf("%d",*ptr);
}
```

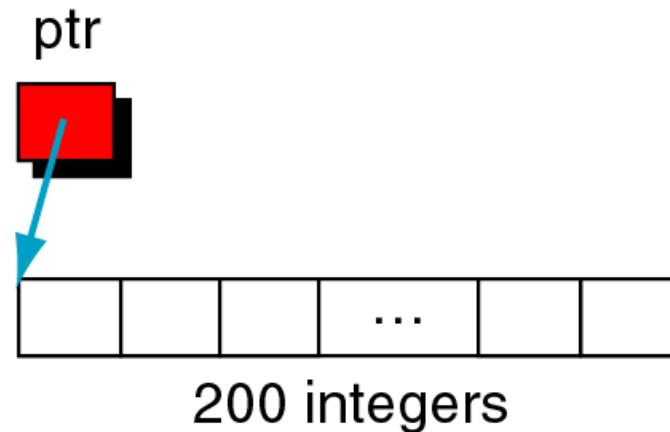
calloc (contiguous memory allocation)

- To allocate memory for arrays
- 1. allocates a contiguous block of memory to contain an array of elements of a specified size
- Parameters – number of elements to be allocated, size of each element
- 2. Returns a pointer to the first element of the allocated array
- Since it is a pointer to an array, the size associated with its pointer is the size of one element not the entire array

calloc (contiguous memory allocation)

- 3. calloc clears memory. So allocated memory is set to zero
- The result is same for malloc and calloc for overflow and zero size

Figure 10-17 calloc



```
if (!(ptr = (int *)calloc (200, sizeof(int))))  
    /* No memory available */  
    exit (100) ;
```

```
/* Memory available */  
...
```

- Allocating memory for an array of 200 integers

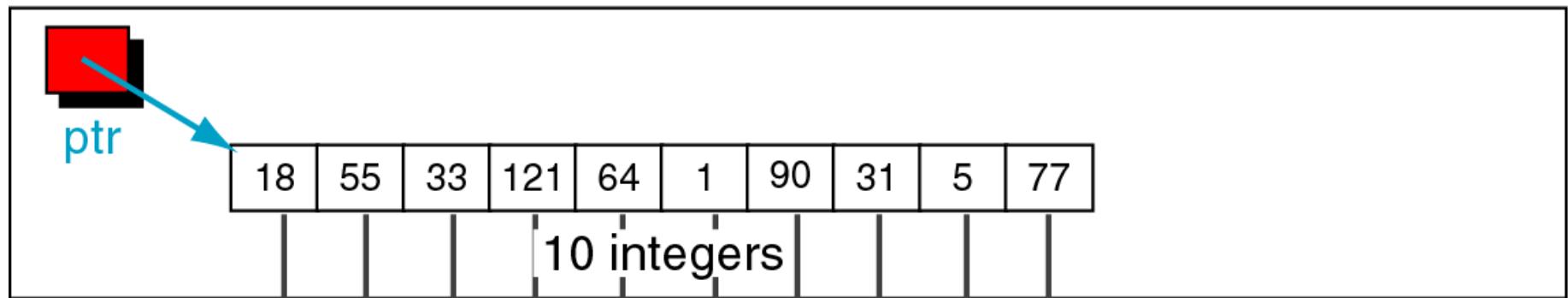
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i;
    int *ptr=(int *) calloc(5,sizeof(int));
    if (ptr==NULL) exit(100);
    for(i=0;i<5;i++)
        scanf("%d",(ptr+i));
    for(i=0;i<5;i++)
        printf("%d\n",*(ptr+i));
    //for(i=0;i<5;i++)
    //printf("%d\n",ptr[i]);
}
```

realloc (reallocation of memory)

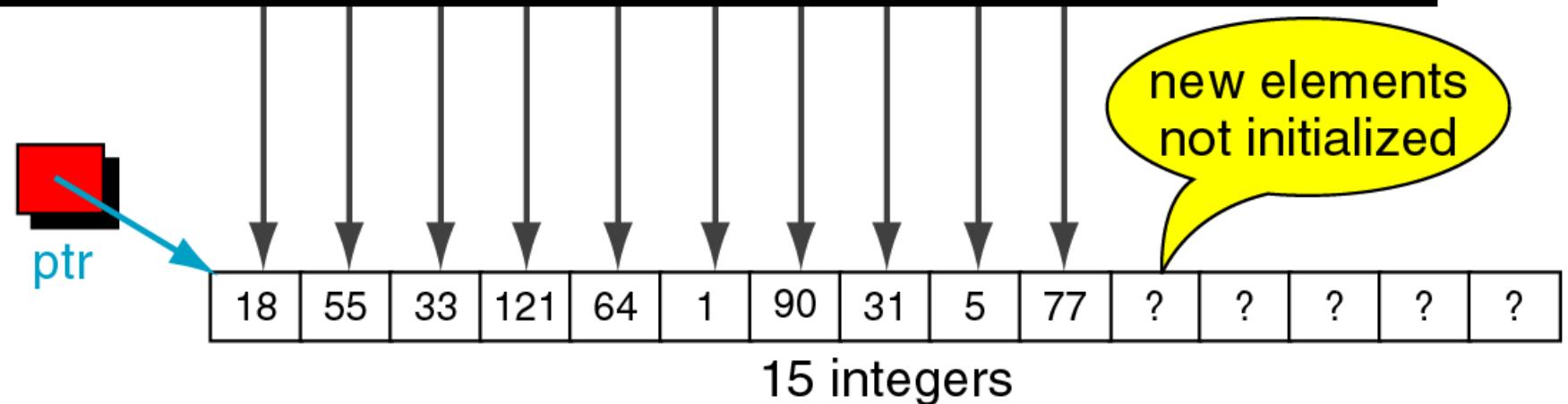
- Given a pointer to a previously allocated block of memory,
 - realloc changes the size of the block by deleting or extending the memory at the end of the block
 - If the memory cannot be extended (due to other allocations), realloc allocates a completely new block
 - Copies the existing memory allocation to the new allocation and deletes the old allocation
- The programmer must ensure that other pointers to the data are correctly changed.
- **Highly inefficient and to be used carefully**

Figure 10-18 realloc

BEFORE



```
ptr = (int *)realloc (ptr, 15 * sizeof(int));
```



AFTER

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

int main()
{
    int i;
    int *ptr=(int *) calloc(5,sizeof(int));
    if (ptr==NULL) exit(100);
    for(i=0;i<5;i++)
        scanf("%d",(ptr+i));
    for(i=0;i<5;i++)
        printf("%d\n",*(ptr+i));
    ptr=(int *)realloc(ptr,10*sizeof(int));
    for(i=0;i<9;i++)
        printf("\n%d\n",ptr[i]);
    ptr[9]=120;
    printf("\n%d\n",ptr[9]);
}
```

11

12

23

34

45

Before reallocation

11

12

23

34

45

After reallocation

11

12

23

34

45

1348221507 unassigned (garbage)

1919381362

1176530273

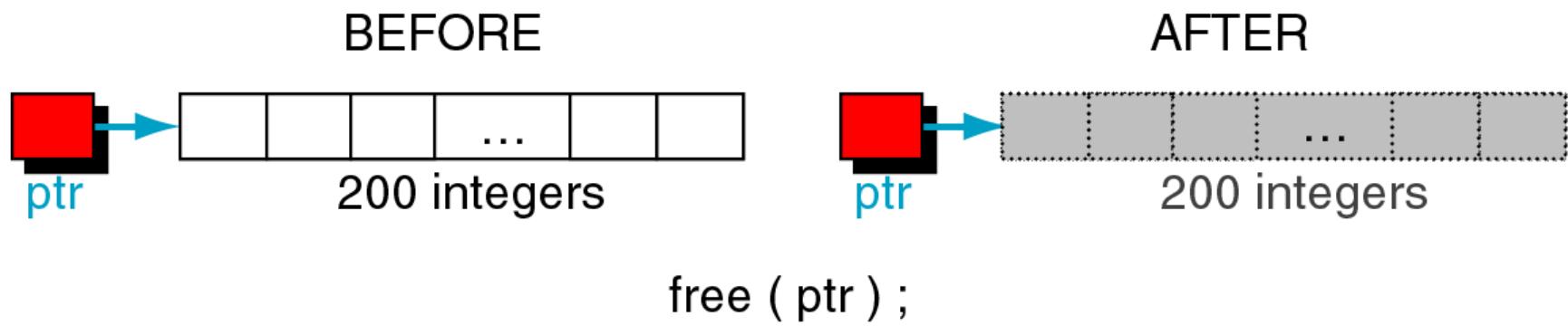
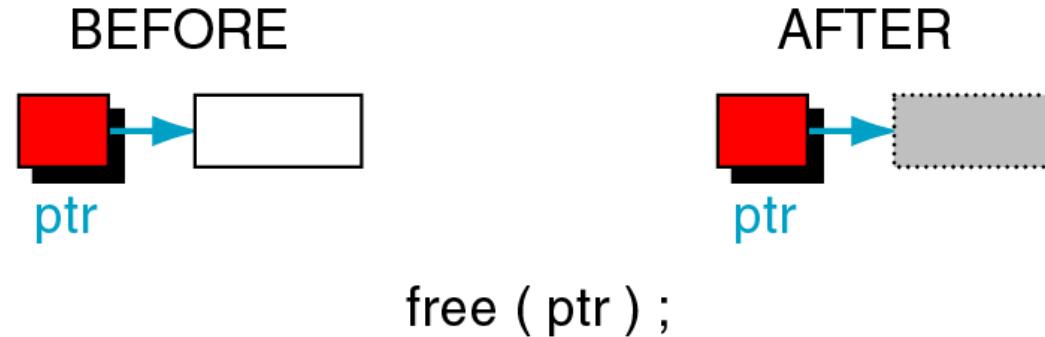
1936026729

120 Assigned value

free (releasing memory)

- When memory allocated by malloc, calloc, realloc are no longer needed
 - They can be freed using free
- It is an error to free memory
 - With a null pointer or
 - A pointer to other than the first element of an allocated block
 - A pointer that is a different type than the pointer that allocated the memory
 - To refer to memory after it has been released

Figure 10-19 Freeing memory examples



- Ex1: to release a single element allocated with a malloc back to the heap
- Ex2: : to release a 200 element array allocated with a calloc, all 200 elements are returned back to the heap

free (releasing memory)

- It is not the pointers that are being released but what they point to
- To release an array of memory that was allocated by calloc, release the pointer only once
 - Releasing memory does not change the value in a pointer
 - It still contains the address in the heap
 - It is a logical error to use the memory once it is released
 - After freeing the memory, clear the pointer by setting to NULL (to ease debugging)

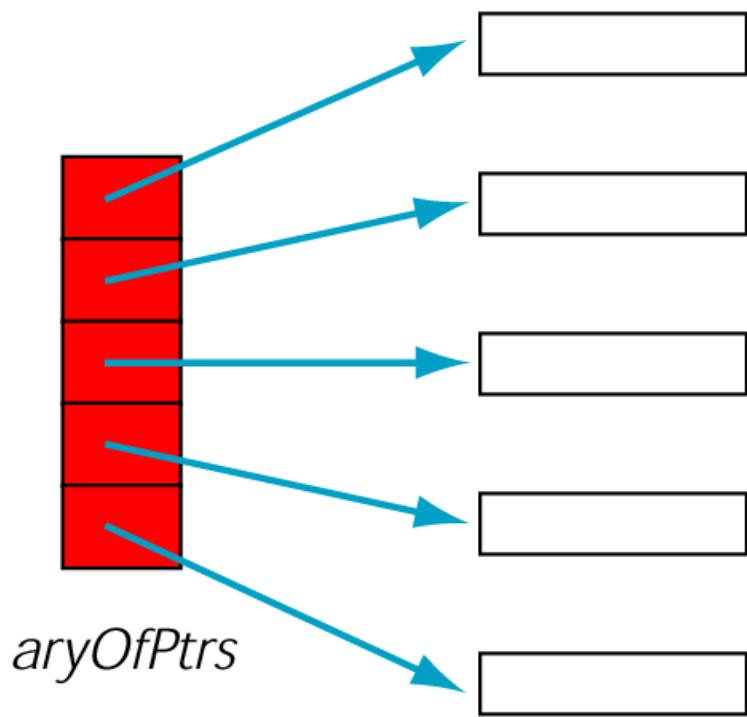
free (releasing memory)

- It is not necessary to clear memory at the end of the program
- OS will release all memory when the program is terminated
- So, free memory whenever it is no longer needed

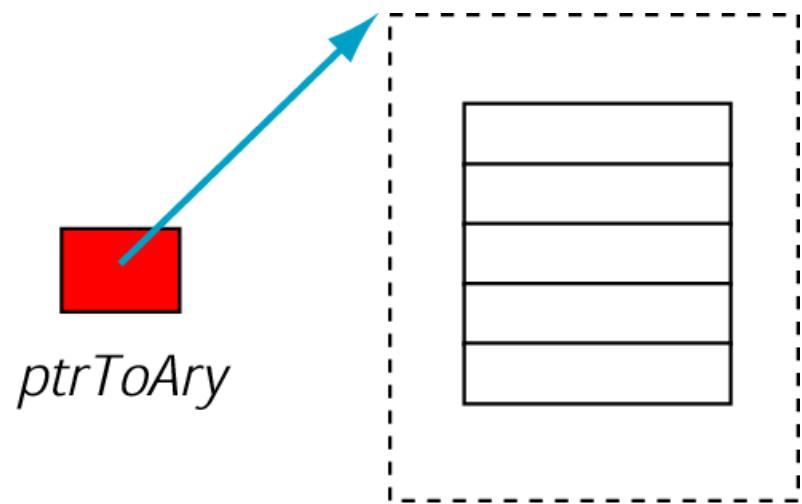
Array of Pointers

- Useful data structure
- Needed when the size of the data in the array is variable
- **Ragged arrays**
- 2D arrays with an uneven right border
- The rows contain different number of elements (size zero to max)
- 2d array wastes a lot of memory for this structure
- So, create n number of 1D arrays that are joined through array of pointers

Figure 10-12 Array of pointers vs. Pointer to Array

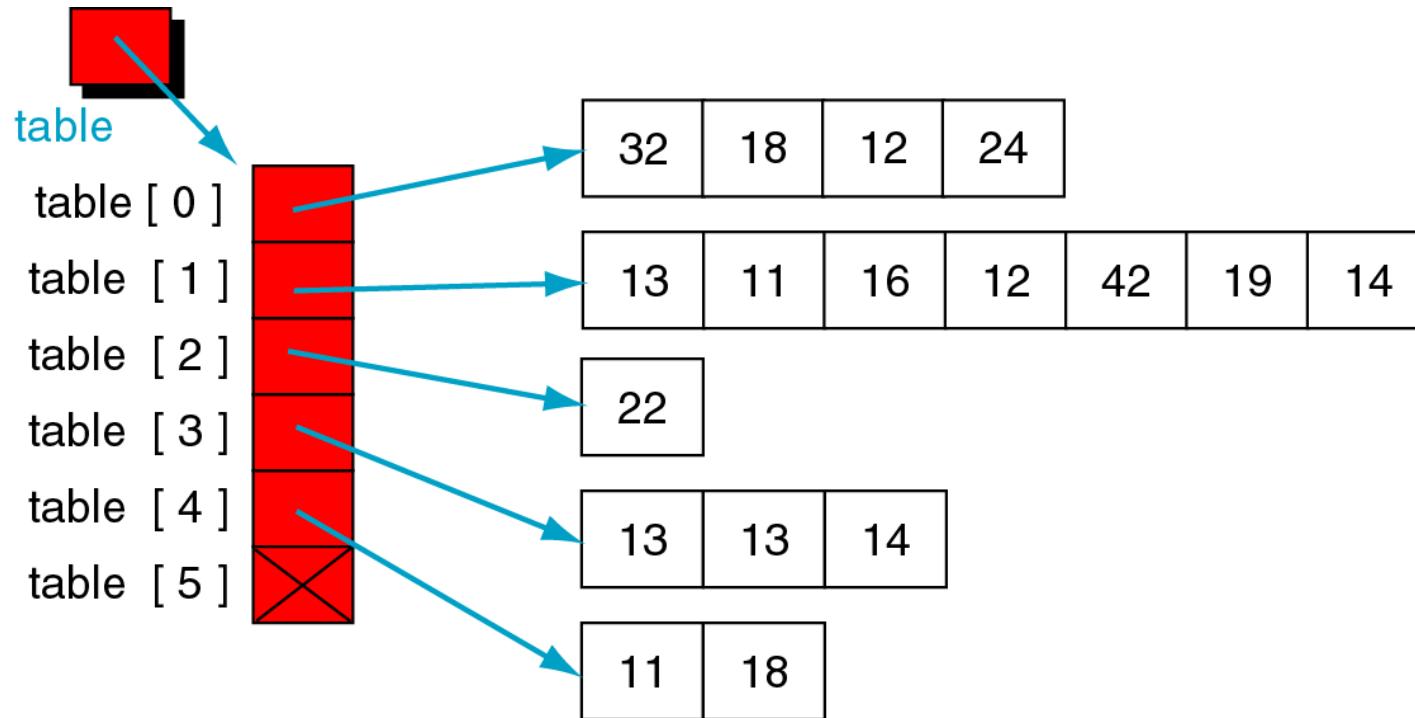


(a) An array of pointers



(b) A pointer to an array

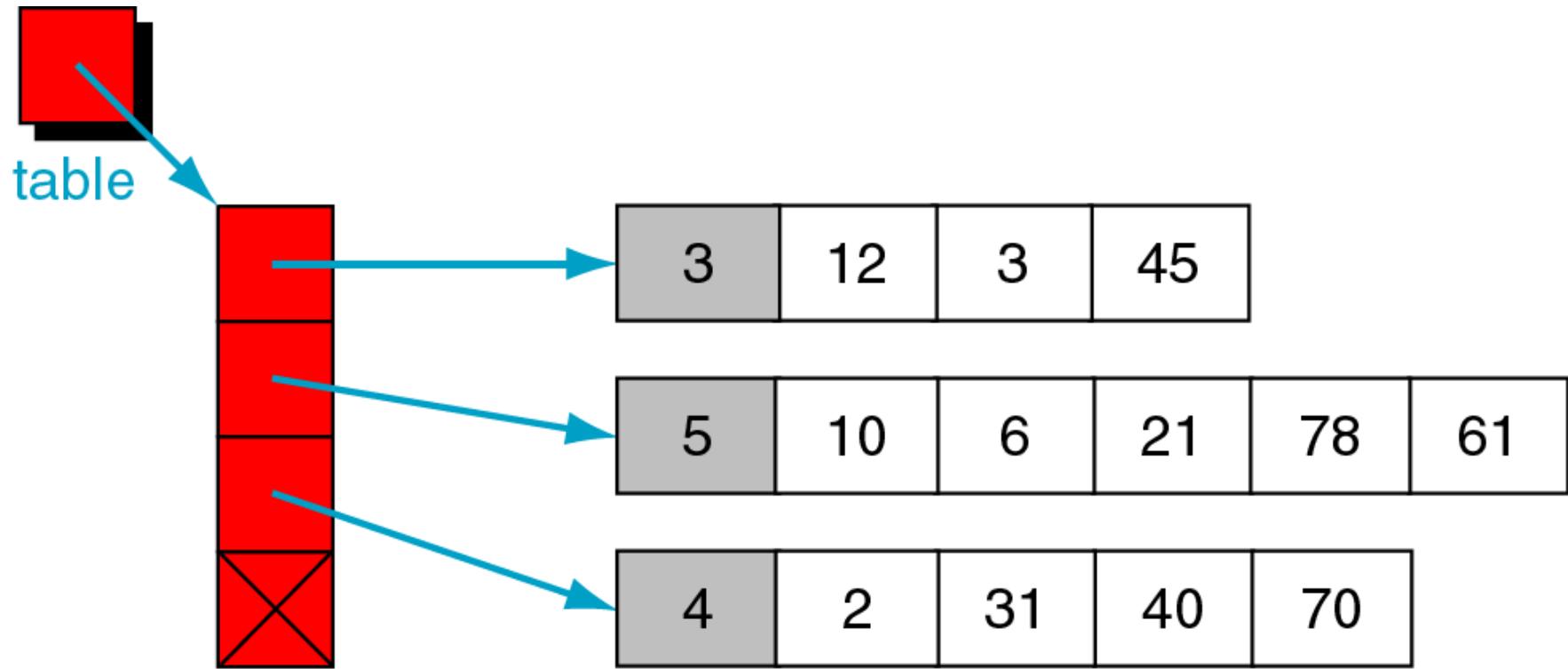
Figure 10-20 A Ragged Array



```
int ** table;
```

```
table = (int **)calloc (rowNum + 1, sizeof(int*));  
  
table[0] = (int*)calloc (4, sizeof(int));  
table[1] = (int*)calloc (7, sizeof(int));  
table[2] = (int*)calloc (1, sizeof(int));  
table[3] = (int*)calloc (3, sizeof(int));  
table[4] = (int*)calloc (2, sizeof(int));  
table[5] = NULL;
```

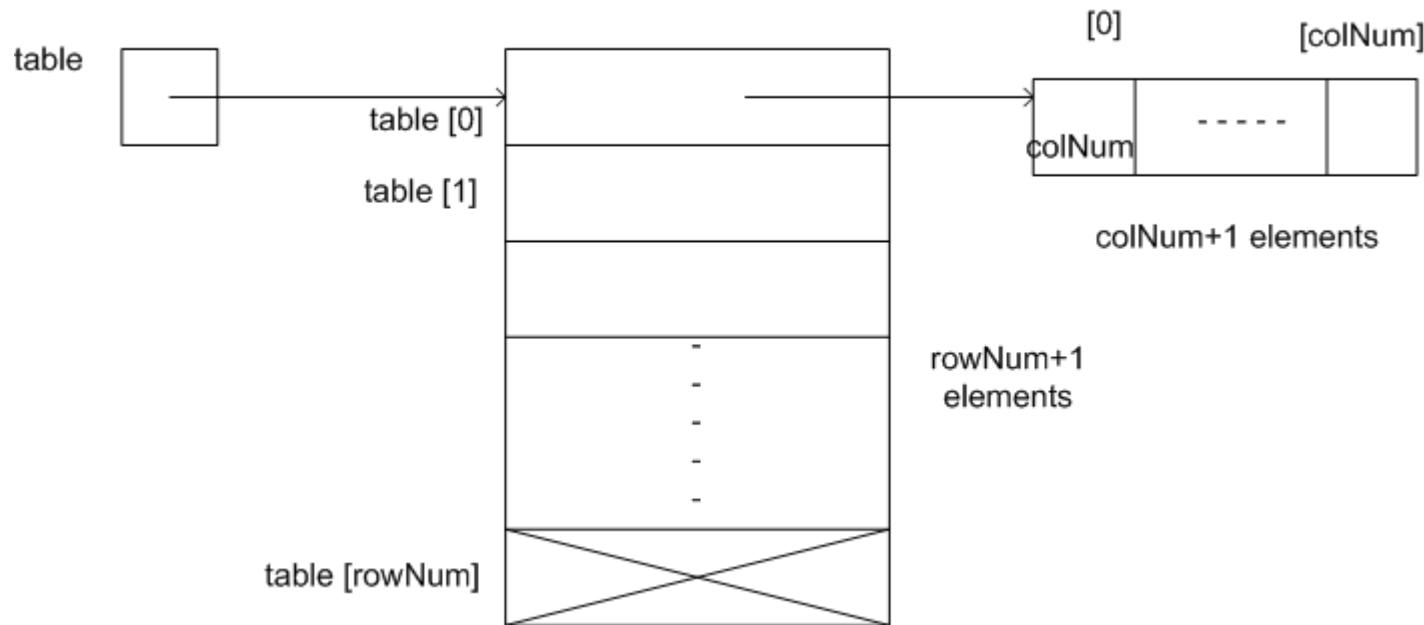
Figure 10-23 ragged array structure



Lab Problem: Construct a ragged array

- The *table* pointer points to the first pointer in an array of pointers.
- Each array pointer points to a second array of integers, the first element of which is the number of elements in the list.
- A sample ragged array structure is shown next

Lab Problem: Construct a ragged array



Lab Problem: Construct a ragged array

- step 1: Declare a ragged array as a variable *table*.
- step 2: Ask the user for row size and set a variable – *rowNum*
- step 3: Allocate space for $(rowNum+1)$ pointers as row pointers. The last row pointer will hold NULL
- step 4: Ask the user for column size and set a variable – *colNum*

Lab Problem: Construct a ragged array

- step 5: Allocate space for $(colNum+1)$ data elements. The first element will hold value contained in colNum itself.
- step 6: Repeat step 3 for all rows
- step 7 : Display ragged array contents.

Lab Problem: Construct a ragged array

```
# include<stdio.h>
# include<stdlib.h>
int main(){
    int rowNum, colNum, i, j;
    int **table;
    printf("\n enter the number of rows \n");
    scanf("%d", &rowNum);
    table = (int **) calloc(rowNum+1, sizeof(int *));
    for (i = 0; i < rowNum; i++) /* this will tell which row we are in */
    {
        printf("enter size of %d row", i+1);
```

Lab Problem: Construct a ragged array

```
scanf("%d", &colNum);
    table[i] = (int *) calloc(colNum+1, sizeof(int));
    printf("\n enter %d row elements ", i+1);
    for (j = 1; j <= colNum; j++)
    {
        scanf("%d", &table[i][j]);
    }
    table[i][0] = colNum;
    printf("size of row number [%d] = %d", i+1,
    table[i][0]); }
```

Lab Problem: Construct a ragged array

```
table[i] = NULL;  
for (i = 0; i < rowNum; i++) /* this will tell which row we  
are in */  
{  
    printf("displaying %d row elements\n", i+1);  
    for (j = 0; j <= *table[i]; j++)  
    {  
        printf("%5d", table[i][j]);  
    }  
    printf("\n");  
}  
1/9/2022  
return 0; }
```

Lab Problem: Construct a ragged array

Sample input and output:

enter the number of rows: 3

enter size of row 1: 4

enter row 1 elements: 10 11 12 13

enter size of row 2: 5

enter row 2 elements: 20 21 22 23 24

enter size of row 3

enter row 3 elements: 30 31 32

displaying

10 11 12 13

20 21 22 23 24

30 31 32

Pointer to an array

```
#include<stdio.h>
int main()
{
    int arr[] = { 3, 5, 6, 7, 9 };
    int *p = arr;
    int (*ptr)[5] = &arr;
    printf("\n p = %p, ptr = %p\n", p, ptr);
    printf("\n *p = %d, *ptr = %p\n", *p, *ptr);
    printf("\n sizeof(p) = %u, sizeof(*p) = %u\n",
           sizeof(p), sizeof(*p));
    printf("\n sizeof(ptr) = %u, sizeof(*ptr) = %u\n",
           sizeof(ptr), sizeof(*ptr));
    printf("%d\t%d", (*ptr)[0], (*ptr)[1]);
    P++;ptr++;
    return 0;
}
```

`p = 0060FEF4, ptr = 0060FEF4`

`*p = 3, *ptr = 0060FEF4`

`sizeof(p) = 4, sizeof(*p) = 4`

`sizeof(ptr) = 4, sizeof(*ptr) = 20`

`3`

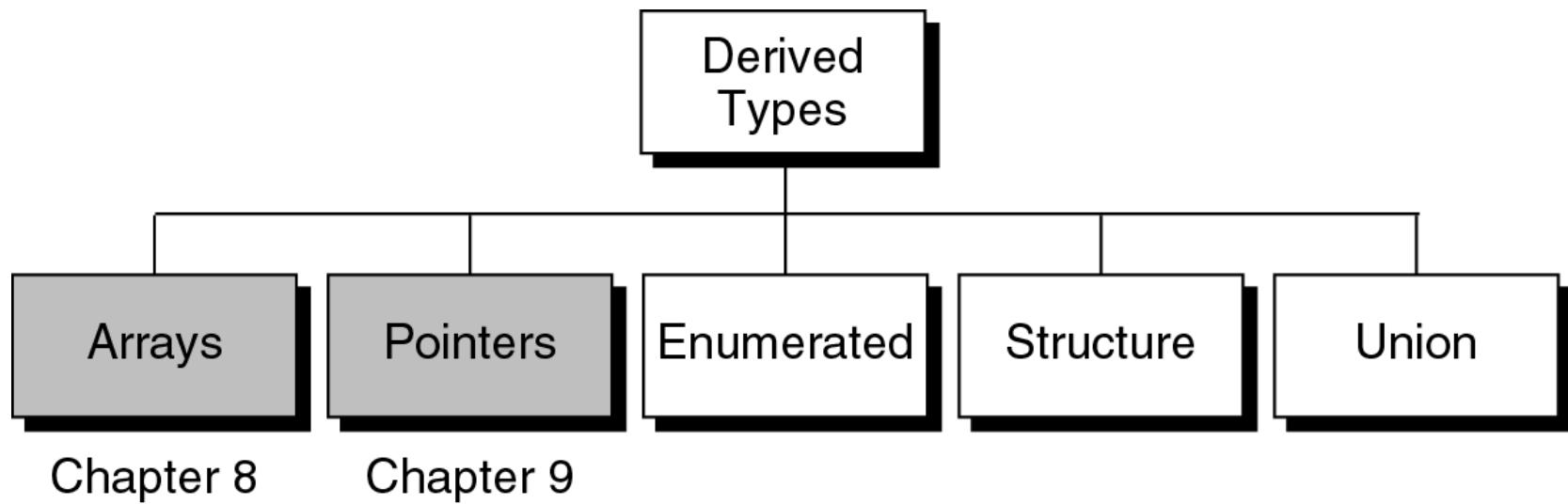
`5`

END

Chapter 12

Derived Types-- Enumerated, Structure

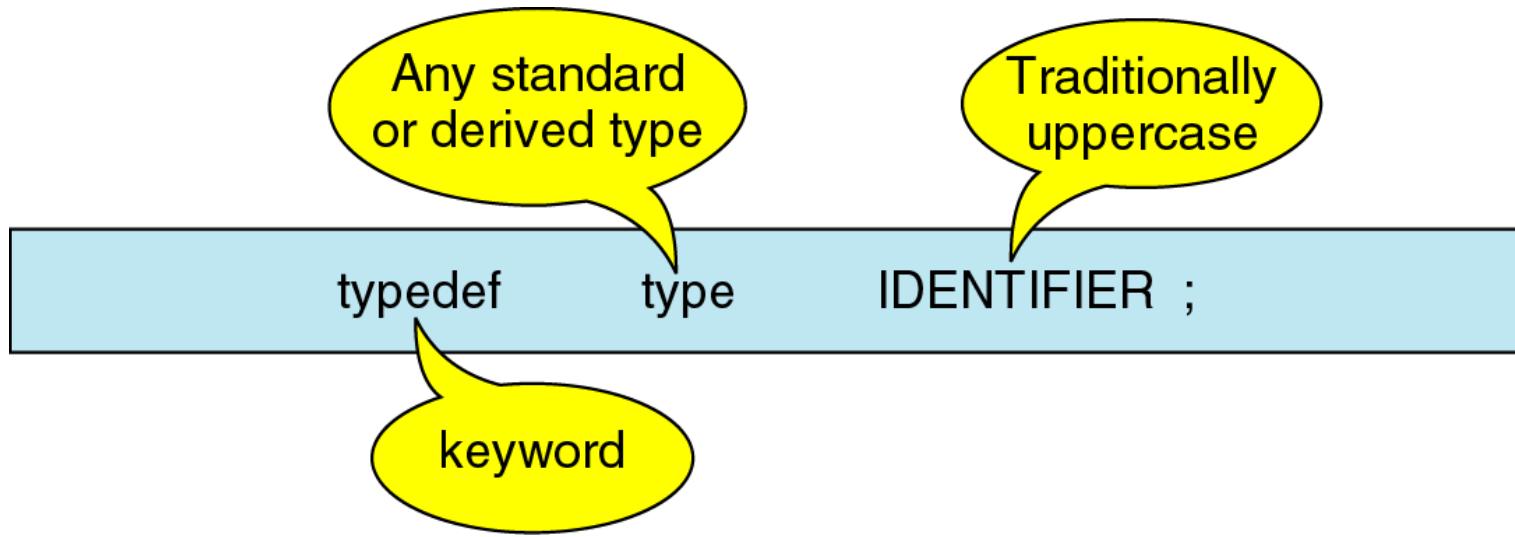
Figure 12-1



Objectives :

- To understand the structure and enumerated types.
- To use type definition statements in programs

Figure 12-2 type definition format



- A `typedef` gives a name to a data type by creating a new type that can then be used wherever a type is permitted
- Adv: to replace a complex name with an easier mnemonic
- To be coded in uppercase for easier readability.
- Ex1. `int roll_number;`
- `Typedef int ROLL_NUMBER;`
- `ROLL_NUMBER n1,n2,n3;`
- Ex2:
Replacing `char *stringPtrArray[20]` by `typedef`
`typedef char *STRING;`
`STRING stringPtrArray [20];`

Figure 12-2 type definition format

- A `typedef` gives a name to a data type by creating a new type that can then be used wherever a type is permitted
- Adv: to replace a complex name with an easier mnemonic
- To be coded in uppercase for easier readability
- Ex:

```
char *stringPtrArray[20]
```

```
typedef char *STRING;  
STRING stringPtrArray [20];
```

Enumerated Types

- Enumeration is a user defined datatype in C language based on the standard integer type.
- Each integer value is given an identifier called an enumeration constant.
- We can thus use the enumerated constants as symbolic names which make our program more readable.

Figure 12-3

```
enum {enumeration constants} variable_identifier ;
```

Format 1: enumerated variable

```
enum tag {enumeration constants} ;
```

```
enum tag variable_identifier ;
```

Format 2: enumerated tag

Figure 12-4

```
enum colors{red, white, blue, green, yellow};
```

```
enum colors aColor ;
```

Enumerated tag

```
typedef enum {red, white, blue, green, yellow} COLORS ;
```

```
COLORS aColor ;
```

Enumerated typedef

```
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};  
//enum week1{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
```

```
int main()  
{  
    enum week day;  
    day = Wed;  
    for(int i=Mon;i<=Sun;i++)  
        printf("%d\t",i);  
    return 0;  
}
```

0 1 2 3 4 5 6

```
enum week{Mon=11, Tue, Wed, Thur=11, Fri, Sat, Sun};  
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun,Mon};
```

Structures

```
int roll_num;  
char *name;  
float gpa;  
char sec;
```



Container

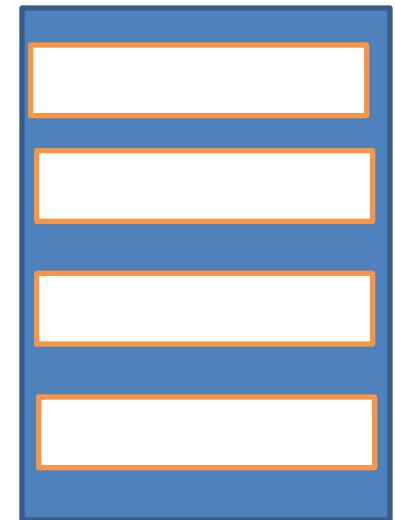
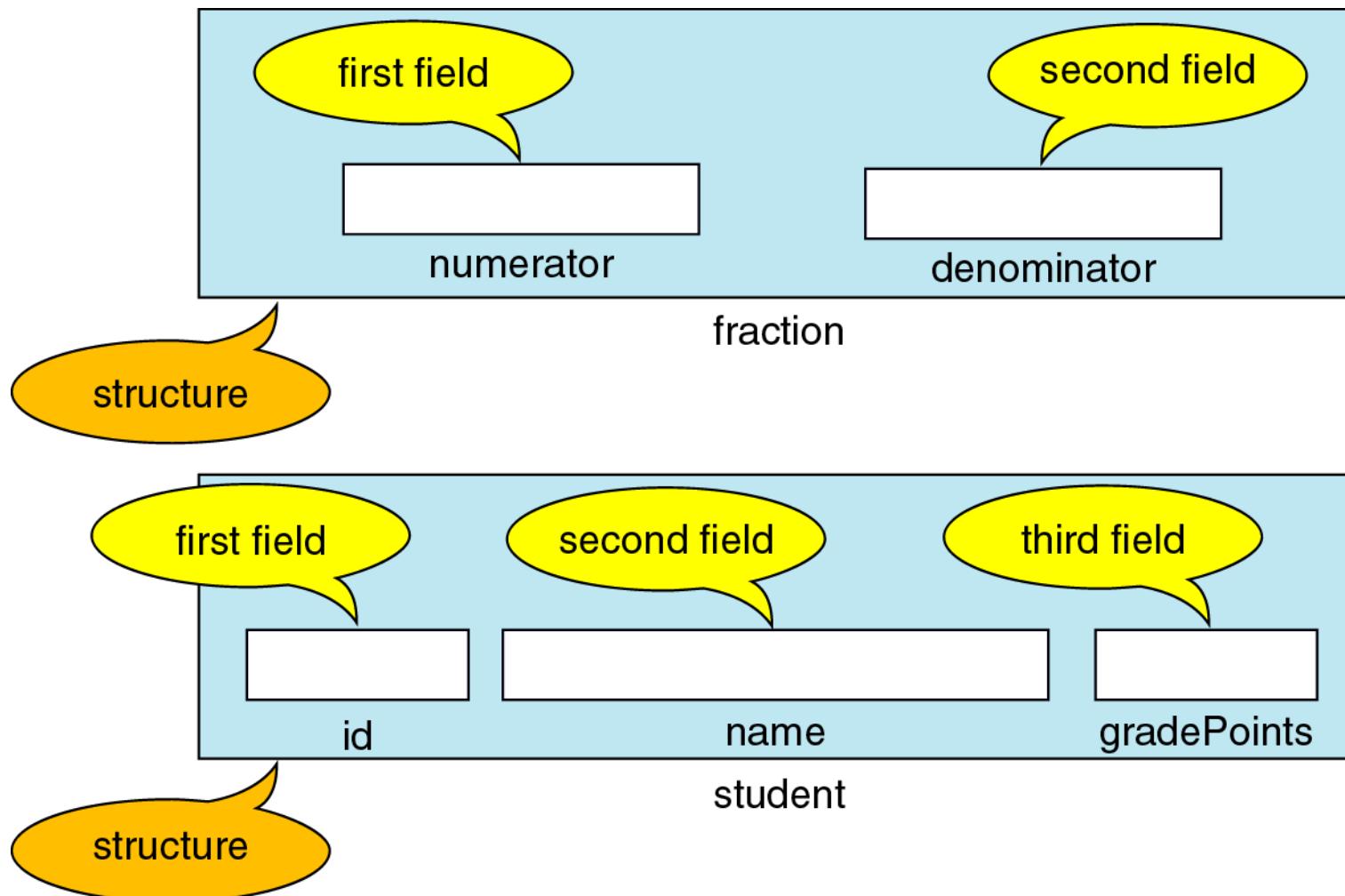


Figure 12-6 Structure Examples



Note: the data in a structure should all be related to one object

Ex1: both integers belong to the same fraction

Ex2: all data relate to one student

Structure

- Structure – a collection of related elements, possibly of different types, having a single name
- Field – each element in a structure is called a field
 - Same as variable in that it has type and exists in memory
 - Differs from a variable in that it is a part of structure
- Structure vs. array
 - Both are derived data types that can hold multiple pieces of data
 - All elements in an array must be of the same type, while the elements in a structure can be of the same or different types

Structure – declaration and definition

- Keyword struct – informs the compiler that it is a collection of related data
- 3 ways to declare/define a structure in C
 1. Structure variable
 2. Tagged structure
 3. Type-defined structure

Figure 12-7 structure variable

```
struct { field-list } variable_identifier ;
```



```
struct  
{  
    type1 fieldName1;  
    type2 fieldName2;  
    ...  
    typeN fieldNameN;  
} structName;
```

- Note: The above definition creates a structure for only one variable definition
- As there is no structure identifier (tag), it cannot be shared
- So, it is not really a type
- This type of declaration format is not to be used

```
struct {  
    char id[10];  
    char name[20];  
    int gpa;  
};
```

```
struct{  
    char id[10];  
    char name[20];  
    int gpa;  
}s1;
```

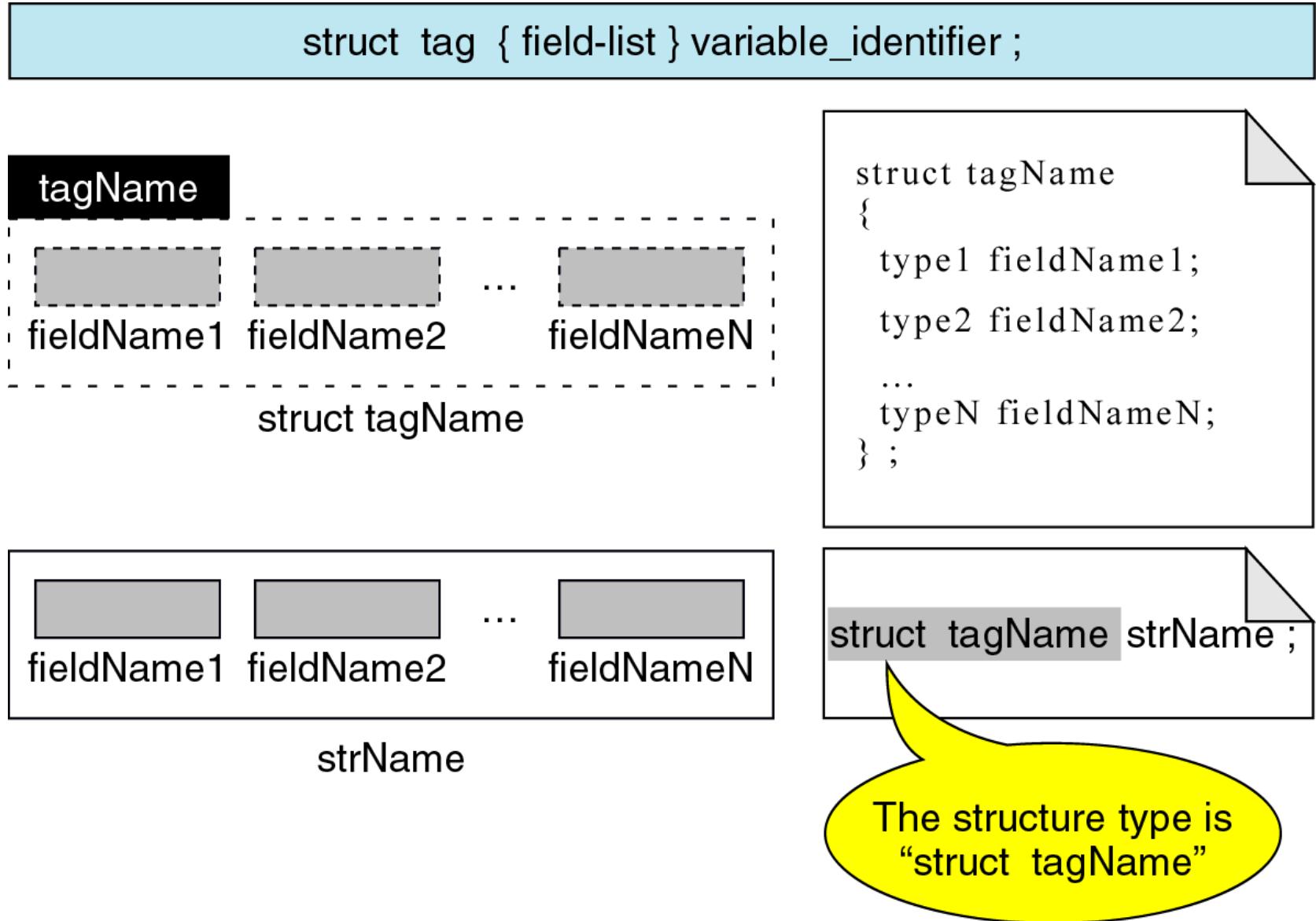
```
struct student{  
    char id[10];  
    char name[20];  
    int gpa;  
}s1;
```

```
typedef struct{  
    char id[10];  
    char name[20];  
    int gpa;  
}STUDENT;
```

```
STUDENT s1, s2;
```

```
#include<stdio.h>
#include<string.h>
struct Student
{
    char name[25];
    int age;
    char branch[10];
    char gender;
};
int main()
{
    struct Student s1;
    s1.age = 18;
    strcpy(s1.name, "Viraaj");
    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);
    return 0;
}
```

Figure 12-8 Tagged structure



Tagged Structure – declaration and definition

- struct tagname – tagname is the identifier for the structure
 - allows to define variables, parameters, and return types
- If a struct is concluded with a semicolon after the closing brace, no variables are defined
 - So structure is a type template with no associated storage
- To define a variable at the same time we declare the structure, list the variables by comma separation

Tagged Structure – declaration and definition

Ex: declare and use student structure:

```
struct student{  
    char id[10];  
    char name[20];  
    int gpa;  
};  
struct student astudent;  
void print_Student(struct student stu);
```

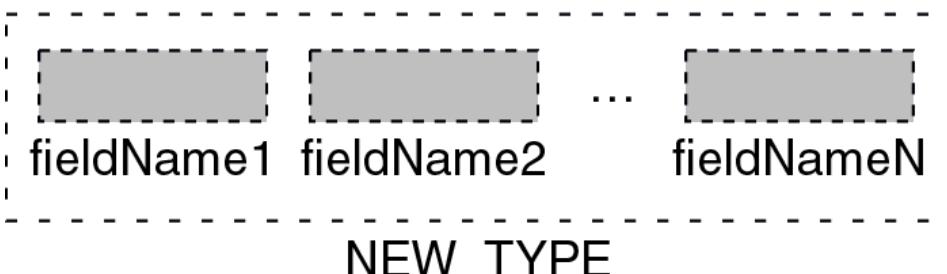
Tagged Structure – declaration and definition

```
struct student{  
    char id[10];  
    char name[20];  
    int gpa;  
};  
struct student astudent;  
void print Student( struct student stu);
```

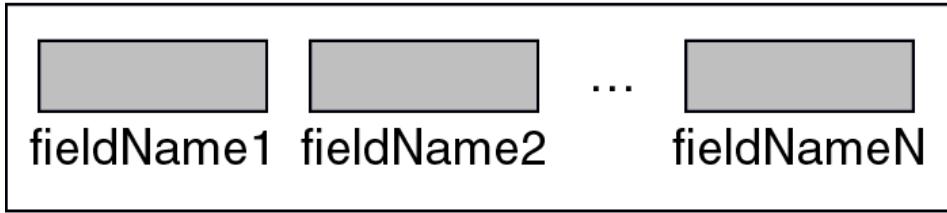
- Follow the above format of declaring structure first and then define variables
 - Declare structure declaration in the global area of the program before main
 - So structure declaration scope is global and can be shared by all functions

Figure 12-9 Type defined structure

```
typedef struct { field-list } TYPE-ID ;
```



```
typedef struct  
{  
    type1 fieldName1;  
    type2 fieldName2;  
    ...  
    typeN fieldNameN;  
} NEW_TYPE;
```



strName

The typedef name can
be used like any type

Type defined Structure – declaration and definition

Ex: declare and use student structure:

```
typedef struct {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;  
  
STUDENT astudent;  
  
void printStudent(STUDENT stu);
```

Type defined Structure – declaration and definition

- The most powerful way to declare a structure is by `typedef`
- `Typedef` vs. tagged structure declaration
 - `Typedef` is to be added before `struct`
 - Identifier at the end of the block is the type definition name not a variable
 - Cannot define a variable with the `typedef` declaration

```
typedef struct {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;  
  
STUDENT astudent;  
  
void printStudent(STUDENT stu);
```

Type defined Structure – declaration and definition

- It is possible to combine the tagged structure and type definition structure in a tagged type definition
- The difference between tagged type definition and a type defined structure is
 - Here, the structure has a tag name in tagged type definition

```
typedef struct tag {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;  
STUDENT astudent;  
void printStudent(STUDENT stu);
```

Figure 12-10 struct format variations

```
struct {  
    ...  
} variable_identifier;
```

structure variable

```
struct tag  
{  
    ...  
} variable_identifier;
```

```
struct tag variable_identifier;
```

tagged structure

```
typedef struct  
{  
    ...  
} TYPE_ID;
```

```
TYPE_ID variable_identifier;
```

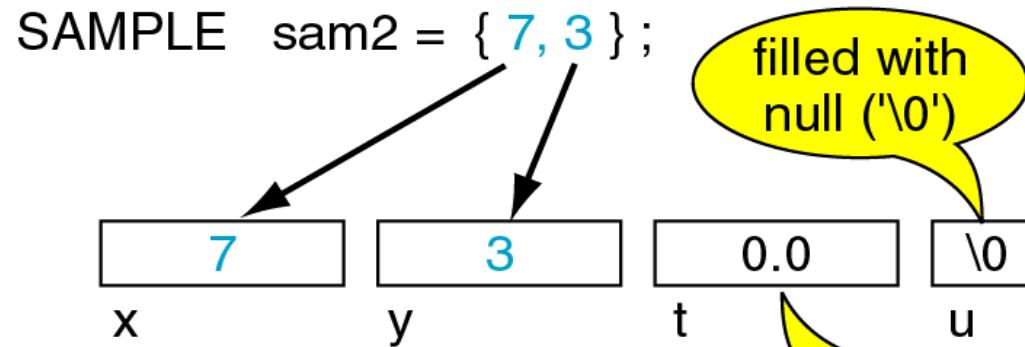
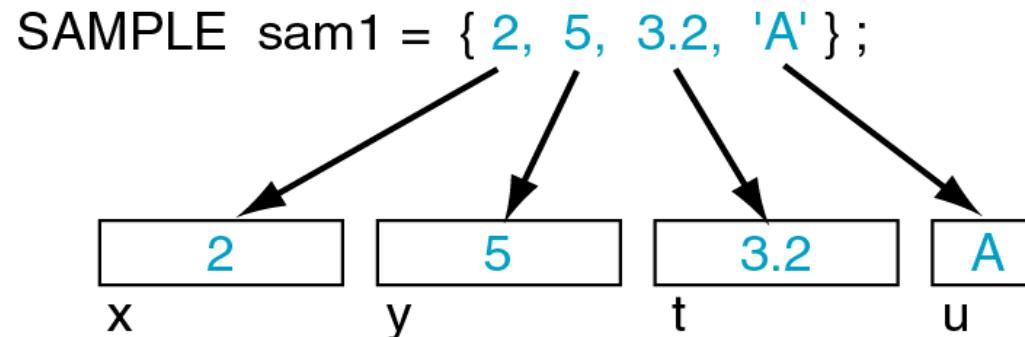
type-defined structure

Initializing structures

- Rules for structure initialization are similar to rules of array initialization
 - The initializers are enclosed in braces and comma separated
 - They must match their corresponding types in the structure definition
 - For a nested structure, the nested initializes must be enclosed in their own set of braces

Figure 12-11 Initializing structures

```
typedef struct
{
    int x;
    int y;
    float t;
    char u;
} SAMPLE;
```



- Ex1: initializer for each field is given – mapped sequentially to their types
- Ex2: initializer for some fields are only given – structure elements will be assigned null values – zero for integers and floating point numbers, \0 for chars and strings (same as in arrays)

Accessing structures

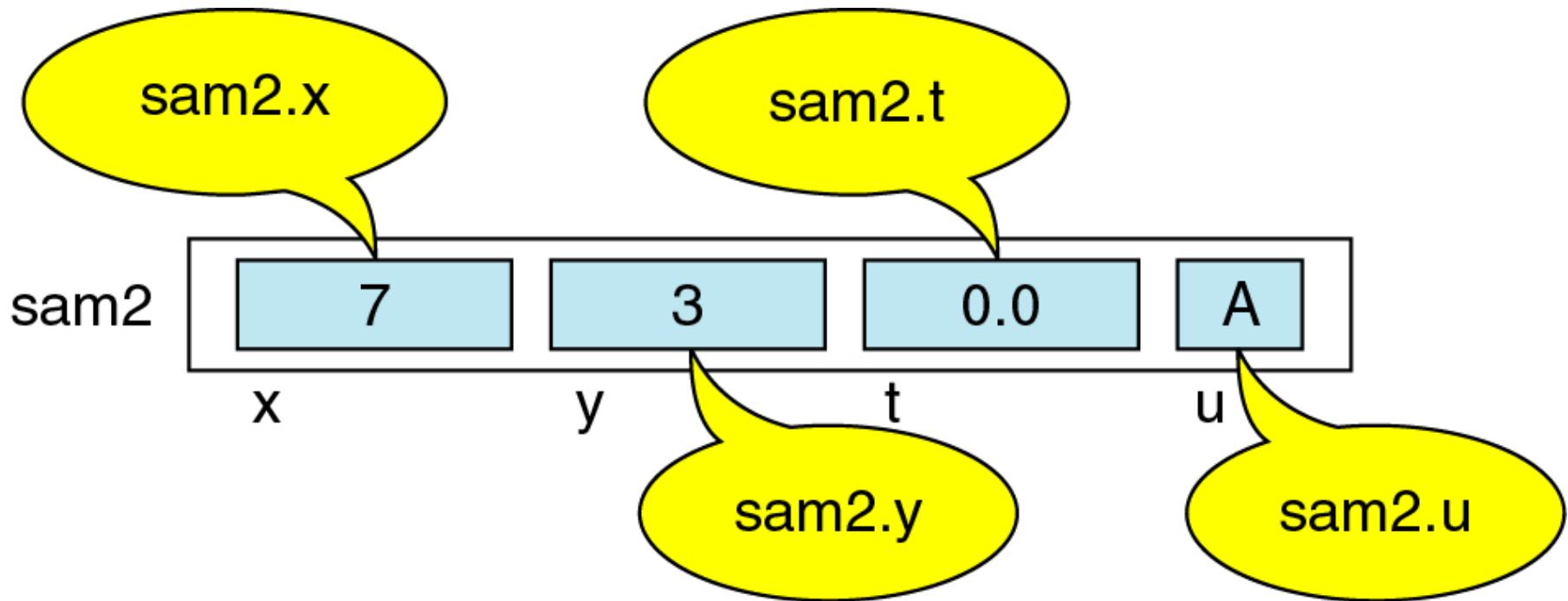
- Same way as we manipulate variables using expressions and operators, the structure fields can also be operated
- Ex:

```
typedef struct {  
    char id[10];  
    char name[20];  
    float gpa;  
} STUDENT;
```

```
STUDENT astudent;
```

- Refer to fields by
astudent.id, astudent.name, astudent.gpa

Figure 12-12 structure member operator



Ex: Reading data into and writing data from structure members
is same as done for variables

```
scanf("%d %d %f %c", &sam2.x, &sam2.y, &sam2.t, &sam2.u);
```

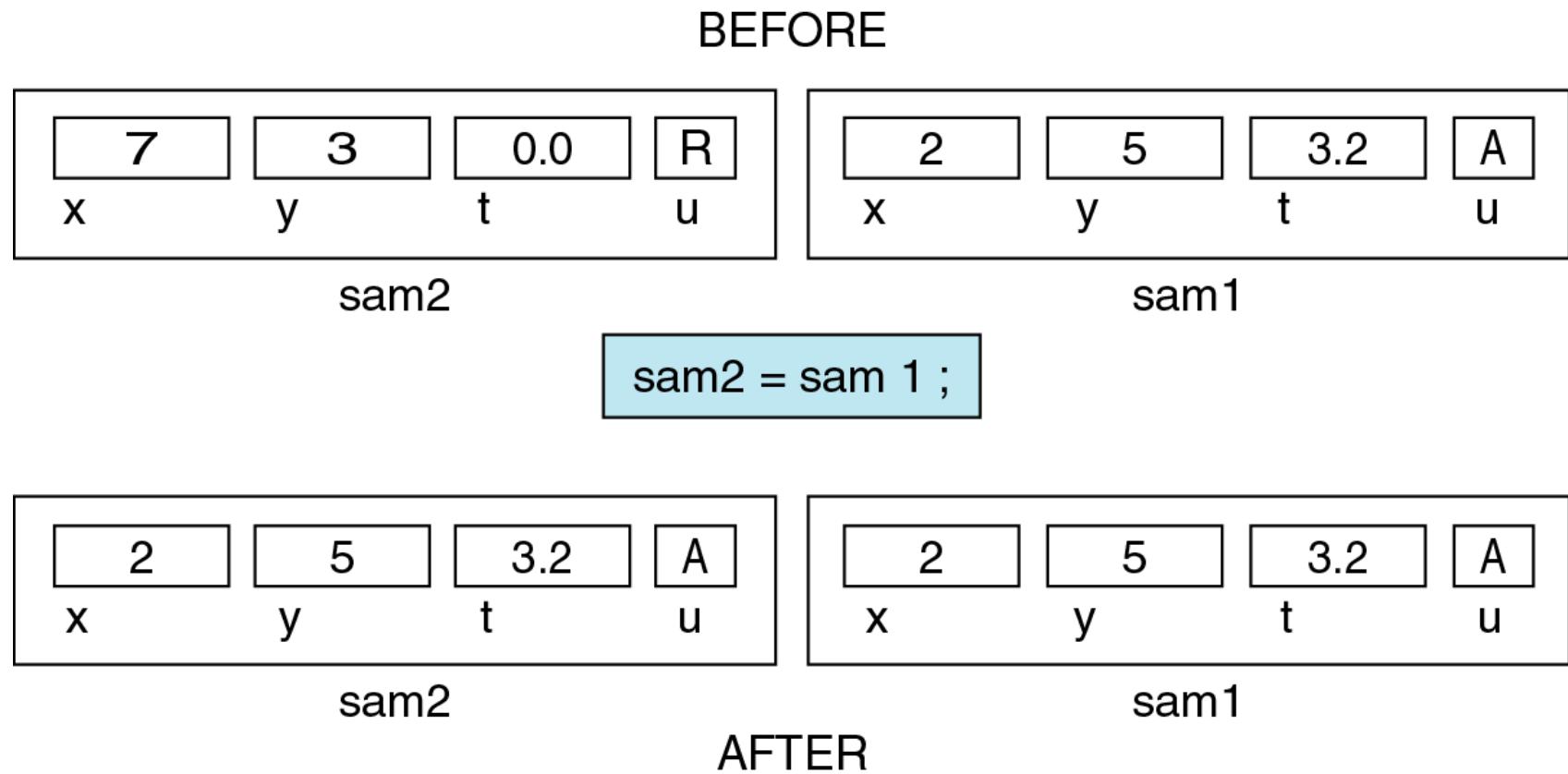
Precedence of Member operator

- Similar to operator [] for array indexing, dot is member operator for structure reference
- 1) The precedence of member operator is higher than that of increment
- Ex: sam2.x++; ++sam2.x;
- No parentheses are required
- 2) &sam1.x is eqt to &(sam1.x). So dot has higher priority than & operator

Operations on Structures

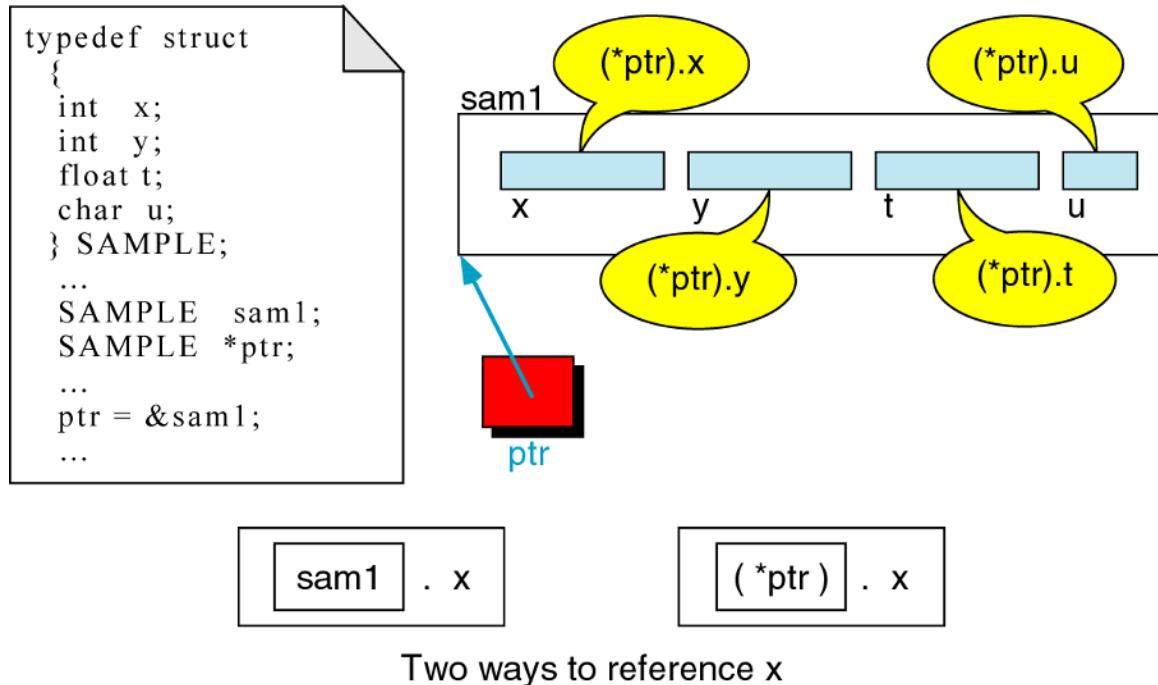
- Assignment operation : assigning one structure to another
- Structure is treated as one entity and only one operation i.e., assignment is allowed on the structure itself
- To copy one structure to another structure of the same type
 - Rather than assigning individual members
 - Assign one to another
- Ex: read values into sam1 from the keyboard. Now copy sam1 to sam2 by
- `sam2=sam1;`

Figure 12-13 Copying a structure



Pointer to structures

- Like other types, structures can also be accessed through pointers

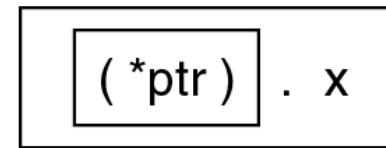
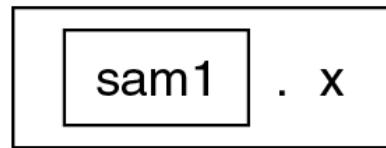
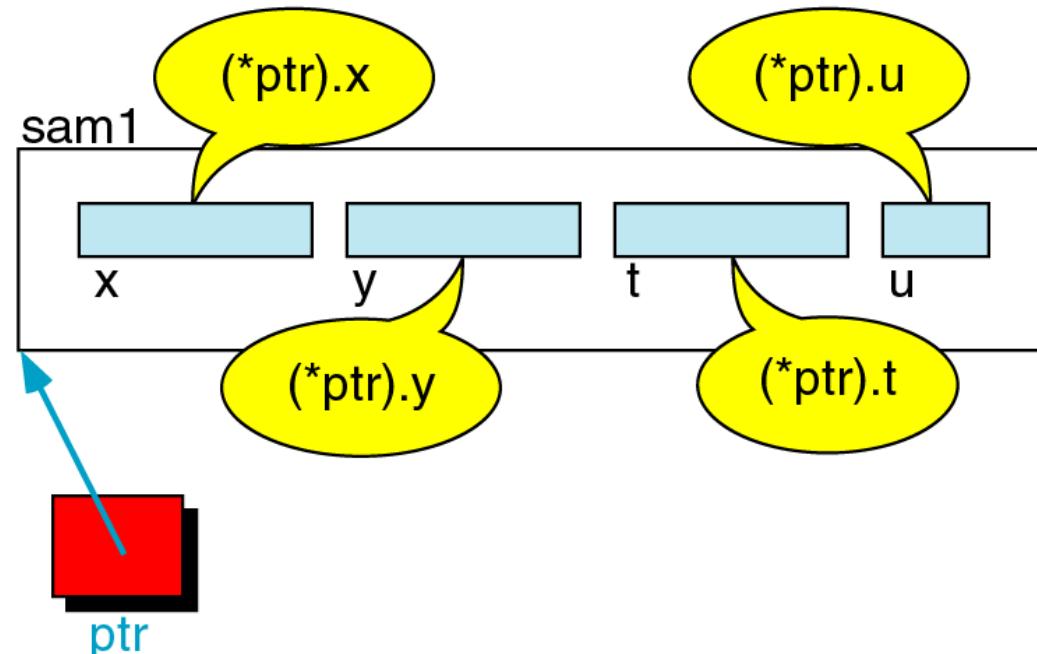


- Accessing structure itself by `*ptr`
- ptr contains the address of the beginning of the structure
- Now we do not only need to use structure name with member operator such as `sam1.x`
we can also use `(*ptr).x`

Figure 12-14 Pointers to structures

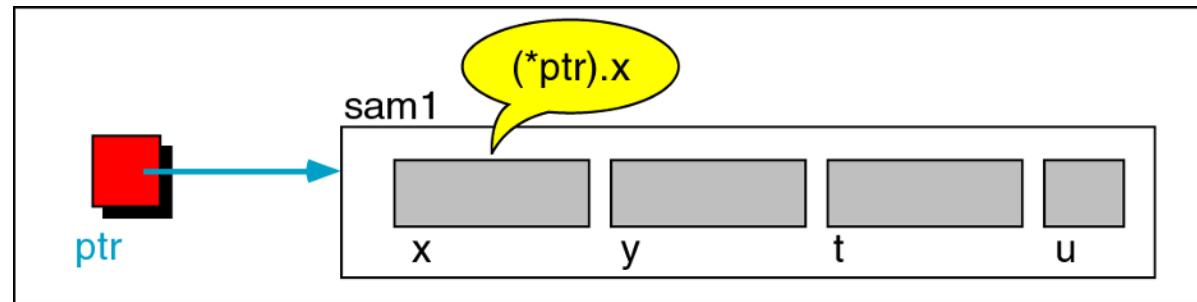
```
typedef struct
{
    int x;
    int y;
    float t;
    char u;
} SAMPLE;
```

```
...  
SAMPLE sam1;  
SAMPLE *ptr;  
...  
ptr = &sam1;  
...
```

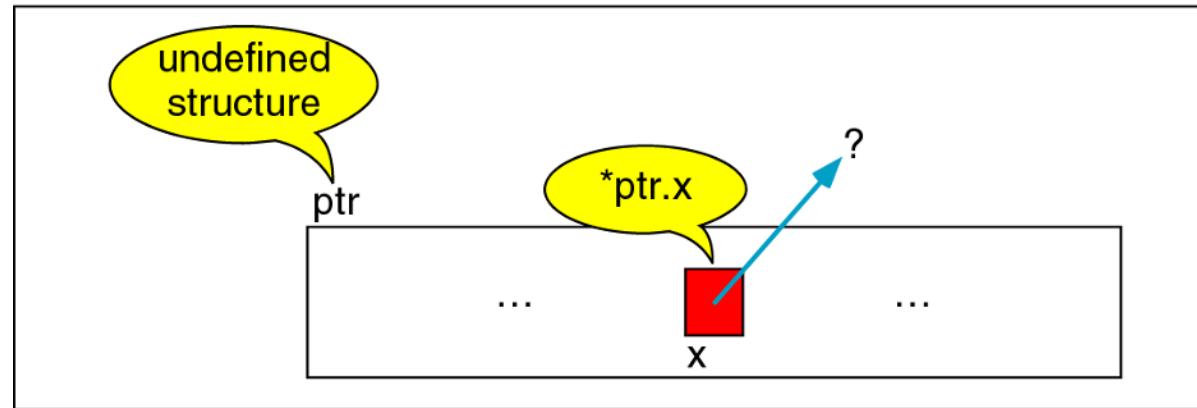


Two ways to reference x

Figure 12-15 Interpretation of invalid pointer use



The correct reference



The wrong way to reference the component

- In the expression `(*ptr).x`, parentheses are necessary as member operator has more priority than indirection operator
- Default interpretation of `*ptr.x` is `*(ptr.x)` which is error because it means that there is a structure called `ptr` (undefined here) containing a member `x` which must be a pointer
- So, a compile-time error is generated as it is not the case

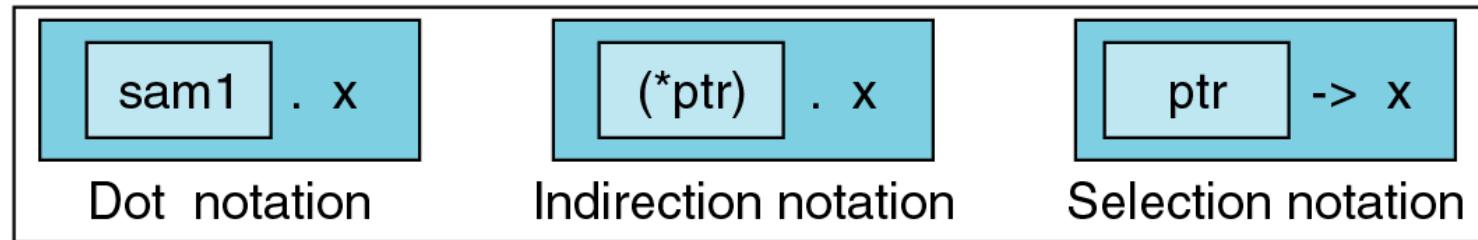
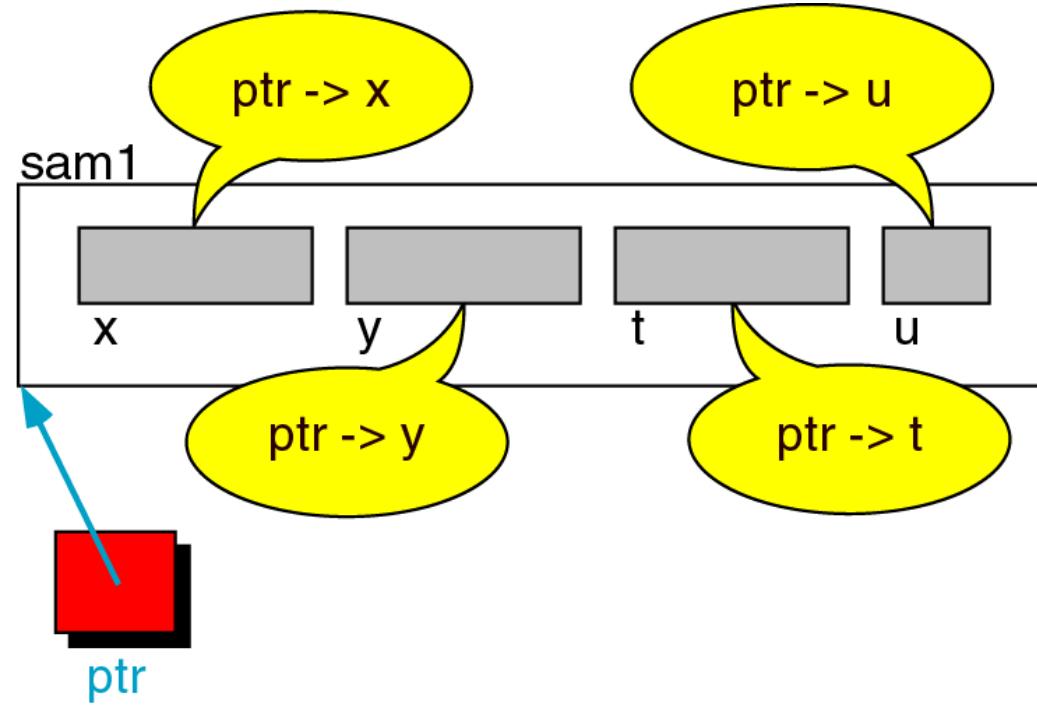
Selection operator

- However, there is a selection operator `->` (minus sign and greater than symbol) to eliminate the problem of pointer to structures
- The priority of selection operator (`->`) and member operator(.) are the same
- The expressions
- `(*pointerName).fieldName` is same as
`pointerName->fieldName`
- But `pointerName -> fieldName` is preferred

Figure 12-16 pointer selection operator

```
typedef struct
{
    int x;
    int y;
    float t;
    char u;
} SAMPLE;
```

```
...
SAMPLE sam1;
SAMPLE *ptr;
...
ptr = &sam1;
...
```



three ways to reference the field `x`

COMPLEX STRUCTURES

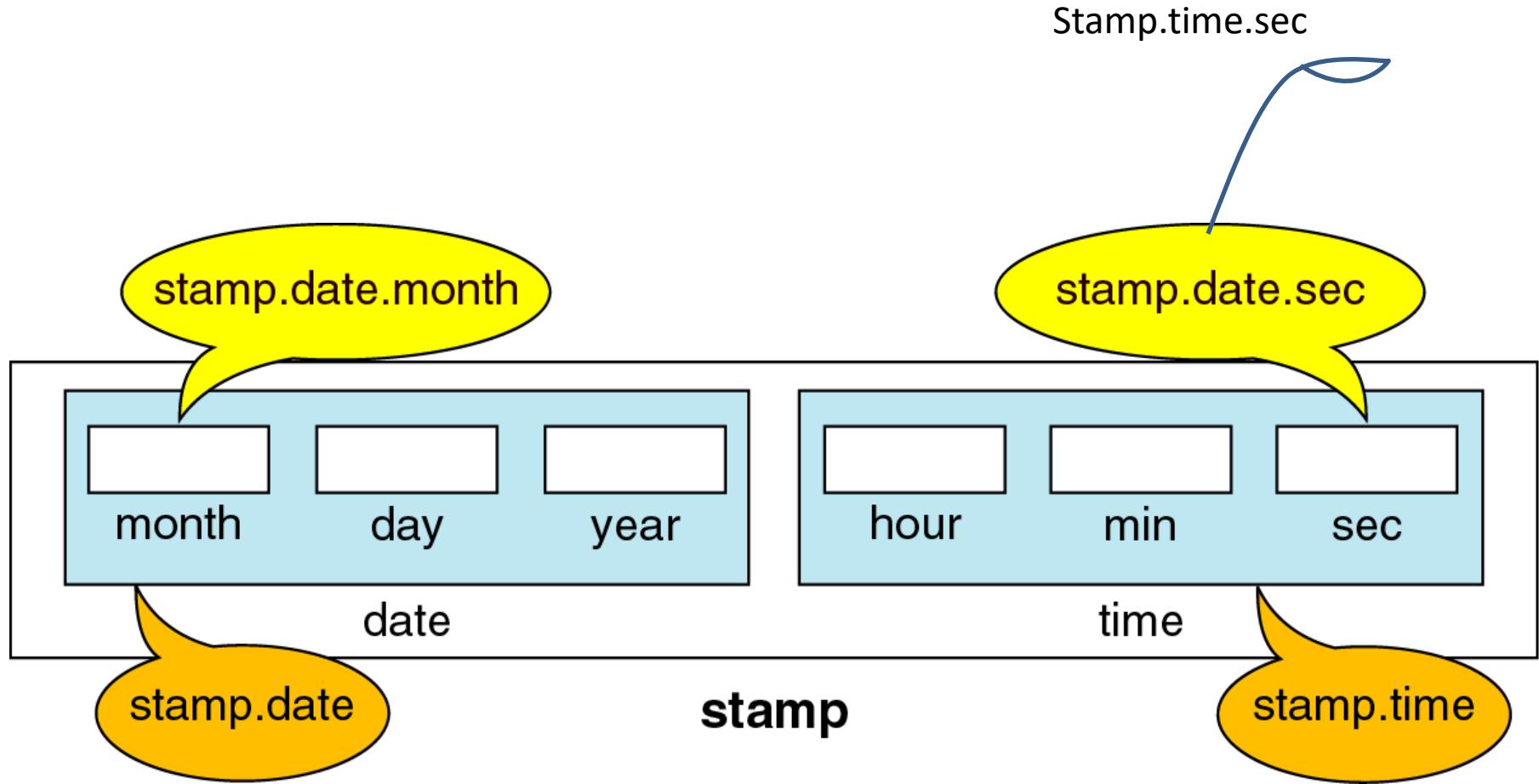
Complex Structures

- **Structures to solve complex problems**
- structures within structures (nested structures)
- arrays within structures
- arrays of structures, etc.

Nested Structures

- When a structure includes another structure, it is a nested structure
- We can have structures as members of a structure

Figure 12-17 Nested structure



- A structure called **stamp** stores the date and time
- The date is a structure that stores date, month and year
- The time is a structure that stores hour, minute and second

Nested Structures – declaration and referencing

- Preferred

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} DATE;  
  
typedef struct {  
    int hour;  
    int min;  
    int sec;  
} TIME;
```

```
typedef struct {  
    DATE date;  
    TIME time;  
} STAMP;  
  
STAMP stamp;
```

Preferred – simpler to declare each structure separately and then group it in a higher level structure

1/9/2022

Not recommended – complex to declare a nested structure with one declaration

42

```
#include<stdio.h>
#include<string.h>
typedef struct {
    int month;
    int day;
    int year;
} DATE;
typedef struct {
    int hour;
    int min;
    int sec;
} TIME;

typedef struct {
    DATE date;
    TIME time;
} STAMP;
```

```
int main()
{
    STAMP stamp;
    DATE date1;
    stamp.date.day=15;
    printf("%d\t",stamp.date.day);
    date1.day=67;
    printf("%d\t",date1.day);
    return 0;
}
```

Nested Structures – declaration and referencing

- Not recommended

```
typedef struct {  
    struct{  
        int month;  
        int day;  
        int year;  
    } date;  
    struct {  
        int hour;  
        int min;  
        int sec;  
    } time;  
} STAMP;  
STAMP stamp;
```

Preferred – simpler to declare each structure separately and then group it in a higher level structure

1/9/2022

Not recommended – complex to declare a nested structure with one declaration

44

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

typedef struct {
    struct{
        int month;
        int day;
        int year;
    } date,x;
    struct {
        int hour;
        int min;
        int sec;
    } time;
} STAMP;

int main()
{
    STAMP stamp;
    stamp.date.day=15;
    printf("%d\t",stamp.date.day);
    stamp.x.day=67;
    printf("%d\t",stamp.x.day);
    return 0;
}
```

Nested Structures – declaration

- In preferred notation
 - Declare the structures separately
 - Nesting must be from inside to out
 - So declare innermost structure first, then the next level working upward, toward the outermost structure
- Ex: in stamp structure
 - Date and time is declared first
 - Outer structure stamp is declared next

Nested Structures –referencing

- Accessing a nested structure
- from the highest level to the member of the innermost structure
- Ex: referencing stamp:
 - stamp
 - stamp.date
 - stamp.date.month
 - stamp.date.day
 - stamp.date.year
- stamp.time
- stamp.time.hour
- stamp.time.min
- stamp.time.sec

job.startTime.time.hour
job.endTime.time.hour

Nested Structures – declaration

- Same structure type can be used in a new structure declaration

Ex: using STAMP, we can declare a new structure JOB

```
typedef struct {  
    STAMP startTime;  
    STAMP endTime;  
} JOB;  
JOB job;
```

- Preferred notation

- Allows flexibility
- Ex: DATE is declared as a separate type definition
- So it is possible to pass the DATE structure to a function without having to pass STAMP

Nested Structure - Initialization

- Same as rules of a simple structure
- Initialize each structure completely before proceeding to the next member
- Each structure is enclosed in a set of braces
- Ex: Initializing stamp
 - Initialize date and then time separated by comma
 - Initializing date involves providing values for month, day , year each separated by commas
 - Initializing time involves providing values for hour, min, sec
- Ex: defining and initialization for stamp
- STAMP stamp = { {8, 18, 2014}, {08, 40, 50} };

```

#include<stdio.h>
#include<string.h>
typedef struct {
    int month;
    int day;
    int year;
} DATE;
typedef struct {
    int hour;
    int min;
    int sec;
} TIME;
typedef struct {
    DATE date;
    TIME time;
} STAMP;
int main()
{
    STAMP stamp={{3,12,20},{2,35,2}};
    printf("%d\t",stamp.date.day);
    printf("%d\t",stamp.time.sec);
    return 0;
}
} o/p: 12          2

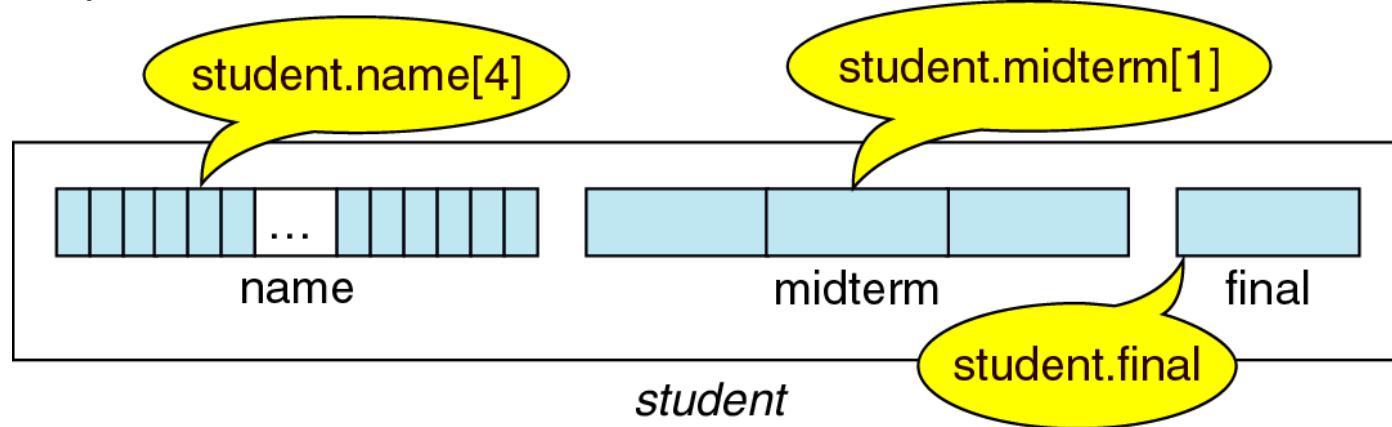
int main()
{
    STAMP stamp={{3,12},{35,2}};
    printf("%d\t",stamp.date.year);
    printf("%d\t",stamp.time.sec);
    return 0;
}
} o/p : 0          0

```

Structures containing arrays

- **Structures can have arrays as members**
- The array can be accessed by index or through pointers
- As with nested structure, arrays can be included within the structure or may be declared separately and then included
- If the array is declared separately, then the declaration must be complete before it can be used in the structure
- Ex: student structure

Figure 12-18 Arrays in structures



```
/* Global Declarations */
typedef struct
{
    char name[26];
    int midterm[3];
    int final;
} STUDENT ;
/* Local Definitions */
STUDENT student;
```

- Student structure contains 2 arrays

Figure 12-18 Arrays in structures

```
/* Global Declarations */  
typedef struct  
{  
    char name[26];  
    int midterm[3];  
    int final int;  
} STUDENT ;  
/* Local Definitions */  
STUDENT student;
```

- Student structure contains 2 arrays

Array with in the structure

```
#include <stdio.h>
typedef struct {
    int roll_no;
    char grade;
    float marks[4];
}STUDENT;
void display(STUDENT a1)
{
    printf("Roll number : %d\n", a1.roll_no);
    printf("Grade : %c\n", a1.grade);
    printf("Marks secured:\n");
    int i;
    for (i = 0; i < 4; i++)
        printf("Subject %d : %.2f\n", i + 1, a1.marks[i]);
}
```

```
void read(STUDENT *a1)
{
    int i,sum=0;
    printf("Enter the Roll_number\n");
    scanf("%d",&a1->roll_no);
    // scanf("%d",&(*a1).roll_no);
    printf("Enter the marks \n");
    for(i=0;i<4;i++)
        scanf("%f",&a1->marks[i]);
    for(i=0;i<4;i++)
        sum=sum+a1->marks[i];
    if(sum/4.0>80)
        a1->grade='A';
    else a1->grade='B';
}
int main()
{
    STUDENT A;
    read(&A);
    display(A);
    return 0;
}
```

Enter the Roll_number

1

Enter the marks

100

100

100

100

Roll number : 1

Grade : A

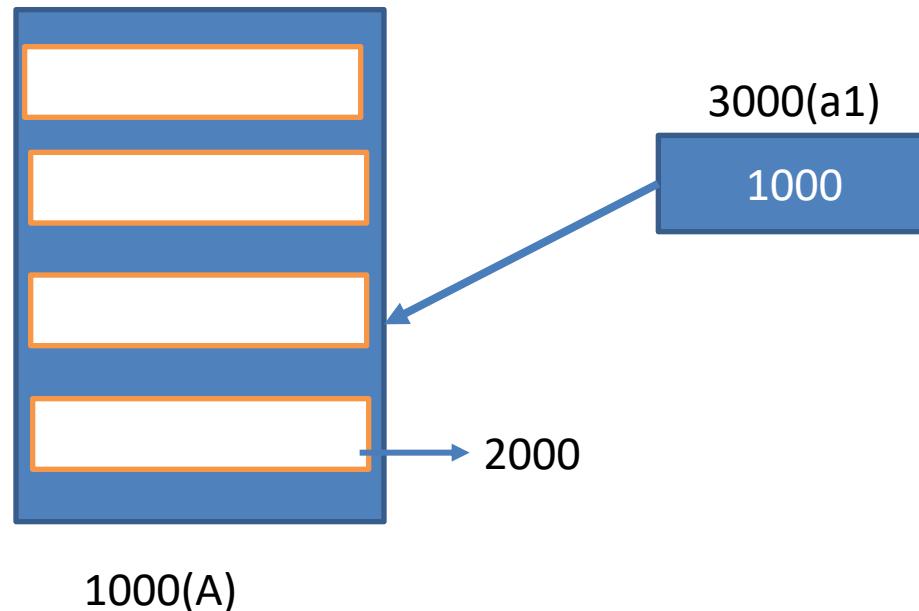
Marks secured:

Subject 1 : 100.00

Subject 2 : 100.00

Subject 3 : 100.00

Subject 4 : 100.00



```
STUDENT read(STUDENT a1)
```

structure returning structure type.

```
{
```

```
    int i,sum=0;
```

```
    printf("Enter the Roll_number\n");
```

```
    scanf("%d",&a1.roll_no);
```

```
    printf("Enter the marks \n"); void display(STUDENT a1)
```

```
    for(i=0;i<4;i++) {
```

```
        scanf("%f",&a1.marks[i]);
```

```
        for(i=0;i<4;i++)
```

```
            sum=sum+a1.marks[i];
```

```
        if(sum/4.0>80)
```

```
            a1.grade='A';
```

```
        else a1.grade='B';
```

```
    return a1;
```

```
}
```

```
int main()
```

```
{ STUDENT A;
```

```
    A=read(A);
```

```
    display(A);
```

```
    return 0;
```

```
}
```

```
1/9/2022
```

```
    printf("Roll number : %d\n", a1.roll_no);
    printf("Grade : %c\n", a1.grade);
    printf("Marks secured:\n");
    int i;
    for (i = 0; i < 4; i++) {
        printf("Subject %d : %.2f\n", i + 1, *(a1.marks+i));
    }
```

//Replacing array with in structure using pointer

```

#include <stdio.h>
typedef struct {
    int roll_no;
    char grade;
    float marks[4];
}STUDENT;
void read(STUDENT a1[])
{
    int i,j,sum=0;
    for(i=0;i<4;i++){
        printf("Enter the Roll_number for student %d\n ",i+1);
        scanf("%d",&a1[i].roll_no);
        printf("Enter the marks of 4 subjects for roll number%d
\n",i+1);
        for(j=0;j<4;j++)
            scanf("%f",&a1[i].marks[j]);
        for(j=0;j<4;j++)
            sum=sum+a1[i].marks[j];
        if(sum/4.0>80)
            a1[i].grade='A';
        else a1[i].grade='B';
        sum=0;
    }
}

void display(STUDENT a1)
{
    printf("Roll number : %d\n", a1.roll_no);
    printf("Grade : %c\n", a1.grade);
    printf("Marks secured:\n");
    int i;
    for (i = 0; i < 4; i++) {
        printf("Subject %d : %.2f\n",i + 1, *(a1.marks+i));
    }
}

int main()
{
    STUDENT A[10];
    printf("create 4 students\n");
    read(A);
    for(int i=0;i<4;i++)
        display(A[i]);
    return 0;
}

```

1/9/2022

Structures containing arrays

- Ex: student structure – referencing through index

student

student.name

student.name[i]

student.midterm

student.midterm[j]

student.final_int

Structures containing arrays

- Ex: student structure – [referencing through pointer](#)
- For an array, we can always use a pointer to refer directly to the array elements
- Ex: referring scores in student structure
- int *pScores

```
pScores = student.midterm;
```

```
totalscores = *pScores + *(pScores+1) + *(pScores+2)
```

Structures containing arrays

- **Array initialization in structures**
- Same rule of structure initialization
- Since array is a separate member, its values must be included in a separate set of braces
- Ex: student structure initialization
- STUDENT student = {"name1", {10, 20, 30}, 40};

Structures containing pointers

- The use of pointers can save memory

```
typedef struct
```

```
{
```

```
    char *month;
```

```
    int day;
```

```
    int year;
```

```
} DATE;
```

Structures containing pointers

- The use of pointers can save memory
- Suppose, we want alphabetic month in stamp structure and not integer month
- Alternative 1: add char month[9] as structure member

```
typedef struct
{
    char month[9];
    int day;
    int year;
} DATE;
```

Structures containing pointers

Alternative 2: add char *month as structure member
typedef struct

```
{  
    char *month;  
    int day;  
    int year;  
} DATE;
```

Given the months of the year defined as strings

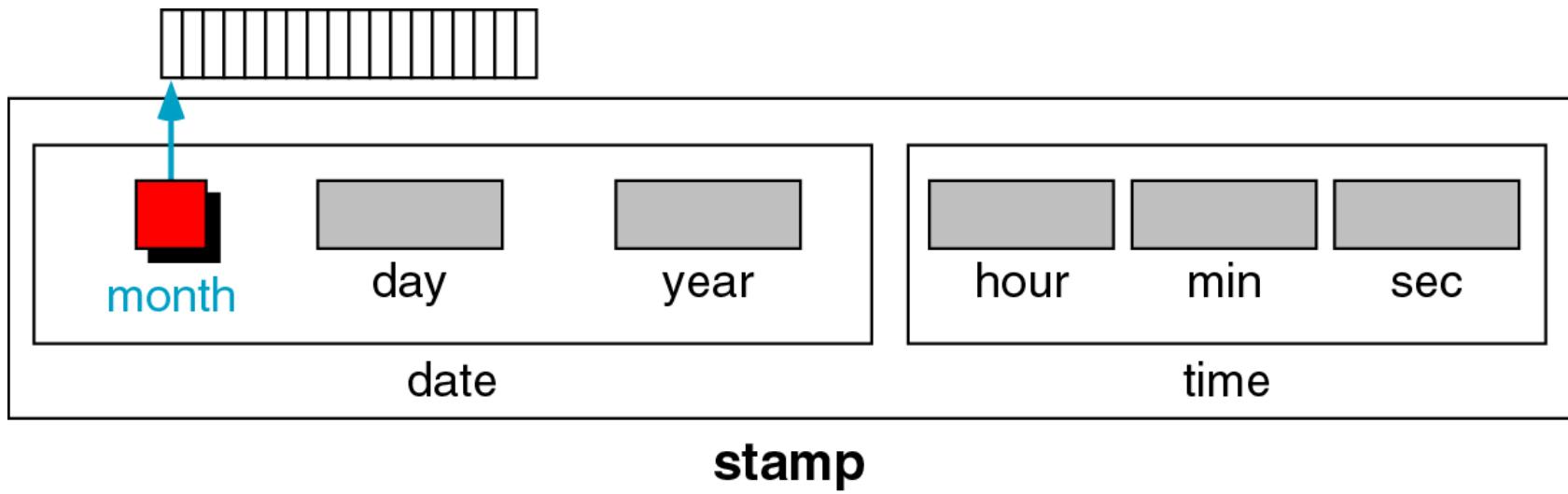
```
char jan[] = "January";  
char feb[] = "February";  
-----
```

```
char dec[] = "December";
```

Assigning month to the structure is now by copying a
pointer to the string

```
stamp.date.month = jan;
```

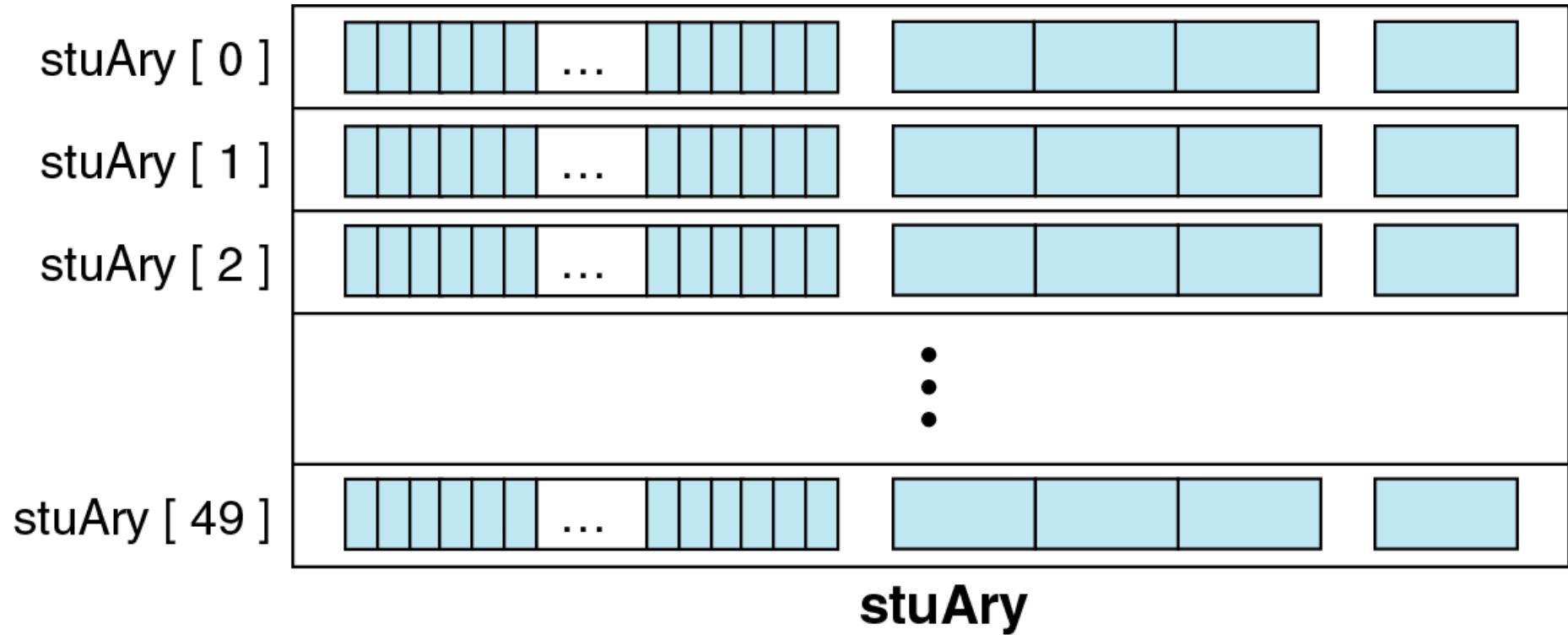
Figure 12-19 Pointers in Structures



Array of Structures

- In an array, we can easily work with data – to calculate average, sorting etc.

Figure 12-20 Array of structures



- Defining STUDENT stuAry[50];
- Accessing by index stuAry[i]
- Accessing by pointer *pstu

Array of Structures (using pointers)

Finding the average of final marks

```
#define SIZE 10
typedef struct{
    char name[25];
    int midterm[3];
    int final;
} STUDENT;
STUDENT stuary[10];
int i, sum = 0;
float average;
```

```
STUDENT *pwalk;
STUDENT *plast;
plast = stuary+SIZE -1;
for(pwalk= stuary; pwalk <= plast;
pwalk++)
    sum = sum + pwalk->final;
average = sum/(float)SIZE;
```

Array of Structures (using pointers)

Finding average of each mid term marks with pointers

```
#define SIZE 10
typedef struct{
    char name[25];
    int midterm[3];
    int final;
}STUDENT;
STUDENT stuarray[10];
int i, sum = 0;
float midtermAvg[3];
```

```
STUDENT *pwalk;
STUDENT *plast;
plast = stuarray+SIZE-1;
for(i = 0; i < 3; i++){
    sum = 0;
    for(pwalk = stuarray; pwalk <= plast;
    pwalk++)
        sum = sum+pwalk->midterm[i];
    midtermAvg[i] = sum/(float)SIZE;
}
```

Array of Structures - PROBLEMS

- Sort an array of student structure using Rollno as the key
- Repeat the problem separately using functions and then using pointers

Structures and Functions

- Structure can be passed to functions in 3 ways
 - 1. Sending individual members
 - 2. Sending the whole structure
 - 3. Passing structures through pointers

Structures and Functions

- 1. Sending individual members
- Actual parameters – use individual members through member operators
- Formal parameters –
 - The called program accordingly writes the type of individual member as int, float, char etc
 - It does not know if the integers were structure members

Figure 12-21 Passing structure member to functions

```
...
res.numerator =
    multiply(fr1.numerator, fr2.numerator);
res.denominator =
    multiply(fr1.denominator, fr2.denominator);
...
```

```
/* ===== multiply ===== */
multiply int x,
           int y)

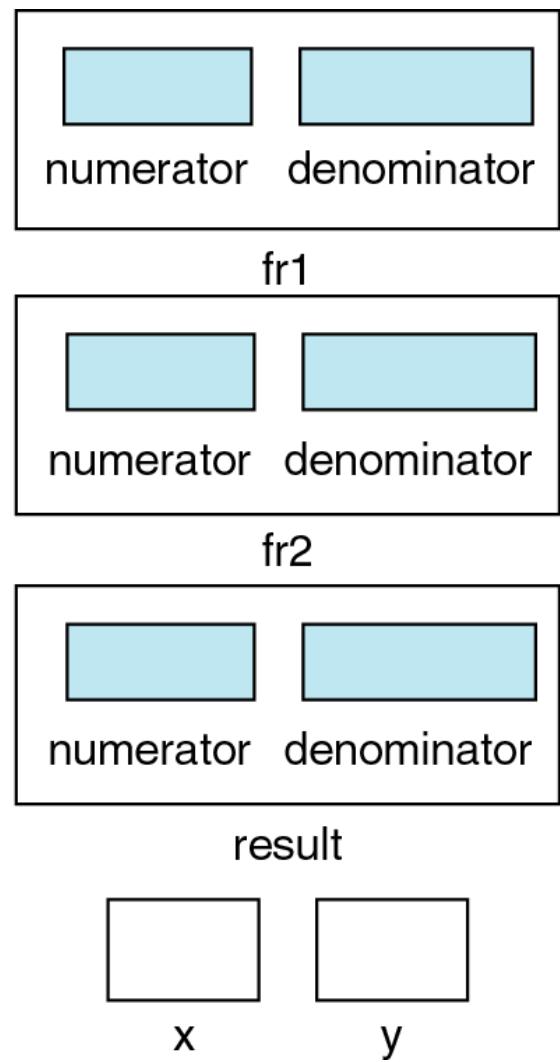
{
    return x * y ;
} /* multiply */
```

Figure 12-21 Passing structure member to functions

```
...
res.numerator =
    multiply(fr1.numerator, fr2.numerator);
res.denominator =
    multiply(fr1.denominator, fr2.denominator);
...

```

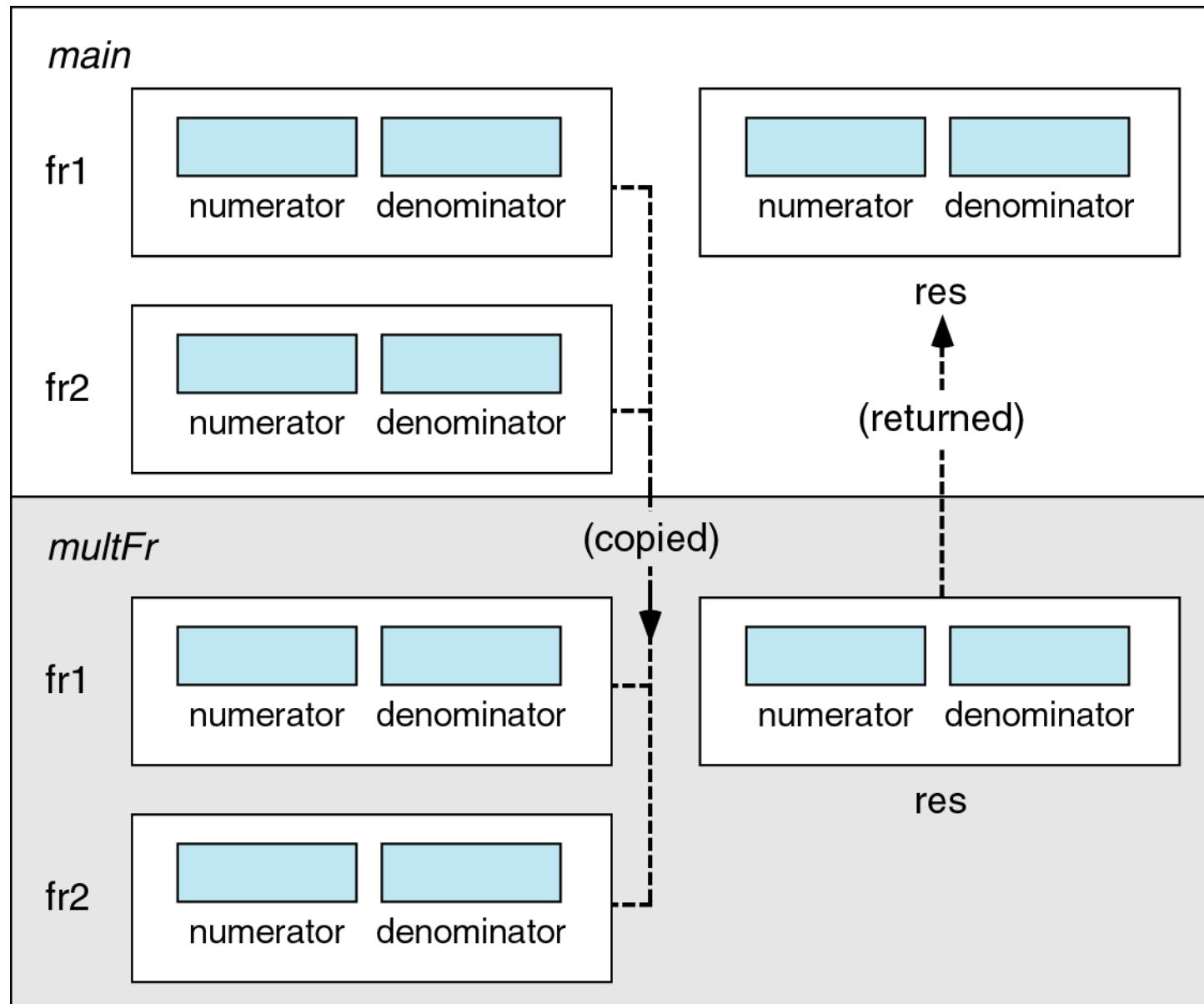
```
/* ===== multiply ===== */
multiply int x,
           int y)
{
    return x * y ;
} /* multiply */
```



Structures and Functions

- 2. Sending the whole structure (since structure is a type)
- Better solution is to pass the entire structure to the function so that function can finish its job in one call
- Actual parameters – use the structure type for declaring the parameters
- Formal parameters –
 - Similarly specify the structure as type in the formal parameters of the called function
 - Similarly return can be specified as the structure type

Figure 12-22 Passing and returning structures



Structures and Functions

Sending the whole structure

```
typedef struct {  
    int numerator;  
    int denominator;  
} FRACTION;  
  
FRACTION getFr();  
  
FRACTION multFr( FRACTION  
fr1, FRACTION fr2);  
  
FRACTION printFr(FRACTION  
result);
```

```
int main() {  
    FRACTION fr1, fr2, res;  
    fr1= getFr();  
    fr2 = getFr();  
    res = multFr(fr1, fr2);  
    printFr(res);  
}
```

Structures and Functions

```
FRACTION getFr() {  
    FRACTION fr;  
    printf("write fraction in the form of x/y");  
    scanf("%d/%d", &fr.numerator, &fr.denominator);  
    return fr;  
}  
FRACTION multFr(FRACTION fr1, FRACTION fr2) {  
    FRACTION res;  
    res.numerator = fr1.numerator * fr2.numerator;  
    res.denominator = fr1.denominator * fr2.denominator;  
    return res;  
}
```

Structures and Functions

```
FRACTION printFr (FRACTION res) {  
    printf("%d/%d", res.numerator, res.denominator);  
}  
}
```

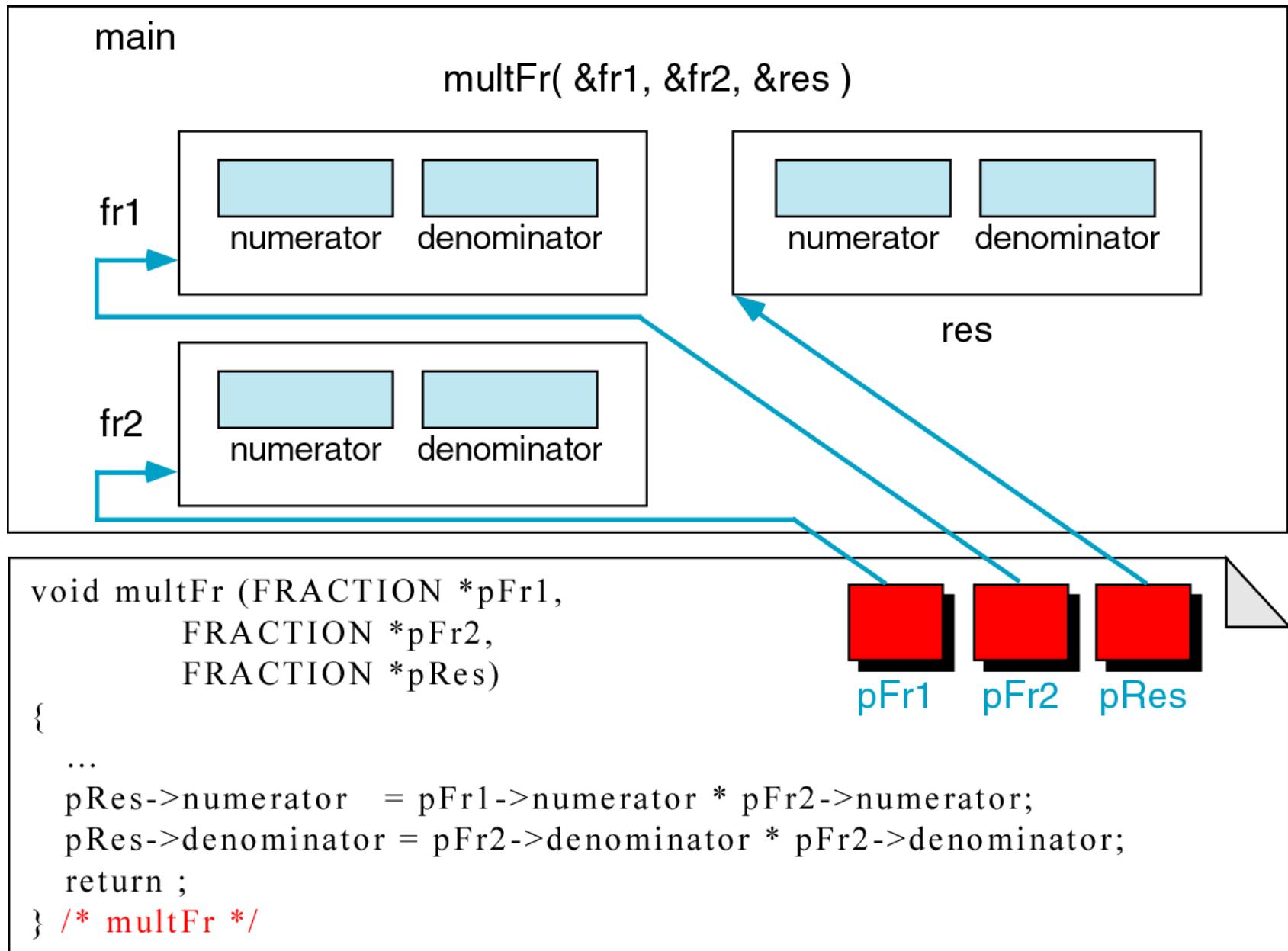
Structures and Functions

- 2. Sending the whole structure (since structure is a type)
- In getFr() function, address and member operator both are used
 - Member operator has more priority, so parentheses are not required
- In getFr() function, two values are returned without even using pointers
 - Using structures, it is possible to return more than one element

Structures and Functions

- 3. Passing structures through pointers

Figure 12-23 Passing structures through pointers



Structures and Functions

3. Passing structures through pointers

```
typedef struct {  
    int numerator;  
    int denominator;  
} FRACTION;  
  
void getFr(FRACTION *pFr);  
  
void multFr(FRACTION *pFr1, FRACTION *pFr2, FRACTION  
*pRes);  
  
FRACTION printFr(FRACTION *pRes);
```

Structures and Functions

```
int main(){
    FRACTION fr1, fr2, res;
    getFr(&fr1);
    getFr(&fr2);
    multFr(&fr1,&fr2,&res);
    printFr(&res);
}
getFr(FRACTION *pFr) {
    printf("write fraction in the form of x/y");
    scanf("%d/%d", &pFr->numerator, &(*pFr).denominator);
}
```

Structures and Functions

```
void multFr(FRACTION *pFr1, FRACTION *pFr2, FRACTION  
*pRes) {  
    pRes->numerator = pFr1->numerator * pFr2->numerator;  
    pRes->denominator = pFr1->denominator * pFr2->  
denominator;  
}  
FRACTION printFr(FRACTION *pRes) {  
    printf("%d/%d", pRes -> numerator, pRes -> denominator);  
}
```

Structures and Functions

- 3. Passing structures through pointers
- `scanf ("%d/%d", &pF->numerator, &(*pFr).denominator)`
- Even with pointers, we need address for `scanf`
- Two notations are equivalent
- Selection operator `->` has higher priority than address operator `&`
 - Parentheses are not required
- Member operator `.` has a higher priority than indirection operator `*` and address operator `&`
 - But `&` and `.` are at the same level, so we need parentheses only for `*` (This is as per textbook. It is probably wrong)

END

Abstract Data Type (ADT)

- An **abstract data type** is a data declaration packaged together with the operations that are meaningful on the data type (composite types).
- In other words, we encapsulate the data and the operation on data and we hide them from the user.
- ADT has
 1. **Declaration of data** (set of values on which it operates)
 2. **Declaration of operation**(set of functions) and hides the representation and implementation details

Array as Abstract Data Type

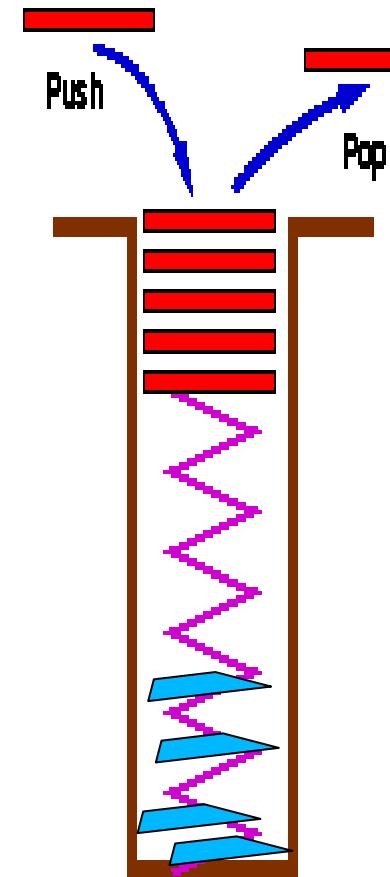
- ADT Array
- **objects:** A set of pairs $\langle \text{index}, \text{value} \rangle$ where for each value of index there is a value from the set item . **Index** is a finite ordered set of one or more dimensions, for example, $\{0, \dots, n-1\}$ for one dimension, $\{(0,0), (0,1), \dots, (2,1), (2,2)\}$ for two dimensions, etc.
- **Functions:** for all $A \in \text{Array}$, $i \in \text{index}$, $x \in \text{item}$, $j, \text{size} \in \text{integer}$
Array Create(j, list) ::= return an array of j dimension where list is a j -tuple whose i th element is the size of i th dimension

Item Retrieve(A, i) ::= if $i \in \text{index}$ retrieve the element from array A indexed by i

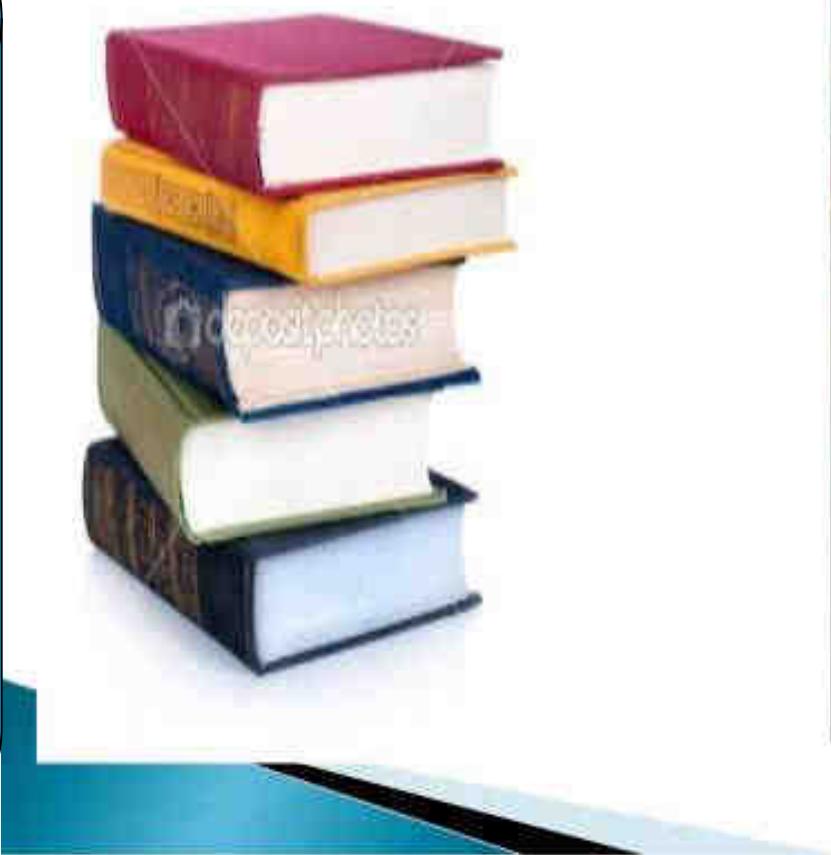
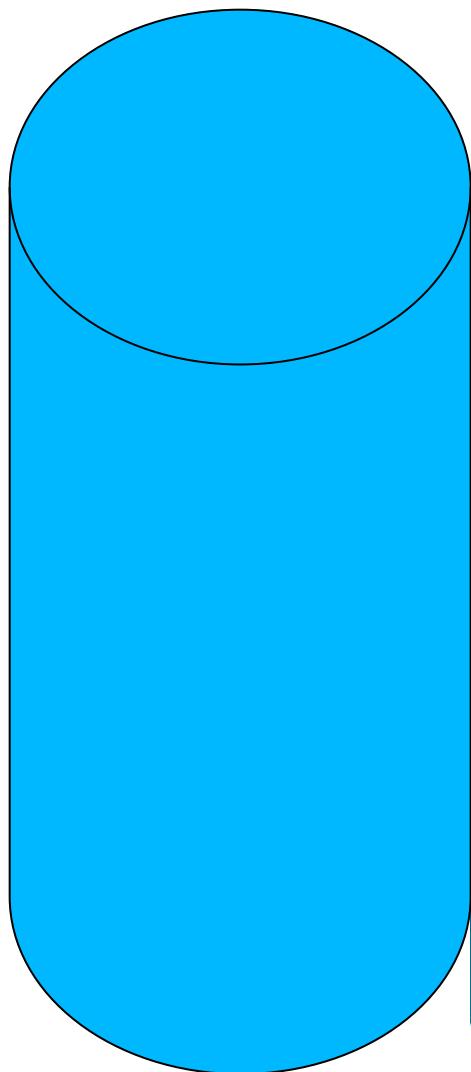
Array Store(A, i, x) ::= if $i \in \text{index}$ store the value x at i th index in the array A

Stacks

- Definition: A Stack (**Linear data structure**) is an **ordered list** in which insertions and deletions are made at one end called the top.
- Insertion is called as **PUSH**
- Deletion of an element is called as **POP**
- Stack is also called as **Last In First Out (LIFO)** list.



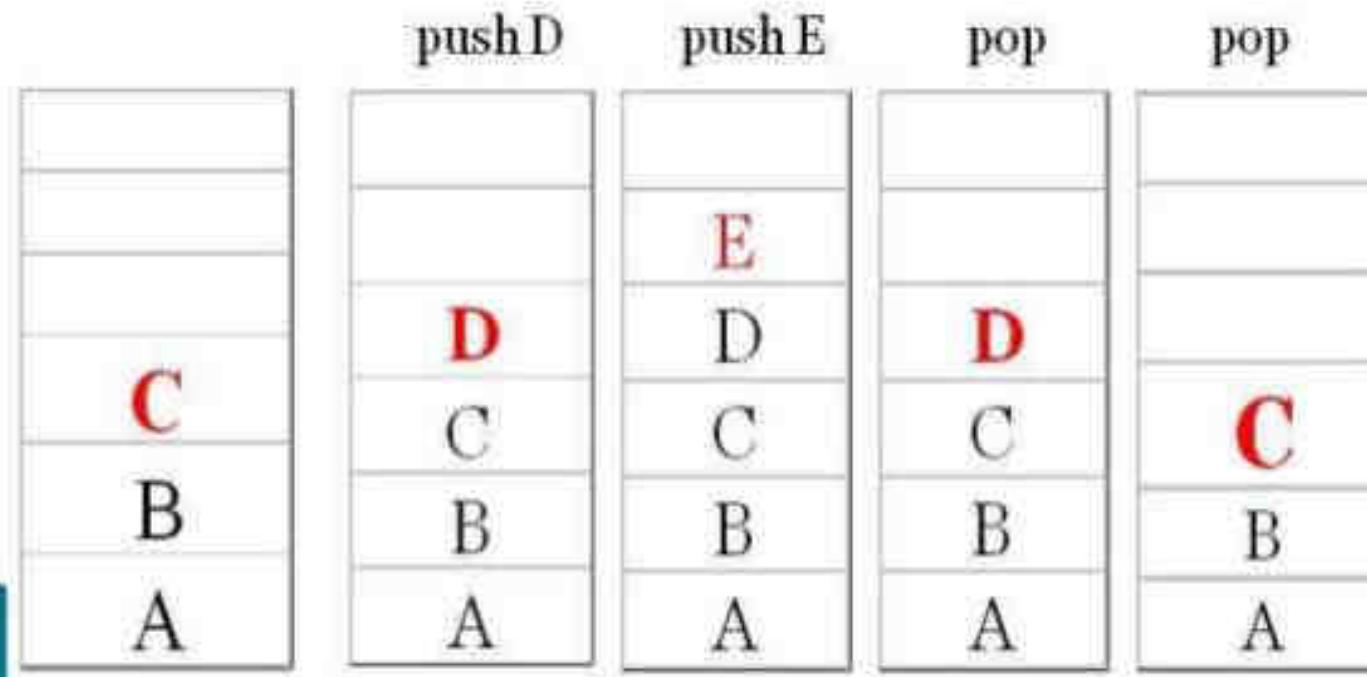
EXAMPLES OF STACK:



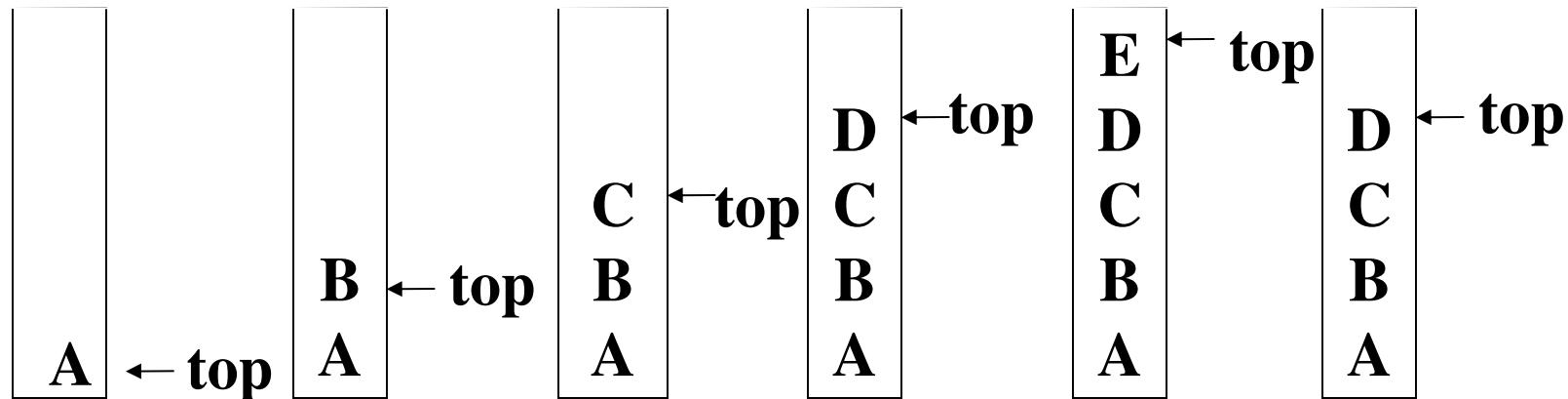
PUSH : It is used to insert items into the stack.

POP: It is used to delete items from stack.

TOP: It represents the current location of data in stack.



stack: a Last-In-First-Out (LIFO) list



Top=-1

*Figure 3.1: Inserting and deleting elements in a stack (p.102)

Abstract Data Type for stack

ADT *Stack* is

objects: a finite ordered list with zero or more elements.

functions:

for all $stack \in Stack$, $item \in element$, $max_stack_size \in \text{positive integer}$

Stack CreateS(max_stack_size) ::=

create an empty stack whose maximum size is
 max_stack_size

Boolean IsFull(stack, max_stack_size) ::=

if (number of elements in $stack == max_stack_size$)
return TRUE
else return FALSE

Stack Push(stack, item) ::=

if (*IsFull(stack)*) $stack_full$
else insert $item$ into top of $stack$ and return

Boolean IsEmpty(stack) ::=
 if(Number of elements are 0)
 return TRUE
 else return FALSE

Element Pop(stack) ::=
 if(IsEmpty(stack)) return
 else remove and return the *item* on the top
 of the stack.

Operations on a Stack

The following operations are performed on the stack...

1.Push (To insert an element on to the stack)

2.Pop (To delete an element from the stack)

3.Display (To display elements of the stack)

Stack data structure can be implemented using:

1.Using Array

2.Using Linked List

- When a stack is implemented using an array, that stack can organize an only limited number of elements.
- When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

```
#include<stdio.h>
#define stackSize 4
void push(char element, char stack[], int *top)
{
    if(*top == stackSize-1)
        {printf("\nStack Full\n");
        return;}
    stack[++(*top)] = element;
}
```

```
void pop(char stack[], int *top){
    if(*top == -1)
    {
        printf("\nThe stack is empty.\n");
        return;
    }
    printf("Element popped: %c \n", stack[(*top)--]);
```

```
int main() {
    char stack[stackSize];
    int top = -1,i;
    push('a', stack, &top);
    push('b',stack, &top);
    push('c', stack, &top);
    for(i=top;i>=0;i--)
        printf("%c\n", stack[i]);
    printf("Top: %d\n", top);
    pop(stack, &top);
}
```

```
printf("Element on top: %c\n",
stack[top]);
pop(stack, &top);
printf("Top: %d\n", top);
pop(stack, &top);
pop(stack,&top);
printf("\n");
push('a', stack, &top);
push('b',stack, &top);
push('c', stack, &top);
push('d', stack, &top);
push('e',stack, &top);
return 0;
```

c

b

A

Top: 2

Element popped: c

Element on top: b

Element popped: b

Top: 0

Element popped: a

The stack is empty.

Stack Full

Implementation: **using array**

```
Stack CreateS(max_stack_size) ::=  
    #define MAX_STACK_SIZE 100 /* maximum stack size */  
    typedef struct {  
        int key;  
        /* other fields */  
    } element;  
    element stack[MAX_STACK_SIZE];  
    int top = -1;
```

Implementation: **using array**

```
Stack CreateS(max_stack_size) ::=  
    #define MAX_STACK_SIZE 100 /* maximum stack size */  
    typedef struct {  
        int key;  
        /* other fields */  
    } element;  
    element stack[MAX_STACK_SIZE];  
    int top = -1;
```

Boolean IsEmpty(Stack) ::= **top < 0**;

Boolean IsFull(Stack) ::= **top >= MAX_STACK_SIZE-1**;

Add to a stack

```
void push(int top, element item)
{
    /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1) {
        stack_full( );
    }
    stack[++top] = item;
}
```

Add to a stack

```
void push(int top, element item)
{
    /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1) {
        stack_full( );
    }
    stack[++top] = item;
}
```

```
Void stack_full()
{
    fprintf(stderr, "Stack is full, cannot add element");
    exit(EXIT_FAILURE);
}
```

Delete from a stack

```
element pop(int *top)
{
    /* return the top element from the stack */
    if (top == -1)
        return stack_empty( ); /* returns and error key */
    return stack[(top)--];
}
```

```
#include<stdio.h>
#define max 6
typedef
{
    int stack[max];
    int top;
}STACK;
int empty(STACK *s)
{
    if(s->top == -1)
    {
        return(1);
    }
    return(0);
}
int full(STACK *s)
{
    if(s->top == max-1)
    {
        return(1);
    }
    return(0);
}
void push(STACK *s, int x)
{
    s->top = s->top+1;
    s->stack[s->top]=x;
}
```

```
int pop(STACK *s)
{
    int x;
    x=s->stack[s->top];
    s->top=s->top-1;
    return(x);
}
void print(STACK s)
{
    int i;
    for(i=s.top;i>=0;i--)
    {
        printf("%d\t",s.stack[i]);
    }
}
```

```
void main()
{
    STACK s;
    int ch,x;
    s.top=-1;
    do{
        printf("1.push\n2.pop\n3.display\n4.exit\n");
        printf("enter your choice\n");
        scanf(" %d",&ch);
```

```
switch(ch)
{
    case 1:printf("enter value\n");
        scanf("%d",&x);
        if(!full(&s))
            push(&s,x);
        else printf("Stack is overflow\n"); break;
    case 2:if(!empty(&s))
    {
        x=pop(&s);
        printf("popped is %d\n",x);
        printf("\nRemaining elements:");
        print(s);
    }
    else printf("stack is empty\n"); break;
    case 3:print(s); break;
}
```

Stack Applications

- **Expression conversion**
 - Infix $(a+b*c) \quad a+b*c^d^e$
 - Prefix $(+a*bc)$
 - Postfix $(abc*+)$
- **Evaluation of expression**
 - $1+2*3 \rightarrow 123*+ \rightarrow 16+ \rightarrow 7$
- **Used to check parenthesis matching in an expression.**
- $a+(b-c*(d-e)/f)-g \rightarrow$ push((), push(() pop pop
- **Recursion handling**

Infix to postfix

1. $A + B \rightarrow AB+$
2. $A+B-C \rightarrow AB+ - C \rightarrow AB+C-$
3. $A + B * C \rightarrow A + BC^* \rightarrow ABC^*+$
4. $A^* B + C \rightarrow AB^* + C \rightarrow AB^*C+$
5. $(a + b - c)^* d - (e + f)$
 $(ab+ - c)^* d - (e + f)$
 $ab+c-^* d - (e + f)$
 $ab+c-^* d - ef+$
 $ab+c- d^* - ef+$
 $ab+c- d^* ef+-$

A/B-C*D^E^F

**A + (B - C) ^D^E
A + BC-DE^^+**

A /B - C * D ^ EF^

A /B - C * DEF^

A /B - C* DEF^

AB/ - C * DEF^

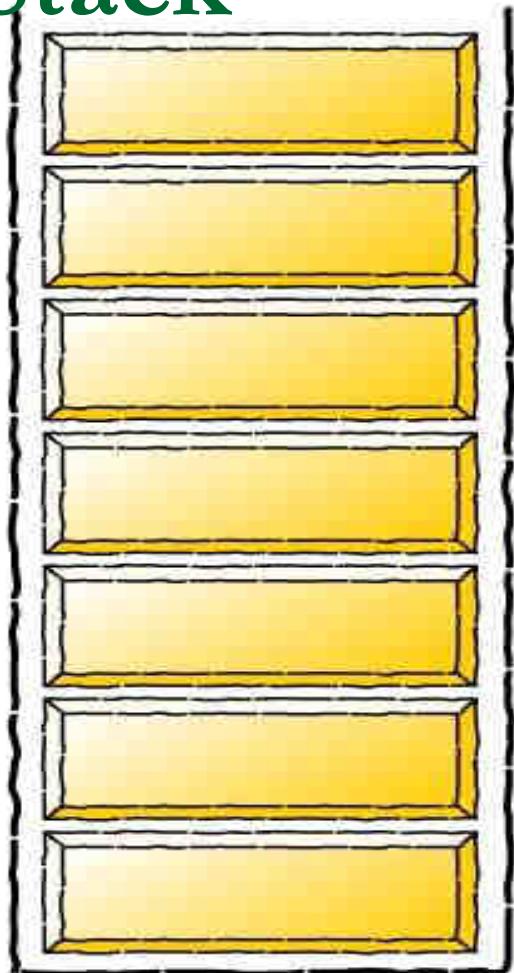
AB/ - CDEF^*

AB/CDEF^*-

Algorithm: -(initially # is pushed on the top of stack with priority value 0)

1. Scan each character (C) of an input string till end from left to right and repeat the steps 1 to 6
2. if C is an operand then append it to postfix array
3. else if C is a right parenthesis then C= POP and append C to the postfix array until a left parentheses is seen on top of stack. pop (POP ones more to remove '(' but do not append it to postfix)
4. else if C is '(' PUSH it on to the stack.
5. else if C is operator Then
 - If the top of the stack is \$/^ and C is also \$/^ then PUSH C
 - else
 - POP and append to postfix as long as priority of C <=stack top
 - PUSH C
6. If stack is not empty, then POP everything from stack and append to postfix.
7. Output the result as postfix array.

Infix to postfix conversion using Stack



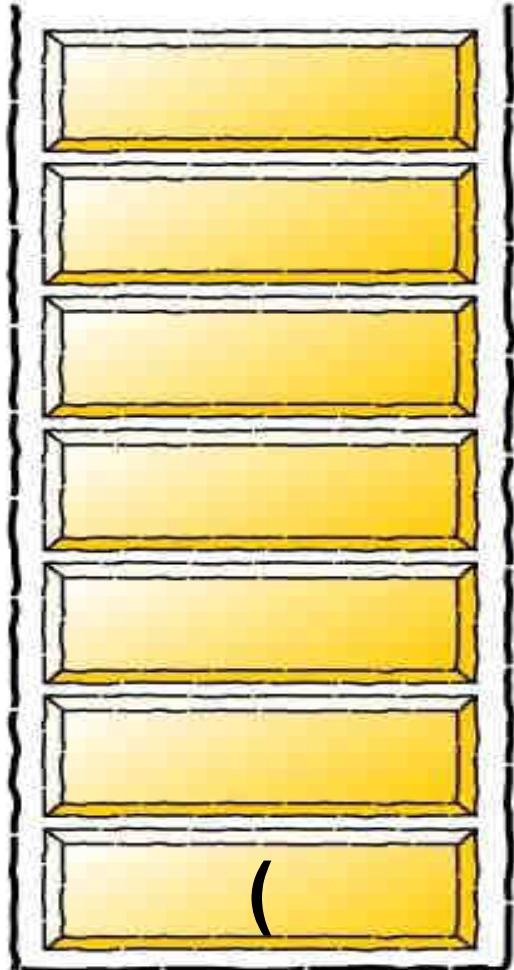
Infix expression

$$(a + b - c) * d - (e + f)$$

Postfix Expression

Infix to postfix conversion

stackVect



infixVect

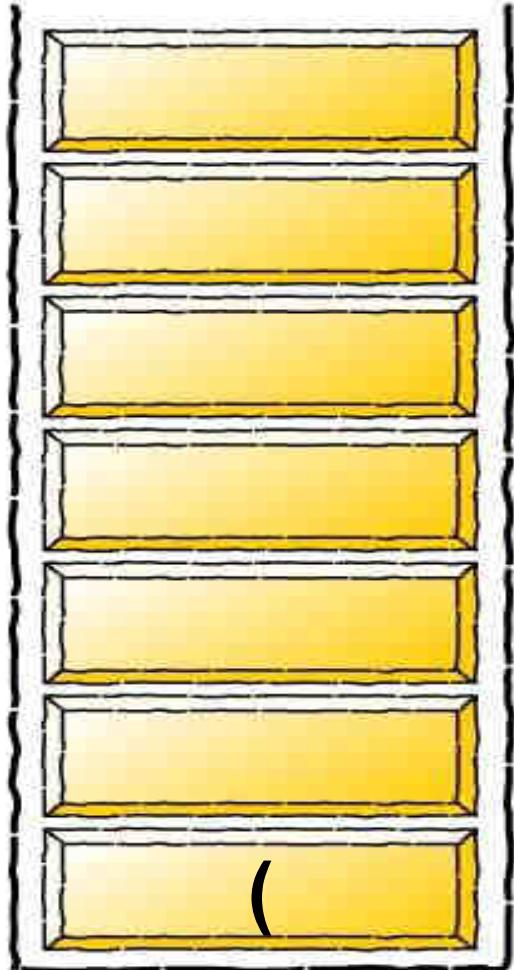
a + b - c) * d - (e + f)

postfixVect



Infix to postfix conversion

stackVect



infixVect

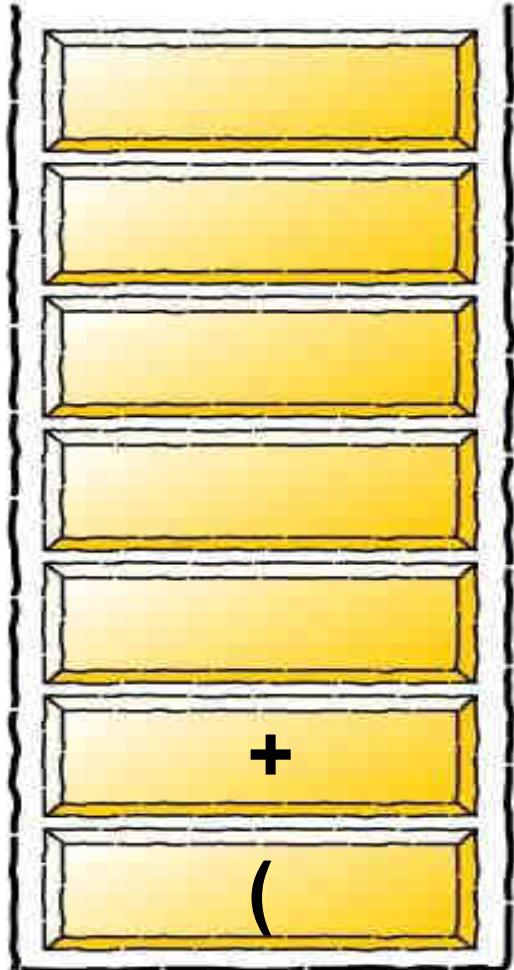
+ b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion

stackVect



infixVect

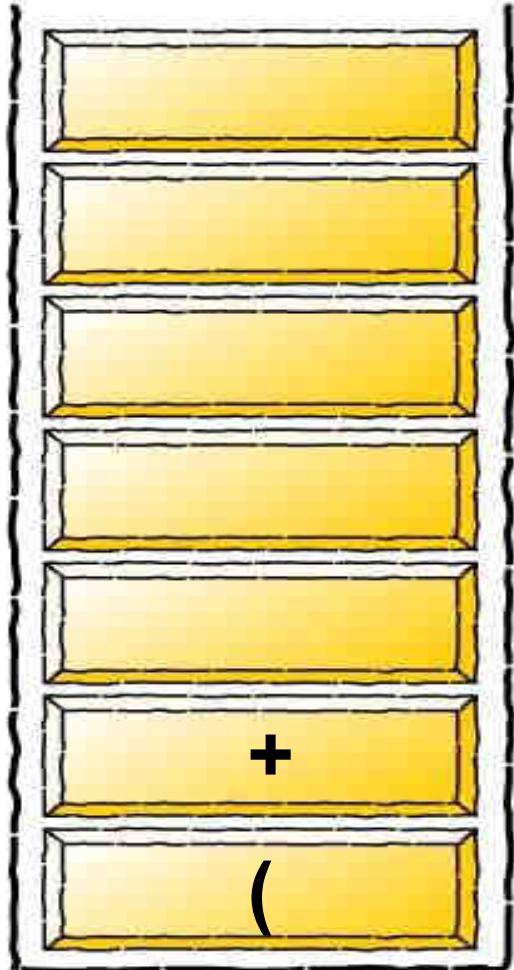
b - c) * d - (e + f)

postfixVect

a

Infix to postfix conversion

stackVect



infixVect

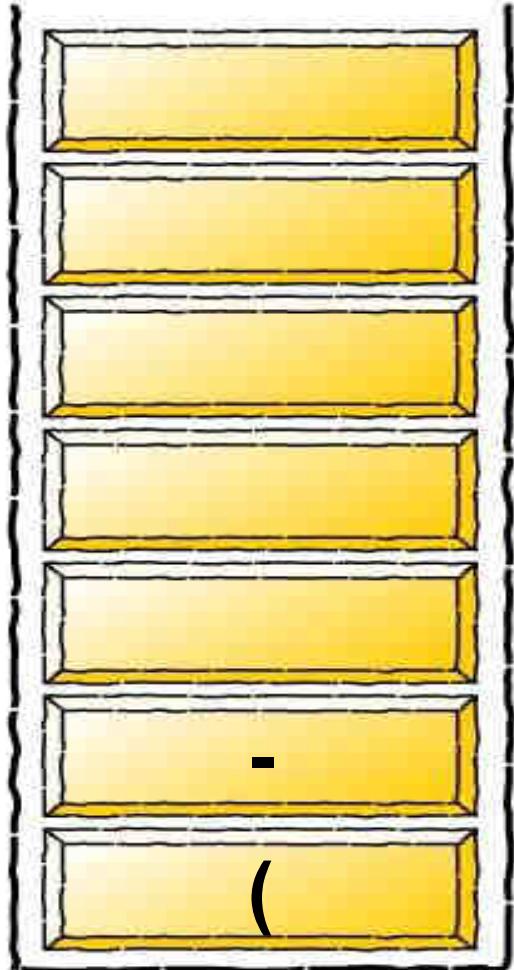
- c) * d - (e + f)

postfixVect

a b

Infix to postfix conversion

stackVect



infixVect

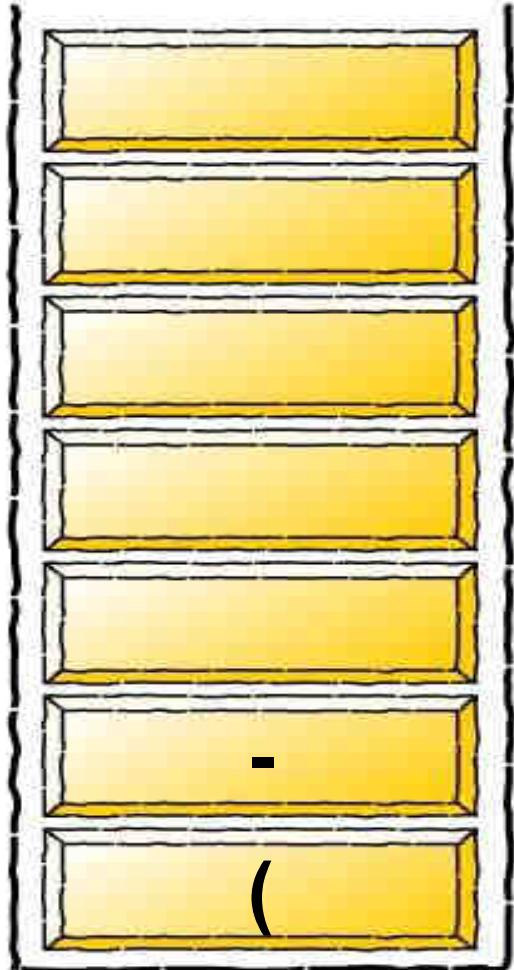
c) * d - (e + f)

postfixVect

a b +

Infix to postfix conversion

stackVect



infixVect

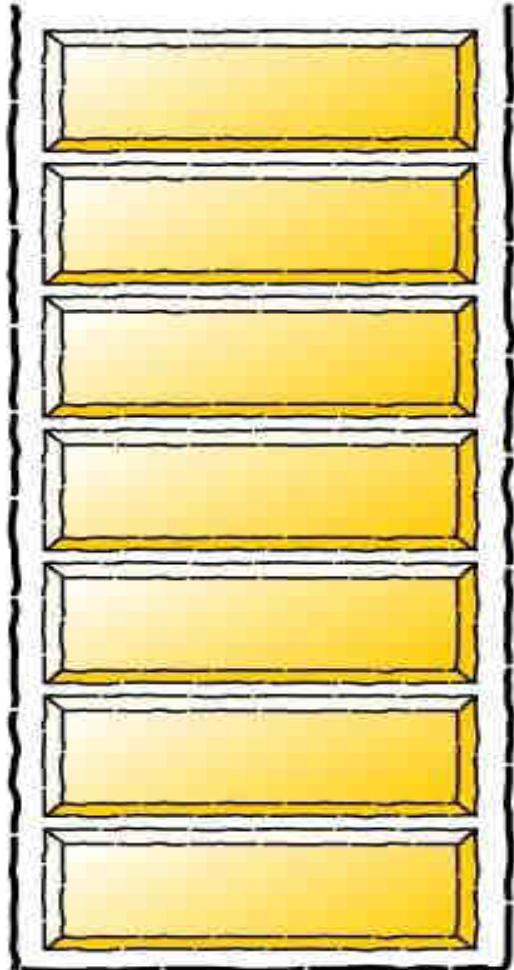
) * d – (e + f)

postfixVect

a b + c

Infix to postfix conversion

stackVect



infixVect

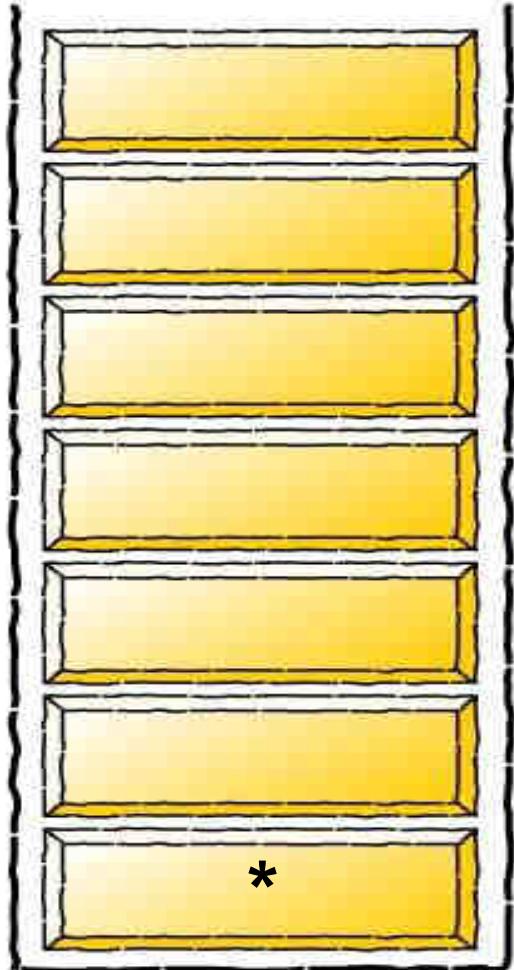
* d - (e + f)

postfixVect

a b + c -

Infix to postfix conversion

stackVect



infixVect

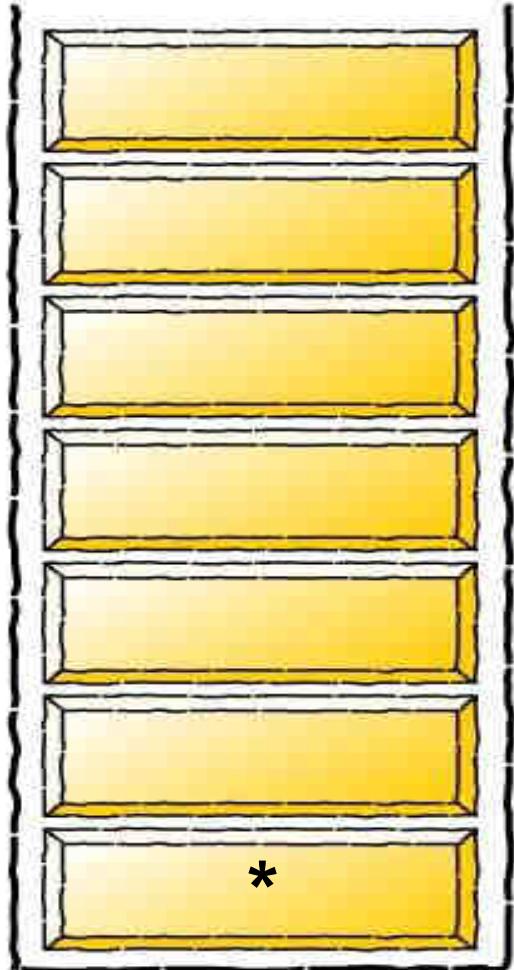
d - (e + f)

postfixVect

a b + c -

Infix to postfix conversion

stackVect



infixVect

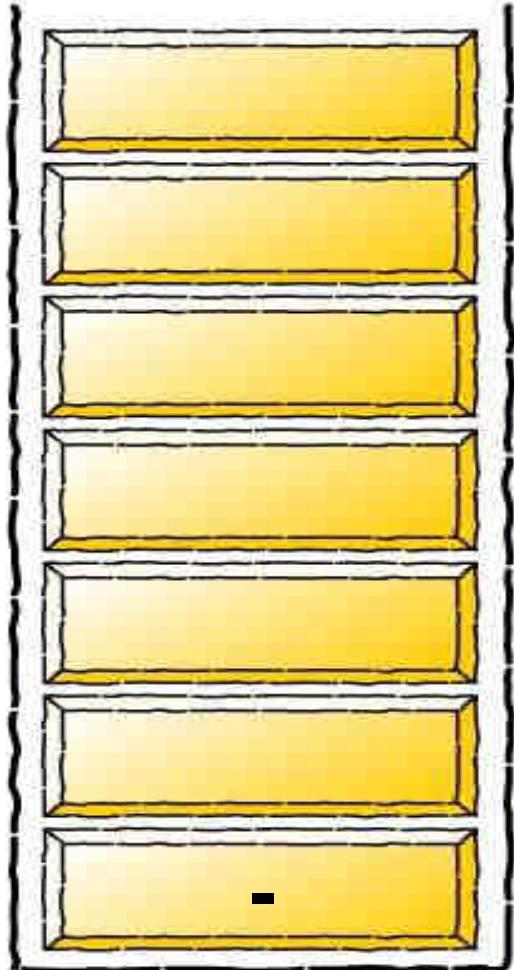
$- (e + f)$

postfixVect

a b + c - d

Infix to postfix conversion

stackVect



infixVect

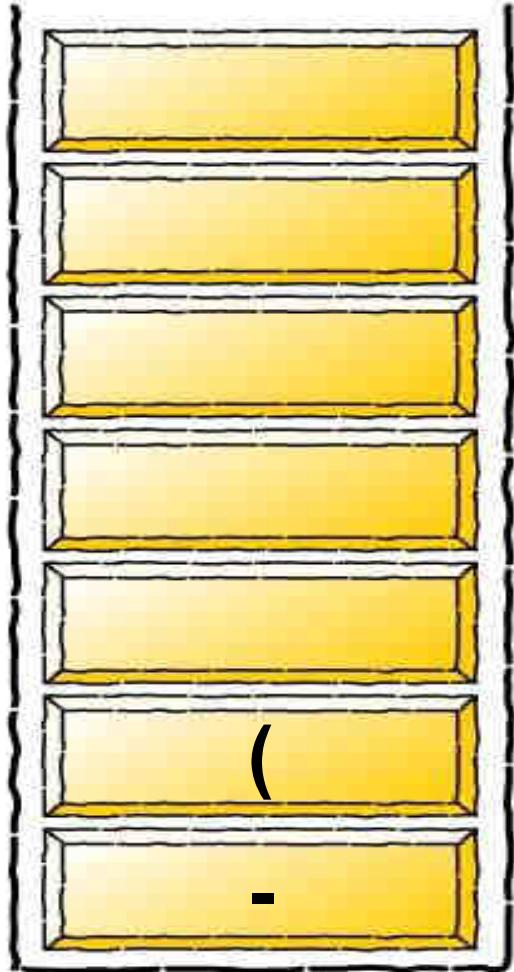
(e + f)

postfixVect

a b + c - d *

Infix to postfix conversion

stackVect



infixVect

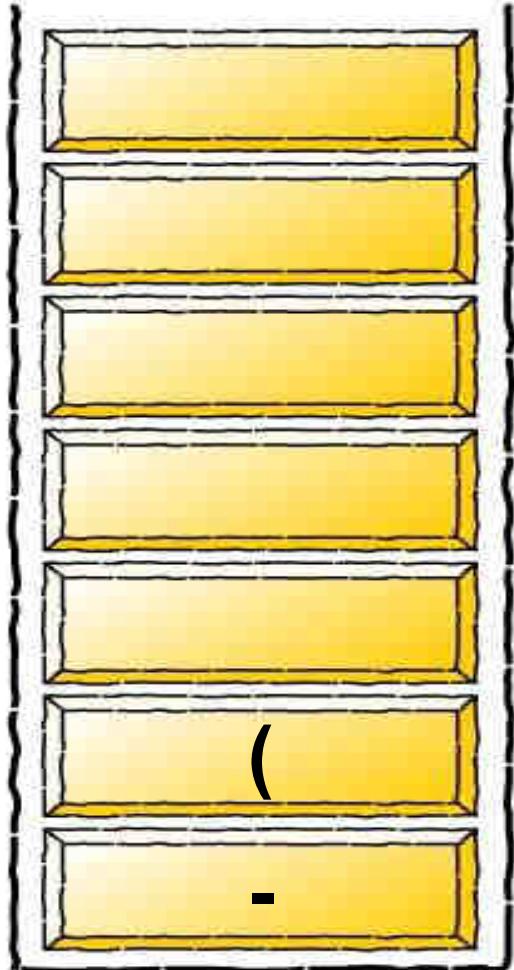
e + f)

postfixVect

a b + c - d *

Infix to postfix conversion

stackVect



infixVect

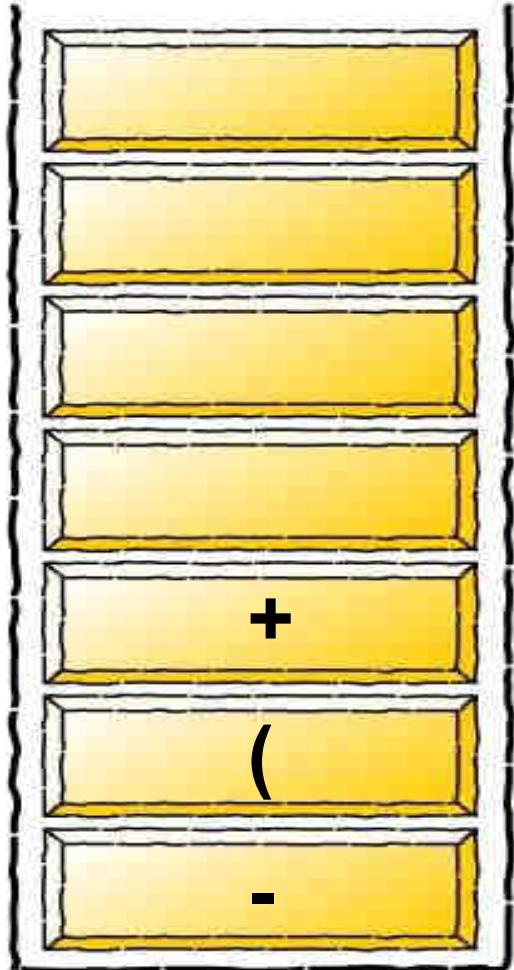
+ f)

postfixVect

a b + c - d * e

Infix to postfix conversion

stackVect



infixVect

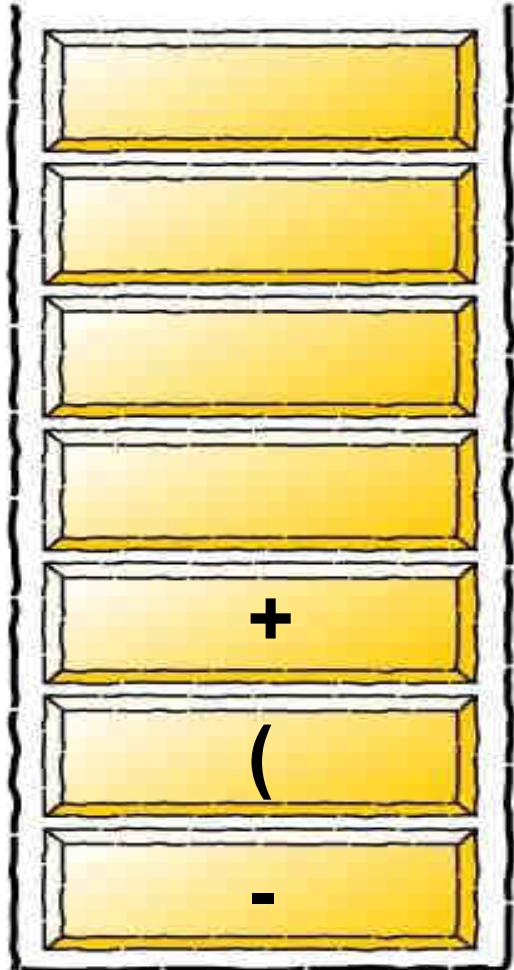
f)

postfixVect

a b + c - d * e

Infix to postfix conversion

stackVect



infixVect

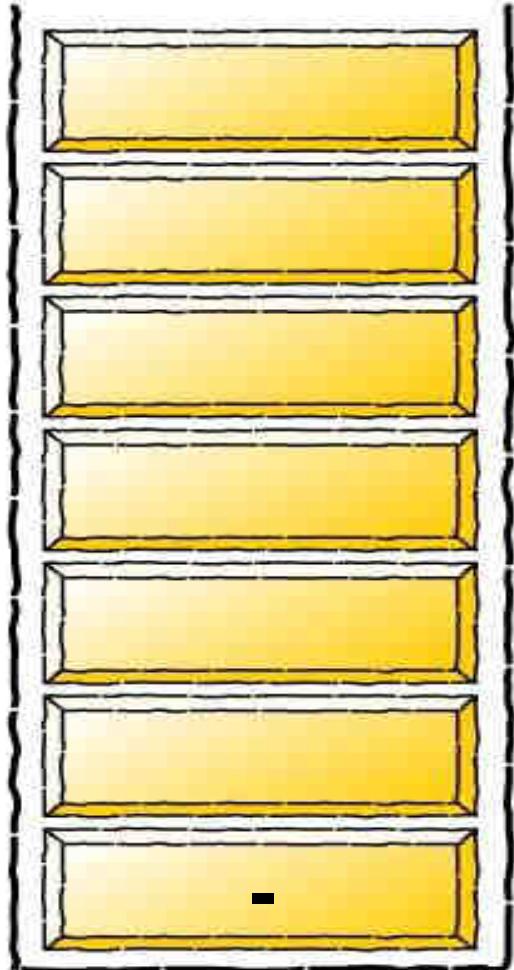
)

postfixVect

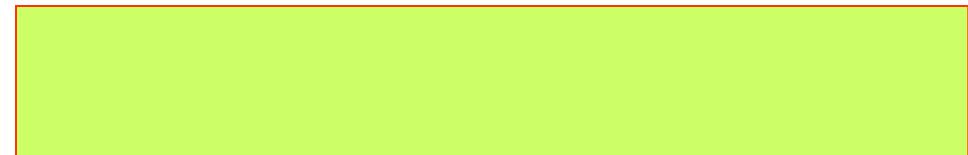
a b + c - d * e f

Infix to postfix conversion

stackVect

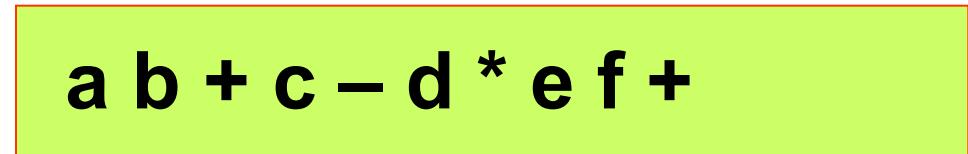


infixVect



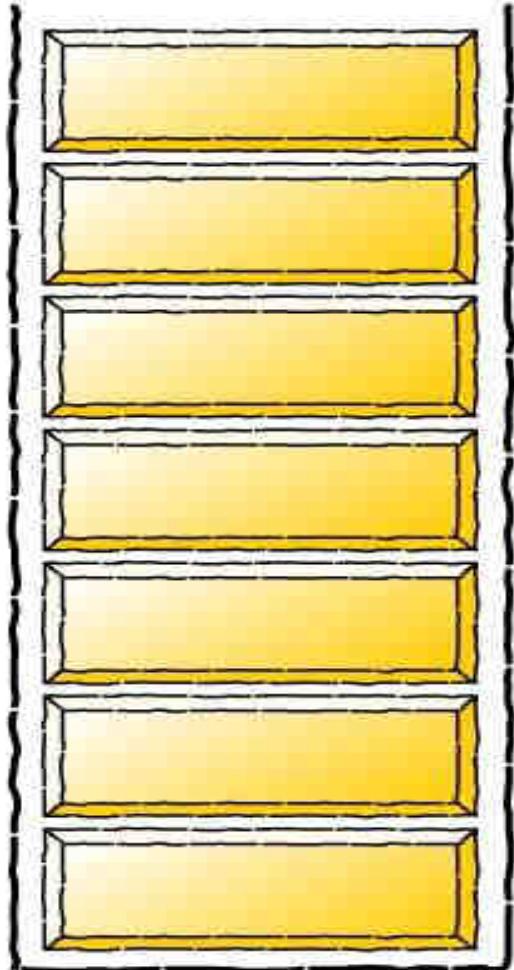
postfixVect

a b + c - d * e f +

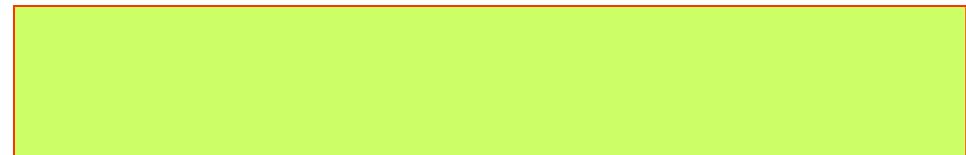


Infix to postfix conversion

stackVect



infixVect



postfixVect

a b + c - d * e f + -

INPUT CHARACTER	OPERATION ON STACK	STACK(#)	POSTFIX EXPRESS
A		Empty	A
*	Push	*	A
B		*	A B
-	Pop and Push	-	A B *
(Push	- (A B *
C		- (A B * C
+	Push	- (+	A B * C
D		- (+	A B * C D
)	Pop and append to postfix till '('	-	A B * C D +
+	Pop and push	+	A B * C D + -
E		+	A B * C D + - E
End of Input	Pop till Empty(#)	Empty	A B * C D + - E +

```

#define SIZE 50
#include<string.h>
#include <stdio.h>
typedef struct
{
char s[SIZE];
int top;
}STACK;

void push(STACK *s1,char elem)
{ s1->top++;
  s1->s[s1->top]=elem;
}

char pop(STACK *s1)
{
return(s1->s[s1->top--]);
}

```

```

int pre(char elem)
{
switch(elem)
{
case '#': return 0;
case '(': return 1;
case '+':
case '-': return 2;
case '*':
case '/': return 3;
case '$': return 4;
}
}
```

```
int main()
{
    STACK s1;    s1.top=-1;
    char infix[50],postfix[50],ch,elem;
    int i=0,k=0;
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infix); push(&s1,'#');
    while( (ch=infix[i++]) != '\0')
    {
        switch(ch)
        {
            case '(': push(&s1,ch);break;
            case ')': while( s1.s[s1.top] != '('
                postfix[k++]=pop(&s1);
                elem=pop(&s1);
                break;
        }
    }
}
```

```

case '+':
case '-':
case '*':
case '/':
case '$': if(s1.s[s1.top]=='$&& ch=='$')
    push(&s1,'$') ;
    else
        {while( pre(ch)<=pre(s1.s[s1.top]) )
            postfix[k++]=pop(&s1);
            push(&s1,ch);
        }          break;
default: postfix[k++]=ch;
    }
}
while( s1.s[s1.top]!='#' ) postfix[k++]=pop(&s1);
postfix[k]='\0';
printf("\n\nGiven Infix Expn: %s\t Postfix Expn: %s\n",infix,postfix);
}

```

Infix to prefix

1. $A + B \rightarrow +AB$

2. $A+B-C \rightarrow +AB - C \rightarrow -+ABC$

3. $A + B * C \rightarrow A + *BC \rightarrow +A*BC$

4. $A * B + C \rightarrow *AB + C \rightarrow +*ABC$

5. $(a + b - c) * d - (e + f)$

$(+ab - c) * d - (e + f)$

$-+abc * d - (e + f)$

***-+abcd - +ef**

-*-+abcd ef+

$A \wedge B \wedge C \rightarrow A \wedge \wedge B C \rightarrow ^\wedge A \wedge B C$

THE ALGORITHM: -(initially # is pushed on the top of stack with priority value 0)

1. Scan symbol (C) from right to left and repeat step 1 to 6 for each element of C until the end of input.
2. If an operand is encountered add it to prefix.
3. If a right parenthesis is encountered push it onto STACK.
4. If left parenthesis is encountered, then
 - a. Repeatedly pop from the STACK and add to prefix (each operator on top of stack until a left parenthesis is encountered)
 - b. Remove the left parenthesis
5. If C scanned is \$/^ and top STACK is \$/^ then:
 - a. POP from STACK and append it to prefix.
 - b. PUSH
6. If C is an operator and (NOT \$/^) then
 - a. Repeatedly pop from STACK and add to Prefix each operator from top of STACK which has > precedence than the operator scanned operator.
 - b. Push (C)
8. If stack is not empty, then POP everything from stack and append to prefix.
9. Output the result as prefix in reverse order.

```
int pre(char elem)
{
    switch(elem)
    {
        case '#': return 0;
        case ')': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
        case '$': return 4;
    }
}
```

```
int main()
{
    STACK s1;
    s1.top=-1;
    char infix[50],prefix[50],ch,elem,temp;
    int i=0,k=0;
    printf("\\n\\nRead the Infix Expression ? ");
    scanf(" %s",infix);
    push(&s1,'#');
    strrev(infix);
```

```
while( (ch=infix[i++]) != '\0')
```

```
{
```

```
switch(ch)
```

```
{
```

```
    case ')': push(&s1,ch);
```

```
    printf("PUSH %c\n",ch);
```

```
        break;
```

```
    case '(': while( s1.s[s1.top] != ')')
```

```
    {
```

```
        temp=pop(&s1);
```

```
        prefix[k++]=temp;
```

```
    }
```

```
    elem=pop(&s1);
```

```
    break;
```

```

case '+':
case '-':
case '*':
case '/':
    while( pre(s1.s[s1.top]) > pre(ch) )
    {
        temp=pop(&s1);
        prefix[k++]=temp;
        // prefix[k++]=pop(&s1);
        printf("POP and APPEND %c\n",temp);
    }
    push(&s1,ch);
    printf("PUSH %c\n",ch);
    break;
case '$': if(s1.s[s1.top]=='$')
{
    temp=pop(&s1);
    prefix[k++]=temp;
    printf("POP and APPEND %c\n",temp);
}

```

```
push(&s1,'$') ;  
                break;  
default:prefix[k++]=ch;  
    } }
```

```
while( s1.s[s1.top]!='#'){  
    temp=pop(&s1);  
    prefix[k++]=temp;  
    prefix[k]='\0';  
    strrev(prefix);  
    strrev(infix);  
    printf("\n\nGiven Infix Expn: %s  Prefix Expn: %s\n",infix,prefix);  
}
```

Balanced parentheses

BALANCED EXPRESSION	UNBALANCED EXPRESSION
$(a + b)$	$(a + b$
$[(c - d) * e]$	$[(c - d * e]$
$\{()\}[]$	$\{[()]\}$

Steps to check for balanced or unbalanced expression:

Input the expression and put it in a character stack.

- Scan the characters from the expression one by one.
- If the scanned character is a starting bracket (‘ (‘ or ‘ { ‘ or ‘ [‘), then push it to the stack.
- If the scanned character is a closing bracket (‘) ’ or ‘ } ’ or ‘] ’), then pop from the stack and if the popped character is the equivalent starting bracket, then proceed. Else, the expression is unbalanced.
- After scanning all the characters from the expression, if there is any parenthesis found in the stack or if the stack is not empty, then the expression is unbalanced.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX 20
struct stack
{
    char stk[MAX];
    int top;
}s;

void push(char item)
{
if (s.top == (MAX - 1)) printf ('Stack is Full\n');
else
{
    s.top = s.top + 1; // Push the char and increment top
    s.stk[s.top] = item;
}}
void pop()
{
if (s.top == - 1)
printf ('Stack is Empty\n');
else
    s.top = s.top - 1; // Pop the char
    and decrement top
}
```

```
int main()
{
char exp[MAX];
int i = 0; s.top = -1;
printf("\nINPUT THE EXPRESSION : ");
scanf("%s",exp);
for(i = 0;i < strlen(exp);i++)
{
if(exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
{
push(exp[i]); // Push the open bracket
}
else if(exp[i] == ')' || exp[i] == ']' || exp[i] == '}') // If a closed
bracket is encountered
{
if(exp[i] == ')')
{ if(s.stk[s.top] == '(')
pop(); // Pop the stack until closed bracket is found
```

```
else
{
printf("\nUNBALANCED EXPRESSION\n");
break;
}}
else if(exp[i] == ']')
{
if(s.stk[s.top] == '[')
pop(); // Pop the stack until closed bracket is found
else
{
printf("\nUNBALANCED EXPRESSION\n");
break;
}}
else if(exp[i] == '}')
{
if(s.stk[s.top] == '{')
pop(); // Pop the stack until closed bracket is found
```

Algorithm: Evaluate prefix expression

Reverse the given expression and scan each digit till the end

- 1. If the character is an operand, push it to the operand stack.**
- 2. If the character is an operator,**
 - 1. pop the operand from the stack, say it's op1.**
 - 2. pop another operand from the stack, say it's op2.**
- 3. perform (op1 operator op2) and push it to stack.**
- 4. pop from the stack and return the result.**

Reversed Prefix Expression: $2563+5*2*/-$

Token	Action	Stack
2	Push 2 to stack	[2]
5	Push 5 to stack	[2, 5]
6	Push 6 to stack	[2, 5, 6]
3	Push 3 to stack	[2, 5, 6, 3]
+	Pop 3 from stack	[2, 5, 6]
	Pop 6 from stack	[2, 5]
	Push 3+6 =9 to stack	[2, 5, 9]
5	Push 5 to stack	[2, 5, 9, 5]
*	Pop 5 from stack	[2, 5, 9]
	Pop 9 from stack	[2, 5]
	Push 5*9=45 to stack	[2, 5, 45]
2	Push 2 to stack	[2, 5, 45, 2]
*	Pop 2 from stack	[2, 5, 45]
	Pop 45 from stack	[2, 5]
	Push 2*45=90 to stack	[2, 5, 90]
/	Pop 5 from stack	[2, 5]
	Pop 90 from stack	[2]
	Push 90/5=18 to stack	[2, 18]
-	Pop 18 from stack	[2]
	Pop 2 from stack	[]
	Push 18-2=16 to stack	[16]

```
#define SIZE 50
#include<string.h>
#include <stdio.h>
typedef struct
{
    int s[SIZE];
    int top;
}STACK;
void push(STACK *s1,int
elem)
{ s1->top++;
    s1->s[s1->top]=elem;
}
```

```
int pop(STACK *s1)
{
    return(s1->s[s1->top--]);
}
```

```
int evaluate(char opr, int n1, int n2)
{
    switch(opr)
    {
        case '+': return (n1+n2);
        case '-': return (n1-n2);
        case '*': return (n1*n2);
        case '/': return (n1/n2);
        case '$': return pow(n1,n2);
    }
}
```

```
int main()
{
    STACK s1;
    s1.top=-1;
    int op1,op2,res;
    char prefix[50],ch,elem;
    int i=0,k=0;
    printf("\n\nRead prefix Expression ? ");
    scanf("%s",prefix);
    strrev(prefix);
    while( (ch=prefix[i++]) != '\0')
    {
        if(isdigit(ch))      push(&s1,ch-'0');
        else {
            op1=pop(&s1);      op2=pop(&s1);
            res=evaluate(ch,op1,op2);
            push(&s1,res);}
    }
    printf("\nThe result is%d",pop(&s1));}
```


Palindrome Check Using Stack

```
typedef struct
{
    char s[SIZE];
    int top;
}STACK;
void push(STACK *s1,char elem)
{ s1->top++;
    s1->s[s1->top]=elem;
}
char pop(STACK *s1)
{
    return(s1->s[s1->top--]);
}
```

```
int main()
{
    STACK s1;
    s1.top=-1;
    char string1[50];
    int i=0;
    printf("\n\nRead the string\n");
    scanf("%s",string1);
    if(isPalindrome(s1,string1))
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

```
int isPalindrome(STACK s1,char str[])
```

```
{
```

```
    int length = strlen(str);
```

```
    int i, mid = length/2;
```

```
    for (i = 0; i < mid; i++)
```

```
        push( &s1,str[i]);
```

```
// Checking if the length of the string
```

```
// is odd, if odd then neglect the
```

```
// middle character
```

```
if (length % 2 != 0)
```

```
    i++;
```

```
// While not the end of the string
```

```
while (str[i] != '\0') {
```

```
    char ele = pop(&s1);
```

```
// If the characters differ then the
```

```
// given string is not a palindrome
```

```
if (ele != str[i])
```

```
    return 0;
```

```
    i++;
```

```
}
```

```
return 1;
```

```
}
```

Decimal to binary conversion

```
typedef struct
```

```
{
```

```
int s[SIZE];
```

```
int top;
```

```
}STACK;
```

```
void push(STACK *s1,int elem)
```

```
{ s1->top++;
```

```
 s1->s[s1->top]=elem;
```

```
}
```

```
int pop(STACK *s1)
```

```
{
```

```
 return(s1->s[s1->top--]);
```

```
}
```

```
int main()
```

```
{
```

```
 STACK s1; s1.top=-1;
```

```
 int num;
```

```
 printf("Enter an integer : ");
```

```
 scanf("%d",&num);
```

```
 while(num!=0)
```

```
{
```

```
 push(&s1,num%2);
```

```
 num=num/2;
```

```
}
```

```
 printf("Binary Equivalent is : ");
```

```
 while(s1.top!=-1)
```

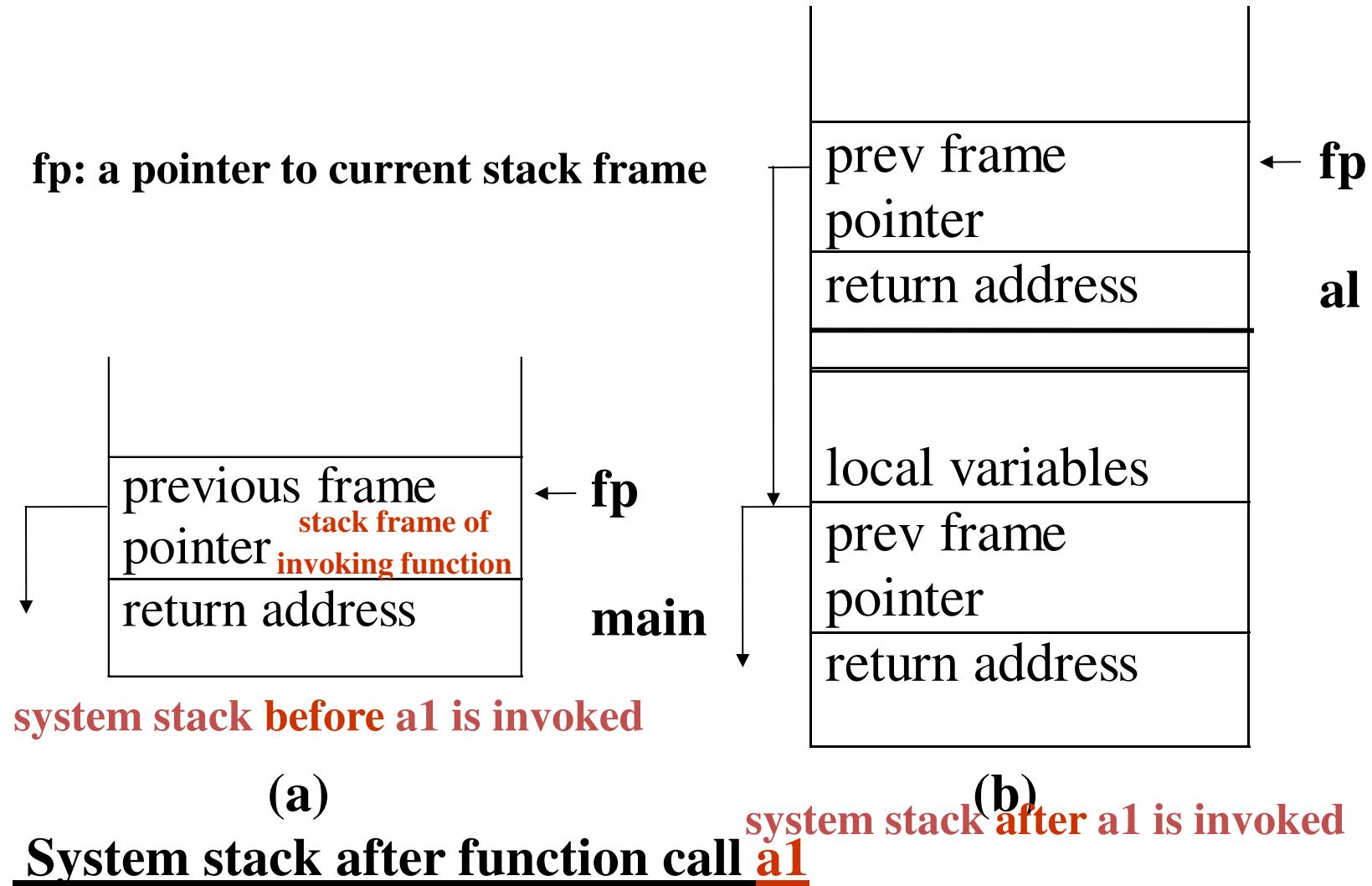
```
 printf("%d",pop(&s1));
```

```
 printf("\n");
```

```
 return 0;
```

```
}
```

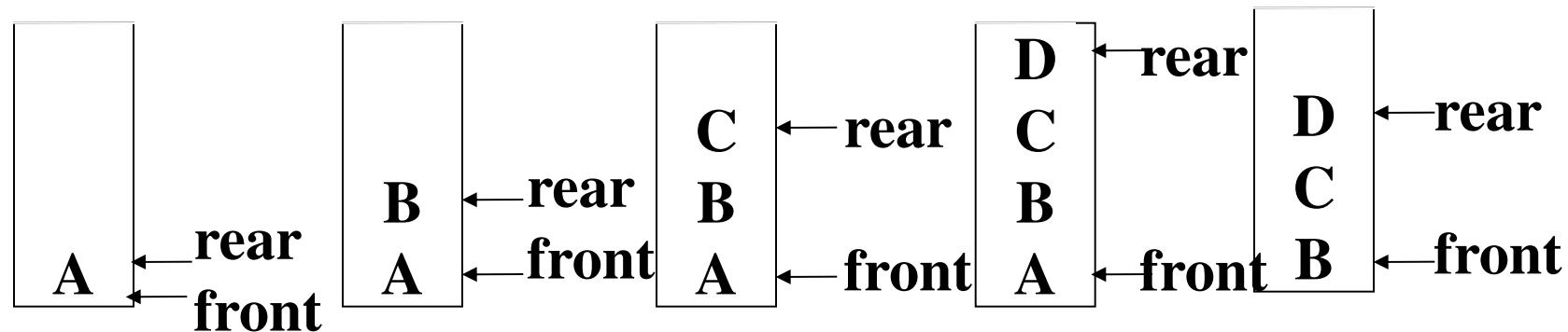
An application of stack: stack frame of function call (activation record)



Queues

- Definition: A queue is an **ordered list** in which insertions and deletions takes place at different ends.
- **New elements are added to rear end**
- **Old elements are deleted from front end**
- Since first element inserted is the first element deleted queues is also known as **First-In-First-Out (FIFO) lists.**

Queue: a First-In-First-Out (FIFO) list



*Figure 3.4: Inserting and deleting elements in a queue (p.106)

Implementation of Queue

```
#include<stdio.h>
#define max 4
typedef struct
{
    int que[max];
    int front;
    int rear;
} QUEUE;
int isfull( QUEUE q)
{
    if(q.rear==max-1) return 1;
    return 0;
}
int isempty(QUEUE q)
{
    if(q.front==q.rear)
        return 1;
    return 0; }
```

```
void insertq( QUEUE *q, int ele)
{ q->que[++q->rear]=ele;
}
```

```
int deleteq(QUEUE *q)
{
    return (q->que[q->front++]);
}
```

```
void displayq(QUEUE q)
{
    int i;
    for(i=q.front+1;i<=q.rear; i++)
        printf("%d\t",q.que[i]);
    return;
}
```

```
int main()
{
    QUEUE q;
    q.front=-1;
    q.rear=-1;
    int ch,i,x;
```

Do
{
printf("\n1.Insert\n2.Delete\n3.
Dispaly\n4.exit\n");
printf("enter your choice\n");
scanf("%d",&ch);

```
switch(ch)
{
    case 1: printf("enter the element\n");
              scanf("%d",&x);
              if(isfull(q))
                  printf("Queue is full\n");
              else
                  insertq(&q,x);
                  break;
    case 2:if(isempty(q))
              printf("Queue is empty\n");
              else {
                  x=deleteq(&q);
                  printf("Deleted element %d\n",x); }   break;
    case 3: if(isempty(q))
              printf("Queue is empty\n");
              else   displayq(q);      break;  } }while(ch!=4);
return 0;}
```

Queues - Application

- Used by operating system (OS) to create job queues.
- If OS does not use priorities then the jobs are processed in the order they enter the system

circular queue

```
typedef struct
{
    int que[max];
    int front;
    int rear;
} Q;
```

```
int isfull( Q q)
{
if((q.rear+1)%max)==q.front)
return 1;
return 0;
}
```

```
int isempty(Q q)
{
if(q.front==q.rear) return 1;
return 0;
}
```

```
void addq(Q *q, int ele)
{
    q->rear=(q->rear+1)%max;
    q->que[q->rear]=ele;
}
```

```
void displayq(Q q)
{
    int i;
    for(i = q.front+1; i != q.rear; i = (i+1)%max)
        printf("%d ", q.que[i]);
    printf("%d ", q.que[i]);
}
```

```
int deleteq(Q *q)
{ int ele;
    q->front=(q->front+1)%max;
    ele=q->que[q->front];
    return ele;
}
```

```
int main()
{
    Q q;
    q.front=0;    q.rear=0;    int ch,i,x;
    do{
        printf("\n1.Insert\n2.Delete\n3.Dispaly\n4.exit\n");
        printf("enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("enter the element\n");
                    scanf("%d",&x);
                    if(isfull(q))
                        printf("Queue is full\n");
                    else
                        insertq(&q,x);
                    break;
            case 2: if(isempty(q))
                    printf("Queue is empty\n");
                    else
                        deleteq(&q);
                    break;
            case 3: dispalyq(q);
                    break;
            case 4: exit(0);
        }
    }while(ch!=4);
}
```

```
case 2: if(isempty(q))
          printf("Queue is empty\n");
        else
        {
          x=deleteq(&q);
          printf("Deleted element %d\n",x);
        }
        break;
case 3: if(isempty(q))
          printf("Queue is empty\n");
        else
          displayq(q);
        break;
}}while(ch!=4);
return 0;}
```

Priority Queue

```
void insertq( QUEUE *q, int ele)
{int i;
if(q->front== -1)
{q->front=0;

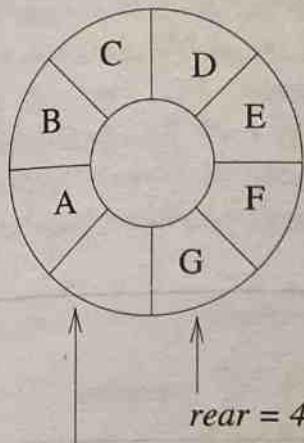
q->que[++q->rear]=ele;
}
else
{
    for(i=q->rear;i>=0;i--)
    if(ele<q->que[i])
        q->que[i+1]=q->que[i];
    else break;
    q->que[i+1]=ele;
    q->rear=q->rear+1;
}
}
```

The possible operation performed on deque are

- 1. Add an element at the rear end**
- 2. Add an element at the front end**
- 3. Delete an element from the front end**
- 4. Delete an element from the rear end**
- 5. Only 1st,3rd and 4th operations are performed by input-restricted deque**
- 6. 1st,2nd and 3rd operations are performed by output-restricted deque.**

- 1. An input-restricted deque is one where deletion can be made from both ends, but insertion can be made at one end only.**
- 2. An output-restricted deque is one where insertion can be made at both ends, but deletion can be made from one end only.**

Circular Queues using Dynamically Allocated Arrays



(a) A full circular queue

queue	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	C	D	E	F	G		A	B

front = 5, rear = 4

(b) Flattened view of circular full queue

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G		A	B								

front = 5, rear = 4

(c) After array doubling

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
C	D	E	F	G										A	B

front = 13, rear = 4

(d) After shifting right segment

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
A	B	C	D	E	F	G									

front = 15, rear = 6

(e) Alternative configuration

Figure 3.7: Doubling queue capacity

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



$front = 5, rear = 4$

(c) After array doubling

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



$front = 13, rear = 4$

(d) After shifting right segment

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



$front = 15, rear = 6$

(e) Alternative configuration

Figure 3.7: Doubling queue capacity

5Q-6

```
void addQ(element item){  
    rear = (rear+1) % capacity;  
    if (front == rear).  
        queueFull(); /* double capacity */  
    queue [rear] = item;  
}
```

Prog 3.9 - Add to a circular queue (using a dynamically allocated array)

```
void queueFull() {
    /* allocate an array with twice the capacity */
    element * newQueue;
    newQueue = malloc(newQueue, 2 * capacity * sizeof(*queue));
    /* copy from queue to newQueue */
    int start = (front + 1) % capacity;
    if (start < 2)
        /* no wrap around */
        copy(queue + start, queue + start + capacity - 1, newQueue);
    else {
        /* queue wraps around */
        copy(queue + start, queue + capacity, newQueue);
        copy(queue, queue + rear + 1, newQueue + capacity - start);
    }
    ... to newQueue */
}
```

```
    }  
    /* switch to new Queue */  
    front = 2 * capacity - 1; // capacity contains old value  
    rear = capacity - 2; // " "  
    capacity *= 2;  
    free(queue);  
    queue = newQueue;
```

{
Prog 3.10 - Doubling queue capacity

Abstract data type of queue

structure *Queue* is

objects: a finite ordered list with zero or more elements.

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_queue_size ∈ positive integer

Queue CreateQ(max_queue_size) ::=

create an empty queue whose maximum size is

max_queue_size

Implementation 1: using array

```
Queue CreateQ(max_queue_size) ::=  
# define MAX_QUEUE_SIZE 100/* Maximum queue size */  
typedef struct {  
    int key;  
    /* other fields */  
    } element;  
  
element queue[MAX_QUEUE_SIZE];  
int rear = -1;  
int front = -1;
```

Abstract data type of queue

Boolean IsFullQ(queue, max_queue_size) ::=
 if(number of elements in queue ==
max_queue_size)
 return TRUE
 else return FALSE

Queue AddQ(queue, item) ::=
 if (IsFullQ(queue)) queue_full
 else insert item at rear of queue and return queue

Implementation 1: using array

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
if (*rear == MAX_QUEUE_SIZE_1) {
    queue_full( );
    return;
}
queue [++*rear] = item;
}
```

*Program 3.3: Add to a queue (p.108)

Boolean IsEmptyQ(*queue*) ::=
 if (*queue* == CreateQ(*max_queue_size*))
 return *TRUE*
 else return *FALSE*

Element DeleteQ(*queue*) ::=
 if (IsEmptyQ(*queue*)) **return**
 else remove and return the *item* at front of queue.

*Structure 3.2: Abstract data type Queue (p.107)

Implementation 1: using array

Boolean IsEmpty(queue) ::= front == rear

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
if ( *front == rear)
    return queue_empty( ); /* return an error key */
return queue [++ *front];
}
```

*Program 3.4: Delete from a queue(p.108)

problem:

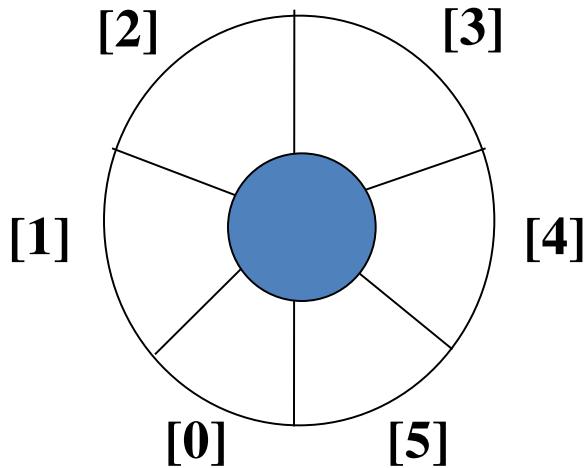
**there may be available space when IsFullQ is true
i.e.. movement is required.**

Implementation 2: regard an array as a circular queue

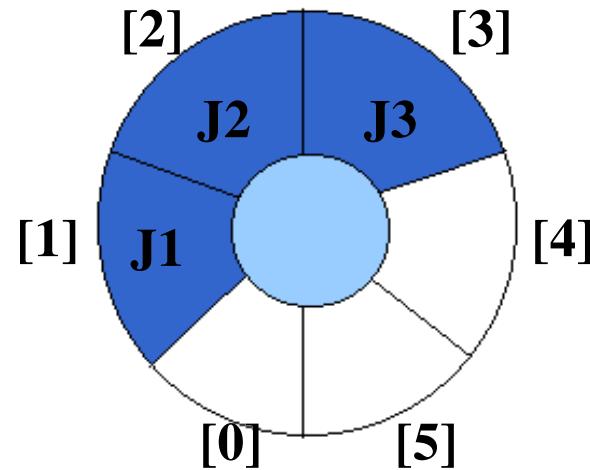
front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE



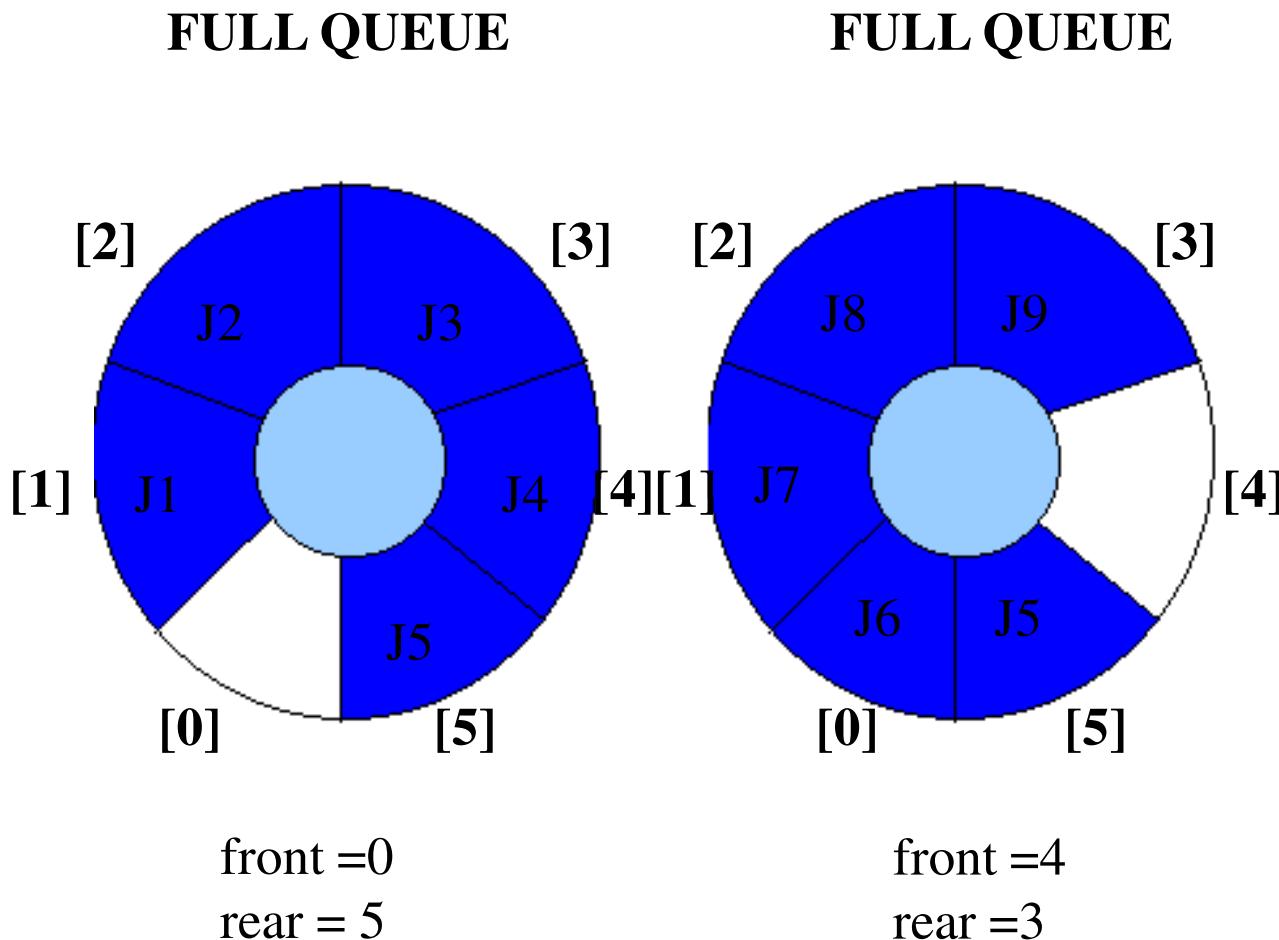
front = 0
rear = 0



front = 0
rear = 3

*Figure 3.6: Empty and nonempty circular queues (p.109)

Problem: one space is left when queue is full



*Figure 3.7: Full circular queues and then we remove the item
(p.110)

Add to a circular queue

```
void addq(int front, int *rear, element item)  
{
```

```
}
```

*Program 3.5: Add to a circular queue (p.110)

Add to a circular queue

```
void addq(int front, int *rear, element item)
{
    /* add an item to the queue */
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print error */
        return;
}

queue[*rear] = item;
```

*Program 3.5: Add to a circular queue (p.110)

Delete from a circular queue

```
element deleteq(int* front, int rear)  
{
```

```
}
```

*Program 3.6: Delete from a circular queue (p.111)

Delete from a circular queue

element deleteq(int* front, int rear)

{

element item;

 /* remove front element from the queue and put it in item
*/

if (*front == rear)

return queue_empty();

 /* queue_empty returns an error key */

***front = (*front+1) % MAX_QUEUE_SIZE;**

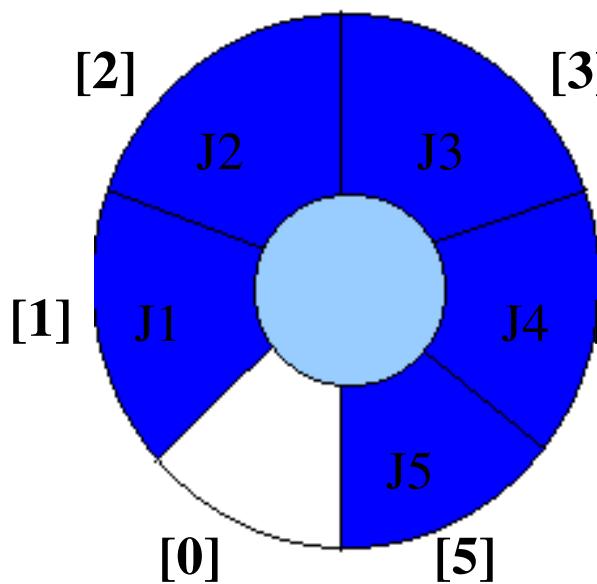
return queue[*front];

}

*Program 3.6: Delete from a circular queue (p.111)

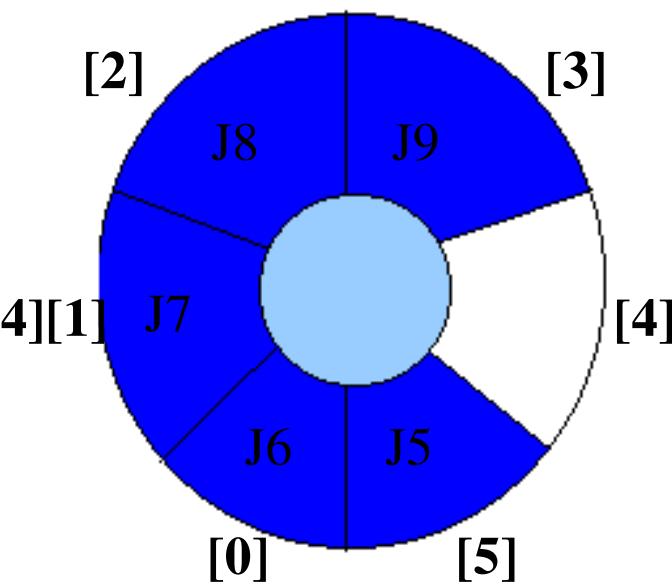
Circular Queue: **full == rear** condition

FULL QUEUE



front = 0
rear = 5

FULL QUEUE



front = 4
rear = 3

Infix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression.** Ex: **a+b*c**

Infix prefix and postfix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression.** Ex: $a+b*c$
- If the operator is written before the operands , it is called **prefix expression** Ex: $+a*bc$
- If the operator is written after the operands, it is called **postfix expression.** Ex: $abc^* +$

Infix, prefix, and Postfix expression

- An expression in infix form is **dependent of precedence during evaluation**
- Ex: to evaluate $a+b*c$, sub expression $a+b$ can be evaluated only after evaluating $b*c$.
- As soon as we get an operator we cannot perform the operation specified on the operands.
- So it takes more time for compilers to check precedence to evaluate sub expression.

Infix, prefix, and Postfix expression

- Both **prefix and postfix representations are independent of precedence of operators.**
- In a **single scan** an entire expression can be evaluated
- **Takes less time to evaluate.**
 - However infix expressions have to be converted to postfix or prefix.

Evaluation of Expressions

$$X = a / b - c + d * e - a * c$$

$$a = 4, b = c = 2, d = e = 3$$

Interpretation 1:

$$((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\cdots$$

Evaluation of Expressions

$X = a / b - c + d * e - a * c$

$a = 4, b = c = 2, d = e = 3$

Interpretation 1:

$$((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\cdots$$

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence ¹	Associativity
()	function call	17	left-to-right
[]	array element		
-> .	struct or union member		
- ++	increment, decrement ²	16	left-to-right
- ++	decrement, increment ³	15	right-to-left
!	logical not		
-	one's complement		
- +	unary minus or plus		
& *	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >=	relational	10	left-to-right
< <=			
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= =	assignment	2	right-to-left
,	comma	1	left-to-right

user

compiler

Infix	Postfix
$2+3*4$	$234*+$
$a*b+5$	$ab*5+$
$(1+2)*7$	$12+7*$
$a*b/c$	
$(a/(b-c+d))* (e-a)^* c$	
$a/b-c+d^* e-a^* c$	

*Figure 3.13: Infix and postfix notation (p.120)

Postfix: no parentheses, no precedence

user

compiler

Infix	Postfix
$2+3*4$	$234*+$
$a*b+5$	$ab*5+$
$(1+2)*7$	$12+7*$
$a*b/c$	$ab*c/$
$(a/(b-c+d))*((e-a)^*c)$	$abc-d+/ea-*c^*$
$a/b-c+d^*e-a^*c$	$ab/c-de^*ac^*-$

Postfix: no parentheses, no precedence

Infix to Postfix Conversion

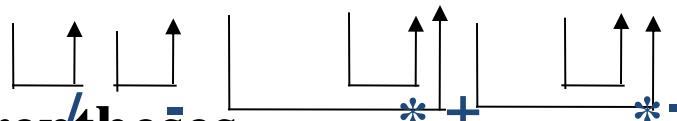
(Intuitive Algorithm/ Manual method)

(1) Fully parenthesize expression

$$\begin{aligned} \textcolor{red}{a / b - c + d * e - a * c} &\rightarrow \\ (\textcolor{red}{((a / b) - c) + (d * e)) - a * c}) \end{aligned}$$

(2) All operators replace their corresponding right parentheses.

$$(\textcolor{red}{((a / b) - c) + (d * e)) - a * c})$$



(3) Delete all parentheses.

$$\textcolor{red}{ab/c-de^*+ac^*-}$$

two passes

Infix to postfix conversion: Sample Exercises

- Convert the following infix expression to postfix expression
- $a+b*c+d*e$
- $a*b+5$
- $(a/(b-c+d))*((e-a)*c)$
- $a/b-c+d*e-a*c$

Evaluation of Postfix Using Stack

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Evaluate postfix expression

Assumptions:

operators: +, -, *, /, % **operands:** single digit integer

Evaluate postfix expression

Assumptions:

operators: +, -, *, /, % **operands:** single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */  
#define MAX_EXPR_SIZE 100 /* max size of expression */
```

```
int stack[MAX_STACK_SIZE]; /* global stack */  
char expr[MAX_EXPR_SIZE]; /* input string -- expression */
```

Evaluate postfix expression

Assumptions:

operators: +, -, *, /, % **operands:** single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */  
#define MAX_EXPR_SIZE 100 /* max size of expression */  
  
typedef enum{lparan, rparen, plus, minus, times, divide,  
             mod, eos, operand} precedence;  
  
int stack[MAX_STACK_SIZE]; /* global stack */  
char expr[MAX_EXPR_SIZE]; /* input string -- expression */
```

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable,
'\0' is the the end of the expression.
```

The stack and top of the stack are global variables.

```
*/  
}
```

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable,
'\0' is the the end of the expression.
```

The stack and top of the stack are global variables.

get_token() is used to return the token type and
the character symbol.

Operands are assumed to be single character digits */

```
}
```

```
precedence get_token(char *symbol, int *n)
```

```
{
```

```
/* get the next token,
```

symbol is the character representation, which is returned,

the token is represented by its enumerated value, which is returned in the function name */

```
}
```

```
int eval(void)
{
    precedence token;
    char symbol;
    int op1, op2;
    int n = 0; /* counter for the expression string */
    int top = -1;

    // Scan left to right
        //If operand push (single digit)
        // If operator pop 2 operands; push the op result
    //If End of Expression, pop the result
```

```
int eval(void)
{
    precedence token;
    char symbol;
    int op1, op2;
    int n = 0; /* counter for the expression string */
    int top = -1;

    // Scan left to right
    token = get_token(&symbol, &n);
    while (token != eos) {
        if (token == operand)
            push(&top, symbol-'0'); /* push operand */
```

```

else {
    /* operator: remove two operands*/
    op2 = pop(&top);
    op1 = pop(&top);
    switch(token) { /* perform operation; result to stack */
        case plus:      push(&top, op1+op2); break;
        case minus:     push(&top, op1-op2); break;
        case times:     push(&top, op1*op2); break;
        case divide:    push(&top, op1/op2); break;
        case mod:       push(&top, op1%op2);
    }
}
token = get_token (&symbol, &n);
} /* End of Expression*/
return pop(&top); /* return result from the stack */
}

```

```
precedence get_token(char *symbol, int *n)
{
    *symbol = expr[(*n)++];
    switch (*symbol) {
        case '(': return lparen;
        case ')': return rparen;
        case '+': return plus;
        case '-': return minus;
```

```
        case '/' : return divide;
        case '*' : return times;
        case '%' : return mod;
        case '\0' : return eos;
        default : return operand;
            /* no error checking, default is operand */
    }
}
```

Infix to Postfix Conversion (Using Stack)

Tken	Stack	Top	Qtp
	[Q] [1] [2]		
a		-1	a
+	+	0	a
b	+	0	ab
*	+	1	ab
c	*	1	abc
es		-1	abc*=

*Figure 3.15: Translation of $a+b*c$ to postfix (p.124)

The orders of operands in infix and postfix are the same.

$$a + b * c, * > +$$

$$a *_1 (b + c) *_2 d$$

Token	Stack [0] [1] [2]			Top	Output
a				-1	a
$*_1$	$*_1$			0	a
($*_1$	(1	a
b	$*_1$	(1	ab
+	$*_1$	(+	2	ab
c	$*_1$	(+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	abc+ $*_1$
d	$*_2$			0	abc+ $*_1$ d
eos	$*_2$			0	abc+ $*_1$ d $*_2$

* Figure 3.16: Translation of $a*(b+c)*d$ to postfix (p.124)

Rules

- (1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
- (2) (has low in-stack precedence, and high incoming precedence.

	()	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp [ ] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp [ ] = {20, 19, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence

icp: incoming precedence

```

void postfix(void)
{
/* output the postfix of the expression. The expression
   string, the stack, and top are global */
char symbol;
precedence token;
int n = 0;
int top = 0; /* place eos on stack */
stack[0] = eos;

// Scan left to right
// If operand print.
else
// If rpar, unstack tokens and print until lpar;
   discard lpar.
   else remove and print symbols if isp>=icp else push()
// Unstack remaining symbols and print

```

```
void postfix(void)
{
/* output the postfix of the expression. The expression
string, the stack, and top are global */
char symbol;
precedence token;
int n = 0;
int top = 0; /* place eos on stack */
stack[0] = eos;
```

```
for (token = get _token(&symbol, &n); token != eos;
     token = get_token(&symbol, &n)) {
    if (token == operand)
        printf ("%c", symbol);
```

```
else if (token == rparen ){
    /*unstack tokens until left parenthesis */
    while (stack[top] != lparen)
        print_token(pop(&top));
    pop(&top); /*discard the left parenthesis */
}
```

```
else if (token == rparen ){
    /*unstack tokens until left parenthesis */
    while (stack[top] != lparen)
        print_token(pop(&top));
    pop(&top); /*discard the left parenthesis */
}
else{
    /* remove and print symbols whose isp is greater
       than or equal to the current token's icp */
    while(isp[stack[top]] >= icp[token] )
        print_token(pop(&top));

    push(&top, token);
}
}
```

```
while ((token = pop(&top)) != eos)
    print_token(token);
print("\n");
}
```

*Program 3.11: Function to convert from infix to postfix (p.126)

Infix to postfix conversion using stack

- Convert the following infix expression to postfix expression using a stack . Show the instances of stack during conversion.
- $a+b*c+d*e$
- $a*b+5$
- $((a/(b-c+d))*((e-a)*c$
- $a/b-c+d*e-a*c$

Stack using Dynamic Arrays

Stack using Dynamic Arrays

```
element *stack= (element *) malloc(sizeof(element));  
int capacity=1; //replace MAX_ITEM_SIZE
```

Void StackFull()

{

```
Stack = (element *) realloc(2*capacity, sizeof(element));  
capacity *=2;  
}
```

```
#include<stdio.h>
#include<stdlib.h>
int capacity=1;
typedef struct
{
    int *stack;
    int top;
}STACK;

int full(STACK *s)
{
    if(s->top == capacity-1)
    {
        s->stack=(int *) realloc(s->stack,sizeof(int)*capacity*2);
        capacity*=2;
    }
    return(0); }
```

```
int empty(STACK *s)
{
    if(s->top == -1)
    {
        return(1);
    }
    return(0);
}
```

```
void push(STACK *s, int x)
{
    s->top = s->top+1;
    s->stack[s->top]=x;
}
```

```
void print(STACK s)
{
    int i;
    for(i=s.top;i>=0;i--)
    {
        printf("%d\t",s.stack[i]);
    }
}
```

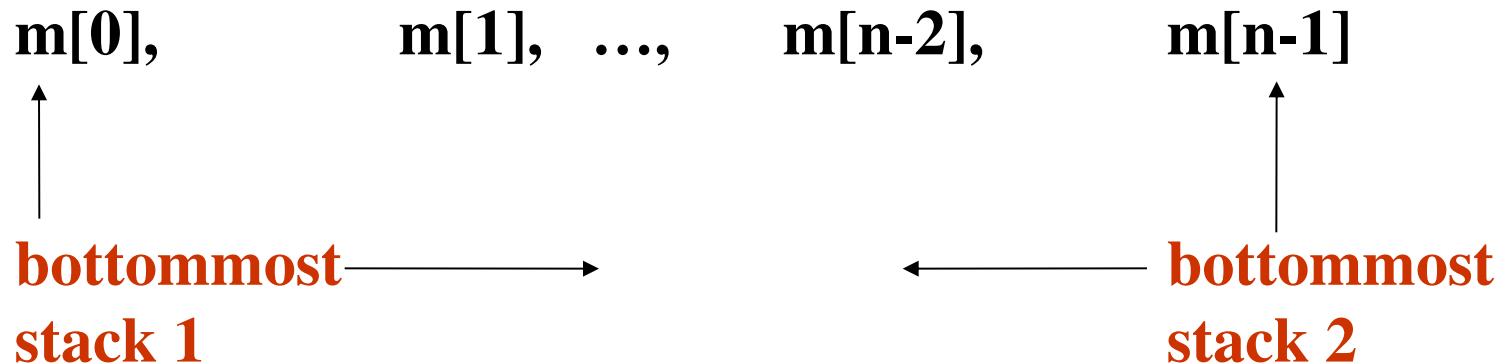
```
int pop(STACK *s)
{
    int x;
    x=s->stack[s->top];
    s->top=s->top-1;
    return(x);
}
```

```
void main()
{
STACK s;
int ch,x; s.top=-1;
s.stack=(int*) malloc(capacity*sizeof(int));
do{
printf("\n1.push\n2.pop\n3.dispaly\n4.exit\n");
printf("enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
    case 1:printf("enter value\n");
        scanf("%d",&x);
        if(!full(&s))
            push(&s,x);
        else
            printf("Stack is overflow\n");
        break;
    case 2:
        if(s.top<=0)
            printf("Stack Underflow\n");
        else
            x=s.stack[s.top];
            s.top--;
            printf("Popped element is %d\n",x);
        break;
    case 3:
        if(s.top<=0)
            printf("Stack Underflow\n");
        else
            printf("Top element is %d\n",s.stack[s.top]);
        break;
    case 4:
        exit(0);
    default:
        printf("Wrong choice\n");
}
}
```

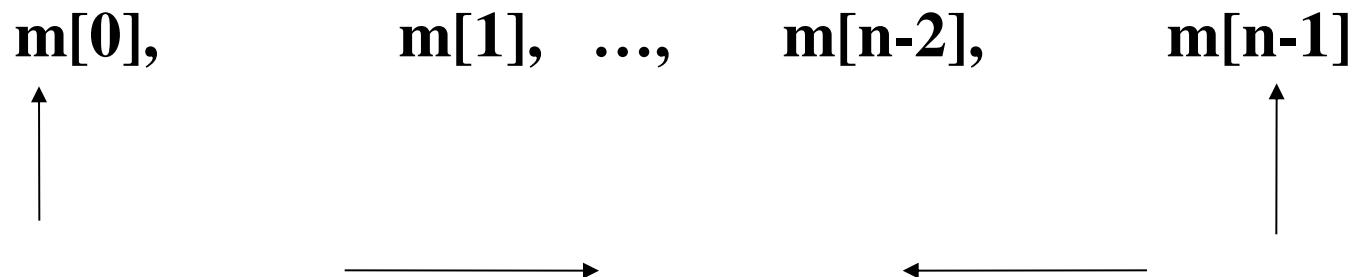
```
case 2:if(!empty(&s))
{
    x=pop(&s);
    printf("popped is %d\n",x);
    printf("\nRemaining elements:");
    print(s);
}
else
    printf('stack is empty\n');
    break;
case 3:if(empty(&s))
    printf("stack is empty\n");
else
    print(s);
    break;
}}while(ch!=4);
}
```

Multiple stacks and queues

Two stacks



Multiple stacks and queues



**More than two stacks (n)
memory is divided into n equal segments**

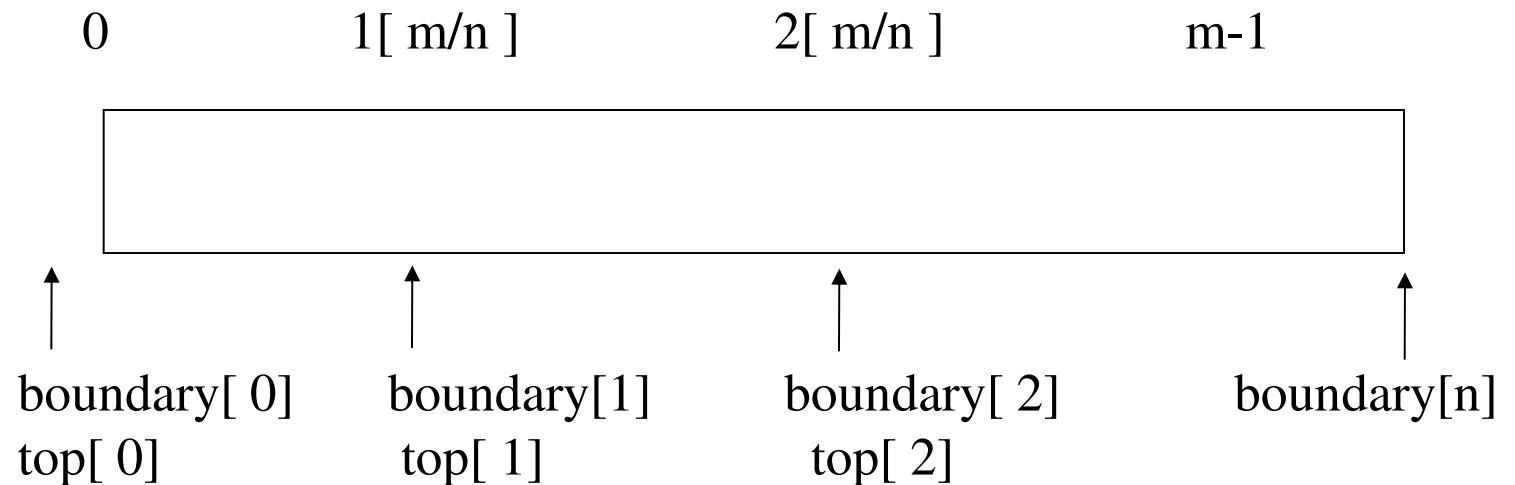
boundary[stack_no]

0 ≤ stack_no < MAX_STACKS

top[stack_no]

0 ≤ stack_no < MAX_STACKS

Initially, $\text{boundary}[i] = \text{top}[i]$.



All stacks are empty and divided into roughly equal segments.

***Figure 3.18: Initial configuration for n stacks in memory [m]. (p.129)**

```
#define MEMORY_SIZE 100 /* size of memory */  
#define MAX_STACKS 10  
    /* max number of stacks plus 1 */
```

```
/* global memory declaration */  
element memory[MEMORY_SIZE];
```

```
int top[MAX_STACKS];  
int boundary[MAX_STACKS];
```

```
int n; /* number of stacks entered by the user */
```

```
top[0] = boundary[0] = -1;  
for (i = 1; i < n; i++)  
    top[i] =boundary[i] =(MEMORY_SIZE/n)*i;
```

boundary[n] = MEMORY_SIZE-1;
***(p.129)**

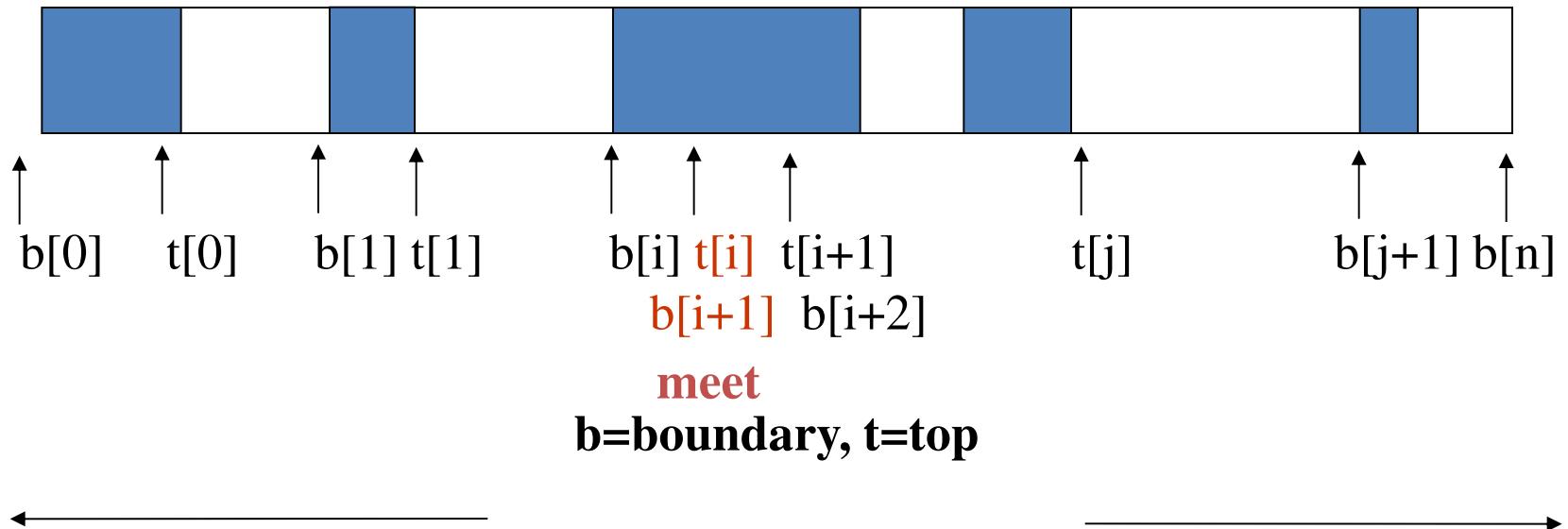
```
void push(int i, element item)
{
    /* add an item to the ith stack */
    if (top[i] == boundary [i+1])
        stack_full(i);    may have unused storage
    memory[++top[i]] = item;
}
```

*Program 3.12: Add an item to the stack *stack-no* (p.129)

```
element pop(int i)
{
    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
        return stack_empty(i);
    return memory[top[i]--;
}
```

*Program 3.13:Delete an *item* from the stack *stack-no* (p.130)

Find j , $i < j < n$
such that $\text{top}[j] < \text{boundary}[j+1]$
or, $0 \leq j < i$



*Figure 3.19: Configuration when stack i meets stack $i+1$,

but the memory is not full (p.130)

Infix	Prefix
a^*b/c	\wedge^*abc
$ab-c+d^*e-a^*c$	$-+/\underline{abc}^*\underline{d}^*\underline{ac}$
$a^*(b+c)d-g$	$\wedge^*a+\underline{bcdg}$

- (1) evaluation
 (2) transformation

*Figure 3.17: Infix and postfix expressions (p.127)

Infix to Prefix

e.g., Infix: A*B+C/D

Infix: (A-B/C)*(A/K-L)

Infix to Prefix

e.g., Infix: A*B+C/D

Prefix: +*AB/CD

Infix: (A-B/C)*(A/K-L)

Prefix: *-A/BC-/AKL

Infix to Prefix

e.g., Infix: $A * B + C / D$

Prefix: $+ * A B / C D$

Infix: $(A - B / C) * (A / K - L)$

Prefix: $* - A / B C - / A K L$

Step 1: Reverse the infix expression i.e $A + B * C$ will become $C * B + A$.

Note while reversing each '(' will become ')' and each ')' becomes '('.

Step 2: Obtain the postfix expression of the modified expression

i.e $CB * A +$.

Step 3: Reverse the postfix expression.

Hence in our example prefix is $+ A * B C$

Infix(A) to Prefix(B)

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6

for each element of A until the STACK is empty

Infix to Prefix

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6

for each element of A until the STACK is empty

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Infix to Prefix

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6
for each element of A until the STACK is empty

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered then:

- a. Repeatedly pop from STACK and add to B each operator
(on the top of STACK) which has same
or higher precedence than the operator.
- b. Add operator to STACK

Infix to Prefix

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6

for each element of A until the STACK is empty

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered then:

- a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has **higher precedence** than the operator.
- b. Add operator to STACK

Step 6. If left parenthesis is encountered then

- a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a right parenthesis is encountered)
- b. Remove the right parenthesis

Step 7. Reverse A

Infix to Prefix

e.g., Infix: A*B+C/D

Prefix: +*AB/CD

Infix: (A-B/C)*(A/K-L)

Prefix: *-A/BC-/AKL

Infix	Prefix
a^*b/c	\wedge^*abc
$ab-c+d^*e-a^*c$	$-\wedge abc^*d^*e^*ac$
$a^*(b+c)d-g$	$\wedge^*a+\wedge bcdg$

- (1) evaluation
(2) transformation

*Figure 3.17: Infix and postfix expressions (p.127)

Prefix to Postfix

Read the Prefix expression in reverse order
(from right to left)

If the symbol is an operand,
then push it onto the Stack

Prefix to Postfix

Read the Prefix expression in reverse order
(from right to left)

If the symbol is an operand,
then push it onto the Stack

If the symbol is an operator,
then pop two operands from the Stack

Create a string by concatenating the two operands
and the operator after them.

string = operand1 + operand2 + operator

And push the resultant string back to Stack

Prefix to Postfix

Read the Prefix expression in reverse order
(from right to left)

If the symbol is an operand,
then push it onto the Stack

If the symbol is an operator,
then pop two operands from the Stack

Create a string by concatenating the two operands
and the operator after them.

string = operand1 + operand2 + operator

And push the resultant string back to Stack

Repeat the above steps
until end of Prefix expression.

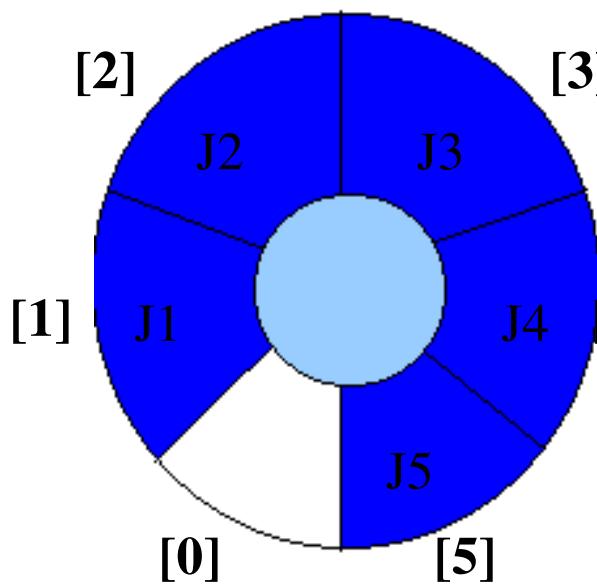
Evaluation of Prefix expression

Hint: Scan the expression from right to left

e.g., $+^*abc$

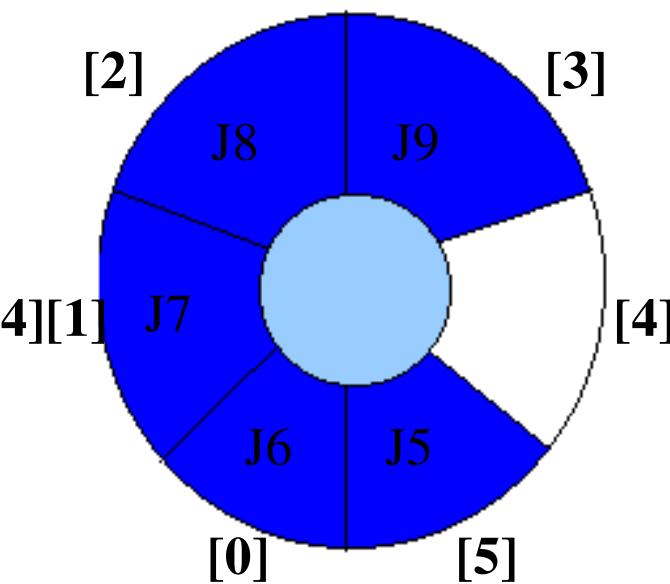
Circular Queue using Dynamically Allocated Arrays

FULL QUEUE



front = 0
rear = 5

FULL QUEUE



front = 4
rear = 3

Circular Queue using Dynamically Allocated Arrays

```
void addq(element item)
```

```
{
```

```
.....
```

```
queueFull();
```

```
.....
```

```
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{
    /*allocate an array with twice the capacity*/
    /* Copy from queue to newQueue*/
    /* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
{
    /*allocate an array with twice the capacity*/
    Element * newQueue;
    newQueue = (newQueue*) (2*capacity * sizeof(*queue));
    /* Copy from queue to newQueue*/
    /* Switch to newQueue
}
```

Circular Queue using Dynamically Allocated Arrays

```
void queueFull()
```

```
{ /*allocate an array with twice the capacity*/
```

```
/* Copy from queue to newQueue*/
```

```
Start = (front+1) % capacity;
```

```
If (Start < 2) /* No wrap around */
```

```
{...}
```

```
Else /* wrap around */
```

```
{...}
```

```
/* Switch to newQueue
```

```
}
```

Circular Queue using Dynamically Allocated Arrays

void queueFull()

```
{ /*allocate an array with twice the capacity*/
```

```
/* Copy from queue to newQueue*/
```

```
Start = (front+1) % capacity;
```

```
If (Start < 2) /* No wrap around */
```

```
copy(queue+start, queue+capacity-1, newQueue);
```

```
Else /* wrap around */
```

```
{copy(queue+start, queue+capacity, newQueue);
```

```
copy(queue, queue+rear-1, newQueue);
```

```
}
```

```
/* Switch to newQueue
```

Circular Queue using Dynamically Allocated Arrays

void queueFull()

```
{ /*allocate an array with twice the capacity*/  
/* Copy from queue to newQueue*/  
/* Switch to newQueue  
front = 2*capacity-1;  
rear = capacity -2;  
capacity *= 2;  
free(queue);  
queue = newQueue;  
}
```

Recursion

Recursion

- Recursion is the name given for expressing anything in terms of itself.
- Recursive function is a function which calls itself until a particular condition is met.

The factorial function

- Given a positive integer n, factorial is defined as the product of all integers between n and 1.
i.e factorial of 4 is $4*3*2*1=24$
- Hence we have the formula

$$\begin{aligned}n! &= 1 && \text{if } n==0 \\n! &= n*(n-1)*(n-2) \dots *1 && \text{if}(n>0)\end{aligned}$$

- $n!=n*(n-1)!$
 $n!=n*(n-1)*(n-2)!$
 $= n*(n-1)*(n-2)*\dots*0! = n*(n-1)*(n-2)*\dots*1$
- Hence this can be achieved by having a function which calls itself until 0 is reached. This is recursive function for factorial.

Ex:

$$5! = 5 * 4! \rightarrow 120$$

$$4 * 3! \rightarrow 24$$

$$3 * 2! \rightarrow 6$$

$$2 * 1! \rightarrow 2$$

$$1 * 0! \rightarrow 1$$

1

Fibonacci sequence

0,1,1,2,3,5,8,.....

- Each element is the sum of two preceding elements.
- Fibonacci of a number is nothing but the value at that position in sequence.

i.e $\text{fib}(0) == 0$

$\text{fib}(1) == 1$

$\text{fib}(2) == 1$

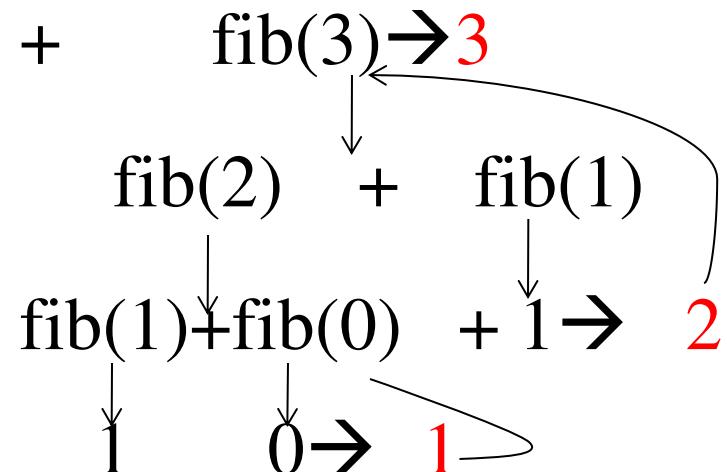
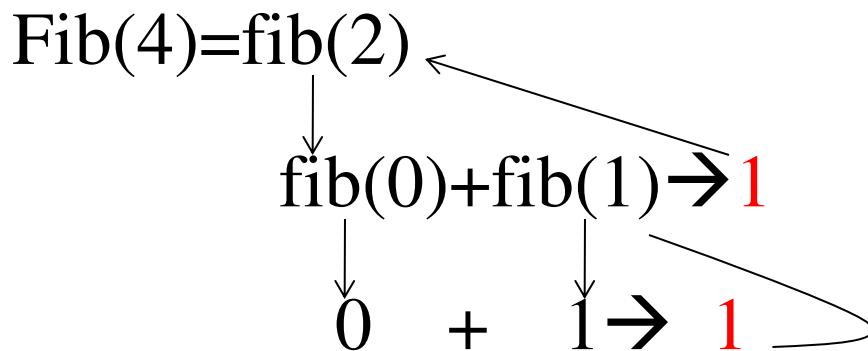
$\text{fib}(3) == 2$ and so on

- Fibonacci is defined in formula as

$\text{fib}(n) = n$

$\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$

if $n == 0$ or $n == 1$
for $n >= 2$



Binary search

- Binary search is an efficient method of search.
 1. element is compared with the middle element in the array. If the middle element is the element to be searched, search is successful.
 2. if element is less than the middle element, then searching is restricted to the first half.
 3. if element is greater than the middle element, then searching is restricted to the second half.
 4. this process is continued until the element is found or not found.

1	2	3	4	<u>5</u>	6	7	8	9
0	1	2	3	4	5	6	7	8

Let the element to be searched is **2**

Middle element is 5 and 2 is less than 5. hence first half is considered, which is

1	2	3	4
0	1	2	3

middle element is 2 and hence search is successful and element is found at position 1

1	2	3	4	<u>5</u>	6	7	8	9
0	1	2	3	4	5	6	7	8

Let the element to be searched is **10**

6	7	8	9
5	6	7	8
8	9		
7	8		

10 is not found.

Properties of recursive algorithms

- Recursive algorithm should terminate at some point, otherwise recursion will never end.
- Hence recursive algorithm should have stopping condition to terminate along with recursive calls.
for ex: in factorial stopping condition is $n!=1$ if $n==0$
In multiplication of 2 numbers, it is $a*b=a$ if $b==1$
In Fibonacci it is $\text{fib}(0)=0$ and $\text{fib}(1)=1$
` In binary search it is $\text{low} > \text{high}$

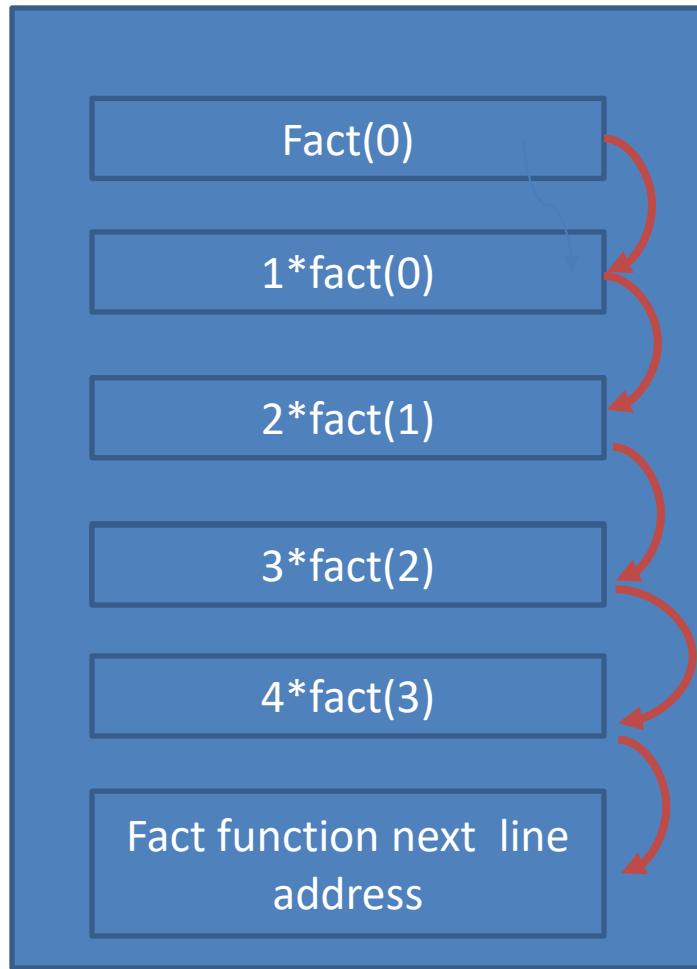
Factorial in C

```
int fact(int n)
{
    int x, y, res;
    if(n==0)
        return 1;
    else
    {
        return n*fact(n-1);

/*
    x=n-1;
    y=fact(x);
    res=n*y;
    return res; */

    }
}
```

Here $y=fact(x)$, function gets called by itself each time with 1 less number than previous one until number gets zero.



Control flow in evaluating fact(4)

4			

n x y res

4	3		
4			

n x y res

3	2		
4	3		
4			

n x y res

2	1		
3	2		
4	3		
4			

n x y res

1	0		
2	1		
3	2		
4	3		
4			

n x y res

1	0	1	1
2	1	1	2
3	2	2	6
4	3	6	24

n x y res

Recursive program to find the nth fibonacci number

```
int fib(int n)
```

```
{
```

```
    if(n==0)
```

```
        return 0;
```

```
    if(n==1)
```

```
        return 1;
```

```
    else
```

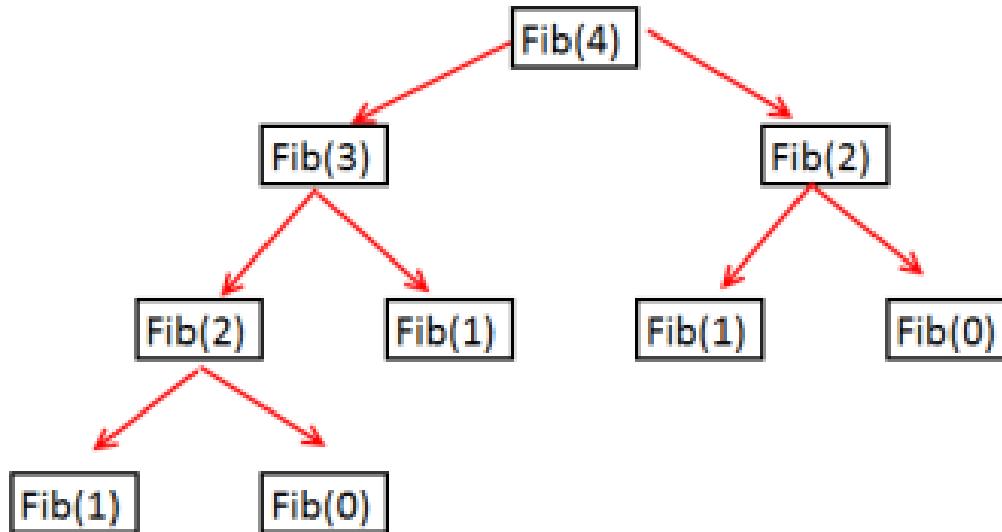
```
{
```

```
        return (fib(n-1) + fib(n-2));
```

```
}
```

```
}
```

Fibonacci Series



$$\text{Fibonacci}(N) = 0$$

$$= 0$$

$$= \text{Fibonacci}(N-1) + \text{Finacchi}(N-2)$$

for n=0

for n=1

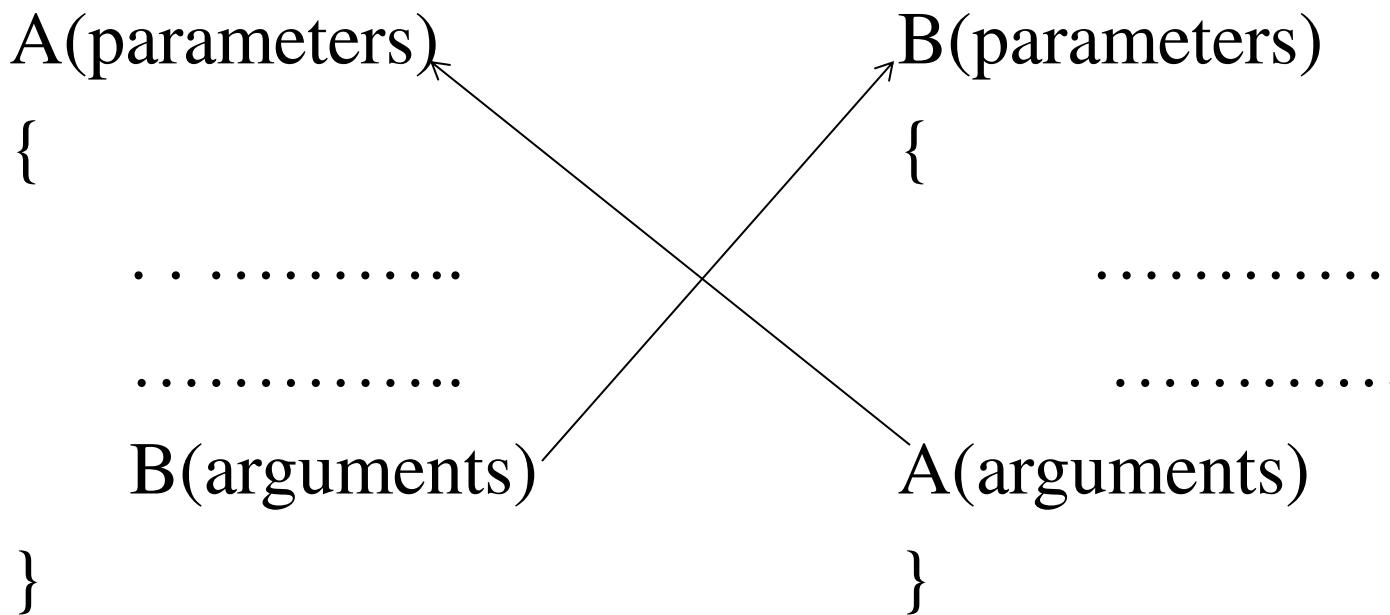
for n>1

Recursive program to do a binary search

```
int binary(int item, int a[], int low, int high)
{
    int mid;
    if(low > high)
        return -1;
    mid=(low+high)/2;
    if(item==a[mid])
        return mid;
    else if(item<a[mid])
    {
        high=mid-1;
        return binary(item, a, low, high);
    }
    else
    {
        low=mid+1;
        return binary(item, a, low, high);
    }
}
```

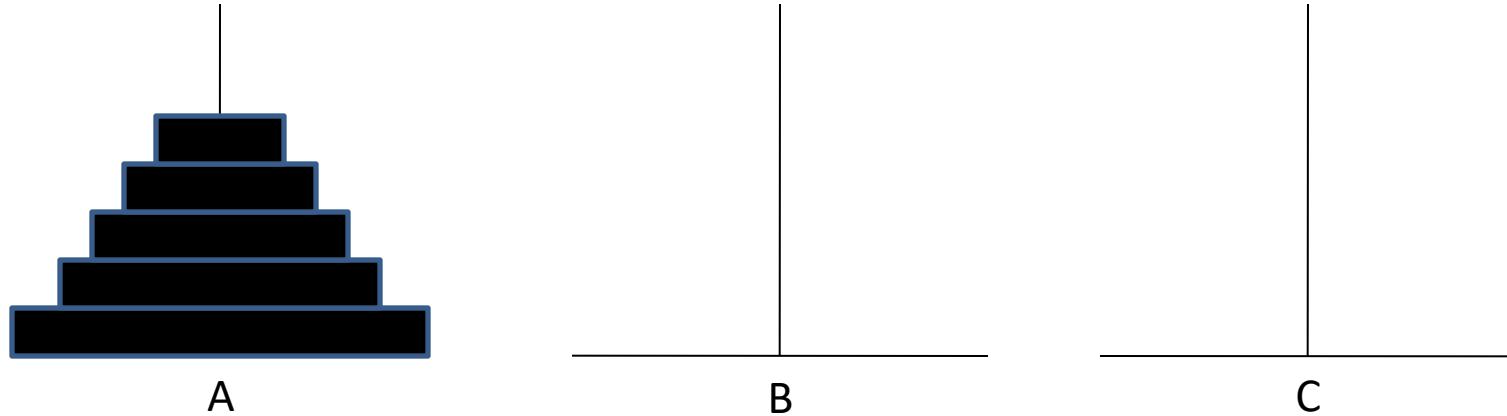
Recursive chains

- Recursive function need not call itself directly. It can call itself indirectly as shown



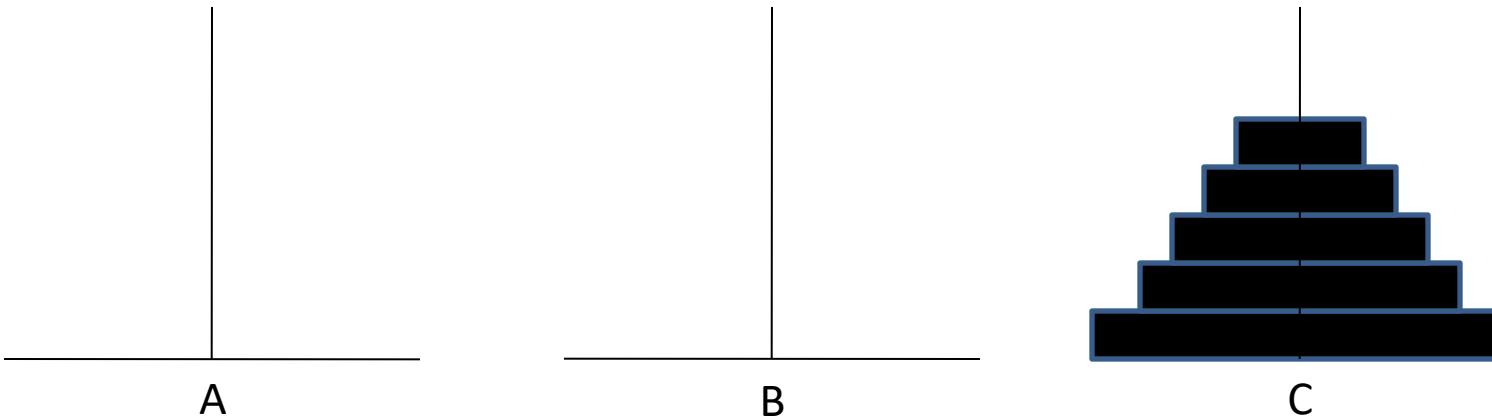
Towers of Hanoi problem

Initial setup



- There are 3 pegs A,B, and C and five disks of different diameters placed on peg A so that a larger disk is always below a smaller disk.
- The aim is to move five disks to peg C using peg B as auxiliary. Only the top disk on any peg may be moved to another peg, and a larger disk may never rest on a smaller one.

After passing all the 5 disks to peg C:

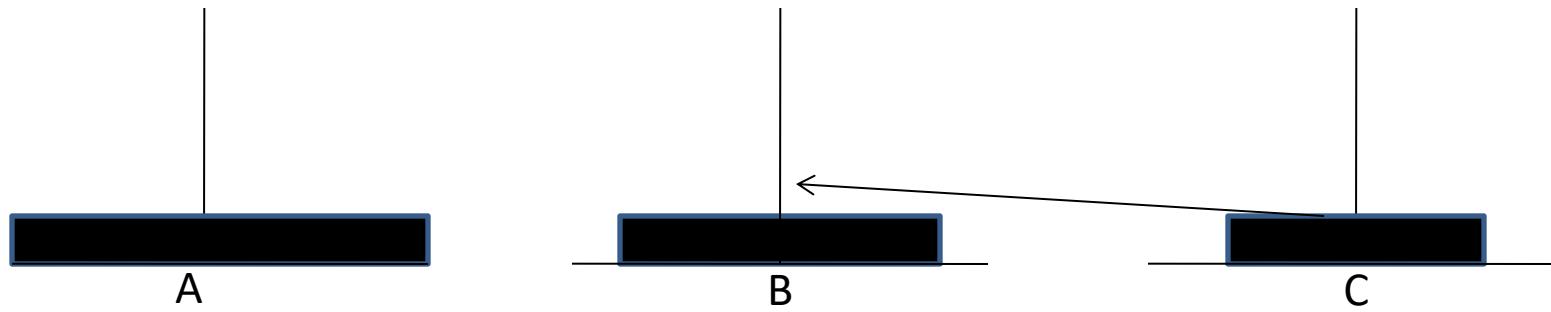
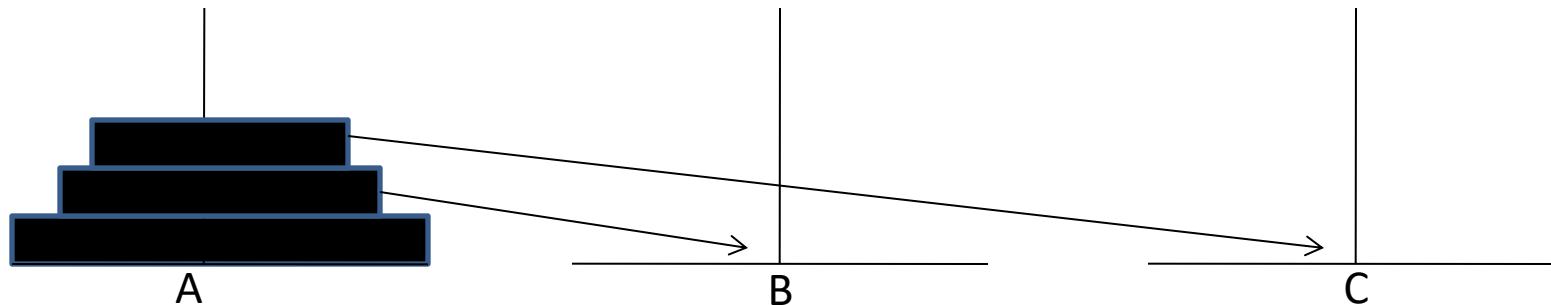


Lets consider the general case of n disks

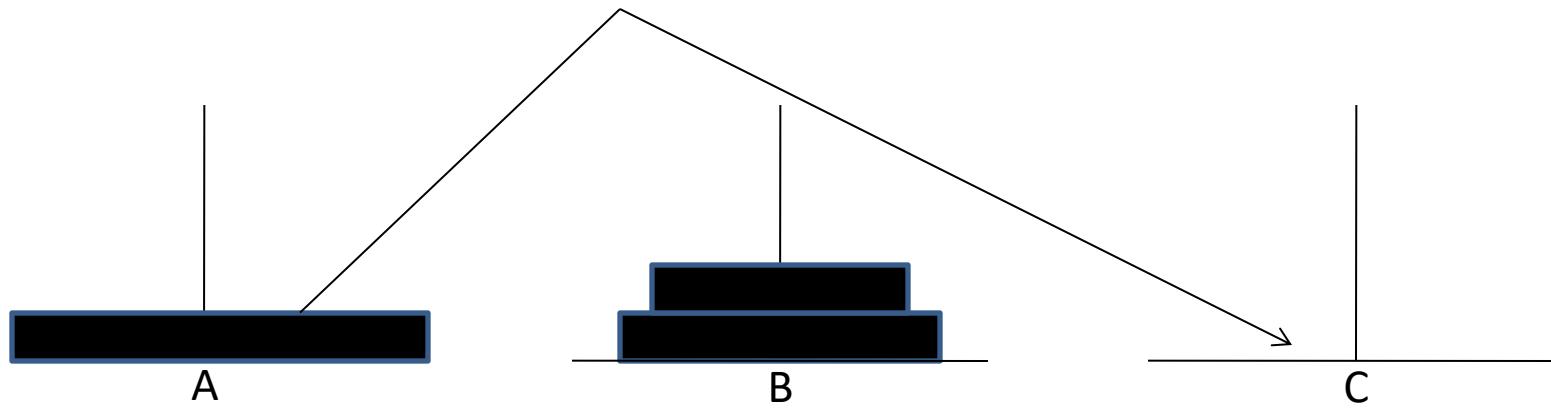
To move n disks from A to C using B as auxiliary

- 1.If $n==1$, move single disk from A to C.
 - 2.Move the top $n-1$ disks from A to B using C as auxiliary.
 - 3.Move the remaining disk from A to C.
 - 4.Move the $n-1$ disks from B to C, using A as auxiliary.
- Here if $n==1$, step1 will produce a correct solution.
 - If $n==2$, we know that we already have a solution for $n-1$. i.e. 1. so steps 2 and 4 can be performed.
 - If $n==3$, we know that we have a solution for $n-1$. i.e. 2. so steps 2 and 4 can be performed.
 - In this way we have solutions for 1,2,3.....up to any value.
 - This clearly indicates the concept of recursion involved and hence this problem can be solved by recursion.

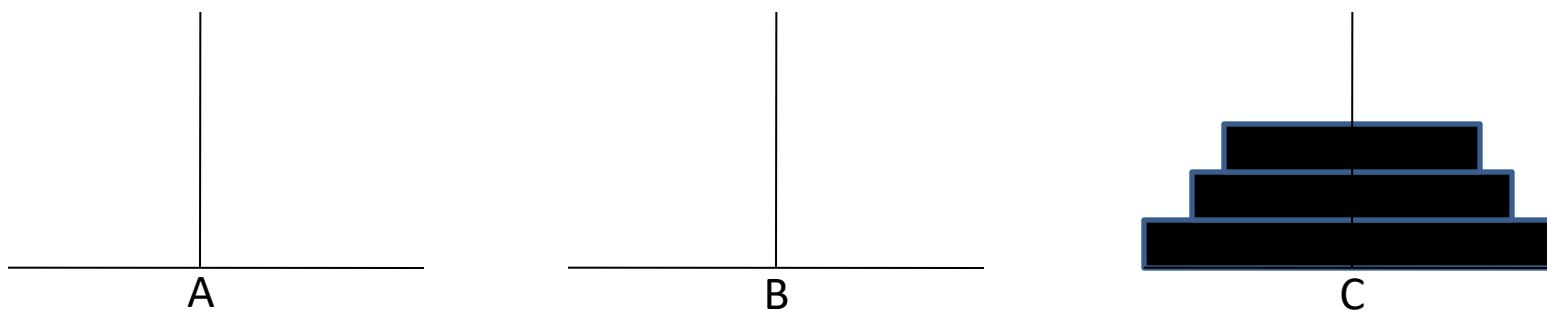
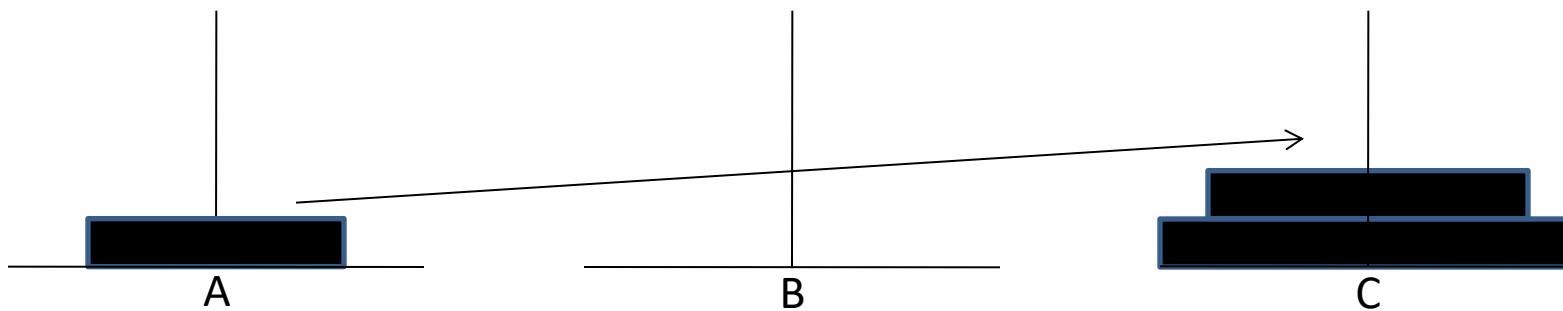
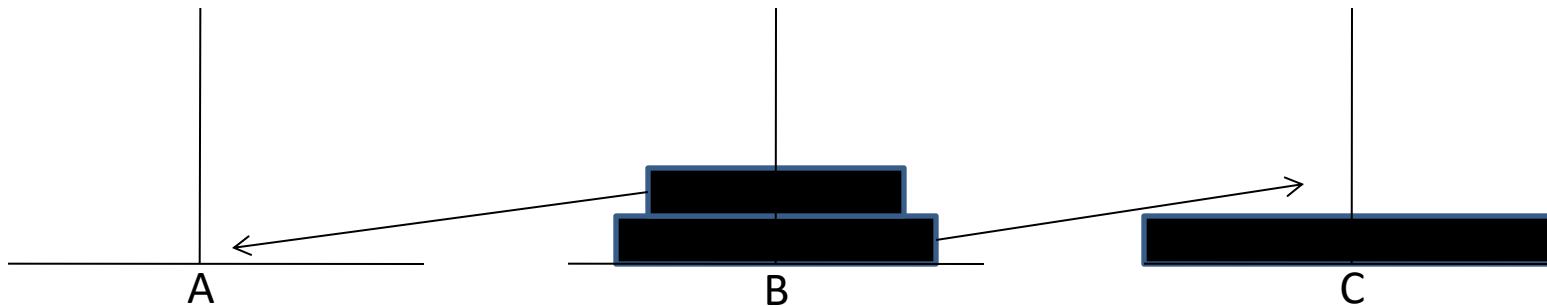
n==3. moving n-1 disks from A to B using C as auxiliary



Moving remaining 1 disk from A to C



Moving n-1 disks from B to C using A as auxiliary



C program for tower of hanoi problem

```
void tower (int n, char source, char temp, char destination) {  
    if(n==1) {  
        printf("move disk 1 from %c to %c \n",source,destination);  
        return;  
    }  
  
    /*moving n-1 disks from A to B using C as auxiliary*/  
    tower(n-1, source, destination, temp);  
  
    printf("move disk %d from %c to %c \n",n,source,destination);  
  
    /*moving n-1 disks from B to C using A as auxiliary*/  
    tower(n-1, temp, source, destination);  
}
```

Length of a string using recursion

```
int StrLen(str[], int index)
```

```
{
```

```
    if (str[index] == '\0') return 0;
```

```
    return (1 + StrLen(str, index + 1));
```

```
}
```

654321

abcdef\0

Length of a string using recursion using static variable

```
int StrLen(char *str)
{
    static int length=0;
    if(*str != '\0')
    {
        length++;
        StrLen(++str);
    }
    return length;
}
```

To Check whether a given String is Palindrome or not using Recursion

```
int isPalindrome(char *inputString, int leftIndex, int rightIndex) {  
    /* Recursion termination condition */  
    if(leftIndex >= rightIndex) return 1;  
    if(inputString[leftIndex] == inputString[rightIndex]) {  
        return isPalindrome(inputString, leftIndex + 1,  
                           rightIndex - 1);  
    }  
    return 0;  
}
```

```
int main(){
    char inputString[100];
    printf("Enter a string for palindrome check\n");
    scanf("%s", inputString);
    if(isPalindrome(inputString, 0, strlen(inputString) - 1))
        printf("%s is a Palindrome \n", inputString);
    else
        printf("%s is not a Palindrome \n", inputString);
    getch();
    return 0;
}
```

To Copy One String to another using Recursion

```
void copy(char str1[], char str2[], int index)
{
    str2[index] = str1[index];
    if (str1[index] == '\0') return;
    copy(str1, str2, index + 1);
}
```

```
void multiply (int m1, int n1, int
              a[10][10], int m2, int n2,
              int b[10][10], int c[10][10])
{
    static int i = 0, j = 0, k = 0;

    if (i >= m1)
    {
        return;
    }
    else if (i < m1)
    {
        if (j < n2)
        {
            if (k < n1)
            {
                c[i][j] += a[i][k] * b[k][j];
                k++;
                multiply(m1, n1, a, m2, n2, b, c);
            }
            k = 0;
            j++;
            multiply(m1, n1, a, m2, n2, b, c);
        }
        j = 0;
        i++;
        multiply(m1, n1, a, m2, n2, b, c);
    }
}
```

Advantages of Recursion

1. Clearer and simpler versions of algorithms can be created using recursion.
2. Recursive definition of a problem can be easily translated into a recursive function.
3. Lot of bookkeeping activities such as initialization etc required in iterative solution is avoided.

Disadvantages

1. When a function is called, the function saves formal parameters, local variables and return address and hence consumes a lot of memory.
2. Lot of time is spent in pushing and popping and hence consumes more time to compute result.

Iteration

- Uses loops
- Counter controlled and body of loop terminates when the termination condition fails.
- Execution is faster and takes less space.
- Difficult to design for some problems.

Recursion

uses if-else and repetitive function calls

Terminates when base condition is reached.

Consumes time and space because of push and pop.

Best suited for some problems and easy to design.

LISTS

Array

successive items located at fixed distance apart

Disadvantage

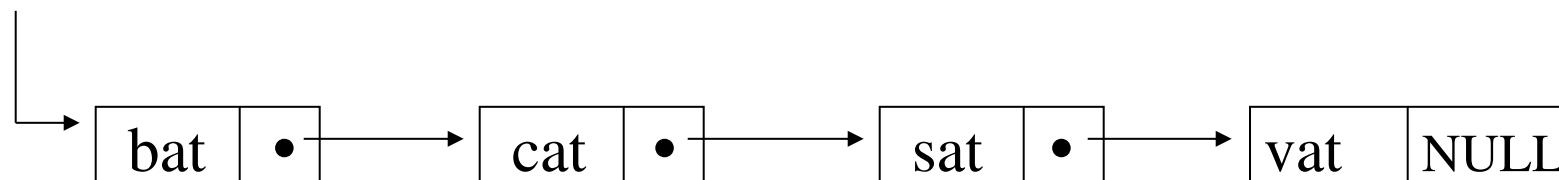
data movements during insertion and deletion

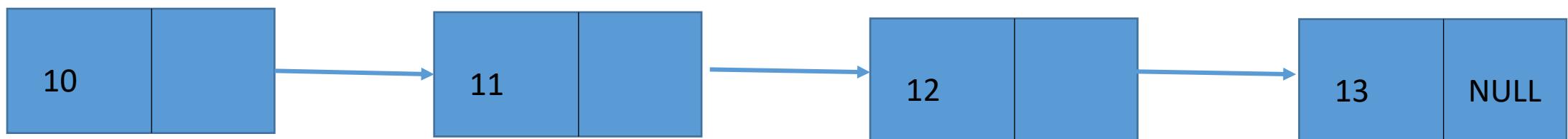
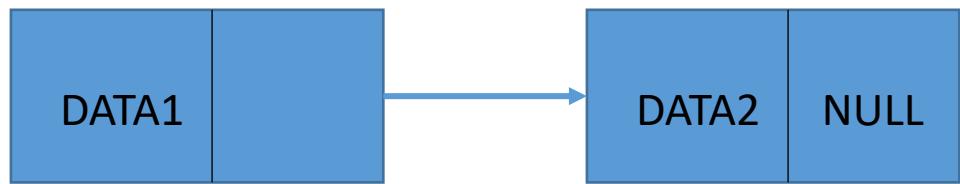
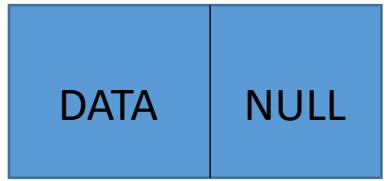
waste space in storing n ordered lists of varying size

possible solution

linked list

A linked list is a sequence of data structures, which are connected together via links.





Types of Linked List

- 1. Simple Linked List** – Item navigation is forward only.
- 2. Doubly Linked List** – Items can be navigated forward and backward.
- 3. Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Operations

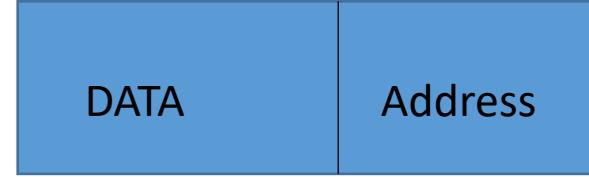
Insertion – Adds an element to the list.

Deletion – Deletes an element from the list.

Display – Displays the complete list.

Search – Searches an element using the given key.

```
typedef struct node
{
    int data;
    struct node *next;
}NODE;
```



```
NODE *n=(NODE*)malloc(sizeof(struct node));
```

```
typedef struct node
{
    int data;
    struct node *next;
}*NODE;
```

```
NODE n=(NODE)malloc(sizeof(struct node));
```



```
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
    int data;
    struct node *next;
}*NODE;
void display(NODE first)
{
    if(first==NULL)
    {
        printf("List is empty\n");
        return;
    }
    while(first!=NULL)
    {
        printf("%d\t",first->data);
        first=first->next;
    }
}
```

```
NODE insert_front(NODE first, int data)
{
    NODE n1;
    n1=(NODE)malloc(sizeof(struct node));
    if(n1==NULL){ printf("Out of memory\n");exit(0);}
    n1->data=data;
    n1->next=NULL;
    if(first==NULL)
        return n1;
    n1->next=first;
    return n1;
}
```

```
NODE insert_Rear(NODE first,int element)
{
NODE n1,f;
n1=(NODE)malloc(sizeof(struct node));
if(n1==NULL){ printf("Out of memory\n");
exit(0);}
n1->data=element;
n1->next=NULL;
if(first==NULL)
return n1;
f=first;
while(first->next!=NULL)
first=first->next;
first->next=n1;
return f;
}
```

```
NODE delete_front(NODE first)
{
NODE temp;
if(first==NULL)
{
    printf("List is empty\n");return NULL;
}
temp=first->next;
printf("The element deleted is %d\n",first->data);
free(first);
return temp;
}
```

```
NODE delete_rear(NODE first)
{
NODE temp,prev=NULL;
if(first==NULL){ printf("List is empty\n");return NULL;}
temp=first;
while(temp->next!=NULL)
{
    prev=temp;
    temp=temp->next;
}
if(prev==NULL)
{ printf("The element deleted is%d\n",first->data);
free(first);
return NULL;
}
printf("The element deleted is %d\n",temp->data);
prev->next=NULL;
free(temp); return first; }
```

```
int main()
{
NODE first=NULL;
int choice, element;
do{
printf("\n1.Insert_Front\n2.Insert_Rear\n
3.Delete_Front\n4.Delete_Rear\n5.display\n6.exit\n ");
printf("Enter the option\n");
scanf("%d",&choice);
switch(choice){
case 1:printf("Enter the element to be inserted\n");
scanf(" %d",&element);
first=insert_front(first,element);
break;
case 2: printf("Enter the element to be inserted\n");
scanf("%d",&element);
first=insert_Rear(first,element);
break;
case 3:first=delete_front(first);break;
case 4:first=delete_rear(first);break;
case 5:printf("The elements of the
list are\n");
display(first);
break;} } while(choice<=5);}
```

Without return type

```
typedef struct list_node *list_pointer;
struct list_node
{
    int data;
    list_pointer link;
};

list_pointer first=NULL;

insert_Front(&first,element);

delete_front(&first);

case 1: printf("Enter the element to be inserted\n");
          scanf("%d",&element);
          insert_Front(&first,element);
          break;
```

```
void insert_Front(list_pointer *first, int data)
{
list_pointer n1,temp,last;
n1=(list_pointer)malloc(sizeof(struct list_node));
n1->data=data;
n1->link=NULL;
if(*first==NULL) *first=n1;
else
{n1->link=*first;
*first=n1;}
}

void insert_Rear(list_pointer *first, int data)
{
list_pointer n1,temp,last;
n1=(list_pointer)malloc(sizeof(struct list_node));
n1->data=data;
n1->link=NULL;
if(*first==NULL) *first=n1;
else
{last=*first;
while((last)->link!=NULL)
last=(last)->link;
(last)->link=n1;}}
```

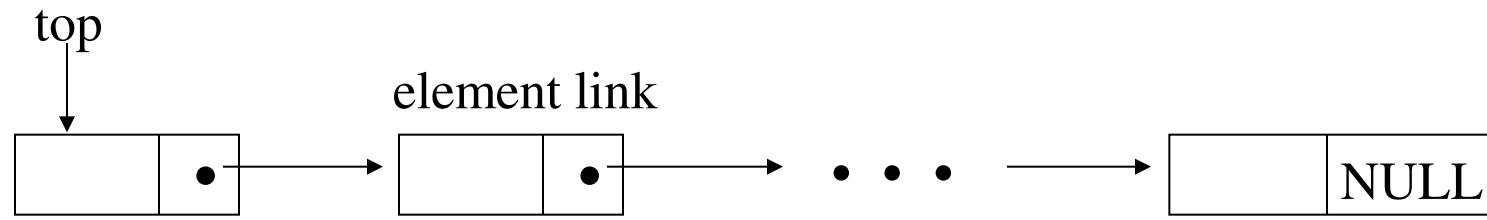
```
void delete_front(list_pointer * first)
{ list_pointer temp;
  if(*first==NULL)
  {
    printf("List is empty\n");
    return ;
    temp=(*first);
    *first=(*first)->link;
    printf("The element deleted is %d\t",temp->data);
    free(temp);
  }
}
```

```
void delete_rear(list_pointer * first)
{ list_pointer temp,prev;
  if(*first==NULL){printf("List is empty\n");return ;}
  if((*first)->link==NULL)
  {
    printf("The element deleted is %d\t",(*first)->data);
    *first=NULL;
    return;  }
  temp=(*first);
  while(temp->link!=NULL)
  {
    prev=temp;
    temp=temp->link;
  }
  printf("The element deleted is %d\t",temp->data);
  prev->link=NULL;
  free(temp);
}
```

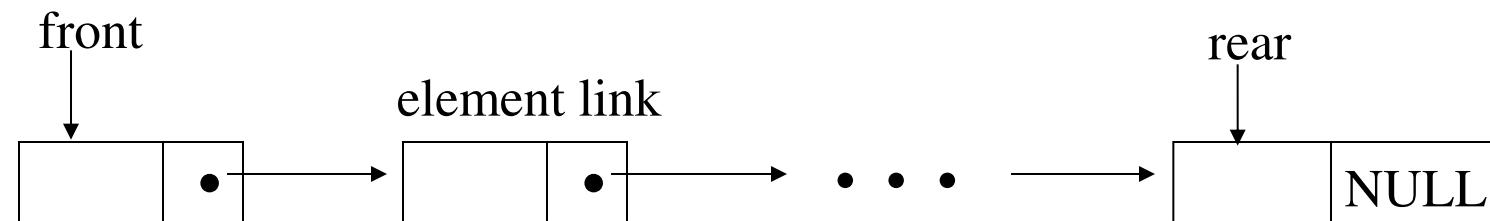
```
void display(list_pointer first)
{
    if(first==NULL)
    {
        printf("List is empty\n");
        return;
    }
    while(first!=NULL)
    {
        printf("%d\t",first->data);
        first=first->link;
    }
}
```

```
int main()
{
list_pointer first=NULL;
int choice, element;
do{
printf("\n1.Insert_Front\n2.Insert_Rear\n3.Display\n4.delete_Front\n5
.delete_Rear\n6.exit\n ");
printf("Enter the option\n");
scanf("%d",&choice);
switch(choice) {
case 1:printf("Enter the element to be inserted\n");
scanf("%d",&element);
insert_Front(&first,element);
break;
case 2:printf("Enter the element to be inserted\n");
scanf("%d",&element);
insert_Rear(&first,element);
break;
case 3:printf("The elements of the list are\n");
display(first);
break;
case 4:delete_front(&first);
break;
case 5:delete_rear(&first);
break;} } while(choice<6);}
```

4.3 DYNAMICALLY LINKED STACKS AND QUEUES



(a) Linked Stack



(b) Linked queue

***Figure 4.10:** Linked Stack and queue (p.147)

```
void push(list_pointer *top, int data)
{
list_pointer n1,temp,last;
n1=(list_pointer)malloc(sizeof(struct
list_node));
n1->data=data;
n1->link=NULL;
if(*top==NULL) *top=n1;
else
{n1->link=*top;
*top=n1;
}
}
```

```
void pop(list_pointer * top)
{ list_pointer temp;
  if(*top==NULL){printf("List is
empty\n");return ;}
  if((*top)->link==NULL)
  {
    printf("The element deleted is
%d\t",(*top)->data);
    *top=NULL;
    return;
  }
  temp=(*top);
  *top=(*top)->link;
  printf("The element popped is %d\t",temp->data);
  free(temp);
}
```

```
void display(list_pointer top)
{
if(top==NULL)
{
printf("List is empty\n");
return;
}
while(top!=NULL)
{
    printf("%d\t",top->data);
    top=top->link;
}
}
```

```
int main()
{
list_pointer top=NULL;
int choice, element;
do{
printf("\n1.Push\n2.Display\n3.Pop\n4.exit\n ");
printf("Enter the option\n");
scanf("%d",&choice);
switch(choice)
{
case 1:printf("Enter the element to be pushed\n");
    scanf("%d",&element);
    push(&top,element);
    break;
case 2:printf("The elements of stack are\n");
    display(top);
    break;
case 3:pop(&top);
}
} while(choice<4);
}
```

Queue

```
void addq(list_pointer *front,list_pointer *rear, int data)
{
list_pointer n1,last;
n1=(list_pointer)malloc(sizeof(struct list_node));
n1->data=data;
n1->link=NULL;
if(*front) (*rear)->link=n1;
else
*front=n1;
*rear=n1;
}
void deleteq(list_pointer *front)
{ list_pointer temp;
if(*front==NULL){printf("List is empty\n");return ;}
temp=(*front);
*front=(*front)->link;
printf("The element deleted is %d\t",temp->data);
free(temp);
}
```

```
void display(list_pointer front)
{
if(front==NULL)
{
printf("List is empty\n");
return;
}
while(front!=NULL)
{
printf("%d\t",front->data);
front=front->link;
}
}
```

```
int main()
{
    list_pointer front=NULL, rear=NULL;
    int choice, element;
    do
    {
        printf("\n1.Insert\n2.Display\n3.delete\n4.exit\n ");
        printf("Enter the option\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:printf("Enter the element to be inserted\n");
                      scanf("%d",&element);
                      addq(&front,&rear,element);
                      break;
            case 2:printf("The elements of the queue are\n");
                      display(front);
                      break;
            case 3:deleteq(&front);
                      break;
        }
    } while(choice<4);
```

Polynomials

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \dots + a_0x^{e_0}$$

Representation

```
typedef struct poly_node *poly_pointer;
```

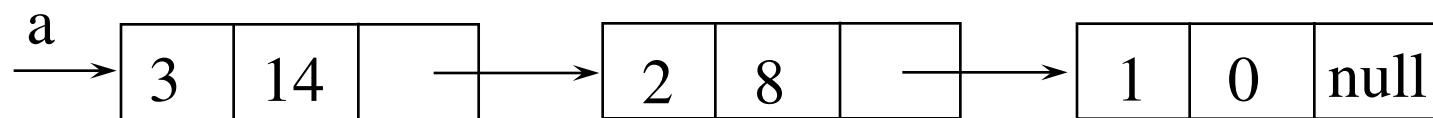
```
typedef struct poly_node {  
    int coef;  
    int expon;  
    poly_pointer link;  
};
```

```
poly_pointer a, b, c;
```

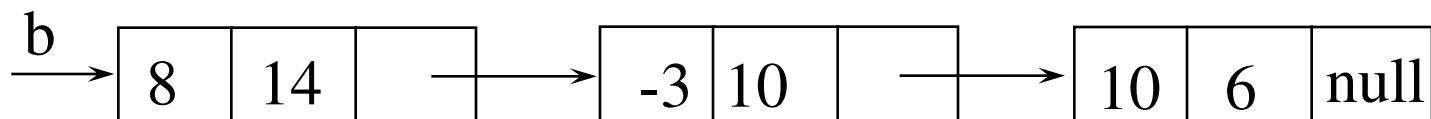
coef	expon	link
------	-------	------

Examples

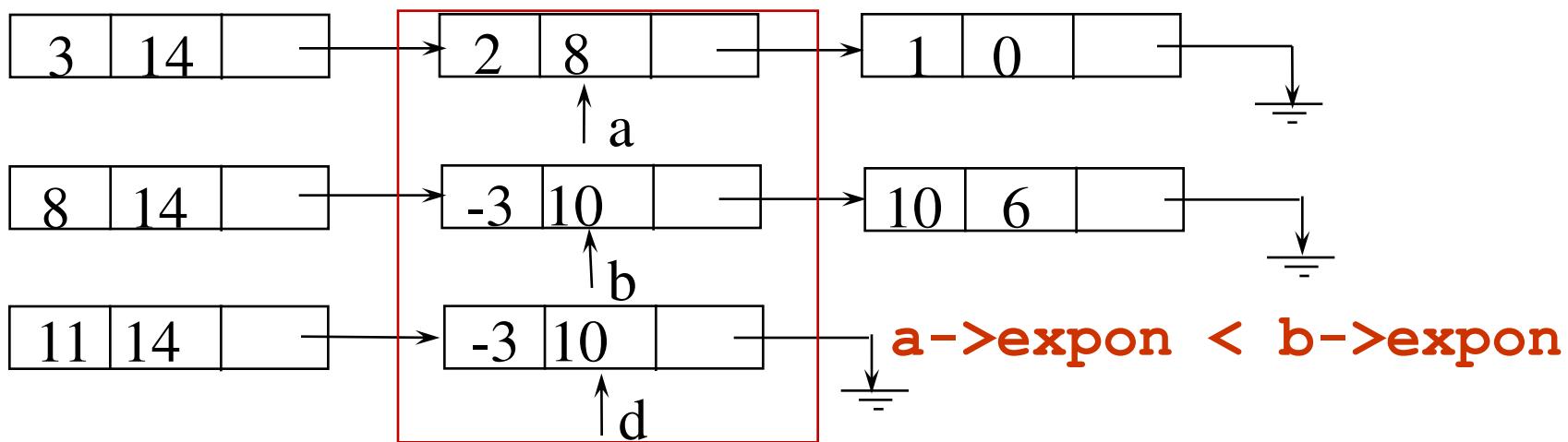
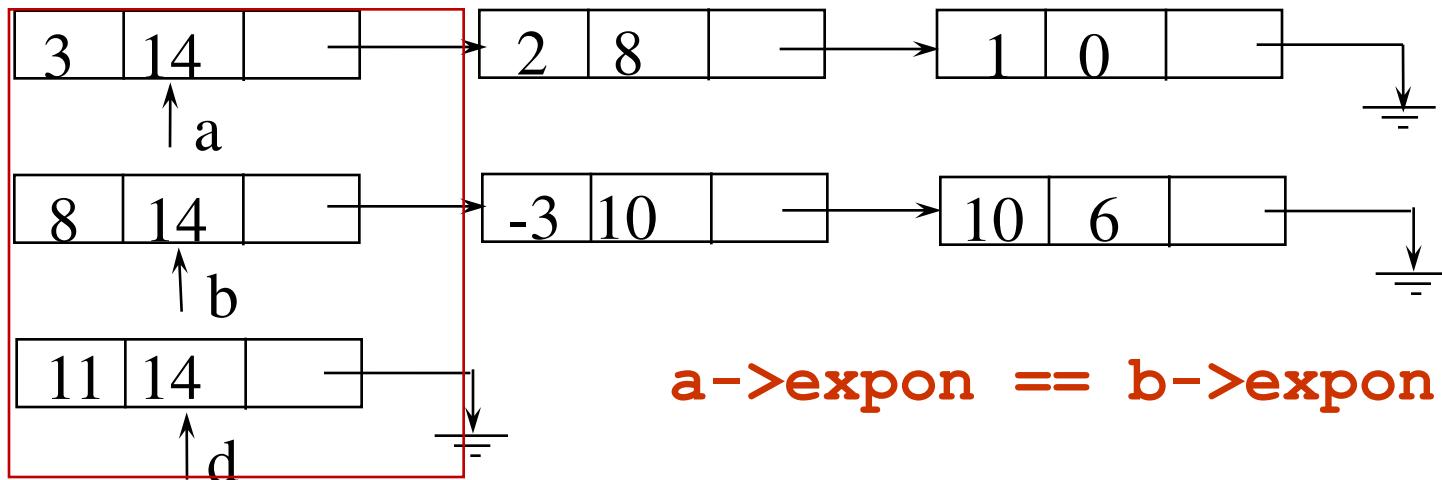
$$a = 3x^{14} + 2x^8 + 1$$



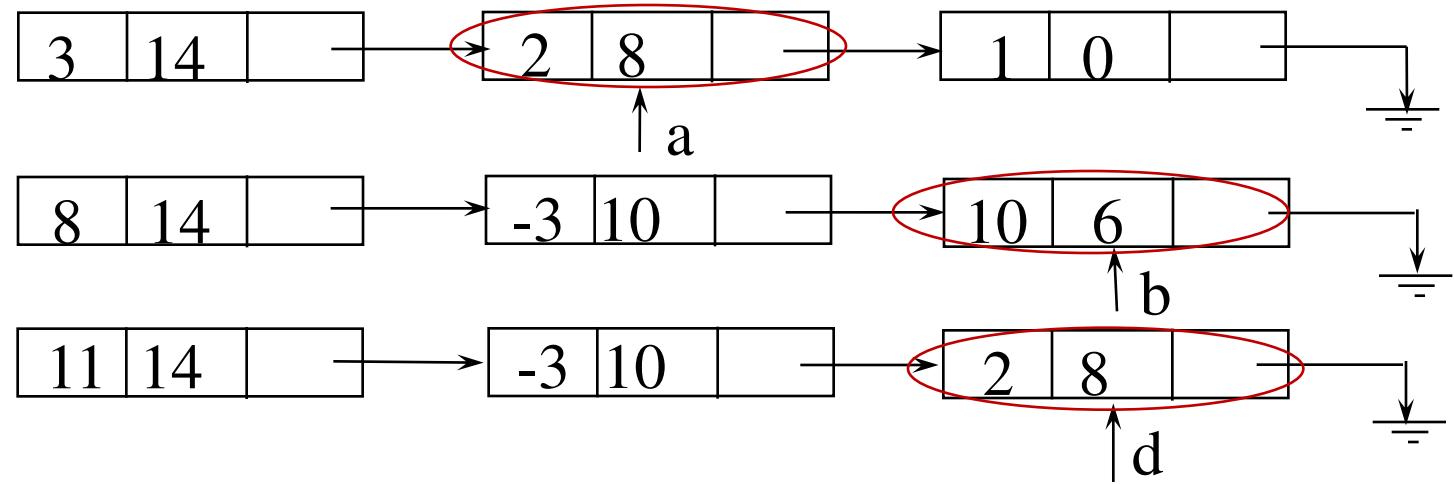
$$b = 8x^{14} - 3x^{10} + 10x^6$$



Adding Polynomials



Adding Polynomials (*Continued*)



```

#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
    int coef;
    int expon;
    struct node * next;
}NODE;

NODE *insert_order(NODE *first, int coe, int expo)
{
    NODE *n=(NODE *)malloc(sizeof(NODE));
    n->coef=coe;
    n->expon=expo;
    n->next=NULL;
    if(first==NULL) return n;
    if(expo>first->expon)
    {
        n->next=first;
        return n;
    }
    NODE *prev=NULL, *cur=first;
    while(cur!=NULL && expo< cur->expon)
    {
        prev=cur;
        cur=cur->next;
    }
    prev->next=n;
    n->next=cur;
    return first;
}

```

```

NODE *add_poly(NODE *L1, NODE *L2)
{
    NODE *L3=NULL;
    int c,sum;
    while(L1 &&L2)
    {
        if(L1->expon==L2->expon) c=0;
        else if(L1->expon>L2->expon) c=1;
        else c=2;
        switch(c)
        {
            case 0:sum=L1->coef+L2->coef;
                if(sum)
                    L3=insert_order(L3,sum,L1->expon);
                L1=L1->next;
                L2=L2->next;
                break;
            case 1:
                L3=insert_order(L3,L1->coef,L1->expon);
                L1=L1->next;
                break;
            case 2:
                L3=insert_order(L3,L2->coef,L2->expon);
                L2=L2->next;
                break;
        }
    }
}

```

```
while(L1)
{
L3=insert_order(L3,L1->coef,L1->expon);
L1=L1->next;
}
while(L2)
{
L3=insert_order(L3,L2->coef,L2->expon);
L2=L2->next;
}
return L3;
}
```

```
void display(NODE *first)
{
if(first==NULL)
{
printf("List is empty\n");
return;
}
while(first!=NULL)
{
printf("%dX%d",first->coef,first->expon);
if(first->next!=NULL) printf("+");
first=first->next;
}
}
```

```
int main()
{
    NODE *first=NULL,
    *L1=NULL,*L2=NULL,*L3;
    int choice, c,e;
    do
    {
        printf("\n1.create\n2.add_poly\n3.exit\n");
        printf("Enter the option\n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: while(1)      {
                            printf("Enter the coefficient \n");
                            scanf("%d",&c); if(c== -999) break;
                            printf("Enter the exponent\n");
                            scanf("%d",&e);
                            L1=insert_order(L1,c,e); }   display(L1);
                
```

```
                while(1)
                {
                    printf("\nEnter the coefficient \n");
                    scanf("%d",&c);
                    if(c== -999) break;
                    printf("Enter the exponent\n");
                    scanf("%d",&e);
                    L2=insert_order(L2,c,e);
                }
                display(L2);
                break;
            case 2:L3=add_poly(L1,L2);
                display(L3);
                break;
        }
    } while(choice<3);
}
```

Addition of two long numbers

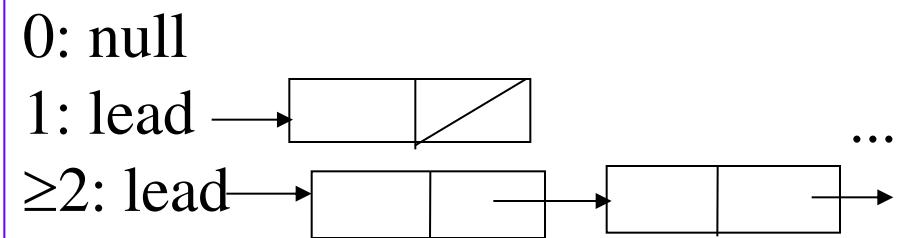
```
list_pointer AddLongInteger(list_pointer A,list_pointer B){  
    int digit,sum,carry;  
    list_pointer L3=NULL;  
    carry = 0;  
    while(A| |B)  
    {  
        sum = (A? A->data:0 )+ (B? B->data:0)+carry;  
        digit = sum%10;  
        carry = sum/10;  
        Insert_Front(&L3,digit);  
        if(A) A=A->link;  
        if(B) B=B->link;  
    }  
    if (carry)  
        Insert_Front(&L3,carry);  
    return L3;  
}
```

```
int main()
{
    list_pointer L1=NULL,L2=NULL,L3=NULL;
    char s[20];
    int choice, i,len;
    printf("Enter the first number\n");
    fflush(stdin);
    scanf("%s",s);
    len=strlen(s);
    for(i=len-1;i>=0;i--)
        Insert_Rear(&L1,s[i]-'0');
    printf("Enter the second number\n");
    fflush(stdin);
    scanf("%s",s);
    len=strlen(s);
    for(i=len-1;i>=0;i--)
        Insert_Rear(&L2,s[i]-'0');
    L3=AddLongInteger(L1,L2);
    printf("\nThe sum of the two numbers is: \t");
    display(L3);
}
```

Invert Single Linked Lists

Use two extra pointers: middle and trail.

```
list_pointer invert(list_pointer lead)
{
    list_pointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```



Circular Singly Linked List

```
void insert_Front(list_pointer *last, int data)
{
list_pointer new_node;
new_node=(list_pointer)malloc(sizeof(struct
list_node));
new_node->data=data;
new_node->link=NULL;
if(*last==NULL)
{new_node->link=new_node;
*last=new_node;
return;
}
new_node->link=(*last)->link;
(*last)->link=new_node;
```

```
void insert_Rear(list_pointer *last, int data)
{
list_pointer new_node;
new_node=(list_pointer)malloc(sizeof(struct list_node));
new_node->data=data;
new_node->link=NULL;
if(*last==NULL)
{new_node->link=new_node;
*last=new_node;
return;
}
new_node->link=(*last)->link;
(*last)->link=new_node;
*last=new_node;
}
```

```
void delete_front(list_pointer *last)
{ list_pointer temp;
 if(*last==NULL)
 {
 printf("List is empty\n");
 return ;
 }
 if((*last)->link==*last)
 {
 printf("The element deleted is %d\t",(*last)->data);
 *last=NULL;
 return;
 }
 temp=(*last)->link;
 (*last)->link=temp->link;
 printf("The element deleted is %d\t",temp->data);
 free(temp);
 }
```

```
void delete_rear(list_pointer * last)
{ list_pointer temp,x;
  if(*last==NULL)
  {
    printf("List is empty\n");
    return ;
  }
  if((*last)->link==*last)
  {
    printf("The element deleted is %d\t",(*last)->data);
    *last=NULL;
    return;
  }
  temp=(*last)->link;
  while(temp->link!=*last)
  {
    temp=temp->link;
    temp->link=(*last)->link;
  }
  printf("The element deleted is %d\t",(*last)->data);
  free(*last);
  *last=temp;
}
```

```
void display(list_pointer last)
{
    list_pointer x;
    if(last==NULL)
    {
        printf("List is empty\n");
        return;
    }
    x=last->link;
    while(x!=last)
    {
        printf("%d\t",x->data);
        x=x->link;
    }
    printf("%d\t",x->data);
}
```

```
int main()
{
list_pointer first=NULL; int choice, element;
do{
printf("\n1.Insert_Front\n2.Insert_Rear\n3.Dis
play\n4.delete_Front\n5.delete_Rear\n6.exit\n
");
printf("Enter the option\n");
scanf("%d",&choice);
switch(choice){
case 1:printf("Enter the element to be
inserted\n");
scanf("%d",&element);
insert_Front(&first,&element); break;
case 2:printf("Enter the element to be
inserted\n");
scanf("%d",&element);
insert_Rear(&first,&element); break;
}
```

```
case 3:printf("The elements of the list are\n");
display(first);
break;
case 4:delete_front(&first);
break;
case 5:delete_rear(&first);
break;
}
} while(choice<6);
}
```

Singly Linked List with Header Node

```
typedef struct list_node *list_pointer;    list_pointer head=(list_pointer)malloc(sizeof(struct list_node));  
struct list_node  
{int data;  
list_pointer link;  
};  
  
void Insert_Front(list_pointer head, int data)  
{  
list_pointer temp;  
temp=(list_pointer)malloc(sizeof(struct  
list_node));  
temp->data=data;  
temp->link=head->link;  
head->link=temp;  
head->data++;  
}
```

```
void Insert_Rear(list_pointer head, int data)
{
list_pointer temp,last_node;
temp=(list_pointer)malloc(sizeof(struct
list_node));
temp->data=data;
temp->link=NULL;
if(head->data==0)
{
    head->link=temp;
    head->data++;
    return;
}
last_node=head->link;
while(last_node->link!=NULL)
    last_node=last_node->link;
last_node->link=temp;
}
```

```
void Delete_Front(list_pointer head)
{ list_pointer temp;
  if(head->data==0){printf("List is empty\n");return ;}
  temp=head->link;
  head->link=temp->link;
  head->data--;
  printf("The element deleted is %d\t",temp->data);
  free(temp);
}
```

```
void Delete_Rear(list_pointer head)
{ list_pointer next,prev;
if(head->data==0)
{printf("List is empty\n");
return;}
next=head;
while(next->link!=NULL)
{
    prev=next;
    next=next->link;
}
printf("The element deleted is %d\t",next->data);
prev->link=NULL;
free(next);
head->data--;
}
```

```
void display(list_pointer head)
{
    list_pointer x;
    if(head->data==0)
    {
        printf("List is empty\n");
        return;
    }
    x=head->link;
    while(x!=NULL)
    {
        printf("%d\t",x->data);
        x=x->link;
    }
}
```

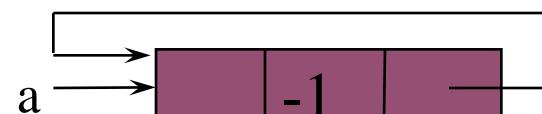
```
int main()
{
list_pointer head=(list_pointer)malloc(sizeof(struct list_node));
head->data=0;
head->link=NULL;
int choice, element;
do{
printf("\n1.Insert_Front\n2.Insert_Rear\n3.Display\n4.delete_Front\n5.delete_Rear
\n6.exit\n ");
printf("Enter the option\n");
scanf("%d",&choice);
switch(choice)
{
case 1:printf("Enter the element to be inserted\n");
        scanf("%d",&element);
        Insert_Front(head,element);
        break;
}
```

```
case 2:printf("Enter the element to be
inserted\n");
    scanf("%d",&element);
    Insert_Rear(head,element);
    break;
case 3:printf("The elements of the list are\n");
    display(head);
    break;
case 4:Delete_Front(head);
    break;
case 5:Delete_Rear(head);
    break;
}
} while(choice<6);
}
```

Head Node

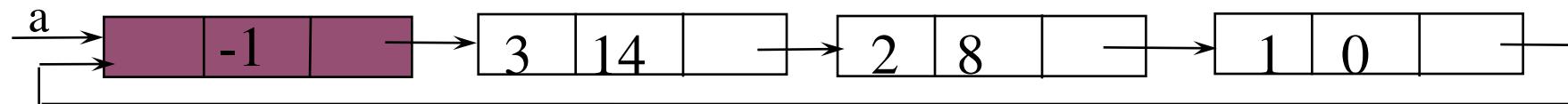
Represent polynomial as circular list.

(1) zero



Zero polynomial

(2) others



$$a = 3x^{14} + 2x^8 + 1$$

4.8 Doubly Linked List

Move in forward and backward direction.

Singly linked list (*in one direction only*)

How to get the preceding node during deletion or insertion?

Using 2 pointers

Node in doubly linked list consists of:

1. *left link field* (llink)
2. *data field* (item)
3. *right link field* (rlink)


```
#include<stdio.h>
#include<stdlib.h>
typedef struct list_node *list_pointer;
struct list_node
{
    int data;
    list_pointer llink;
    list_pointer rlink;
};
```

```
void Insert_Front(list_pointer *first, int data)
{
    list_pointer new_node;
    new_node=(list_pointer)malloc(sizeof(struct
list_node));
    new_node->data=data;
    new_node->llink=NULL;
    new_node->rlink=NULL;
    if(*first==NULL) {*first=new_node; return;}
    (*first)->llink=new_node;
    new_node->rlink=*first;
    *first=new_node; }
```

```
void Insert_Rear(list_pointer *first, int data)
{
list_pointer new_node,last_node;
new_node=(list_pointer)malloc(sizeof(struct
list_node));
new_node->data=data;
new_node->llink=NULL;
new_node->rlink=NULL;
if(*first==NULL) {*first=new_node;return;}
last_node=*first;
while(last_node->rlink!=NULL)
last_node=last_node->rlink;
last_node->rlink=new_node;
new_node->llink=last_node;
}
```

```
void Delete_Front(list_pointer * first)
{ list_pointer temp;
if(*first==NULL){printf("List is empty\n");return ;}
if((*first)->rlink==NULL)
{
    printf("The element deleted is %d\t",(*first)->data);
    *first=NULL;
    return;
}
temp=(*first);
*first=(*first)->rlink;
printf("The element deleted is %d\t",temp->data);
free(temp);
(*first)->llink=NULL;
}
```



```
void Delete_Rear(list_pointer * first)
{ list_pointer temp;
if(*first==NULL){printf("List is empty\n");return ;}
if((*first)->rlink==NULL)
{
    printf("The element deleted is %d\t",(*first)->data);
    *first=NULL;
    return;
}
temp=(*first);
while(temp->rlink)
    temp=temp->rlink;
temp->llink->rlink=NULL;
printf("The element deleted is %d\t",temp->data);
free(temp);
}
```

```
void display(list_pointer first)
{
    if(first==NULL)
    {
        printf("List is empty\n");
        return;
    }
    while(first!=NULL)
    {
        printf("%d\t",first->data);
        first=first->rlink;
    }
}
```

```
int main()
{
    list_pointer first=NULL;  int choice, element;
    do{
        printf("\n1.Insert_Front\n2.Insert_Rear\n3.Display\n4.delete_Front\n5.delete_Rear\n6.exit\n ");
        printf("Enter the option\n");
        scanf("%d",&choice);
        switch(choice){
            case 1:printf("Enter the element to be inserted\n");
                scanf("%d",&element);
                Insert_Front(&first,element);
                break;
            case 2:printf("Enter the element to be inserted\n");
                scanf("%d",&element);
                Insert_Rear(&first,element);      break;
            case 3:printf("The elements of the list are\n");
                display(first);      break;
            case 4:Delete_Front(&first);      break;
            case 5:Delete_Rear(&first);      break;
        } } while(choice<6);}
```

Circular Doubly Linked Lists with Header Node

```
#include<stdio.h>
#include<stdlib.h>
typedef struct list_node *list_pointer;
struct list_node
{
    int data;
    list_pointer llink;
    list_pointer rlink;
};
```

```
int main()
{
    list_pointer head=(list_pointer)malloc(sizeof(struct list_node));
    head->data=0;
    head->llink=head;
    head->rlink=head;
    int choice, element;
    Do
```

.

.

.

.

.

```
void Insert_Front(list_pointer head, int data)
{
list_pointer new_node;
new_node=(list_pointer)malloc(sizeof(struct list_node));
new_node->data=data; new_node->llink=NULL;new_node->rlink=NULL;
if(head->rlink==head)
{
    new_node->rlink=head;
    head->rlink=new_node;
    new_node->llink=head ;
    head->llink=new_node;
    head->data++; return;
}
new_node->rlink=head->rlink;
head->rlink->llink=new_node;
new_node->llink=head;
head->rlink=new_node;
head->data++;
```

```
void Insert_Rear(list_pointer head, int data)
{
list_pointer new_node;
new_node=(list_pointer)malloc(sizeof(struct
list_node));
new_node->data=data;
new_node->llink=NULL;
new_node->rlink=NULL;
if(head->rlink==head)
{
new_node->rlink=head;
head->rlink=new_node;
new_node->llink=head ;
head->llink=new_node;
head->data++;
return;
}

new_node->llink=head->llink;
head->llink->rlink=new_node;
new_node->rlink=head;
head->llink=new_node;
head->data++;
}
```

```
void Delete_Front(list_pointer head)
{
    list_pointer temp;
    if(head->data==0){printf("List is empty\n");return ;}
    temp=head->rlink;
    temp->rlink->llink=head;
    head->rlink=temp->rlink;
    head->data--;
    printf("The element deleted is %d\t",temp->data);
    free(temp);
}
```

```
void Delete_Rear(list_pointer head)
{ list_pointer temp;
  if(head->data==0){printf("List is empty\n");return ;}
  temp=head->llink;
  temp->llink->rlink=head;
  head->llink=temp->llink;
  head->data--;
  printf("The element deleted is %d\t",temp->data);
  free(temp);
}
```

```
void display(list_pointer head)
{
list_pointer temp;
if(head->llink==head)
{
printf("List is empty\n");
return;
}
temp=head->rlink;
while(temp->rlink!=head)
{
    printf("%d\t",temp->data);
    temp=temp->rlink;
}
printf("%d\t",temp->data);
}
```

```
int main()
{
    list_pointer head=(list_pointer)malloc(sizeof(struct
list_node));
    head->data=0; head->llink=head; head->rlink=head;
    int choice, element;
    do{
        printf("\n1.Insert_Front\n2.Insert_Rear\n3.Display\n4.delete_F
ront\n5.delete_Rear\n6.exit\n ");
        printf("Enter the option\n");
        scanf("%d",&choice);
        switch(choice){
            case 1:printf("Enter the element to be inserted\n");
                scanf("%d",&element);
                Insert_Front(head,element);      break;
            case 2:printf("Enter the element to be inserted\n");
                scanf("%d",&element);
                Insert_Rear(head,element);      break;
            case 3:printf("The elements of the list are\n");
                display(head);
                break;
            case 4:Delete_Front(head);      break;
            case 5:Delete_Rear(head);      break;
        } } while(choice<6);}
```



```
//Long Integer addition using Circular Doubly Linked List
//with header node
#include "DLL with header.h"
//Function creates linked list representing the
long integer
Nodeptr ReadLongInteger() {
    Nodeptr head;
    char str[100];
    int i,n;
    printf("Enter the string representing long integer : ");
    scanf("%s", str);
    for(n=0;str[n];n++);
    head = getnode();
    head->llink = head->rlink = head;
//Extract each digit from left and insert at the front of
the list
    for(i=n-1;i>=0;i--)
        InsertFront(str[i]-'0', head);
    return head; }
```

```
Nodeptr AddLongInteger(Nodeptr A,Nodeptr B) {  
    int digit,sum,carry;  
    Nodeptr head,r,R,a,b;  
    carry = 0;  
    a=A->llink;  
    b=B->llink;  
    head = getnode();  
    head->llink= head->rlink = head;  
  
    while(a!= A && b != B) {  
        sum = a->data + b->data + carry;  
        digit = sum%10;  
        carry = sum/10;  
        InsertFront(digit,head);  
        a = a->llink;  
        b = b->llink;  
    }  
}
```

```
//Identify the bigger number
if (a!= A) {
    r=a; R=A;      }
else
{ r=b;R=B;      }
//add carry to remaining digits of bigger number
while(r!= R) {
    sum = r->data + carry;
    digit = sum%10;
    carry = sum/10;
    InsertFront(digit,head);
    r = r->llink;
}
//Insert the last carry, if present.
if (carry)
    InsertFront(carry,head);
return head;
}
```

```
int main() {  
  
    Nodeptr a,b,sum;  
    a = ReadLongInteger();  
    b = ReadLongInteger();  
    printf("\nhead Integer : \n");  
  
    Display(a);  
    printf("\nSecond Integer : \n");  
  
    Display(b);  
    sum=AddLongInteger(a,b);  
    printf("\nSum of 2 given Integers : \n");  
  
    Display(sum);  
    return 0;  
}
```

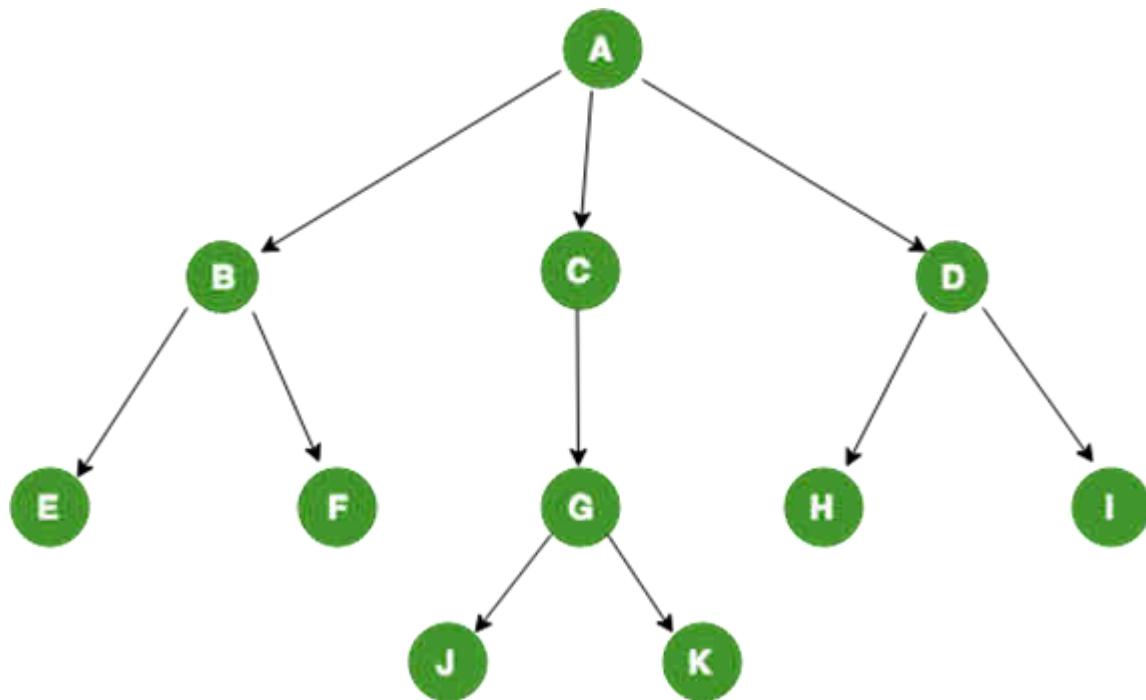
CHAPTER 5

Trees

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.

Tree is a non-linear **data structure**. ... A **tree** can be represented using various primitive or user defined **data types**. To implement **tree**, we can make use of arrays/ linked lists.



Relations in a Tree:

- A is the root of the tree
- A is Parent of B, C and D
- B is child of A
- B, C and D are siblings
- A is grand-parent of E, F, G, H and I

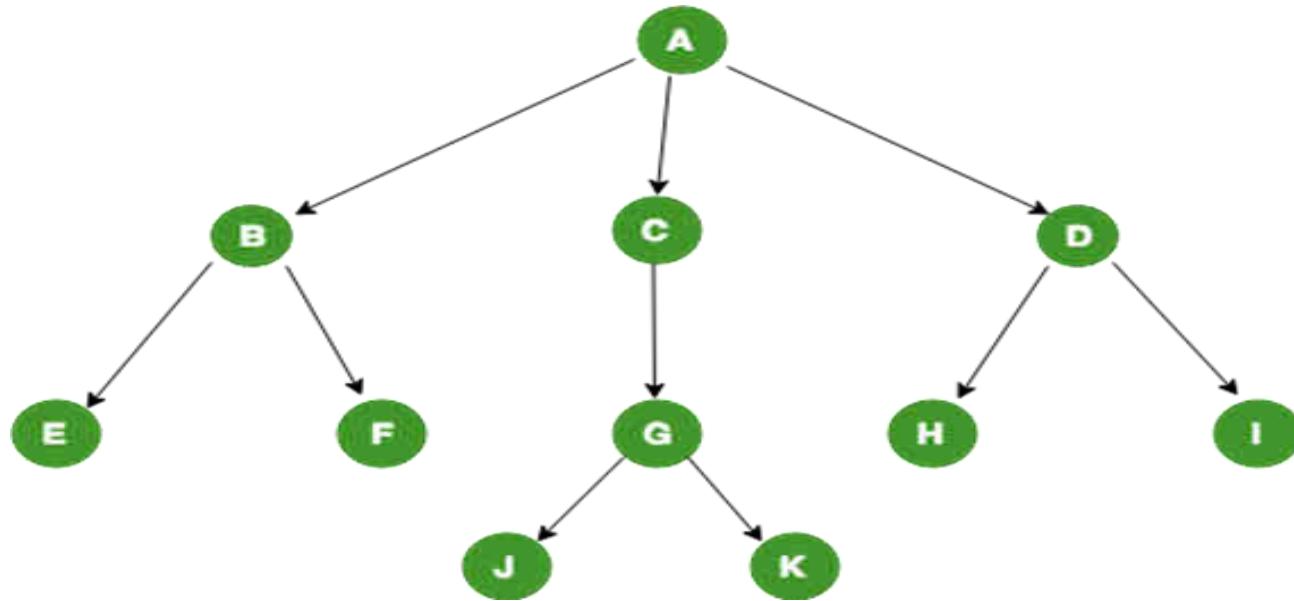
Properties of Tree:

- Every tree has a special node called the root node. The root node can be used to traverse every node of the tree. It is called root because the tree originated from root only.
- If a tree has N vertices(nodes) than the number of edges is always one less than the number of nodes(vertices) i.e $N-1$. If it has more than $N-1$ edges it is called a graph not a tree.
- Every child has only a single Parent but Parent can have multiple child.

Types of Trees in Data Structure

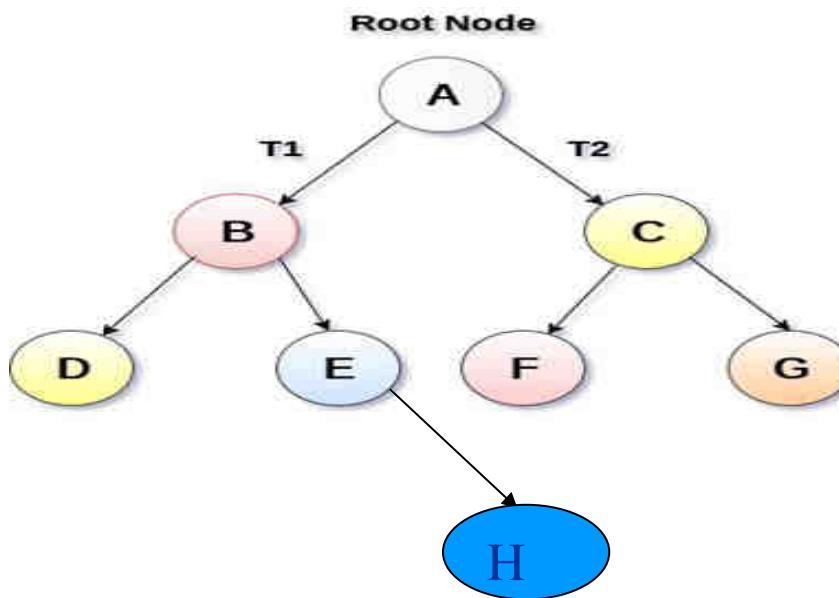
General Tree

A tree is called a general tree when there is no constraint imposed on the hierarchy of the tree. In General Tree, each node can have infinite number of children. This tree is the super-set of all other types of trees. The tree shown in Fig 1 is a General Tree.



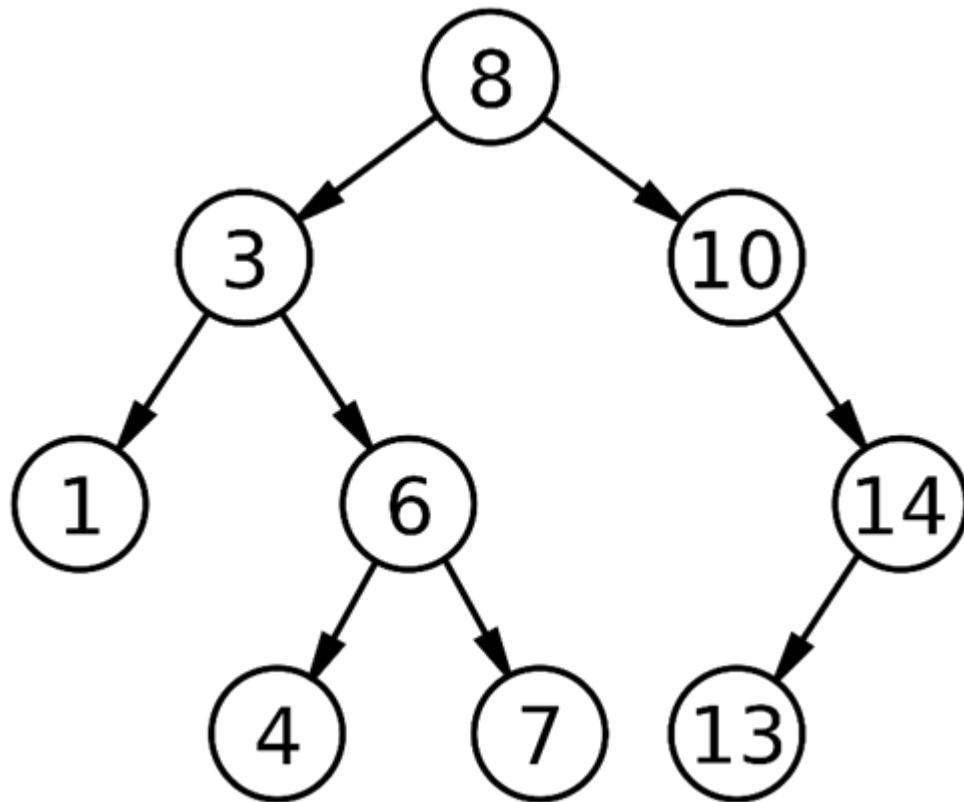
Binary Tree

Binary tree is the type of tree in which each parent can have at most two children. The children are referred to as left child or right child. This is one of the most commonly used trees. When certain constraints and properties are imposed on Binary tree it results in a number of other widely used trees like:



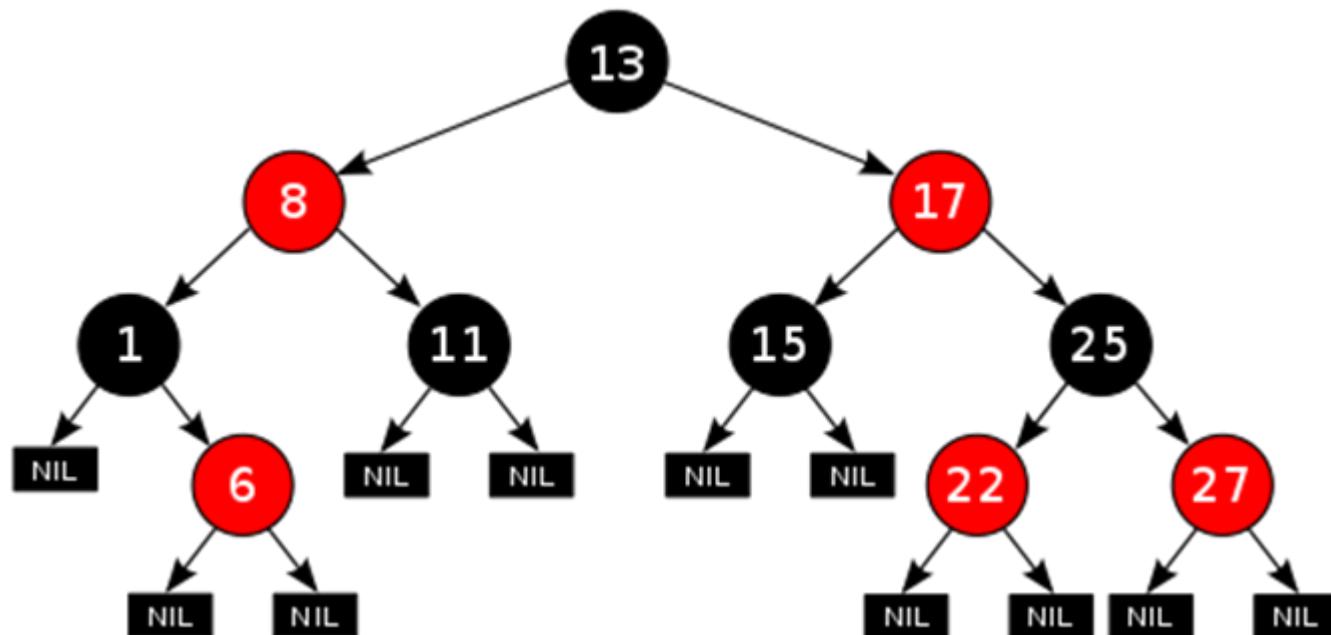
BST (Binary Search Tree):

Binary Search Tree (BST) is an extension of Binary tree with some added constraints. In BST, the value of the left child of a node must be smaller than or equal to the value of its parent and the value of the right child is always larger than or equal to the value of its parent.

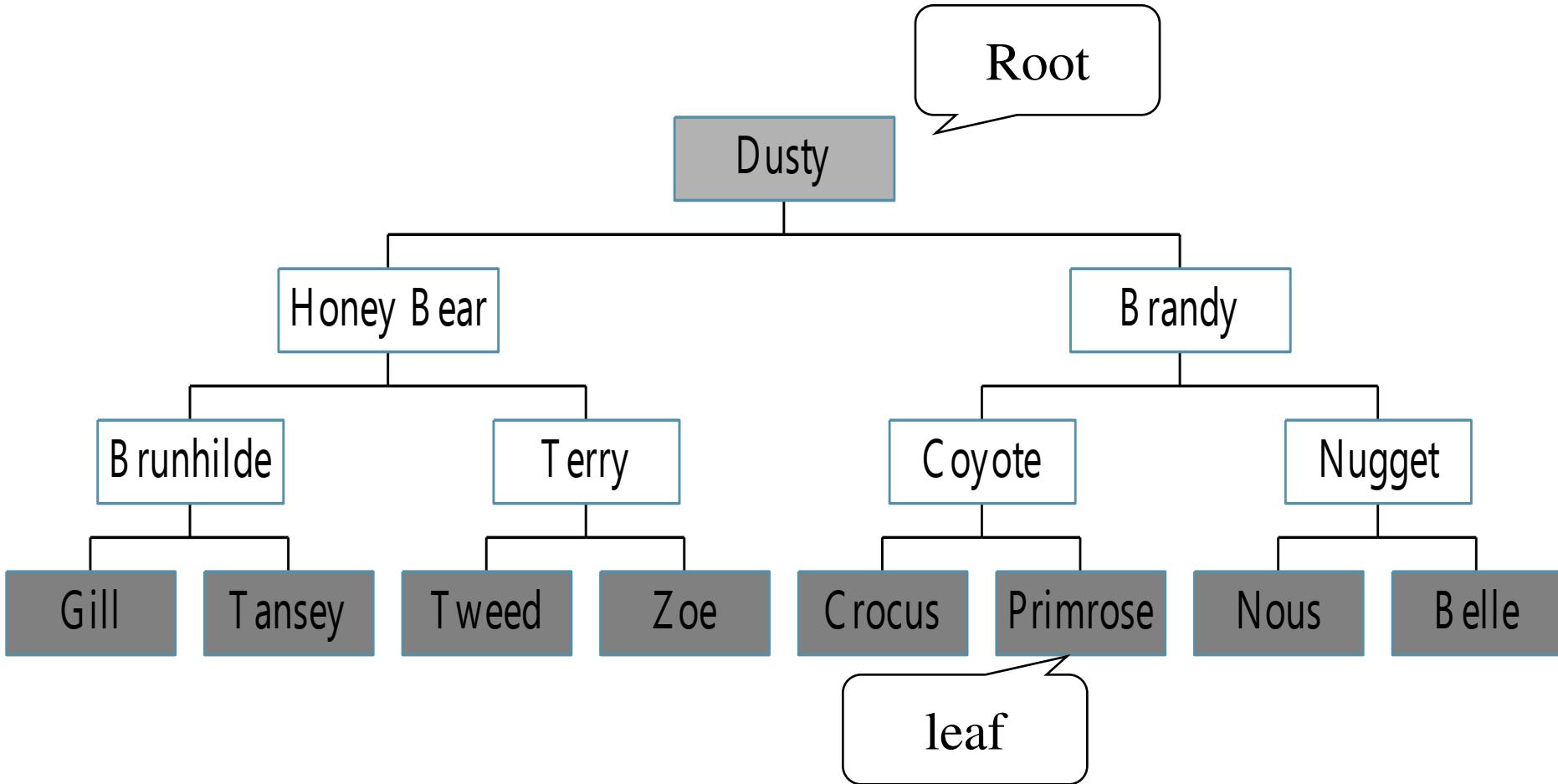


RBT(Red-Black-Tree):

Red-Black is another type of self-balancing tree. The name Red-Black is given to it because each node in a Red-Black tree is either painted Red or Black according to the properties of the Red-Black Tree. This make sure that the tree remains balanced.



Trees



Definition of Tree

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree.
- We call T_1, \dots, T_n the subtrees of the root.

Level and Depth

node (13)

degree of a node

leaf (terminal)

nonterminal

parent

children

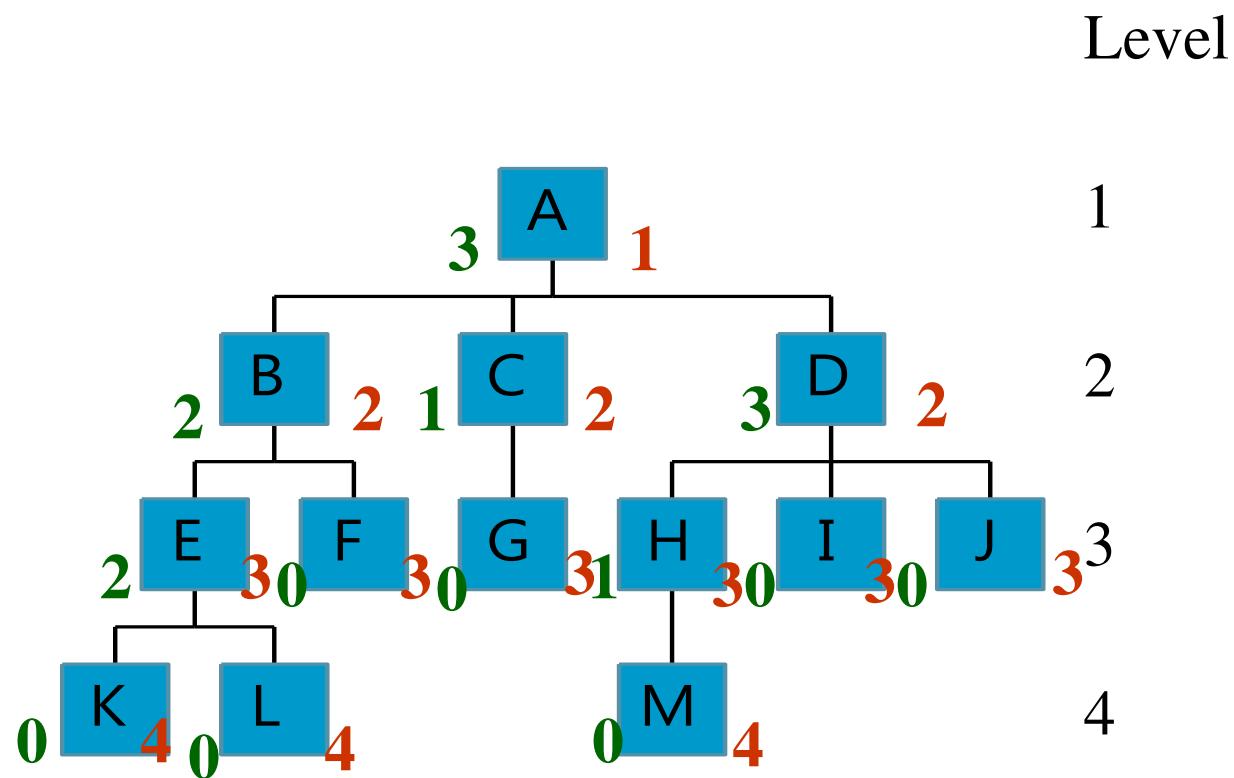
sibling

degree of a tree (3)

ancestor

level of a node

height of a tree (4)



Terminology

- The degree of a node is the number of subtrees of the node
 - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

Representation of Trees

n List Representation

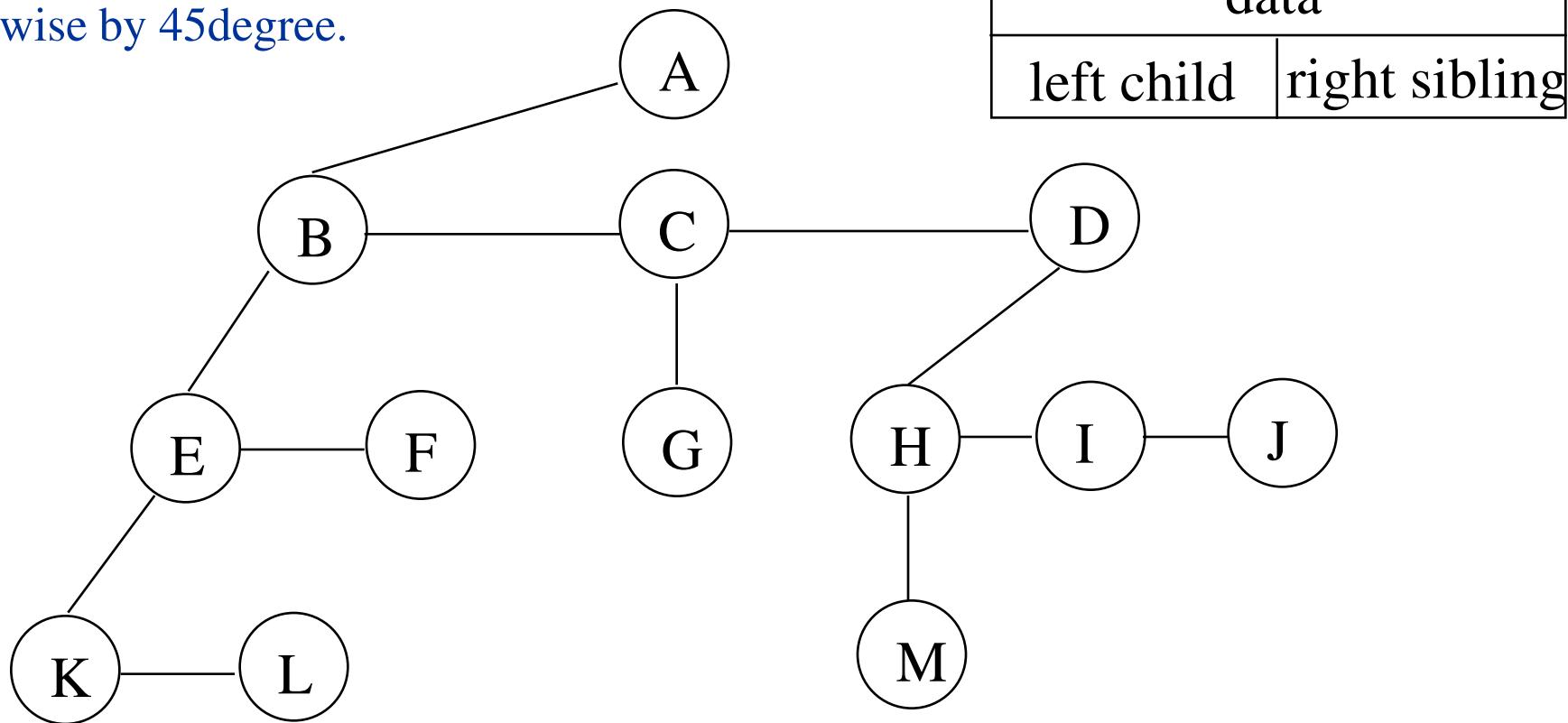
- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- The root comes first, followed by a list of sub-trees

data	link 1	link 2	...	link n
------	--------	--------	-----	--------

How many link fields are
needed in such a representation?

Left Child - Right Sibling

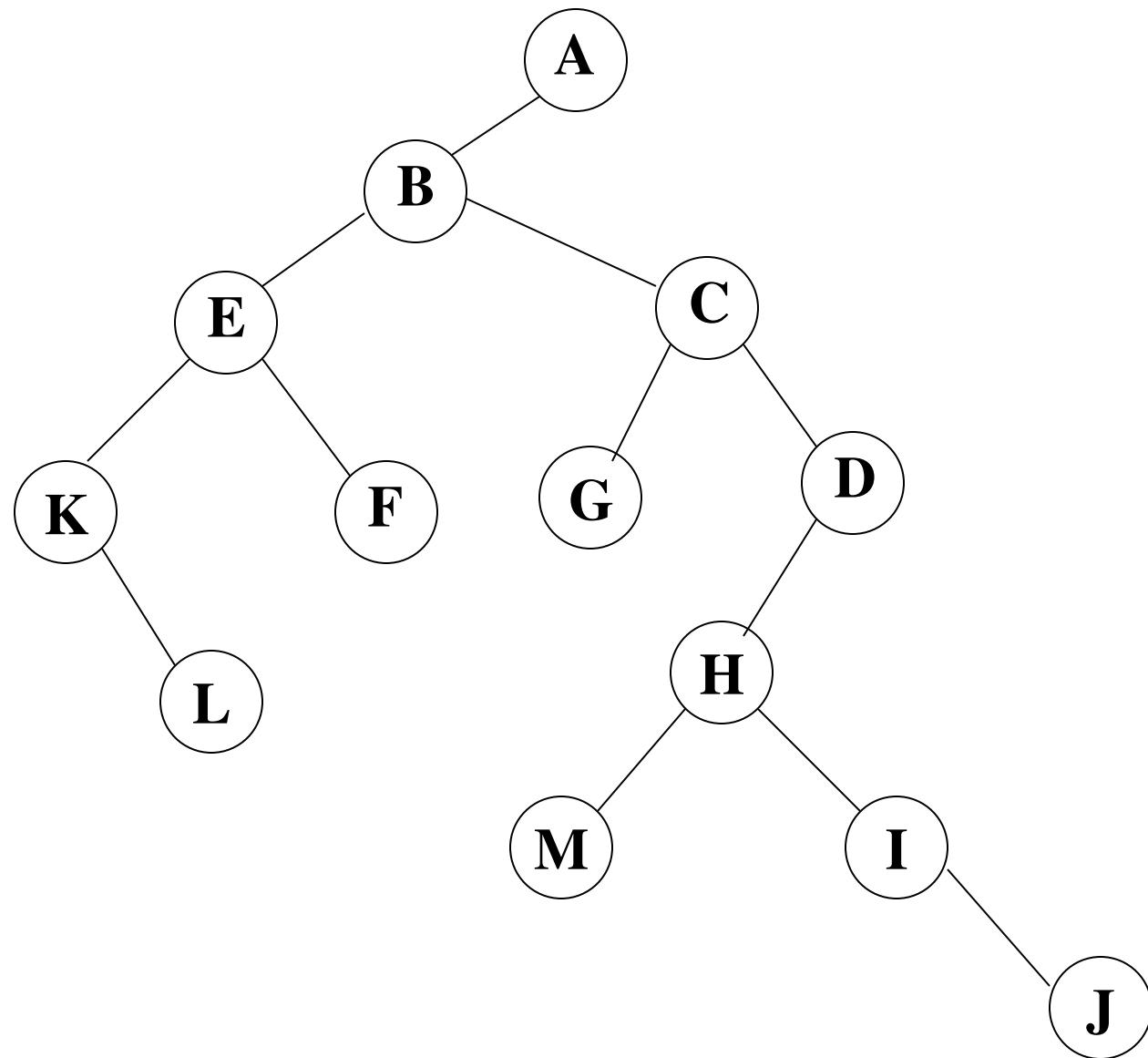
To obtain the degree 2tree representation of a tree, rotate right-sibling pointers clockwise by 45degree.



Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
 - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)



Abstract Data Type Binary_Tree

structure *Binary_Tree*(abbreviated *BinTree*) is objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

functions:

for all $bt, bt1, bt2 \in BinTree, item \in element$

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(bt)::= if (bt==empty binary tree) return TRUE else return FALSE

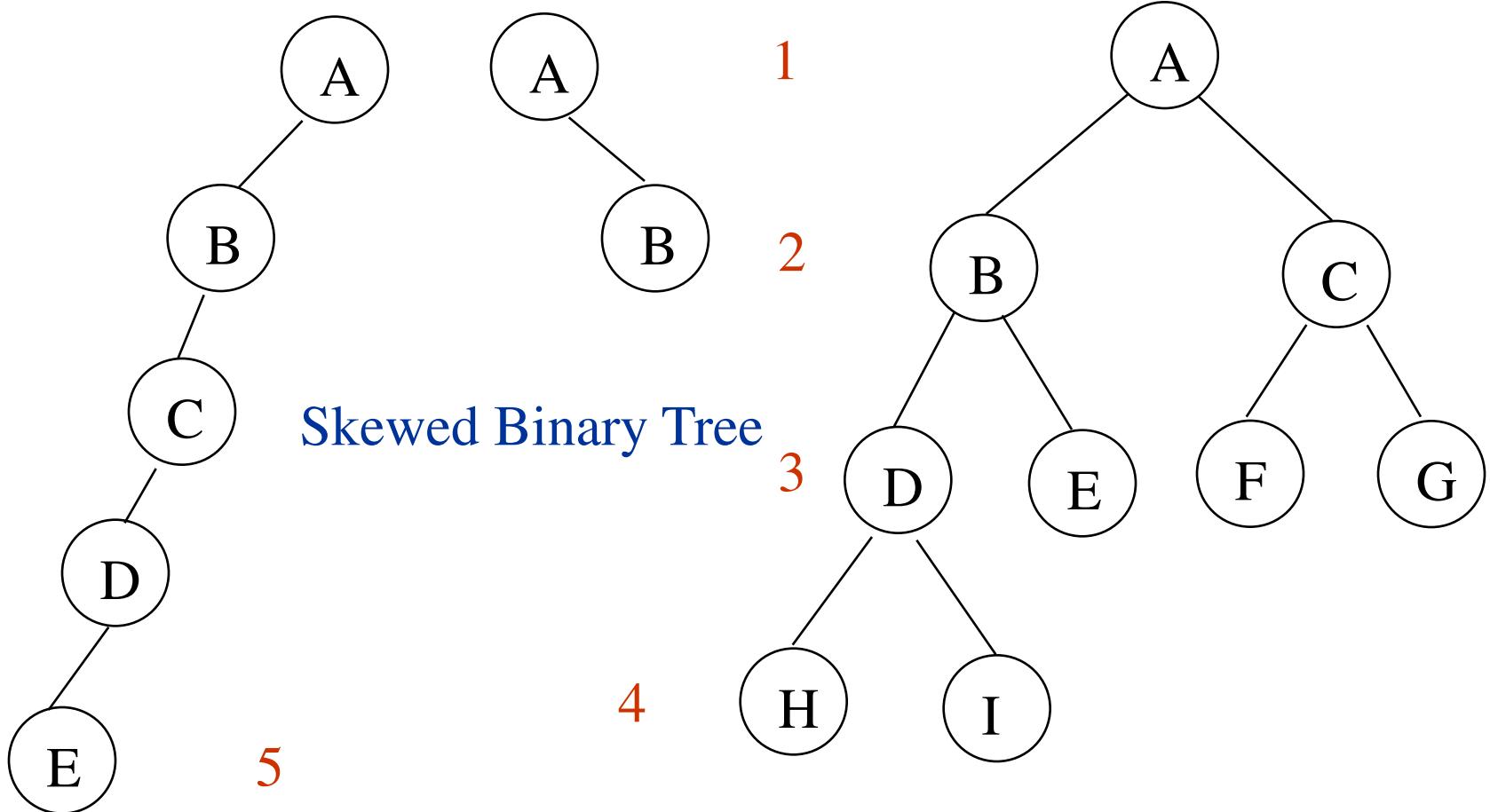
BinTree MakeBT(*bt1, item, bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

Samples of Trees



Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
- The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Prove by induction.

$$\sum_{i=1}^k 2^{i-1} = 2^k - 1$$

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$

proof:

Let n and B denote the total number of nodes & branches in T .

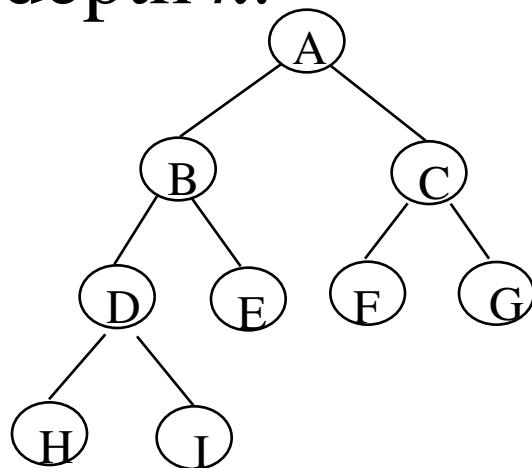
Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$$\begin{aligned}n &= n_0 + n_1 + n_2, \quad B + 1 = n, \quad B = n_1 + 2n_2 \implies n_1 + 2n_2 + 1 = n, \\n_1 + 2n_2 + 1 &= n_0 + n_1 + n_2 \implies n_0 = n_2 + 1\end{aligned}$$

A full binary tree is a tree in which every node other than the leaves has two children. A complete binary tree is a **binary tree in which every level, except possibly the last, is completely filled**, and all nodes are as far left as possible

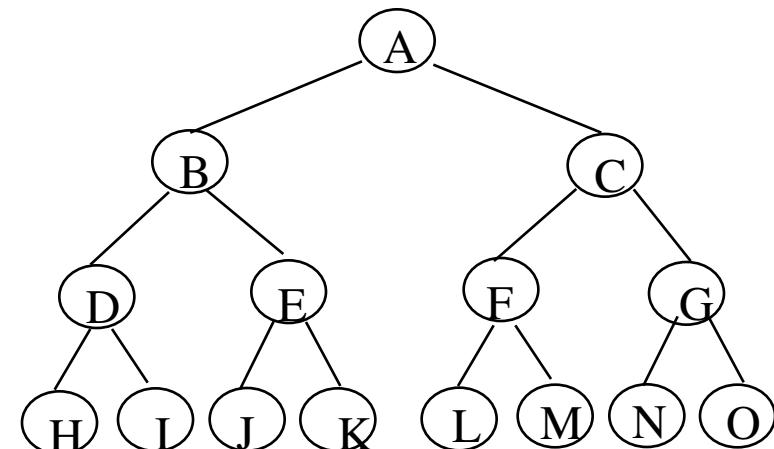
Full BT VS Complete BT

- n A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.
- n A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k .



Complete binary tree

CHAPTER 5

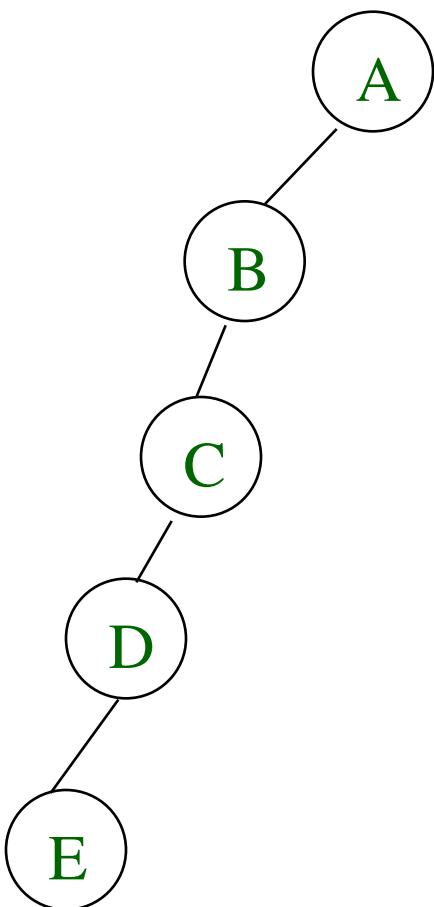


Full binary tree of depth 4

Binary Tree Representations

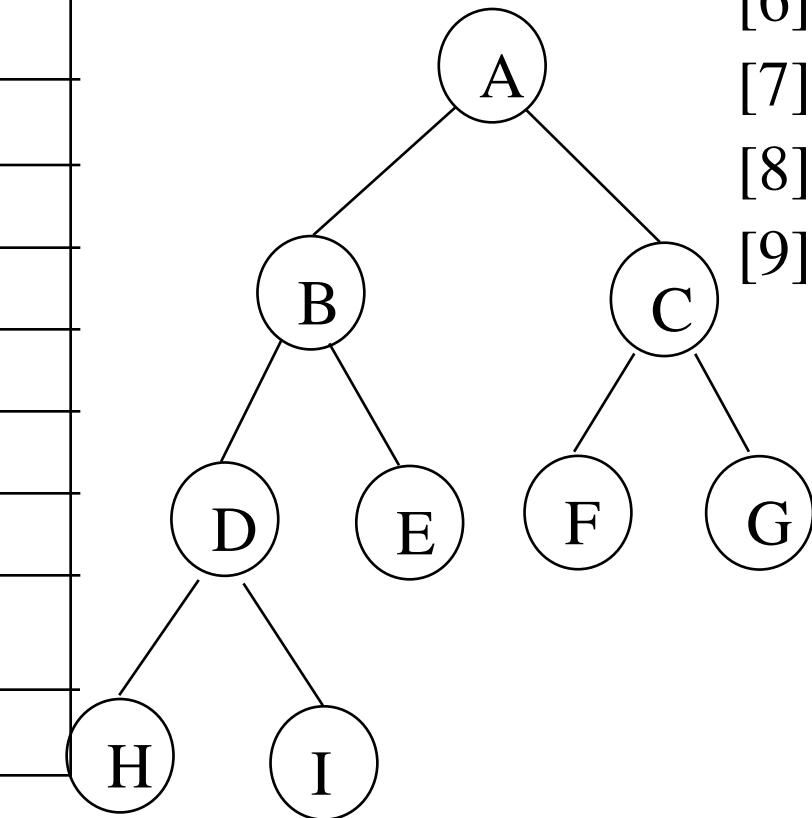
- n If a complete binary tree with n nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:
 - $\text{parent}(i)$ is at $i/2$ if $i \neq 1$. If $i=1$, i is at the root and has no parent.
 - $\text{left_child}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
 - $\text{right_child}(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.

Sequential Representation



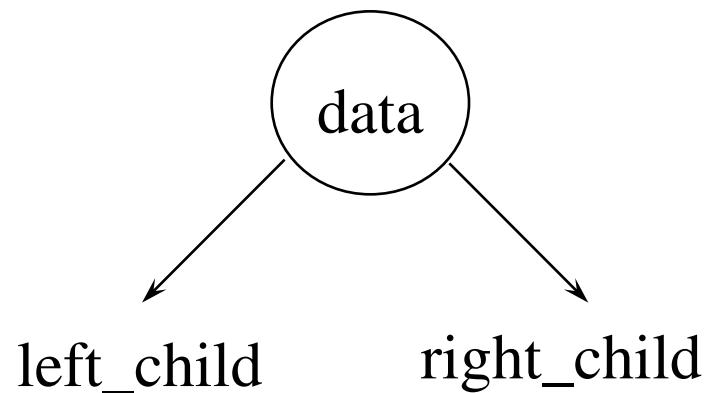
[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

- (1) waste space
(2) insertion/deletion problem



Linked Representation

```
typedef struct node *tree_pointer;  
typedef struct node {  
    int data;  
    tree_pointer left_child, right_child;  
};
```



Binary Tree Traversals

- n Let L, V, and R stand for moving left, visiting the node, and moving right.
- n There are six possible combinations of traversal
 - LVR, LRV, VLR, VRL, RVL, RLV
- n Adopt convention that we traverse left before right, only 3 traversals remain
 - LVR, LRV, VLR
 - inorder, postorder, preorder


```
#include<stdio.h>
#include<stdlib.h>
typedef struct node *tree_pointer;
typedef struct node {
    char data;
    tree_pointer left_child, right_child;
};

void inorder(tree_pointer ptr)
{
    if(ptr)
    {
        inorder(ptr->left_child);
        printf(" Value is %5c\n",ptr->data);
        inorder(ptr->right_child);
    }
}
```

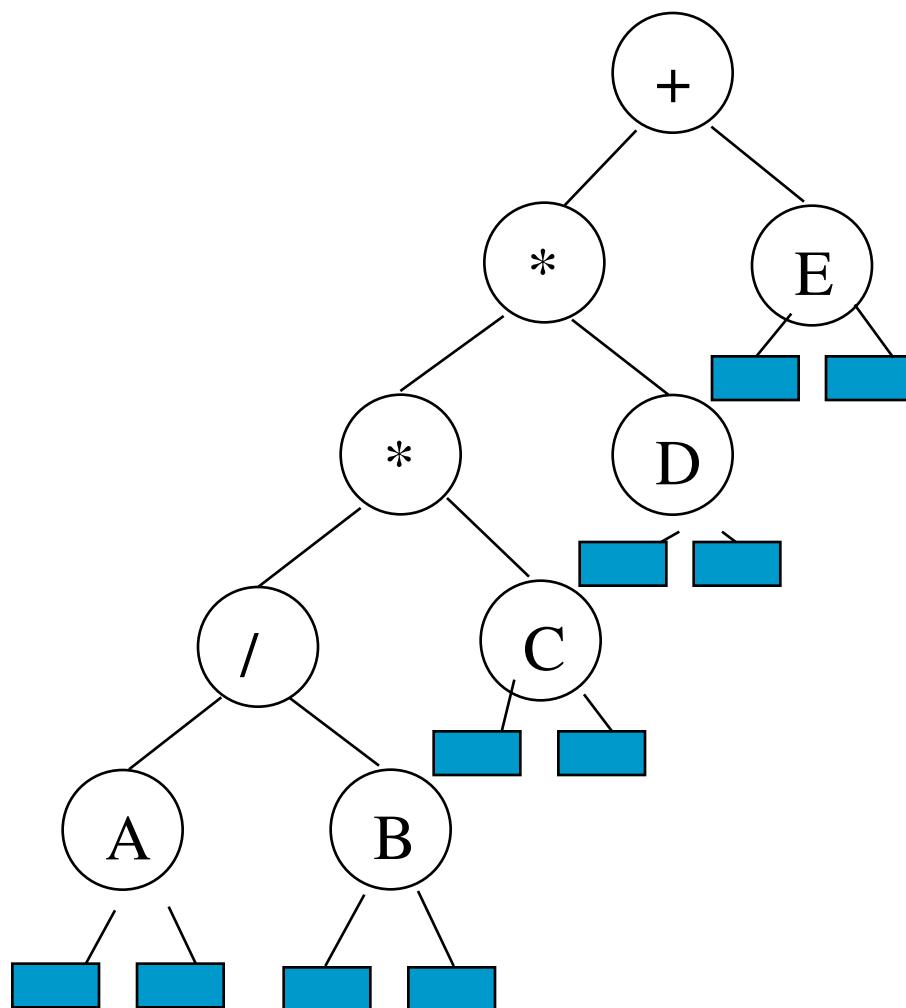
```
void insert(tree_pointer *root,char x,char *s)
{
    int i=0;
    tree_pointer p,previous,current;
    p=(tree_pointer )malloc(sizeof(struct node));
    p->data=x;
    p->left_child=NULL;
    p->right_child=NULL;
    if(*root==NULL){*root=p;return;}
    previous=NULL;      current=root;
    while(current!=NULL)
    {
        previous=current;
        if(*(s+i)=='L'||*(s+i)=='l')  current=current->left_child;
        else
            current=current->right_child;
        i++;
    }
}
```

```
if(current!=NULL || *(s+i)!="\0")
{
    printf("insertion is not possible\n");
    return ;
}
i=i-1;
if(*(s+i)=='L'||*(s+i)=='l')
    previous->left_child=p;
else
    previous->right_child=p;
}
```

```
void main()
{
int op;
tree_pointer root;
char path[10],n;
root=(tree_pointer )malloc(sizeof(struct node));
root->data='+';
root->right_child=root->left_child=NULL;
do
{
    printf("\n 1.Insertion");
    printf("\n 2.Inorder");
    printf("\n 3.Quit");
    printf("\n Enter your choice\n");
    scanf("%d",&op);
```

```
switch (op)
{
    case 1: printf("\n Enter the element to insert\n");
              fflush(stdin);
              scanf("%c",&n);
    if(root==NULL)insert(&root,n,'x');
    else
    {
        printf("Enter the path\n");
        flush(stdin);  scanf("%s",path);
        insert(&root,n,path);}
    break;
    case 2: inorder(root);
    break;
    default: exit(0);
}
}while(op<3); }
```

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
    }
}
```

A / B * C * D + E

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

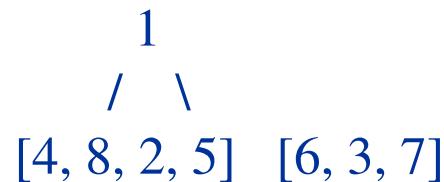
Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

Let us see the process of constructing tree from $\text{in[]} = \{4, 8, 2, 5, 1, 6, 3, 7\}$ and $\text{post[]} = \{8, 4, 5, 2, 6, 7, 3, 1\}$

- 1) We first find the last node in $\text{post}[]$. The last node is “1”, we know this value is root as root always appear in the end of postorder traversal.
- 2) We search “1” in $\text{in}[]$ to find left and right subtrees of root. Everything on left of “1” in $\text{in}[]$ is in left subtree and everything on right is in right subtree.



- 3) We recur the above process for following two.
 -b) Recur for $\text{in}[] = \{6, 3, 7\}$ and $\text{post}[] = \{6, 7, 3\}$
.....Make the created tree as right child of root.
 -a) Recur for $\text{in}[] = \{4, 8, 2, 5\}$ and $\text{post}[] = \{8, 4, 5, 2\}$.
.....Make the created tree as left child of root.

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(tree_pointer node)
{
    int top= -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node=node->left_child)
            add(&top, node);/* add to stack */
        node= delete(&top);
                    /* delete from stack */
        if (!node) break; /* empty stack */
        printf("%D", node->data);
        node = node->right_child;
    }
}
```

O(n)

Trace Operations of Inorder Traversal

Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

```
typedef struct node *tree_pointer;
typedef struct node {
    char data;
    tree_pointer left_child, right_child;
};

void push(int *top, tree_pointer element, tree_pointer stack[])
{
    if(*top == MAX_STACK_SIZE - 1)
        {printf("\Stack Full\n");
         return;}
    stack[>(*top)] = element; }

tree_pointer pop(tree_pointer stack[], int *top)
{ if(*top == -1)
    return NULL;
return(stack[(*top)--]); }
```

```
void iter_inorder(tree_pointer node)
{
    int top= -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node=node->left_child)
            push(&top, node,stack);/* add to stack */
        node= pop(stack,&top);/* delete from stack */
        if(!node) break;
        printf("%c\t", node->data);
        node = node->right_child;
    }
}
```

```
void iter_preorder(tree_pointer node)
{
    int top= -1; /* initialize stack */
    tree_pointer stack[MAX_STACK_SIZE];
    for (;;) {
        for (; node; node=node->left_child)
        {
            printf("%c\t", node->data);
            push(&top, node,stack);/* add to stack */
        }
        node= pop(stack,&top);/* delete from stack */
        if(!node) break;
        node = node->right_child;
    }
}
```

```
typedef struct node *tree_pointer;           Iterative postorder
struct node {
int data;
tree_pointer left_child, right_child;
};
typedef struct
{
tree_pointer address;
int flag;
}STACK;
void push(int *top,tree_pointer element, STACK stack[])
{
if(*top == MAX_STACK_SIZE-1)
{printf("\Stack Full\n"); return; }
stack[++(*top)].address = element;
}
tree_pointer pop(STACK stack[], int *top)
{
if(*top== -1)
return NULL;
return(stack[(*top)--].address);
```

```
void iterative_postorder(tree_pointer ptr)
{
STACK stack[MAX_STACK_SIZE];
int top=-1;
for (;;){
while(ptr){
    push(&top,ptr,stack);
    stack[top].flag=1;
    ptr=ptr->left_child;}
    while(stack[top].flag==2)  {
        ptr=pop(stack,&top);
        if(!ptr)return;
        printf("%d\t",ptr->data);  }
    ptr=stack[top].address;
    stack[top].flag=2;
    ptr=ptr->right_child;
}
}
```

Level Order Traversal

(using queue)

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
    int front = rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    if (!ptr) return; /* empty queue */
    addq(front, &rear, ptr);
    for (;;) {
        ptr = deleteq(&front, rear);
```

```

if (ptr) {
    printf("%d", ptr->data);
    if (ptr->left_child)
        addq(front, &rear,
              ptr->left_child);
    if (ptr->right_child)
        addq(front, &rear,
              ptr->right_child);
}
else break;
}

```

+ * E * D / C A B


```
#include<stdio.h>
#include<stdlib.h>
#define MAX_QUEUE_SIZE 20
typedef struct node *tree_pointer;
struct node {
    int data;
    tree_pointer left_child, right_child;
};

void addq(int *rear,tree_pointer q[],tree_pointer ptr)
{
    q[(*rear)++]=ptr;
}

tree_pointer deleteq(int *fron,int rear,tree_pointer q[])
{
    if(*fron==rear) return NULL;
    return(q[(*fron)++]);
}
```

```
void level_order(tree_pointer ptr)
{
    int front=0;
    int rear = 0;
    tree_pointer queue[MAX_QUEUE_SIZE];
    //if (!ptr) return;
    addq(&rear,queue,ptr);
    for (;;) {
        ptr = deleteq(&front,rear,queue);
        if(ptr) {
            printf("%d\t", ptr->data);
            if (ptr->left_child)
                addq(&rear,queue,ptr->left_child);
            if (ptr->right_child)
                addq(&rear,queue,ptr->right_child);
        }
        else break;
    }
}
```

```
tree_pointer insert()
{
    tree_pointer temp;
    int ele;
    printf("Enter The element\n");
    scanf("%d",&ele);
    if(ele== -1) return NULL;
    temp=(tree_pointer )malloc(sizeof(struct node));
    temp->data=ele;
    printf("Enter left child of %d:\n",ele);
    temp->left_child=insert();
    printf("Enter right child of %d:\n",ele);
    temp->right_child=insert();
}
```

```
void main()
{
int op; tree_pointer root=NULL;
do
{
    printf("\n 1.Insertion"); printf("\n 2.Levelorder");
    printf("\n 3.Quit");    printf("\n Enter your choice\n");
    scanf("%d",&op);
    switch (op)
    {
        case 1: root=insert();           break;
        case 2: if(root==NULL)
                  printf("Tree is empty\n");   else
                  printf("\n The elements are\n");
                  level_order(root); } break;
        default: exit(0);
    } }while(op<3); }
```


BST

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node *tree_pointer;
typedef struct node {
    int data;
    tree_pointer left_child, right_child;
```

```
void BST(tree_pointer *root,int item)
{
    tree_pointer temp,current,previous;
    temp=(tree_pointer )malloc(sizeof(struct node));
    temp->data=item;
    temp->left_child=NULL;
    temp->right_child=NULL;
    if(*root==NULL){ *root=temp; return; }
    previous=NULL;
    current=*root;
    while(current!=NULL)
    {
        previous=current;
        if(item<current->data)
            current=current->left_child;
        else
            current=current->right_child;
    }
}
```

```
if(item<previous->data)
    previous->left_child=temp;
else
    previous->right_child=temp;
```

```
void delete_BST(tree_pointer *root, int item)
{
    tree_pointer del_node, parent, child_del_node, suc;
/* if(*root==NULL)
{
    printf("Tree is empty\n");
    return;
} */
parent=NULL;
del_node=*root;
while(del_node!=NULL && item!=del_node->data)
{
    parent=del_node;
    del_node=(item<del_node->data)?del_node-
>left_child:del_node->right_child;
}
if(del_node==NULL)
{
    printf("Element not found\n");
}
```

```
if(del_node->left_child==NULL)
    child_del_node=del_node->right_child;
else if(del_node->right_child==NULL)
    child_del_node=del_node->left_child;
else{ suc=del_node->right_child;
      while(suc->left_child!=NULL)
          suc=suc->left_child;
      suc->left_child=del_node->left_child;
      child_del_node=del_node->right_child;
}
if(parent==NULL)
{ *root=child_del_node;
return;
}
if(del_node==parent->left_child)
    parent->left_child=child_del_node;
else
    parent->right_child=child_del_node;
}
```

Copying Binary Trees

```
tree_poointer copy(tree_pointer original)
{
tree_pointer temp;
if (original) {
    temp=(tree_pointer) malloc(sizeof(node));
    if (IS_FULL(temp)) {
        fprintf(stderr, "the memory is full\n");
        exit(1);
    }
    temp->left_child=copy(original->left_child);
    temp->right_child=copy(original->right_child);
    temp->data=original->data;
    return temp;
}
return NULL;
}
```

postorder

Equality of Binary Trees

the same topology and data

```
int equal(tree_pointer first, tree_pointer second)
{
    /* function returns FALSE if the binary trees first
       and
       second are not equal, otherwise it returns TRUE */

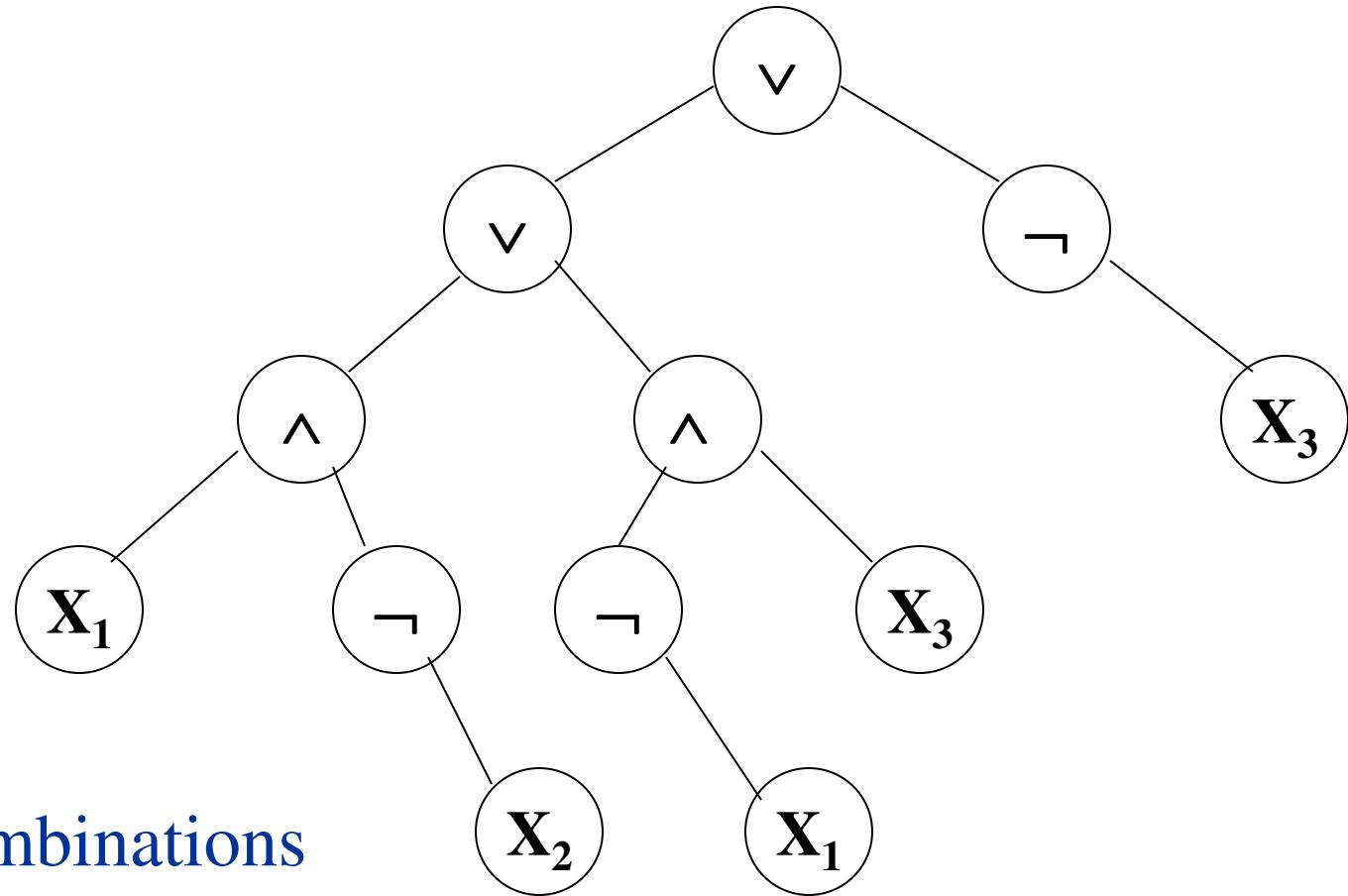
    return ((!first && !second) || (first && second &&
        (first->data == second->data) &&
        equal(first->left_child, second->left_child)
        &&
        equal(first->right_child, second-
        >right_child)))
}
```

Propositional Calculus Expression

- n A variable is an expression.
- n If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.
- n Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$).
- n Example: $x_1 \vee (x_2 \wedge \neg x_3)$
- n satisfiability problem: Is there an assignment to make an expression true?

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

(t,t,t)
 (t,t,f)
 (t,f,t)
 (t,f,f)
 (f,t,t)
 (f,t,f)
 (f,f,t)
 (f,f,f)



2^n possible combinations
for n variables

postorder traversal (postfix evaluation)

node structure

<i>left_child</i>	<i>data</i>	<i>value</i>	<i>right_child</i>
-------------------	-------------	--------------	--------------------

```
typedef enum { not, and, or, true, false } logical;  
typedef struct node *tree_pointer;  
typedef struct node {  
    tree_pointer list_child;  
    logical      data;  
    short int    value;  
    tree_pointer right_child;  
} ;
```

First version of satisfiability algorithm

```
for (all  $2^n$  possible combinations) {  
    generate the next combination;  
    replace the variables by their values;  
    evaluate root by traversing it in postorder;  
    if (root->value) {  
        printf(<combination>);  
        return;  
    }  
}  
printf("No satisfiable combination \n");
```

Post-order-eval function

```
void post_order_eval(tree_pointer node)
{
/* modified post order traversal to evaluate a propositional
calculus tree */
if (node) {
    post_order_eval(node->left_child);
    post_order_eval(node->right_child);
    switch(node->data) {
        case not: node->value =
                    !node->right_child->value;
        break;
    }
}
```

```
case and:    node->value =
            node->right_child->value &&
            node->left_child->value;
            break;

case or:     node->value =
            node->right_child->value ||
            node->left_child->value;
            break;

case true:   node->value = TRUE;
            break;

case false:  node->value = FALSE;
}

}

}
```

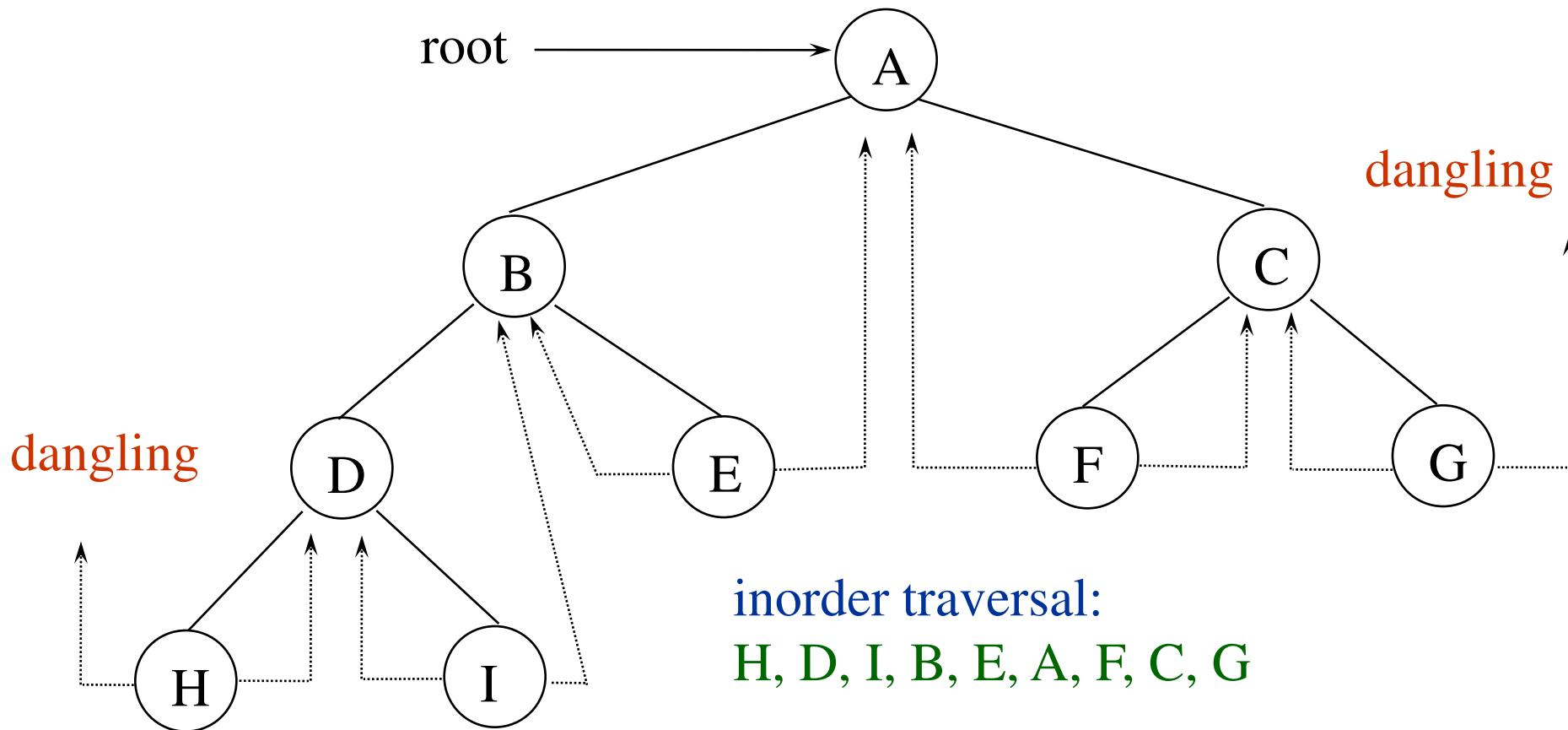
Threaded Binary Trees

- Two many null pointers in current representation of binary trees
 - n: number of nodes
 - number of non-null links: n-1
 - total links: 2n
 - null links: $2n - (n-1) = n+1$
- Replace these null pointers with some useful “threads”.

Threaded Binary Trees (*Continued*)

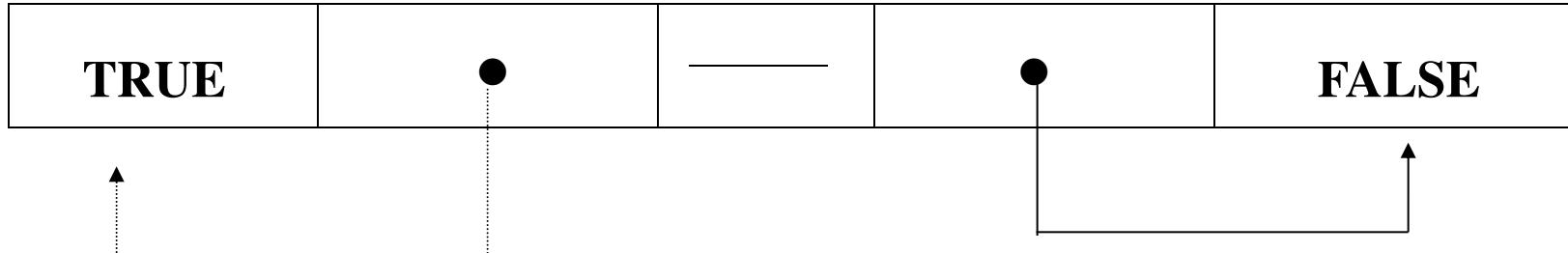
- If `ptr->left_child` is null,
 replace it with a pointer to the node that would be
 visited *before* `ptr` in an *inorder traversal*
- If `ptr->right_child` is null,
 replace it with a pointer to the node that would be
 visited *after* `ptr` in an *inorder traversal*

A Threaded Binary Tree



Data Structures for Threaded BT

left_thread left_child data right_child right_thread

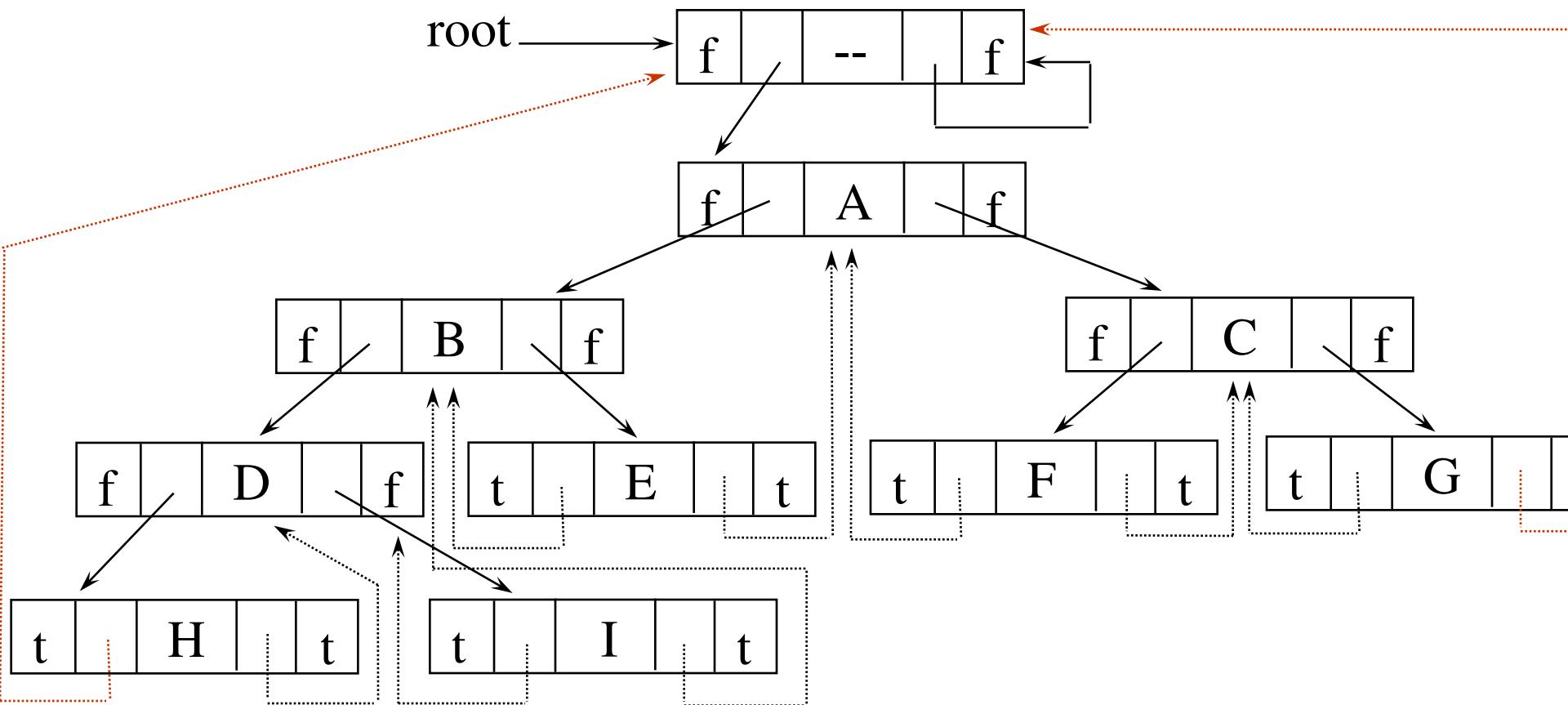


TRUE: thread

FALSE: child

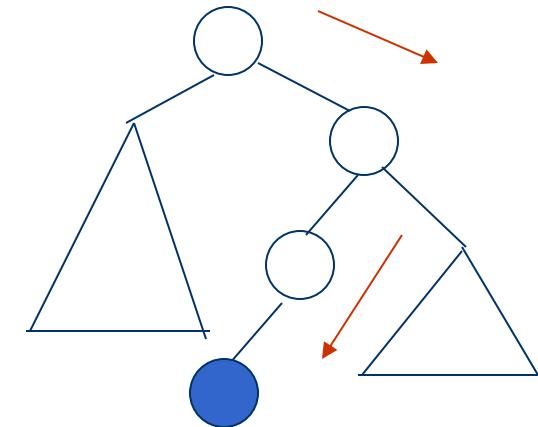
```
typedef struct threaded_tree  
*threaded_pointer;  
  
typedef struct threaded_tree {  
    short int left_thread;  
    threaded_pointer left_child;  
    char data;  
    threaded_pointer right_child;  
    short int right_thread; };
```

Memory Representation of A Threaded BT



Next Node in Threaded BT

```
threaded_pointer insucc(threaded_pointer  
tree)  
{  
    threaded_pointer temp;  
    temp = tree->right_child;  
    if (!tree->right_thread)  
        while (!temp->left_thread)  
            temp = temp->left_child;  
    return temp;  
}
```



Inorder Traversal of Threaded BT

```
void tinorder(threaded_pointer tree)
{
    /* traverse the threaded binary tree
       inorder */
    threaded_pointer temp = tree;
    for (;;) {
        temp = insucc(temp);
        if (temp==tree) break;
        printf("%3c", temp->data);
    }
}
```

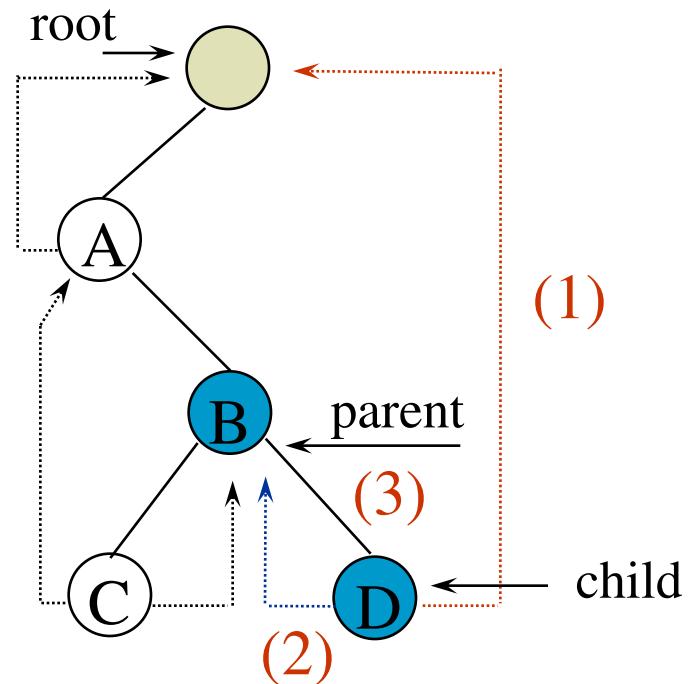
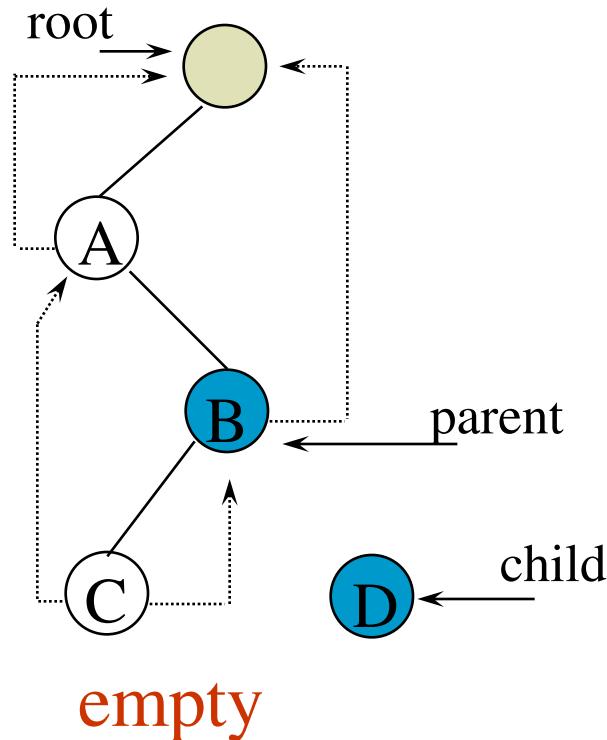
O(n)

Inserting Nodes into Threaded BTs

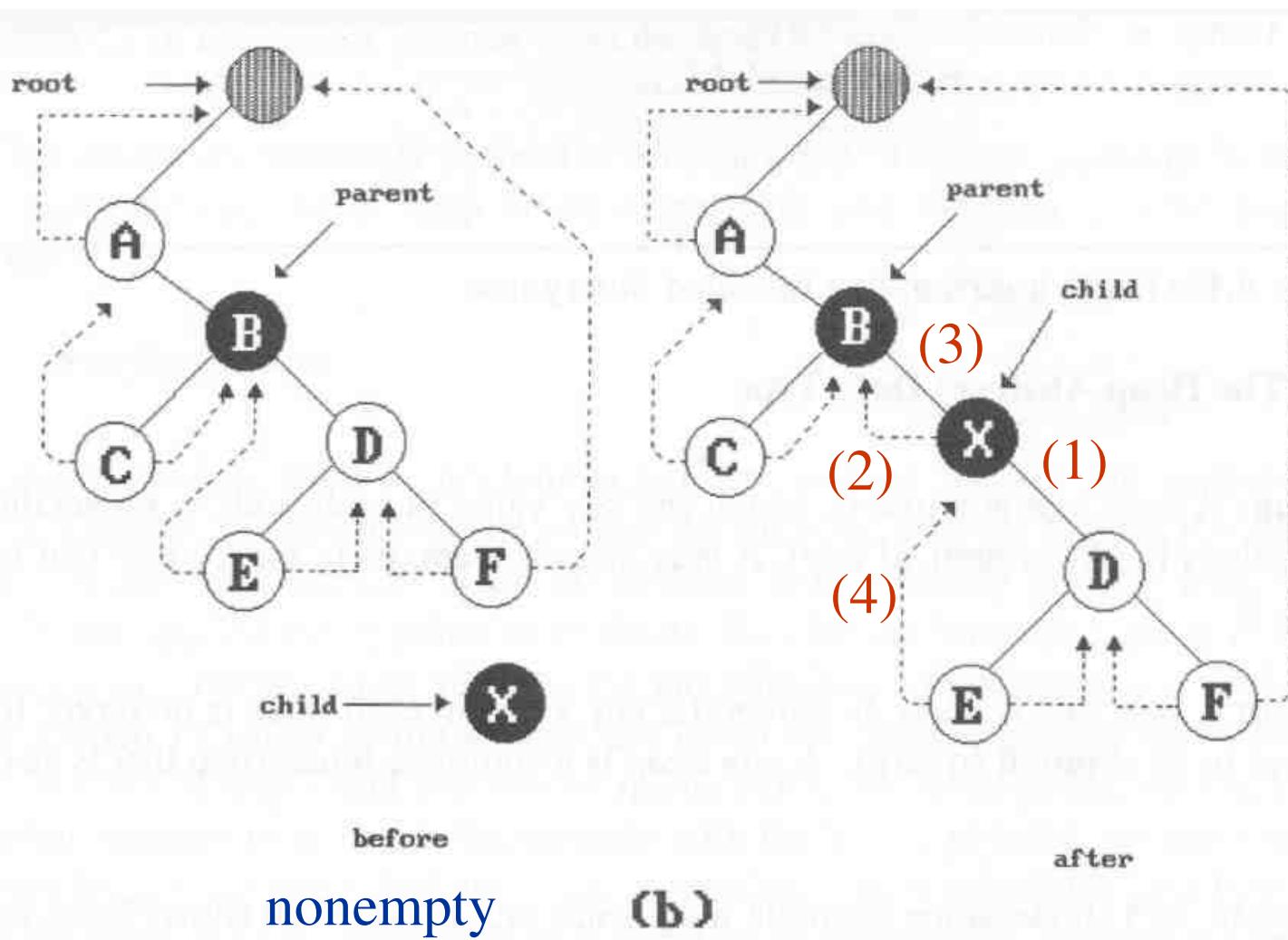
- n Insert child as the right child of node parent
 - change parent->right_thread to FALSE
 - set child->left_thread and child->right_thread to TRUE
 - set child->left_child to point to parent
 - set child->right_child to parent->right_child
 - change parent->right_child to point to child

Examples

Insert a node D as a right child of B.



*Figure 5.24: Insertion of child as a right child of parent in a threaded binary tree (p.217)



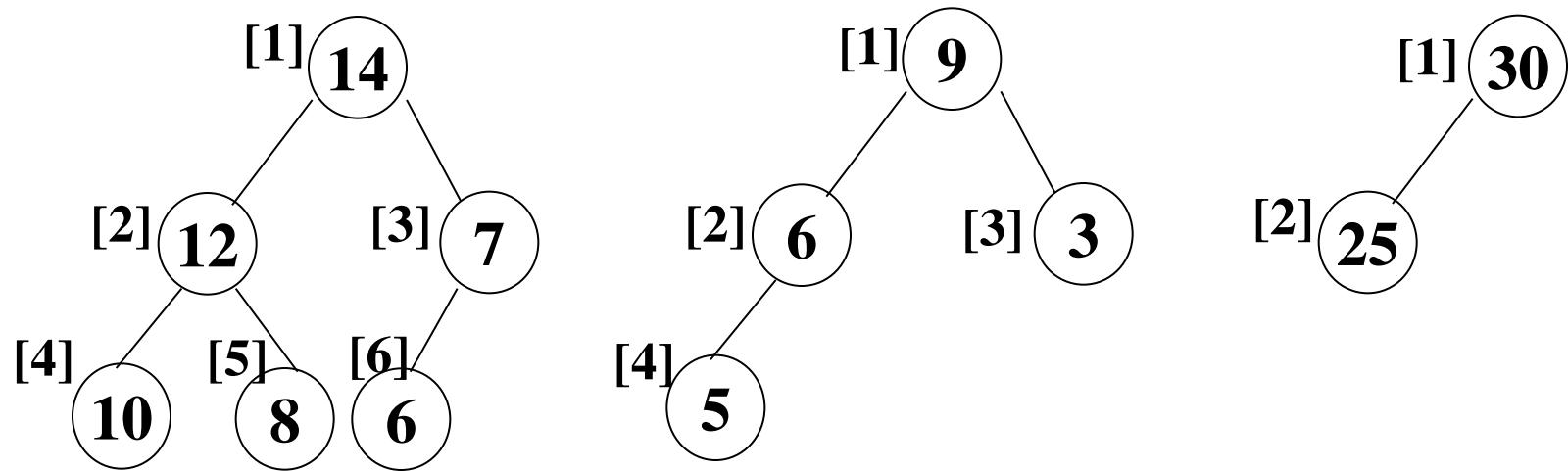
Right Insertion in Threaded BTs

```
void insert_right(threaded_pointer parent,
                  threaded_pointer child)
{
    threaded_pointer temp;
(1) child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
(2) child->left_child = parent;  case (a)
    child->left_thread = TRUE;
(3) parent->right_child = child;
    parent->right_thread = FALSE;
    if (!child->right_thread) { case (b)
(4)     temp = insucc(child);
        temp->left_child = child;
    }
}
```

Heap

- n A *max tree* is a tree in which the key value in each node is **no smaller than** the key values in its children. A *max heap* is a **complete binary tree** that is also a max tree.
- n A *min tree* is a tree in which the key value in each node is **no larger than** the key values in its children. A *min heap* is a **complete binary tree** that is also a min tree.
- n Operations on heaps
 - creation of an empty heap
 - insertion of a new element into the heap;
 - deletion of the largest element from the heap^{CHAPTER 5}⁸⁸

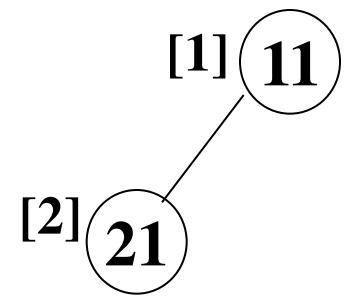
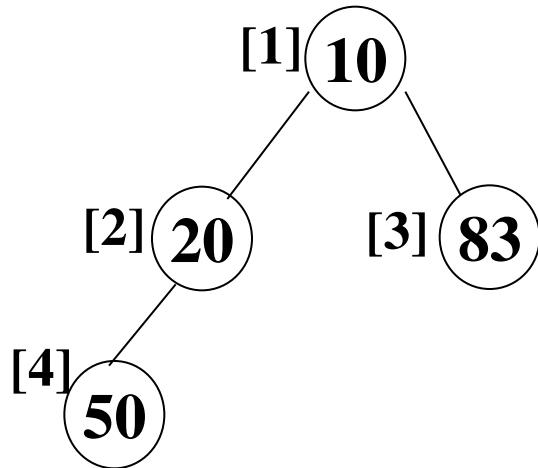
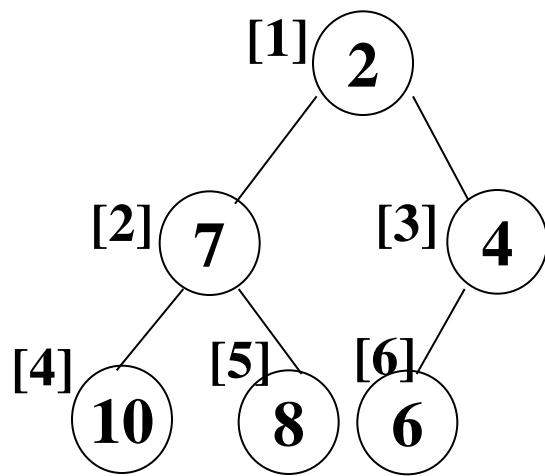
***Figure 5.25:** Sample max heaps (p.219)



Property:

The root of max heap (min heap) contains the largest (smallest).

***Figure 5.26:** Sample min heaps (p.220)



structure MaxHeap ADT for Max Heap

objects: a complete binary tree of $n > 0$ elements organized so that the value in each node is at least as large as those in its children
functions:

for all *heap* belong to *MaxHeap*, *item* belong to *Element*, *n*, *max_size* belong to integer

MaxHeap Create(*max_size*)::= create an empty heap that can hold a maximum of *max_size* elements

Boolean HeapFull(*heap*, *n*)::= if (*n*==*max_size*) return TRUE
else return FALSE

MaxHeap Insert(*heap*, *item*, *n*)::= if (!HeapFull(*heap*,*n*)) insert item into heap and return the resulting heap
else return error

Boolean HeapEmpty(*heap*, *n*)::= if (*n*>0) return FALSE
else return TRUE

Element Delete(*heap*,*n*)::= if (!HeapEmpty(*heap*,*n*)) return one instance of the **largest** element in the heap
and remove it from the heap

Application: priority queue

- „ machine service
 - amount of time (min heap)
 - amount of payment (max heap)
- „ factory
 - time tag

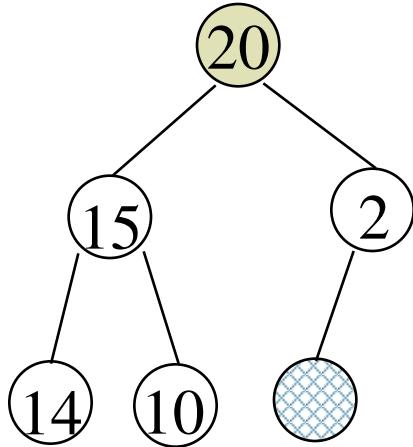
Data Structures

- n unordered linked list
- n unordered array
- n sorted linked list
- n sorted array
- n heap

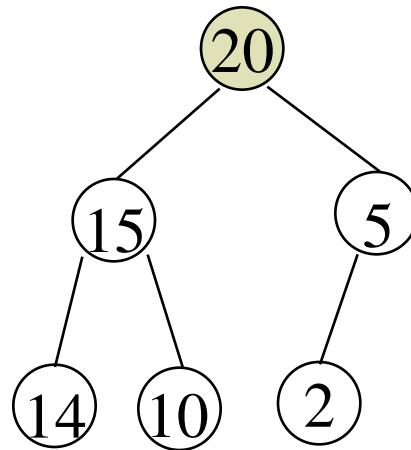
***Figure 5.27:** Priority queue representations (p.221)

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

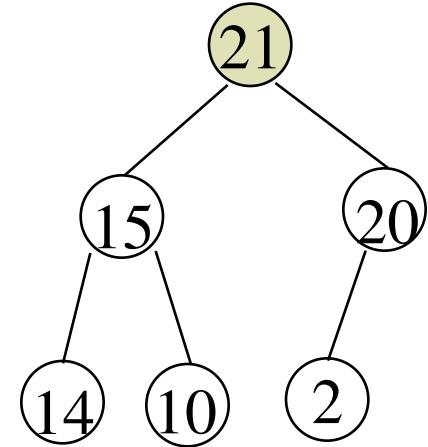
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap

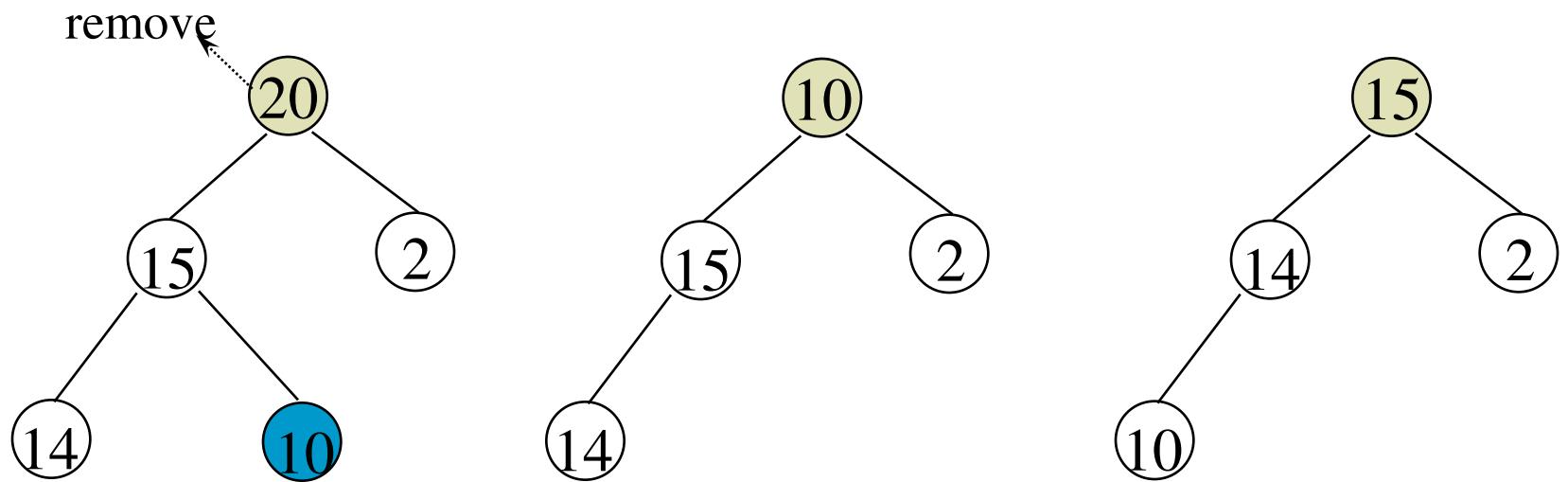


insert 21 into heap

Insertion into a Max Heap

```
void insert_max_heap(element item, int *n)
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "the heap is full.\n");
        exit(1);
    }
    i = ++(*n);
    while ((i!=1) && (item.key>heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;           $2^k - 1 = n \implies k = \lceil \log_2(n+1) \rceil$ 
    }
    heap[i]= item;  $O(\log_2 n)$ 
}
```

Example of Deletion from Max Heap



Deletion from a Max Heap

```
element delete_max_heap(int *n)
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(1);
    }
    /* save value of the element with the
       highest key */
    item = heap[1];
    /* use last element in heap to adjust heap
    temp = heap[(*n)--];
    parent = 1;
    child = 2;
```

```

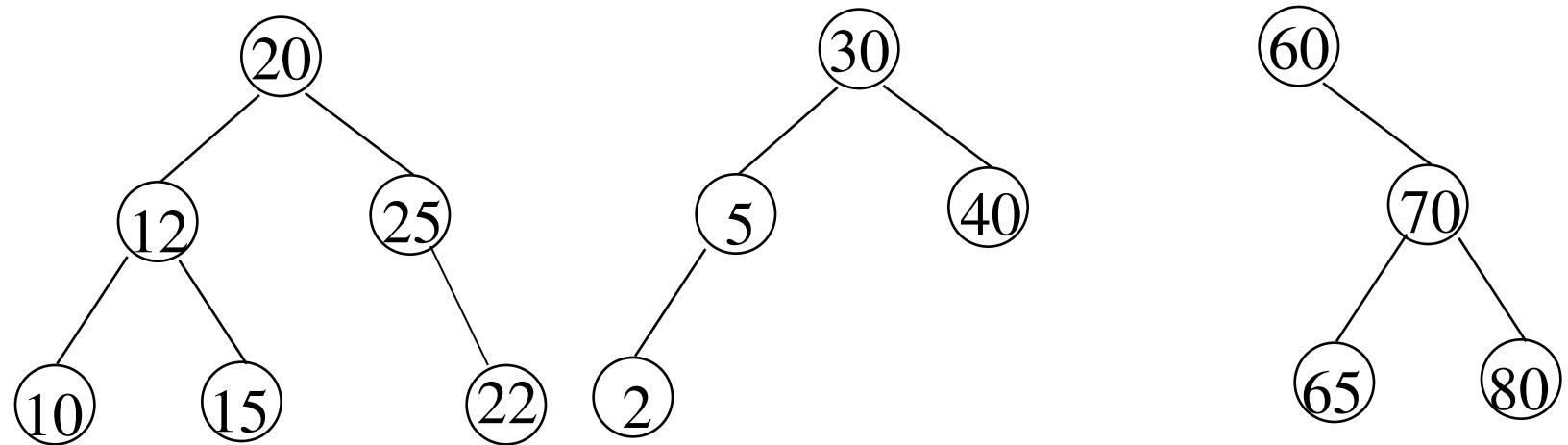
while (child <= *n) {
    /* find the larger child of the current
       parent */
    if ((child < *n) &&
        (heap[child].key < heap[child+1].key))
        child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    child *= 2;
}
heap[parent] = temp;
return item;
}

```

Binary Search Tree

- n Heap
 - a min (max) element is deleted. $O(\log_2 n)$
 - deletion of an arbitrary element $O(n)$
 - search for an arbitrary element $O(n)$
- n Binary search tree
 - Every element has a unique key.
 - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
 - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

```
tree_pointer search(tree_pointer root,
                     int key)
{
    /* return a pointer to the node that
       contains key. If there is no such
       node, return NULL */

    if (!root) return NULL;
    if (key == root->data) return root;
    if (key < root->data)
        return search(root->left_child,
                      key);
    return search(root->right_child, key);
}
```

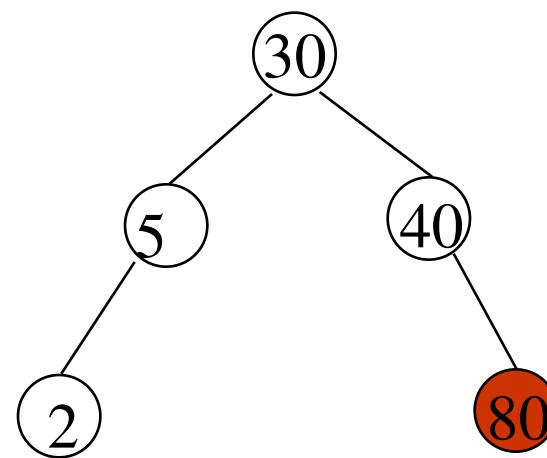
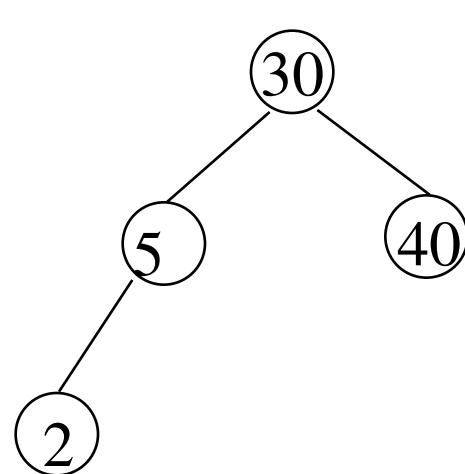
Another Searching Algorithm

```
tree_pointer search2(tree_pointer tree,
                     int key)

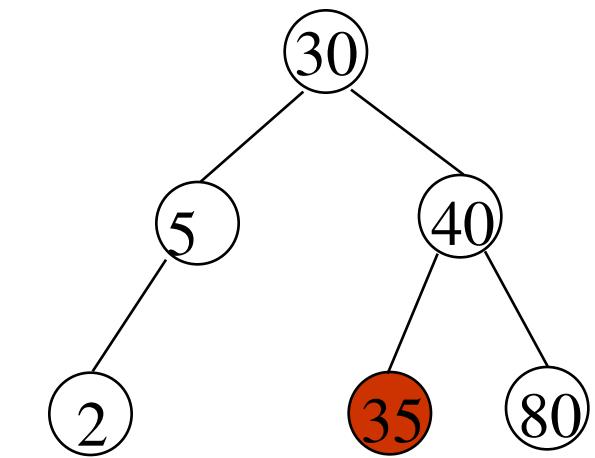
{
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else tree = tree->right_child;
    }
    return NULL;
}
```

$O(h)$

Insert Node in Binary Search Tree



Insert 80



Insert 35

BST Creation

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node *tree_pointer;
typedef struct node {
    int data;
    tree_pointer left_child, right_child;
};
```

```
void BST(tree_pointer *root,int item)
{
    tree_pointer temp,current,previous;
    temp=(tree_pointer )malloc(sizeof(struct node));
    temp->data=item;
    temp->left_child=NULL;
    temp->right_child=NULL;
    if(*root==NULL){ *root=temp; return; }
```

```
previous=NULL;  
current=*root;  
while(current!=NULL)  
{  
    previous=current;  
    if(item<current->data)  
        current=current->left_child;  
    else  
        current=current->right_child;  
}  
if(item<previous->data)  
    previous->left_child=temp;  
else  
    previous->right_child=temp;  
}
```

Deleting a Node from BST

Case 1: Empty LST and Empty RST

Case 2: Empty LST and Non Empty RST

Case 3: Non empty LST and Empty RST

Case 4: Non empty LST and Non empty RST


```
void delete_BST(tree_pointer *root, int item)
{
    tree_pointer del_node, parent, child_del_node, suc;
/* if(*root==NULL)
{
    printf("Tree is empty\n");
    return;
} */
parent=NULL;
del_node=*root;
while(del_node!=NULL && item!=del_node->data)
{
    parent=del_node;
    del_node=(item<del_node->data)?
    del_node->left_child:del_node->right_child;
}
```

```
if(del_node==NULL)
{
    printf("Element not found\n");
    return;
}
if(del_node->left_child==NULL)
    child_del_node=del_node->right_child;
else if(del_node->right_child==NULL)
    child_del_node=del_node->left_child;
else{ suc=del_node->right_child;
      while(suc->left_child!=NULL)
          suc=suc->left_child;
      suc->left_child=del_node->left_child;
      child_del_node=del_node->right_child;
}
i
```

```
f(parent==NULL)
{ *root=child_del_node;
return;
}
if(del_node==parent->left_child)
    parent->left_child=child_del_node;
else
    parent->right_child=child_del_node;
}
printf("\n Enter the element to insert\n");
scanf("%d",&n);
BST(&root,n);
break;
if(root==NULL)
    printf("Tree is empty\n");
else{
    printf("Enter the node to be deleted\n");
    scanf("%d",&n);
    delete_BST(&root,n); } CHAPTER 5
break;
```

```
struct node
{
    int key;
    struct node *left, *right;
};
```

```
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```

```
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
}
```

```
// if key is same as root's key, then This is the node
// to be deleted
else
{
    // node with only one child or no child
    if (root->left == NULL)
    {
        struct node *temp = root->right;
        free(root);
        return temp;
    }
    else if (root->right == NULL)
    {
        struct node *temp = root->left;
        free(root);
        return temp;
    }
}
```

```
// node with two children: Get the inorder successor (smallest
// in the right subtree)
struct node* temp = minValueNode(root->right);

// Copy the inorder successor's content to this node
root->key = temp->key;

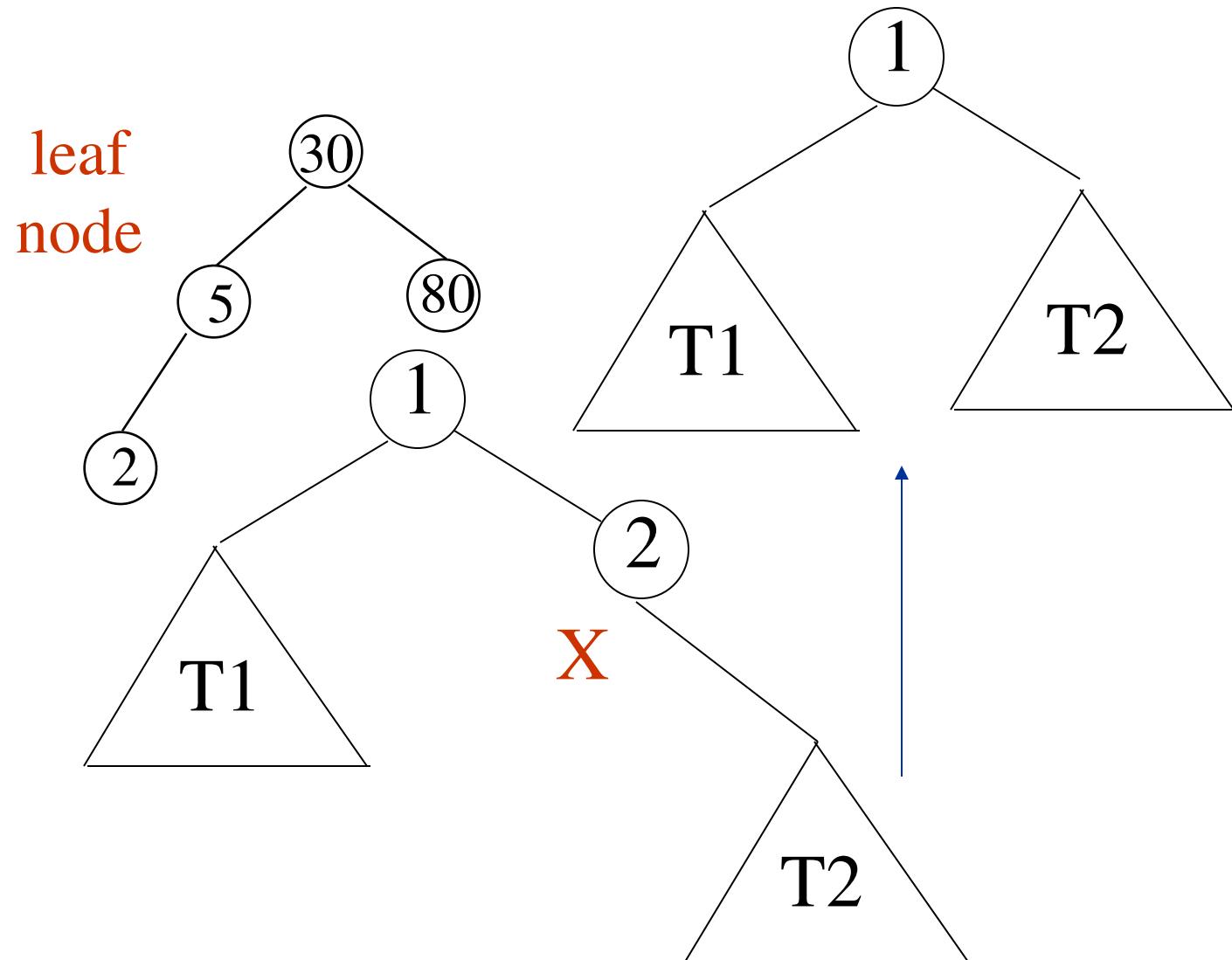
// Delete the inorder successor
root->right = deleteNode(root->right, temp->key);
}

return root;
}
```

Insertion into A Binary Search Tree

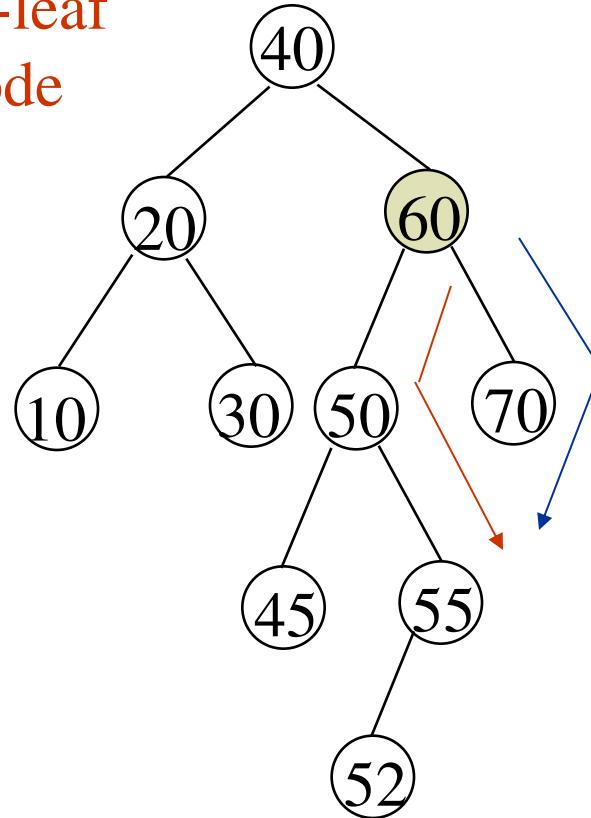
```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
     temp = modified_search(*node, num);
if (temp || !(*node)) {
    ptr = (tree_pointer) malloc(sizeof(node));
    if (IS_FULL(ptr)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    ptr->data = num;
    ptr->left_child = ptr->right_child = NULL;
    if (*node)
        if (num<temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
    else *node = ptr;
}
}
```

Deletion for A Binary Search Tree

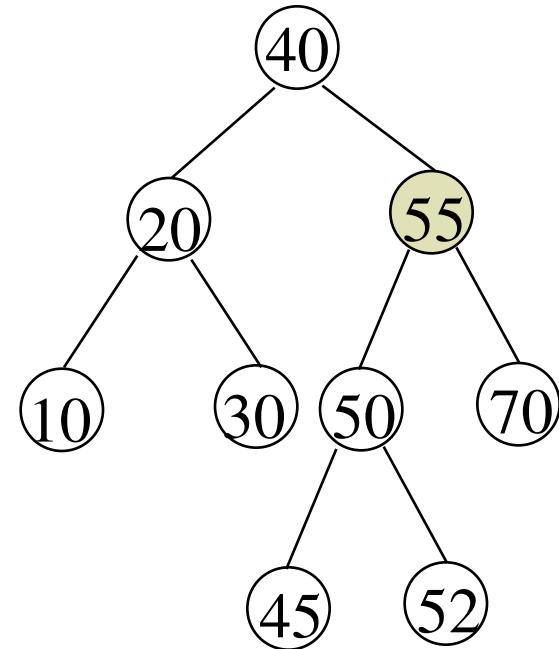


Deletion for A Binary Search Tree

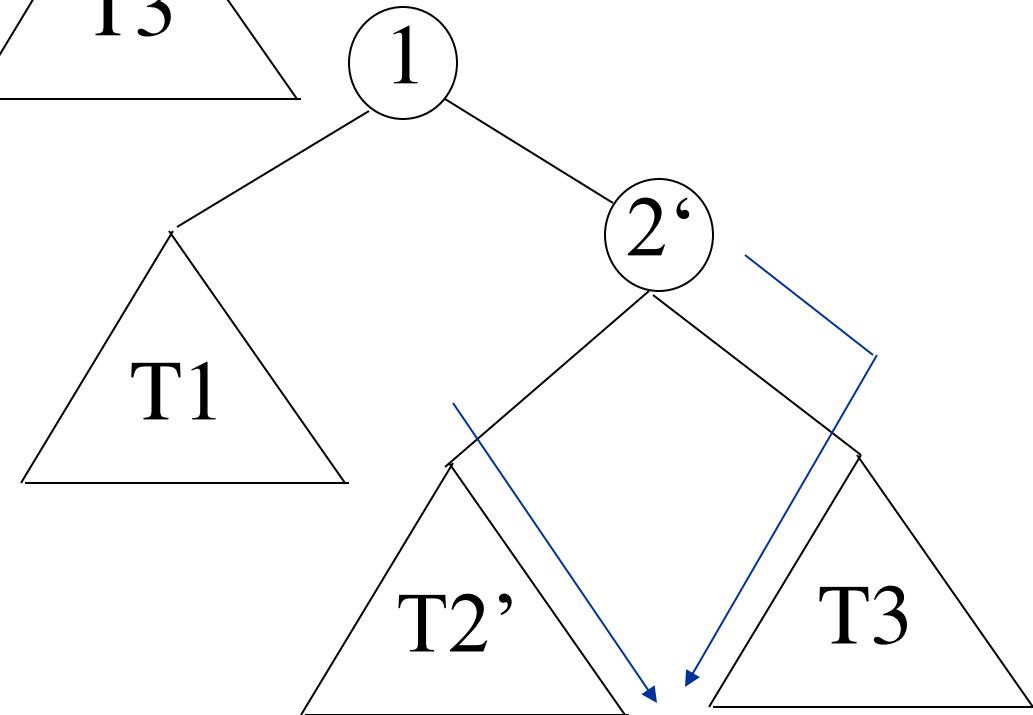
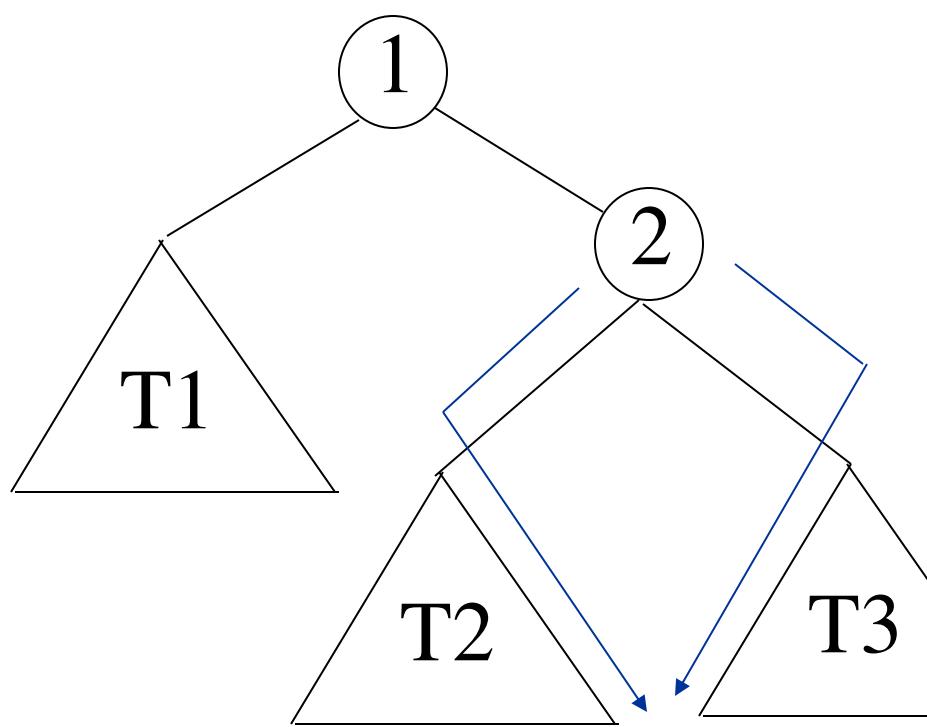
non-leaf
node



Before deleting 60



After deleting 60



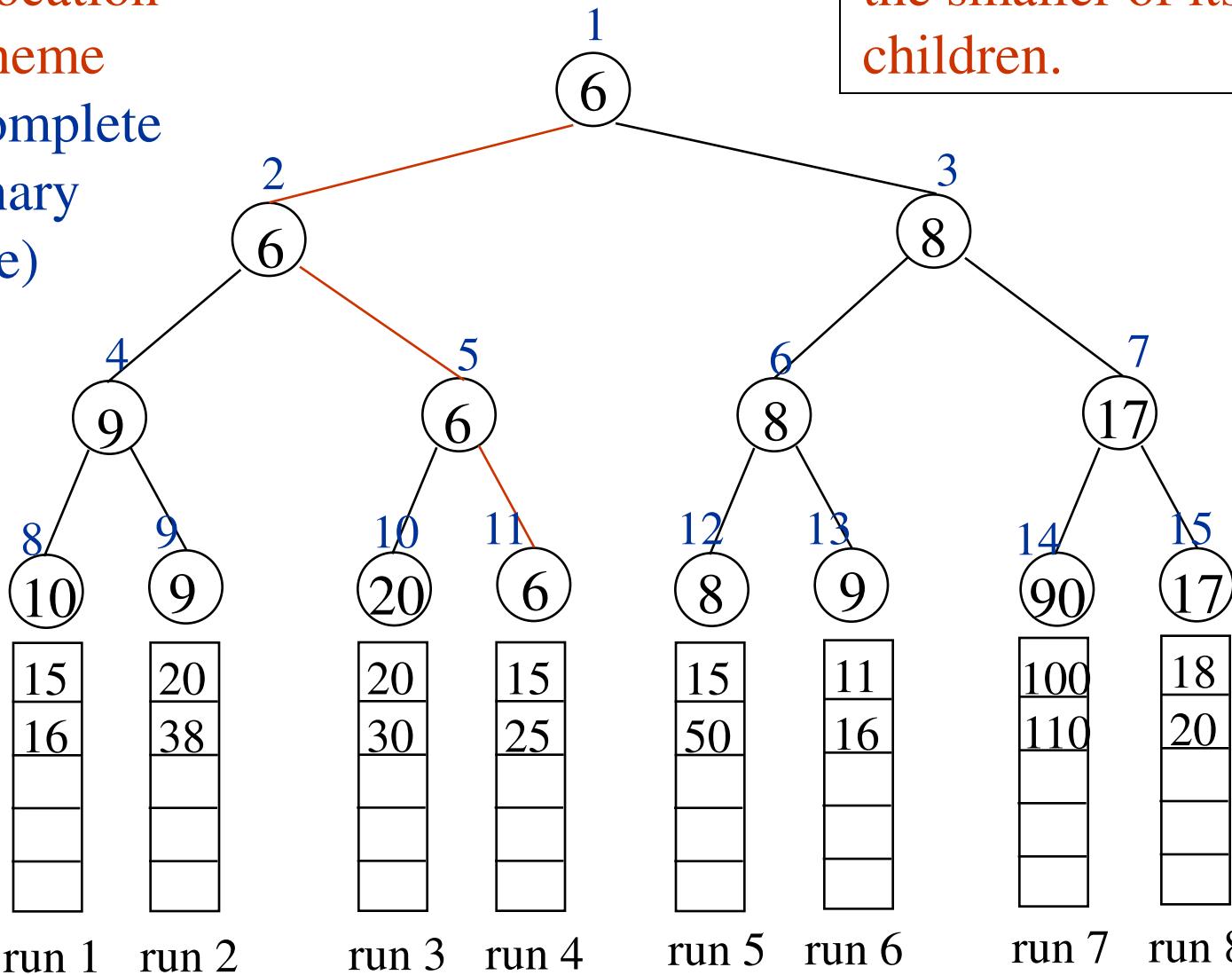
Selection Trees

- (1) winner tree
- (2) loser tree

sequential
allocation
scheme
(complete
binary
tree)

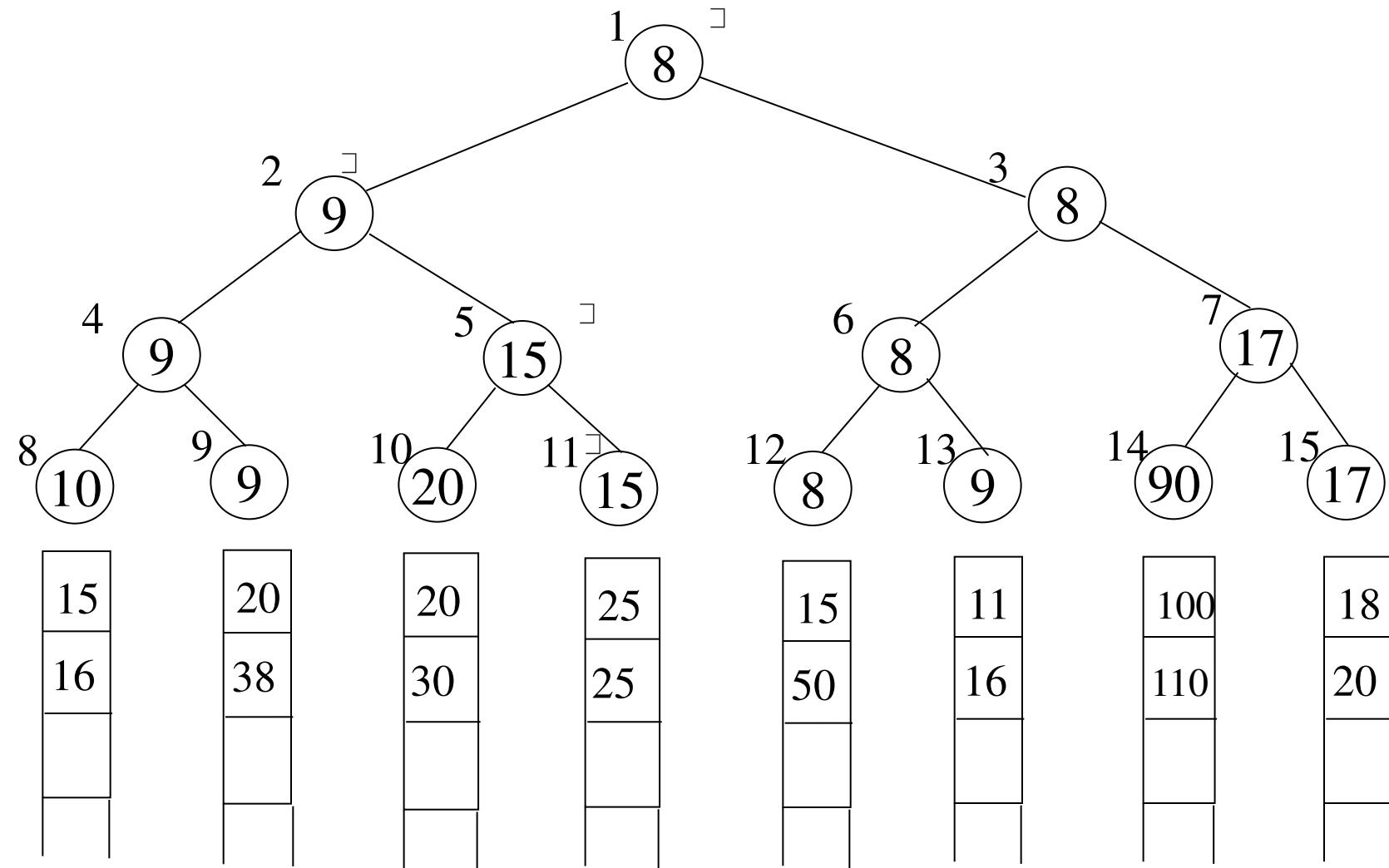
winner tree

ordered sequence



Each node represents
the smaller of its two
children.

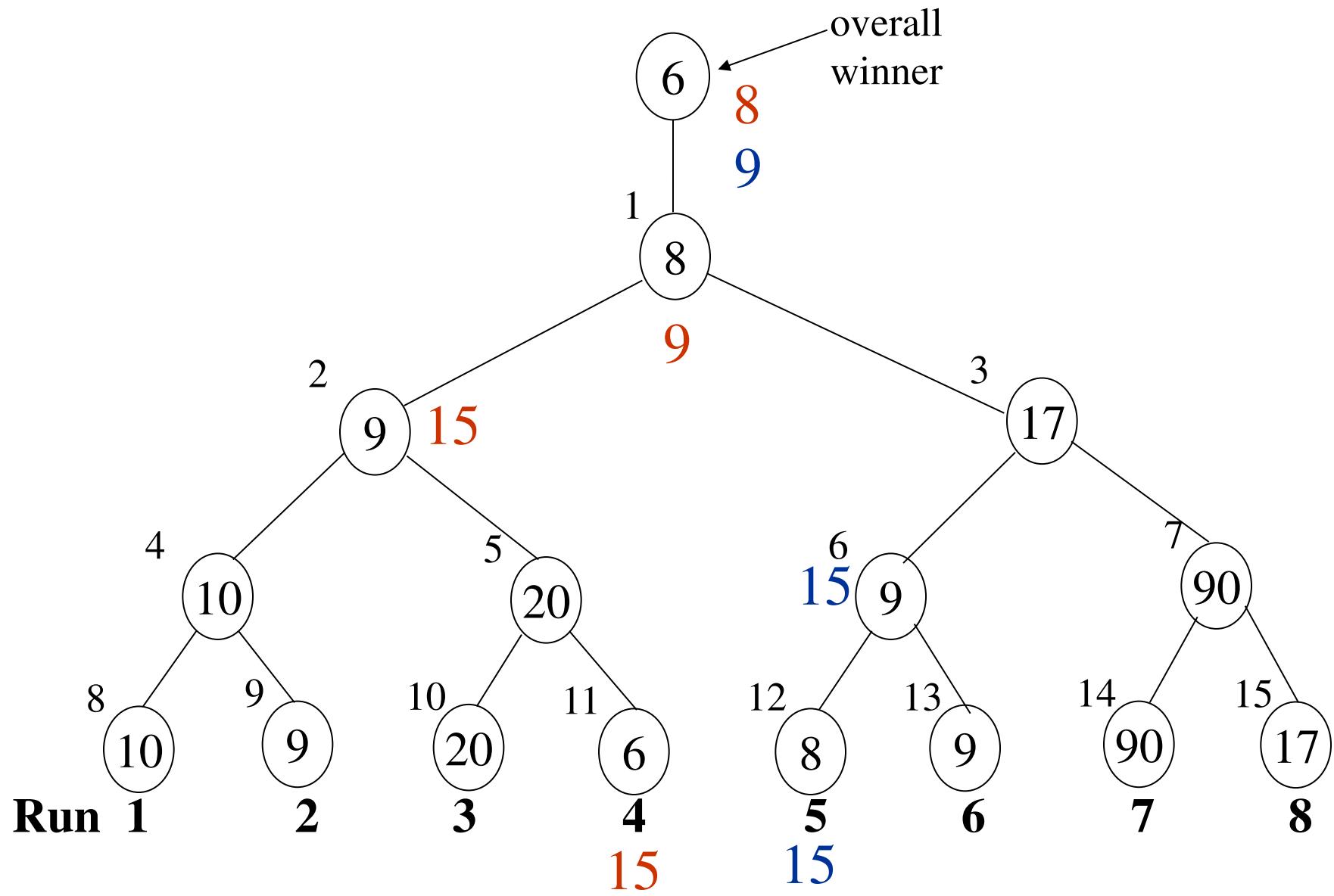
***Figure 5.35:** Selection tree of Figure 5.34 after one record has been output and the tree restructured(nodes that were changed are ticked)



Analysis

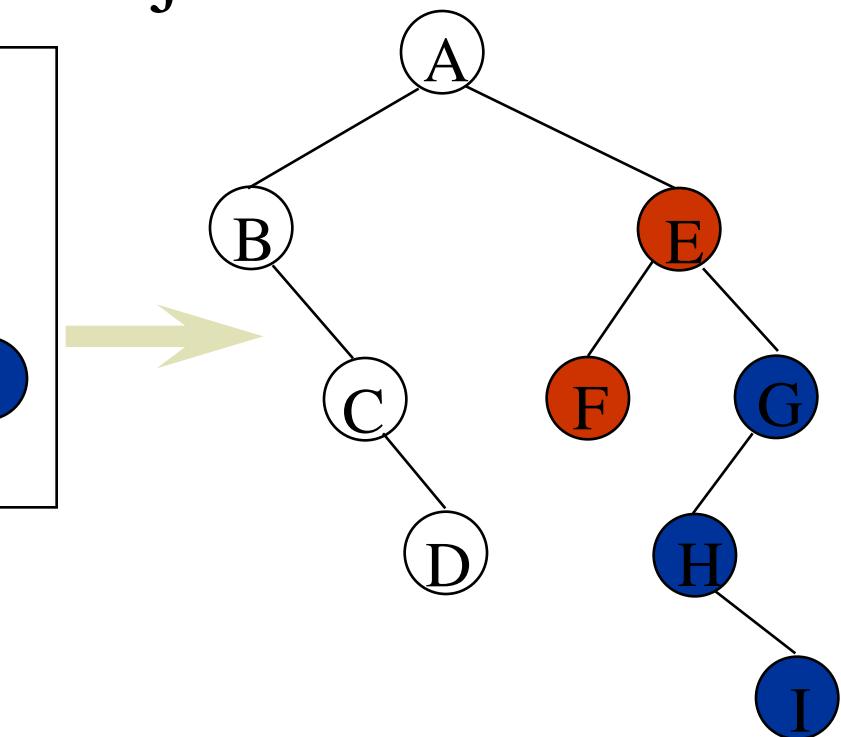
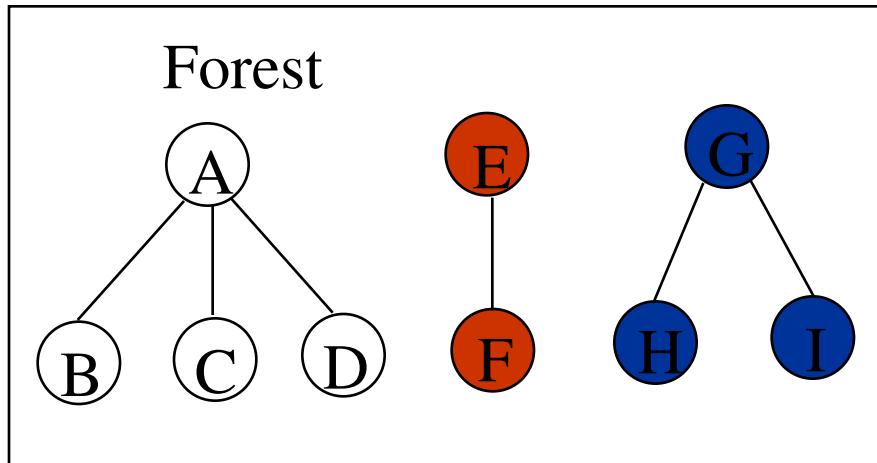
- n K: # of runs
- n n: # of records
- n setup time: $O(K)$ (K-1)
- n restructure time: $O(\log_2 K)$ $\lceil \log_2(K+1) \rceil$
- n merge time: $O(n \log_2 K)$
- n slight modification: **tree of loser**
 - consider the parent node only (vs. sibling nodes)

***Figure 5.36:** Tree of losers corresponding to Figure 5.34 (p.235)



Forest

- n A forest is a set of $n \geq 0$ disjoint trees

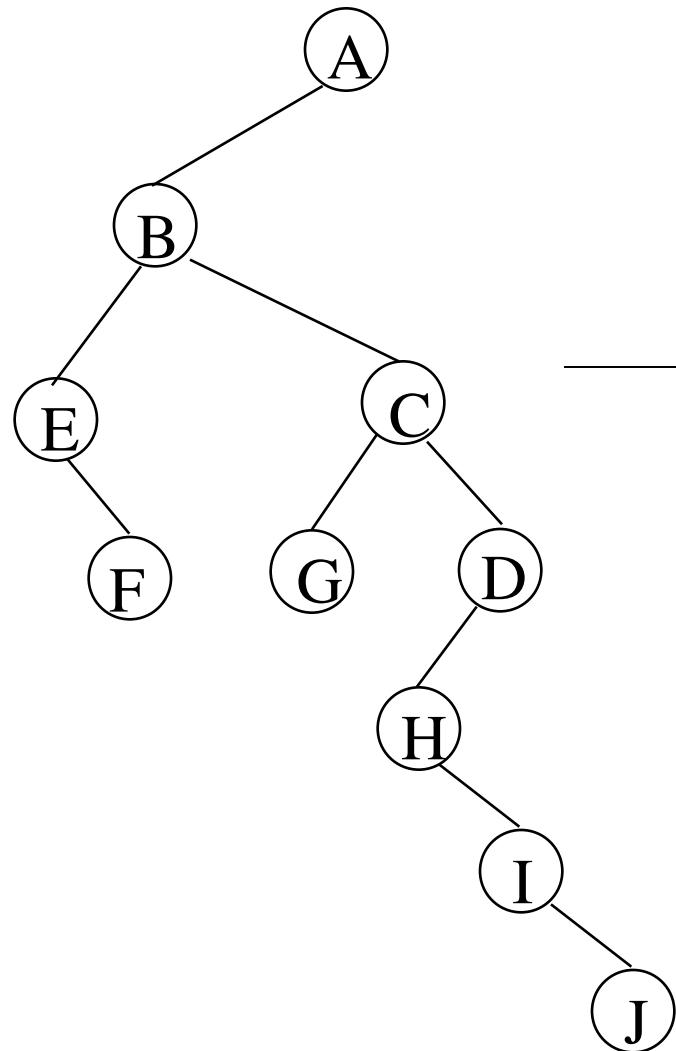


Transform a forest into a binary tree

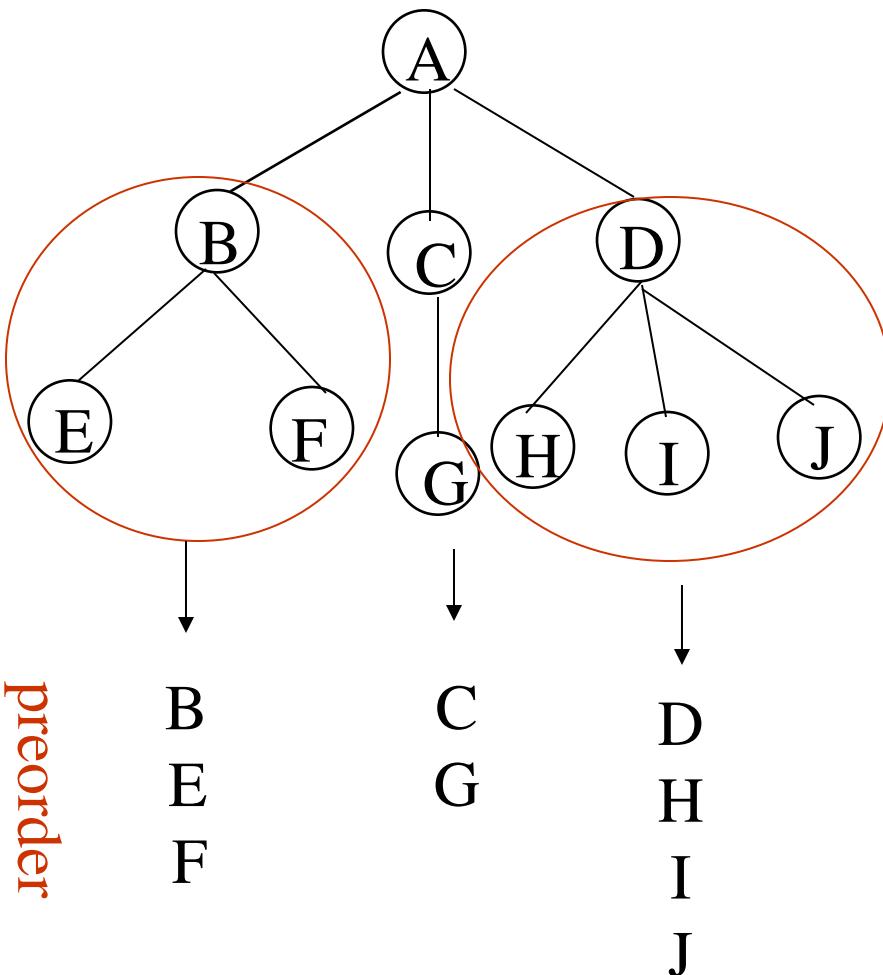
- n T_1, T_2, \dots, T_n : a forest of trees
 $B(T_1, T_2, \dots, T_n)$: a binary tree corresponding to this forest
- n algorithm
 - (1) empty, if $n = 0$
 - (2) has root equal to $\text{root}(T_1)$
 - has left subtree equal to $B(T_{11}, T_{12}, \dots, T_{1m})$
 - has right subtree equal to $B(T_2, T_3, \dots, T_n)$

Forest Traversals

- n Preorder
 - If F is empty, then return
 - Visit the root of the first tree of F
 - Traverse the subtrees of the first tree in tree preorder
 - Traverse the remaining trees of F in preorder
- n Inorder
 - If F is empty, then return
 - Traverse the subtrees of the first tree in tree inorder
 - Visit the root of the first tree
 - Traverse the remaining trees of F in inorder

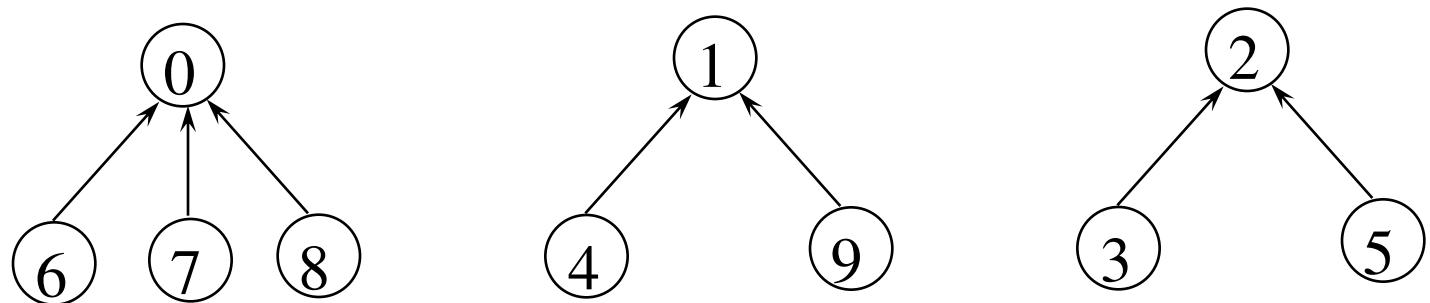


inorder: EFBGCHIJDA
 preorder: ABEFCGDHIJ



Set Representation

- n $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, $S_3 = \{2, 3, 5\}$



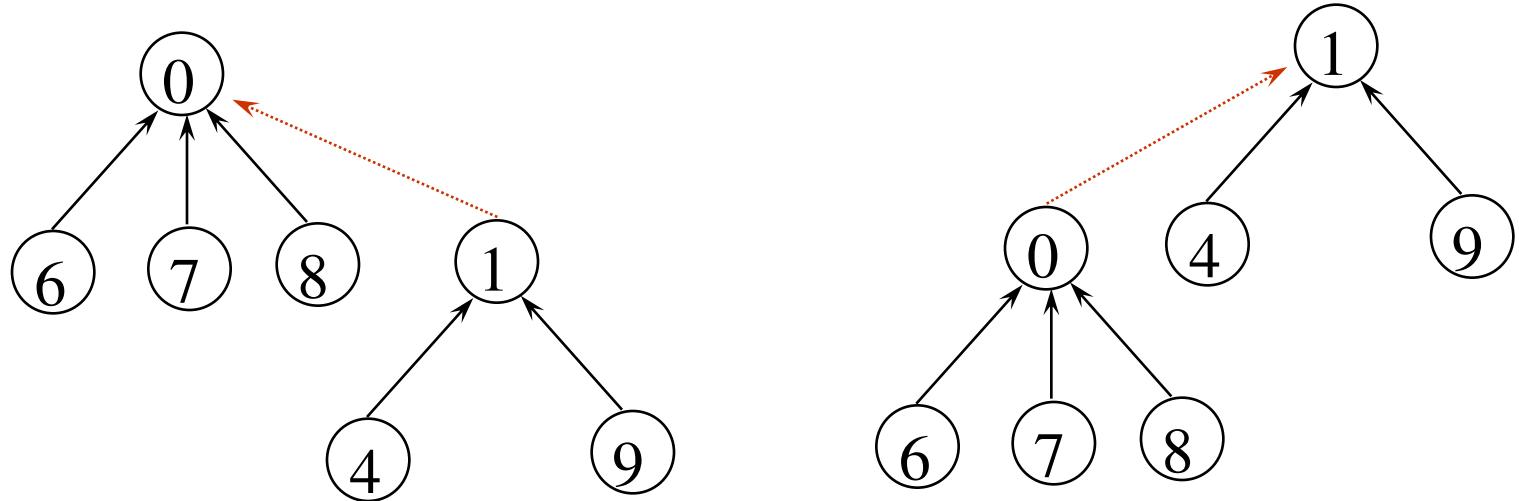
$$S_i \cap S_j = \emptyset$$

- n Two operations considered here
 - *Disjoint set union* $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$
 - *Find(i)*: Find the set containing the element i .

$$3 \in S_3, 8 \in S_1$$

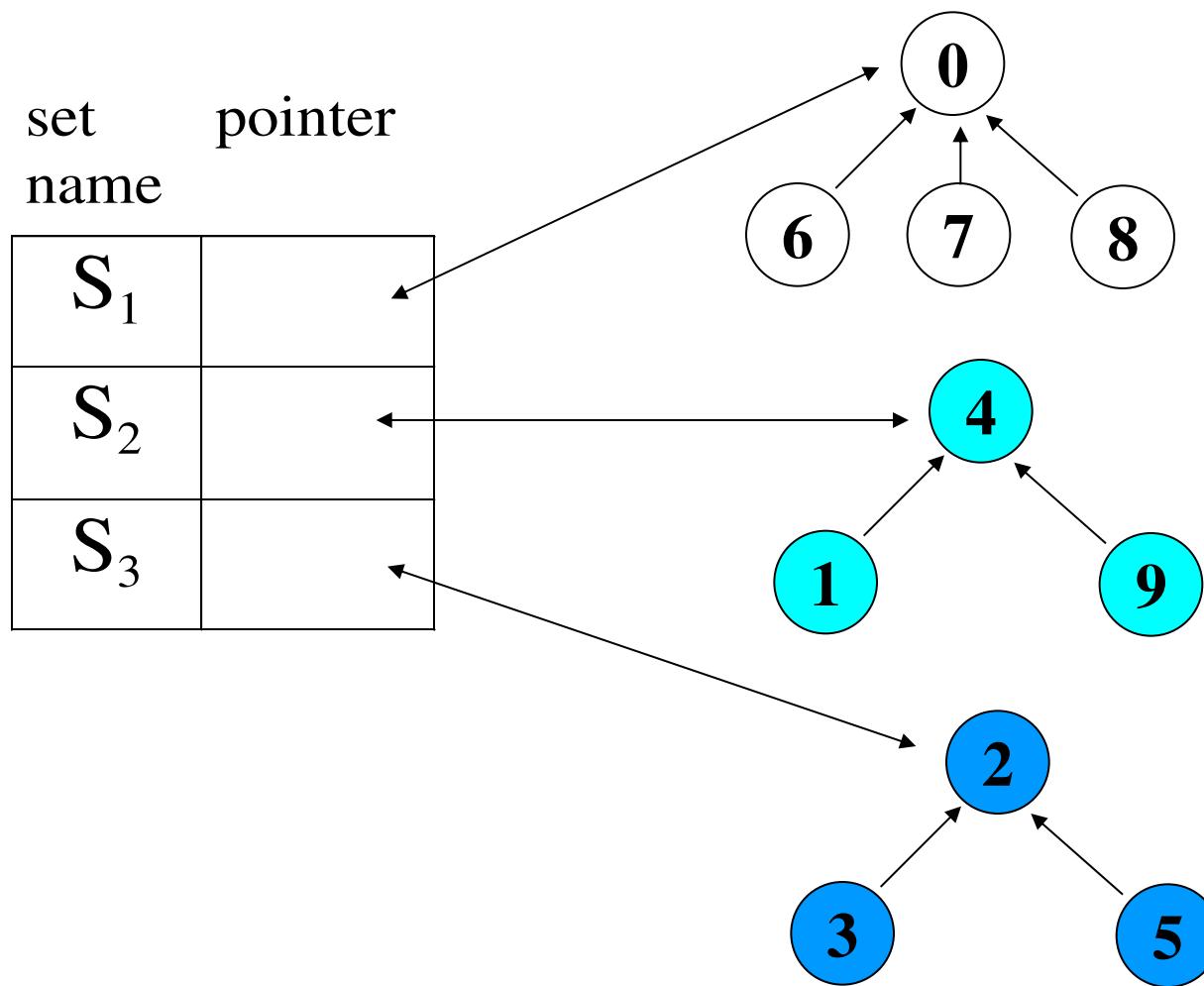
Disjoint Set Union

Make one of trees a subtree of the other



Possible representation for $S_1 \cup S_2$

***Figure 5.41: Data Representation of S_1 , S_2 and S_3** (p.240)



Array Representation for Set

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

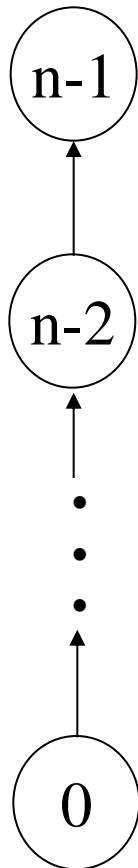
```
int find1(int i)
{
    for (; parent[i]>=0; i=parent[i])
        return i;
}

void union1(int i, int j)
{
    parent[i]= j;
}
```

*Figure 5.43:Degenerate tree (p.242)

union operation
 $O(n) \quad n-1$

find operation
 $O(n^2) \quad \sum_{i=2}^n i$



union(0,1), find(0)
union(1,2), find(0)
.
.
.
union(n-2,n-1),find(0)

degenerate tree

*Figure 5.44: Trees obtained using the weighting rule(p.243)

weighting rule for union(i,j): if # of nodes in $i <$ # in j then j the parent of i

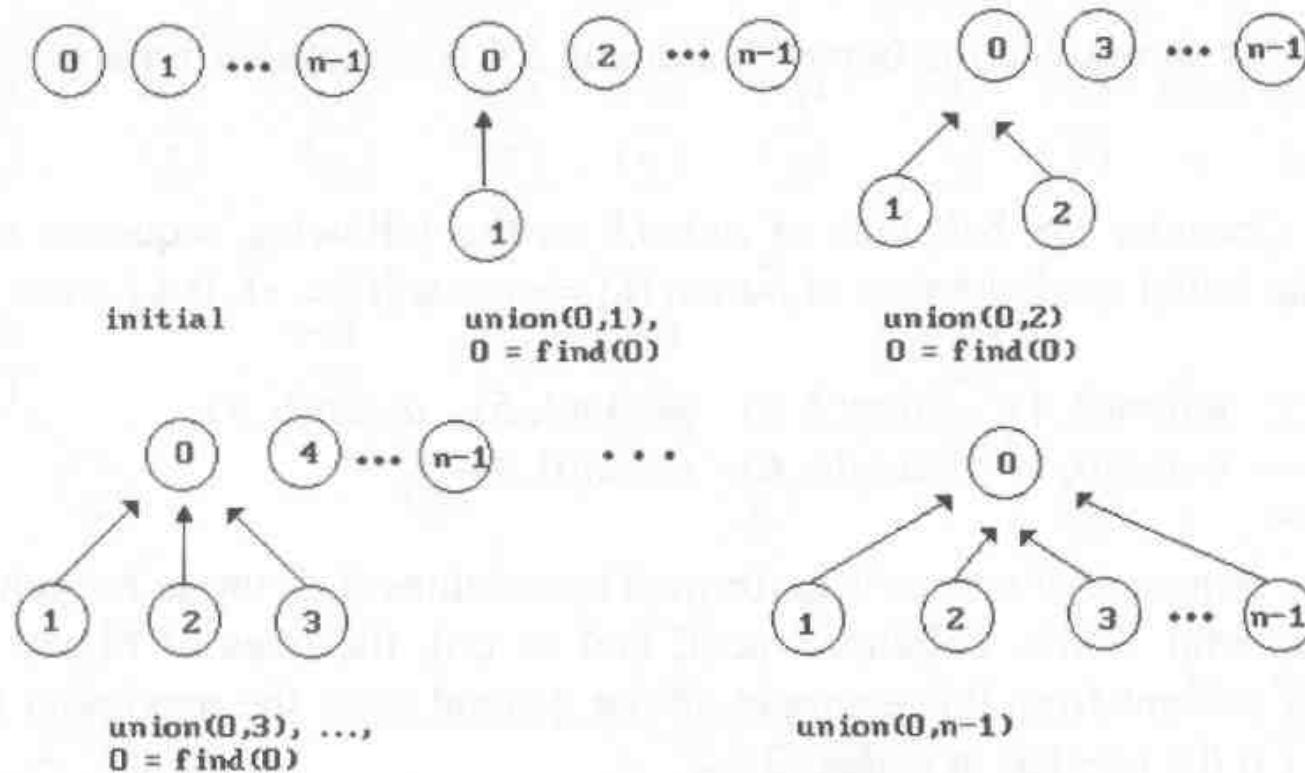


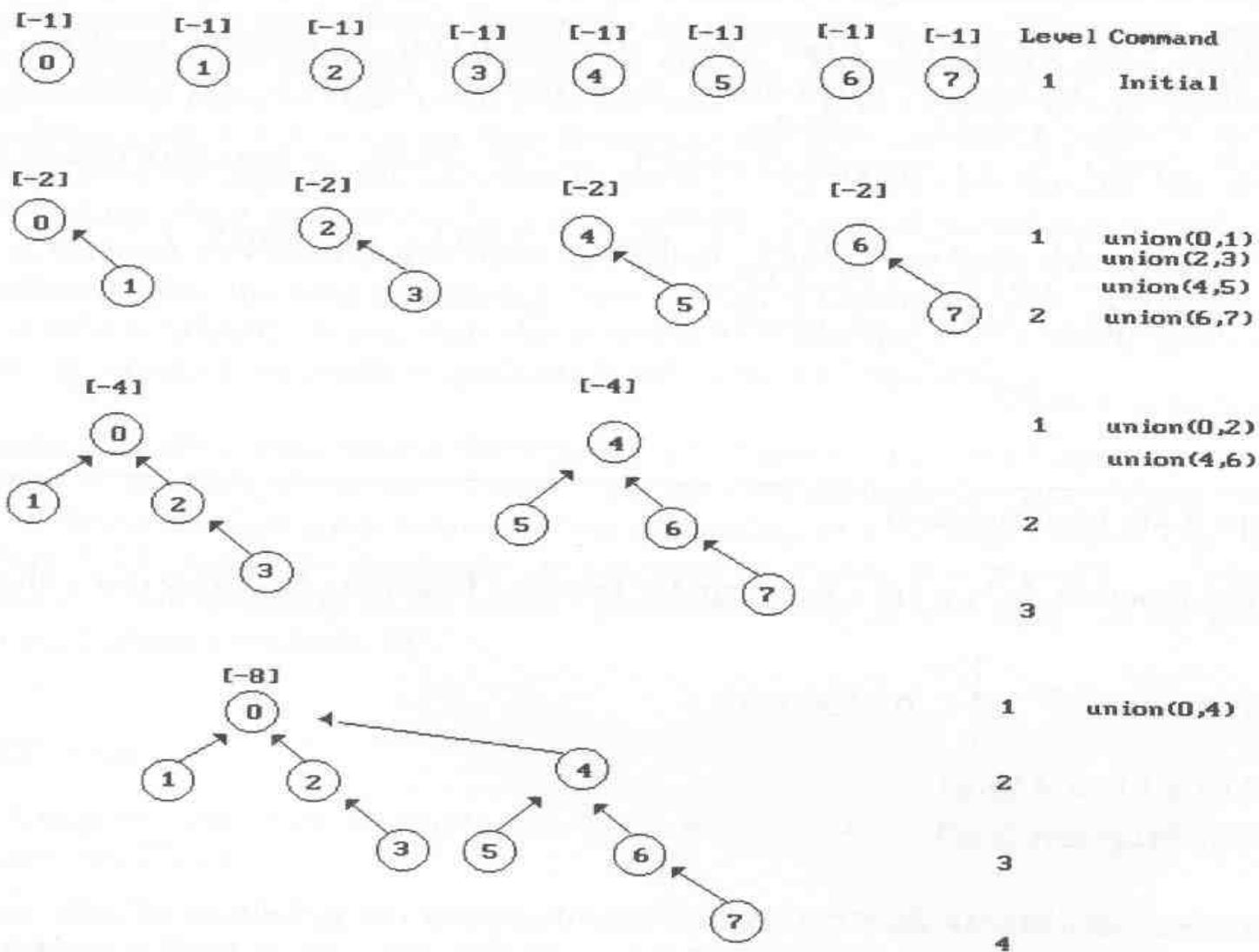
Figure 5.44: Trees obtained using the weighting rule

Modified Union Operation

```
void union2(int i, int j)
{
    Keep a count in the root of tree
    int temp = parent[i]+parent[j];
    if (parent[i]>parent[j]) {
        parent[i]=j;      i has fewer nodes.
        parent[j]=temp;
    }
    else { j has fewer nodes
        parent[j]=i;
        parent[i]=temp;
    }
}
```

If the number of nodes in tree i is less than the number in tree j, then make j the parent of i; otherwise make i the parent of j.

Figure 5.45: Trees achieving worst case bound (p.245)

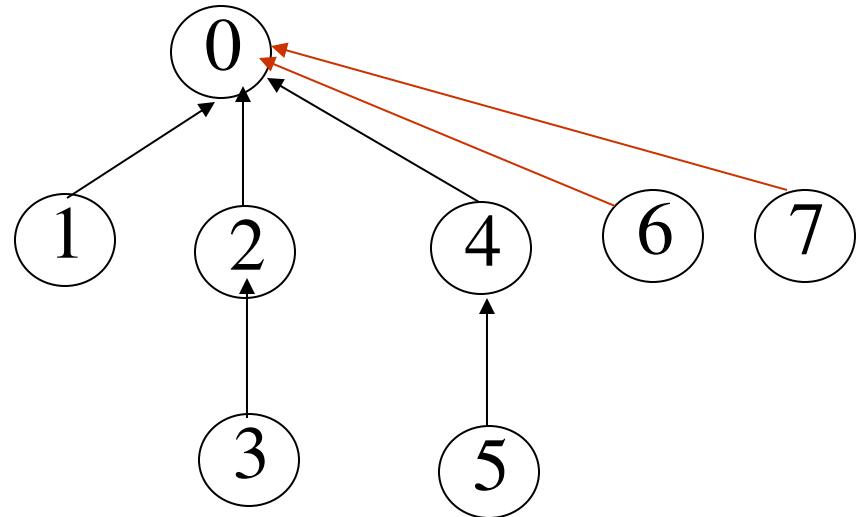
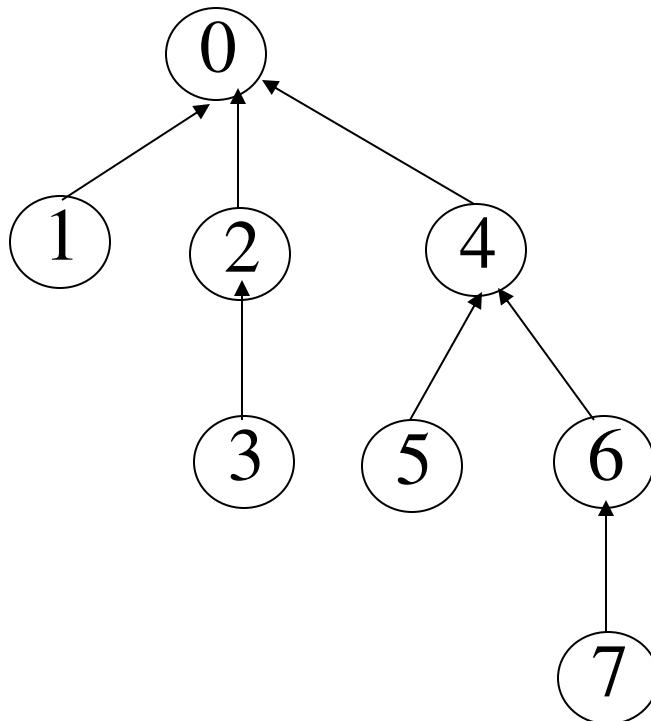


$\lfloor \log_2 8 \rfloor + 1$

Modified *Find(i)* Operation

```
int find2(int i)
{
    int root, trail, lead;
    for (root=i; parent[root]>=0;
          root=parent[root]);
    for (trail=i; trail!=root;
         trail=lead) {
        lead = parent[trail];
        parent[trail]= root;
    }
    return root;
}
```

If j is a node on the path from i to its root then make j a child of the root



find(7) find(7) find(7) find(7) find(7) find(7) find(7)

go up	3	1	1	1	1	1	1	1
reset	2							

12 moves (vs. 24 moves)

Applications

- n Find equivalence class $i \equiv j$
- n Find S_i and S_j such that $i \in S_i$ and $j \in S_j$
(two finds)
 - $S_i = S_j$ do nothing
 - $S_i \neq S_j$ union(S_i, S_j)
- n example
 - 0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8,
3 \equiv 5, 2 \equiv 11, 11 \equiv 0
 - {0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

preorder:
inorder:

A B C D E F G H I
B C A E D G H F I

