Data Structures

Sparse Matrix

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.

Sparse Matrix

$$
\begin{array}{c|ccc}
 & \text{col 1} & \text{col 2} & \text{col 3} \\
\hline
\text{row 1} & -27 & 3 & 4 \\
\text{row 2} & 6 & 82 & -2 \\
\text{row 3} & 109 & -64 & 11 \\
\text{row 4} & 12 & 8 & 9 \\
\text{row 5} & 48 & 27 & 47
\end{array}
\quad 5*3
$$

$$
\begin{array}{c|cccccc}
 & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} & \text{col6} \\
\hline
\text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
\quad 6*6
$$

(a) 15/15

(b) 8/36

sparse matrix
data structure?

<u>Two matrices</u>

SPARSE MATRIX ABSTRACT DATA TYPE

Structure *Sparse_Matrix* is

objects: a set of triples, <*row, column, value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all *a*, *b* ∈ *Sparse_Matrix*, *x* is *item*, *i*, *j*, *max_col*, *max_row* in *index*

Sparse_Marix Create(*max_row, max_col*) ::=

return a *Sparse_matrix* that can hold up to *max_items* = *max _row ,max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col.*

Sparse_Matrix Transpose(*a*) ::=

return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=

if the dimensions of a and b are the same

return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.

else return error

Sparse_Matrix Multiply(*a*, *b*) ::=

if number of columns in a equals number of rows in **b**

return the matrix *d* produced by multiplying a by *b* according to the formula: $d\,[i]\,[j] = \Sigma(a[i][k] \cdot b[k][j])$ where $d\,(i, j)$ is the (i,j)th element

else return error.

* **Structure :** Abstract data type Sparse-Matrix (p.68)

(1) Represented by a two-dimensional array.
Sparse matrix wastes space.

(2) Each element is characterized by <row, col, value>.

# of rows (columns)

# of nonzero terms

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

(a)

$\xrightarrow{\text{transpose}}$

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

(b)

row, column in ascending order

*__Figure__ Sparse matrix and its transpose stored as triples

Sparse_matrix Create(max_row, max_col) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
    typedef struct {
            int col;
            int row;
            int value;
            } term;
    term a[MAX_TERMS]
```

# of rows (columns)
# of nonzero terms

Transpose a Matrix

(1) for each row i

take element <i, j, value> and store it

in element <j, i, value> of the transpose.

difficulty: where to put <j, i, value>

(0, 0, 15) ====> (0, 0, 15)

(0, 3, 22) ====> (3, 0, 22)

(0, 5, -15) ====> (5, 0, -15)

(1, 1, 11) ====> (1, 1, 11)

Move elements down very often.

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

```
void transpose (term a[], term b[])
/* b is set to the transpose of a */
{
    int n, i, j, currentb;
    n = a[0].value;  /* total number of elements */
    b[0].row = a[0].col;  /* rows in b = columns in a */
    b[0].col = a[0].row;  /*columns in b = rows in a */
    b[0].value = n;
    if (n > 0) {                  /*non zero matrix */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
        /* transpose by columns in a */
           for( j = 1; j <=  n; j++)
           /*  find elements from the current column */
           if (a[j].col == i) {
          /* element is in current column, add it to b */
```

```
columns
    elements
        b[currentb].row = a[j].col;
        b[currentb].col  = a[j].row;
        b[currentb].value = a[j].value;
        currentb++;
      }
  }
}
```

*** Program :** Transpose of a sparse matrix

Scan the array “columns” times.
The array has “elements” elements. ==> O(columns*elements)

Discussion: compared with 2-D array representation

O(columns*elements) vs. O(columns*rows)

elements --> columns * rows when nonsparse
O(columns*columns*rows)

Problem: Scan the array “columns” times.

Solution:

Determine the number of elements in each column of the original matrix.
==>
Determine the starting positions of each row in the transpose matrix.

a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

```
void fast_transpose(term a[ ], term b[ ])
{
/* the transpose of a is placed in b */
  int row_terms[MAX_COL], starting_pos[MAX_COL];
  int i, j, num_cols = a[0].col, num_terms = a[0].value;
  b[0].row = num_cols; b[0].col = a[0].row;
  b[0].value = num_terms;
  if (num_terms > 0){ /*nonzero matrix*/
            for (i = 0; i < num_cols; i++)
columns          row_terms[i] = 0;
            for (i = 1; i <= num_terms; i++)
elements         row_term [a[i].col]++
            starting_pos[0] = 1;
            for (i =1; i < num_cols; i++)
columns          starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
```

elements

```
for (i=1; i <= num_terms, i++) {
        j = starting_pos[a[i].col]++;
        b[j].row = a[i].col;
        b[j].col = a[i].row;
        b[j].value = a[i].value;
}
    }
}
```

***Program** Fast transpose of a sparse matrix

Data Structures

Strings

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.

Strings

```
Char  name[25]={“RIT”};
Char cname[]={“RIT”};
```

Strings : ADT

ADT string is
Objects : a finite set of zero or more characters
Functions:
For all s,t belongs to string,
i,j,m belongs to non negative integers.

String null(m) ::= return a string whose maximum length
is m characters, but is initially set to NULL
Integer compare(s,t)::= if s equals t return 0
else if s precedes t return -1
else return +1

Contd…

Boolean isNull(s) ::= if compare(s,null) return False
else return True

Integer Length(S)::= if(compare(s,null) return the number
of characters in s.
else return 0.

String concat(s,t) ::= if(compare(t,null)) return a string whose
elements are those of s followed by
those of t,
else return s.

String substr(s,i,j) ::= if(j>0) &&(i+j-1)<length(s))
return the string containing the characte
of s at position i,i+1,….i+j-1.
else return null.

C string functions & Examples

- Strcat(s,t)
- Strncat(s,t,n)
- Strcmp(s,t)
- Strncmp(s,t,n)
- Strcpy(s,t)
- Strncpy(s,t,n)
- Strlen(s)
- Strchr(s,c) : return ptr to first occurrence of c in s
- Strrchr(s,c) : return ptr to last occurrence of c in s
- Strtok(s,delim) : return string surrounded by delim in s
- Strstr(s,pat) : return ptr to start of pat in s
- Strspn(s,spanset) : return length of span in s
- Strcspn(s,spanset)
- Strpbrk(s,spanset): return ptr to first occurrence of char from spanset

Pattern Matching

Char pat[30],string[50],*t;
If(strstr(s,p) printf("pat Is in str!");
Else printf("pat not found in str!");

Write a String insertion Function

```
Void strnins(char *s,char *t,int i)
{
   char string[30],*temp=string;
   if(i<0 &&i>strlen(s))
           {
                   printf("out of boundary!"); exit(0);
           }
   if(strlen(s))
           {
                   strcpy(s,t);
           }
```

Contd…

```
else if(strlen(t))
{
    strncpy(temp,s,i);
     strcat(temp,t);
     strcat(temp,(s+i));
     strcpy(s,temp);
 }

}
```

Exercise

Write a User defined function to return the token from a string surrounded by a delimiter.

Pattern matching by checking end indices first

```
int nfind(char *string,char *pat)
{
        int i,j,start=0;
        int lasts=strlen(string)-1;
        int lastp=strlen(pat)-1;
        int endmatch=lastp;
```

Contd..

```
for(i=0;endmatch<=lasts;endmatch++,start++)
{
        if(string[endmatch]==pat[lastp])
        for(j=0,i=start;j<lastp
                &&string[i]==pat[j];i++,j++)
                ;
        if(j==lastp)
                return start;
        }
        return -1;
}
```

KMP Algorithm

Failure Function

Pat: abcabcacab

J	0	1	2	3	4	5	6	7	8	9
Pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

KMP Algorithm for pattern matching

```
Int pmatch(char *string,char *pat)
{
   int i=0,j=0;
   int lens=strlen(string);
Int lenp=strlen(pat);
While(i<lens && j<lenp)
{
  if(string[i]==pat[j])
         {
                 i++;j++;
         }
Else if(j==0) i++;
Else j=failure[j-1]+1;
} return ((j==lenp)?(i-lenp):-1);
}
```

KMP Algorithm : Failure function

```
Void fail(char *pat)
{
        int n=strlen(pat);
        failure[0]=-1;
        for(j=1;j<n;j++)
        {
                i=failure[j-1];
                while((pat[j]!=pat[i+1])&&(i>=0))
                        i=failure[i];
                if(pat[j]==pat[i+1])
                        failure[j]=i+1;
                else failure[j]=-1;
        }
}
```

Data Structures

UNIT-1

PREPARED BY

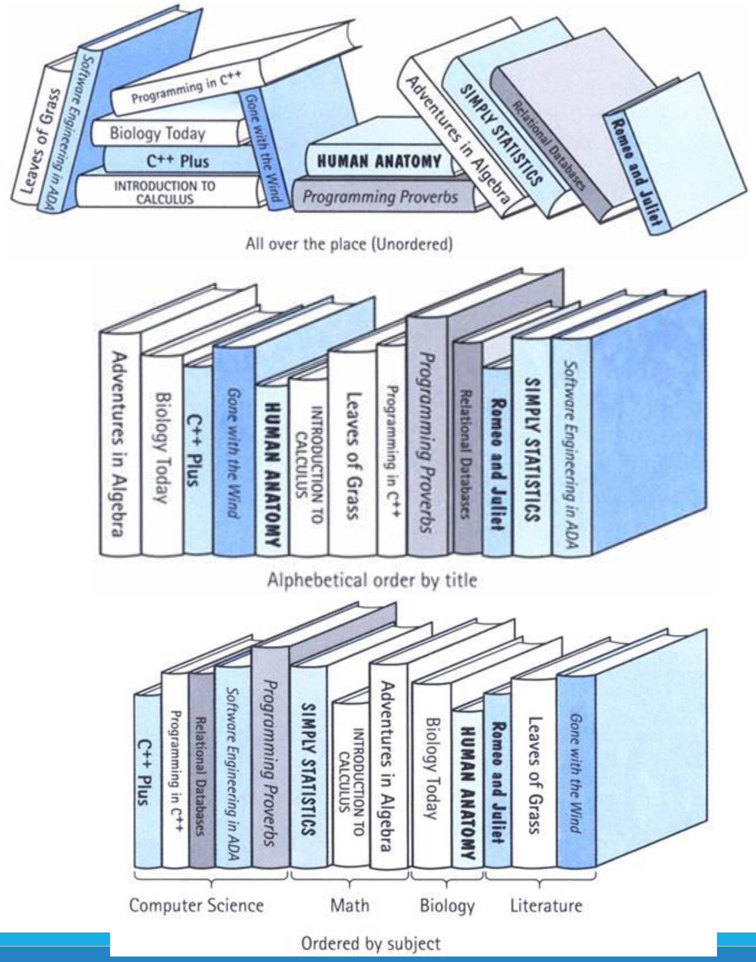
PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

What is Data Structures?

- Example: Library
  - is composed of elements (books)
  - Accessing a particular book requires knowledge of the arrangement of the books
  - Users access books only through the librarian



Basic Data Structures

Structures include

- linked lists
- Stack, Queue
- binary trees
- …and others

Variables

ADDRESS

- For every variable there are two attributes: <u>address</u> and <u>value</u>

In memory with address 3: value: 45.

In memory with address 2: value "Dave"

1	4096
2	"Dave"
3	45
4	"Matt"
5	95.5
6	"wbru"
7	0
8	"zero"

POINTERS

1. It is a variable whose value is also an address.
2. A pointer to an integer is a variable that can store the address of that integer

Pointers

1. Declaration
2. Assigning variable’s address to pointer
3. NULL value in pointer
4. Checking for NULL value
5. Type casting

Dynamic memory allocation

1. Heap
2. Allocating storage during run time
3. Using functions
   - malloc
   - free

Pointers - Example

```
int i,*pi;
pi=(int *)malloc(sizeof(int));
if(pi==NULL)
        {
                printf("memory space is not avail!");
                exit(0);
        }
*pi=567;
printf("%d",*pi);
free(pi);
```

Macro definition for memory allocation

```
#define MALLOC(p,s)
if(!((p)=malloc(s)))
{
printf("cant allocate memory");
exit(0);
}
```

Calling Macro in main() Program

```
int  *pi;
MALLOC(pi,sizeof(int))
```

Exercises

1. Write a C program to add two numbers using pointers. Assign variables to pointers and do addition.

2. Write a C program to add two numbers using pointers. Allocate space to pointers and then assign the vales and do addition.

3. Write a C program to do addition of two numbers using pointers and Macro definition

References

1. http://mitra.ac.in
2. Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed",Fundamentals of Data Structures in C",Computer Science Press, 1992.

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

Algorithm

Definition

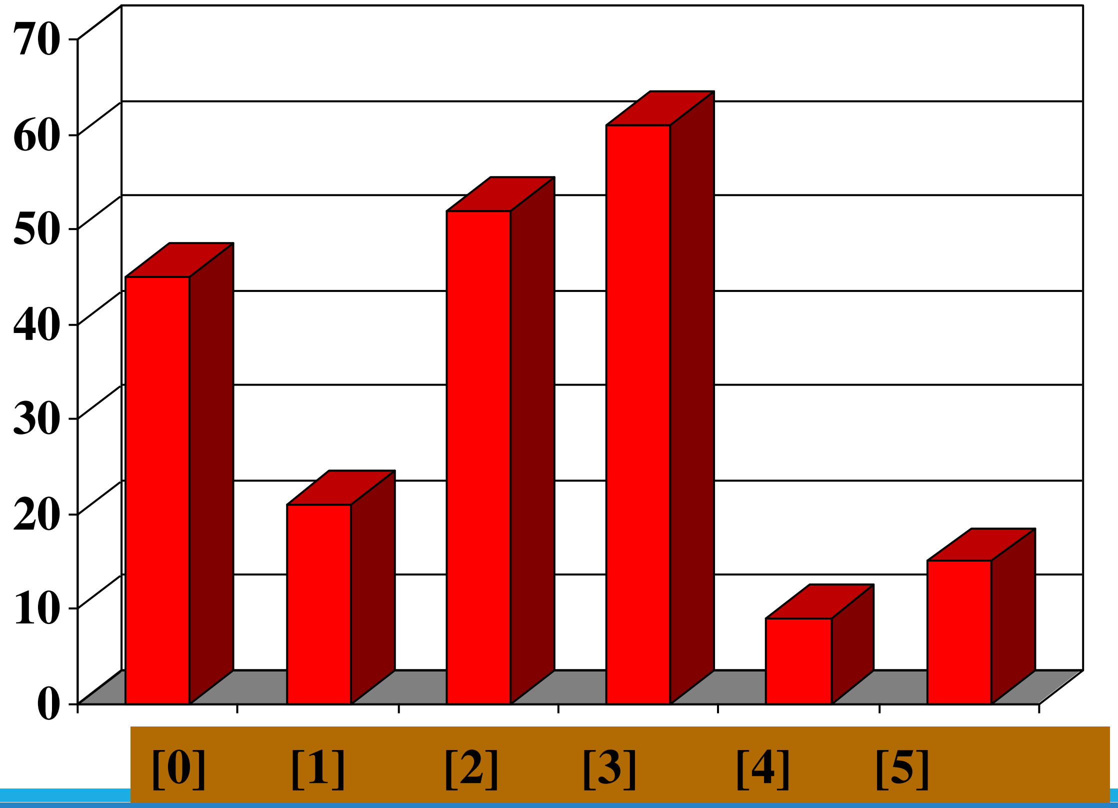
An *algorithm* is a finite set of instructions that accomplishes a particular task.

Criteria

- **input**
- **output**
- **definiteness: clear and unambiguous**
- **finiteness: terminate after a finite number of steps**
- **effectiveness: instruction is basic enough to be carried out**

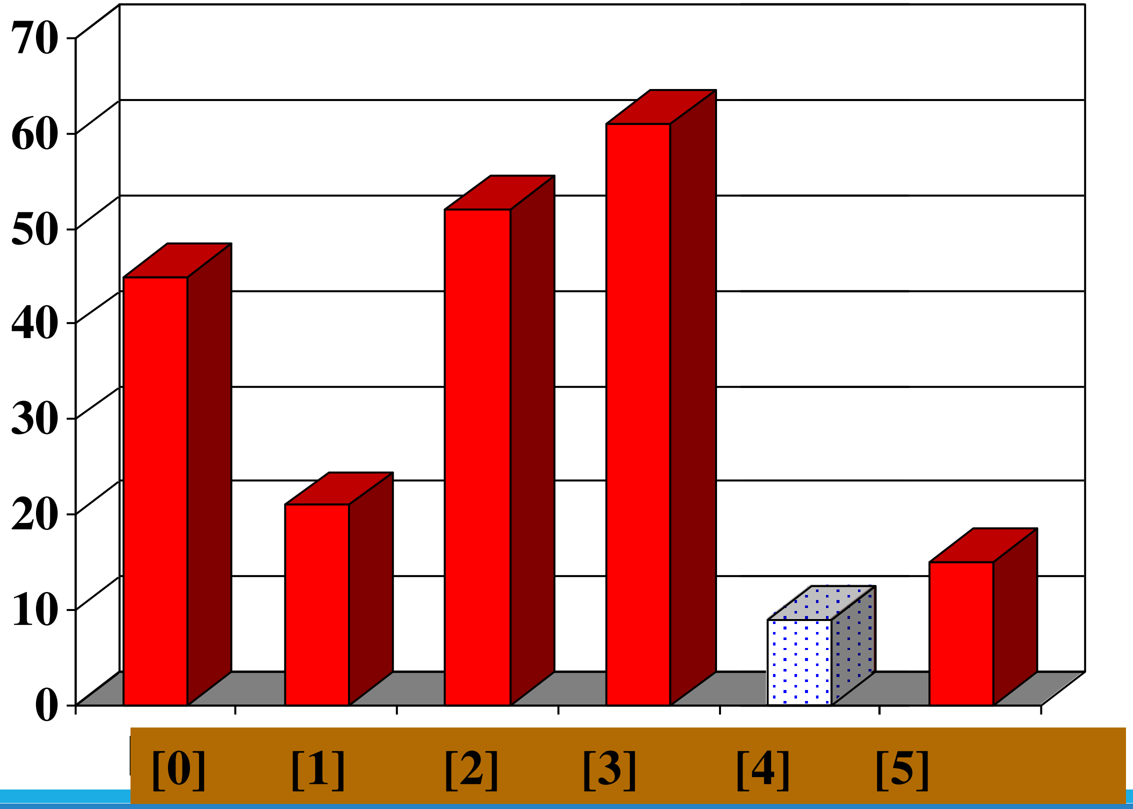
Sorting an Array of Integers

Example: we are given an array of six integers that we want to sort from smallest to largest



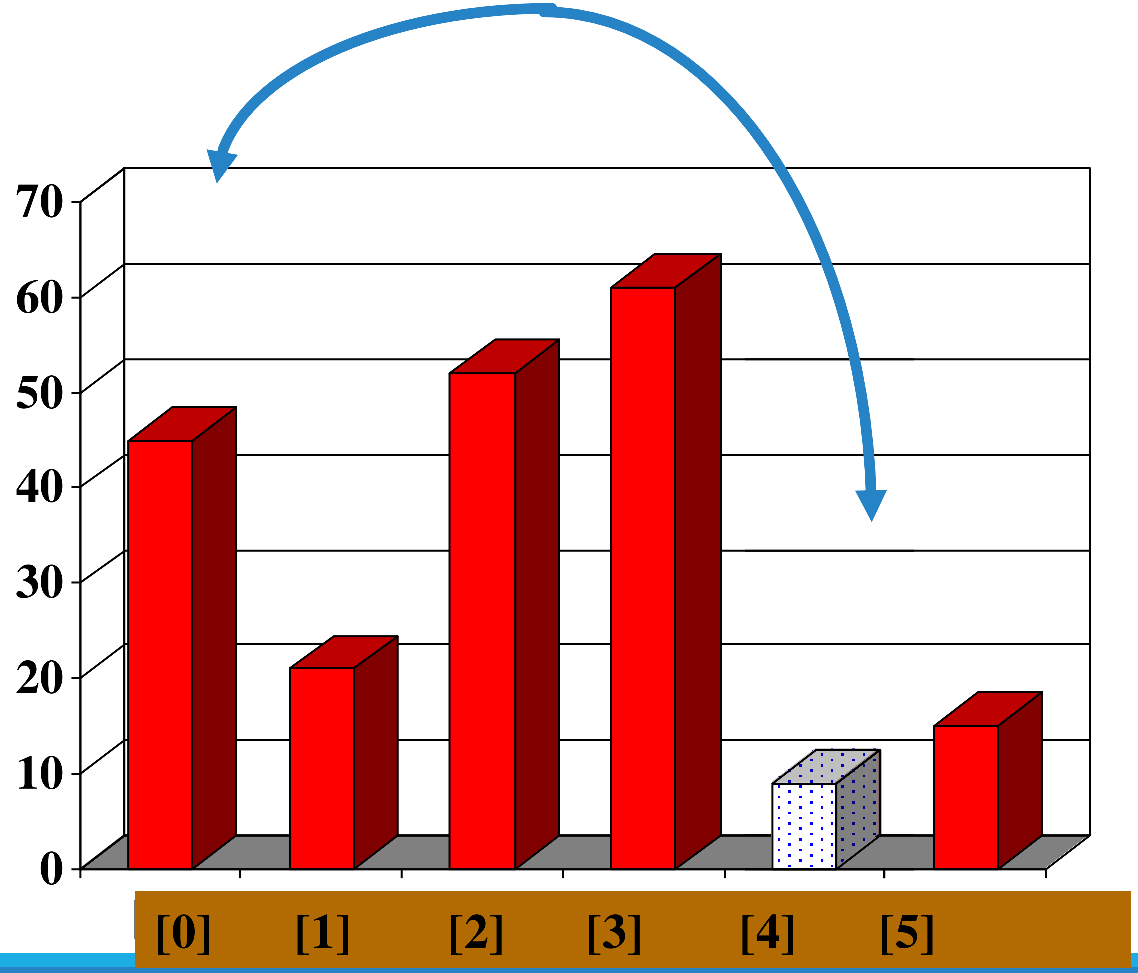
The Selection Sort Algorithm

Start by finding the smallest entry.



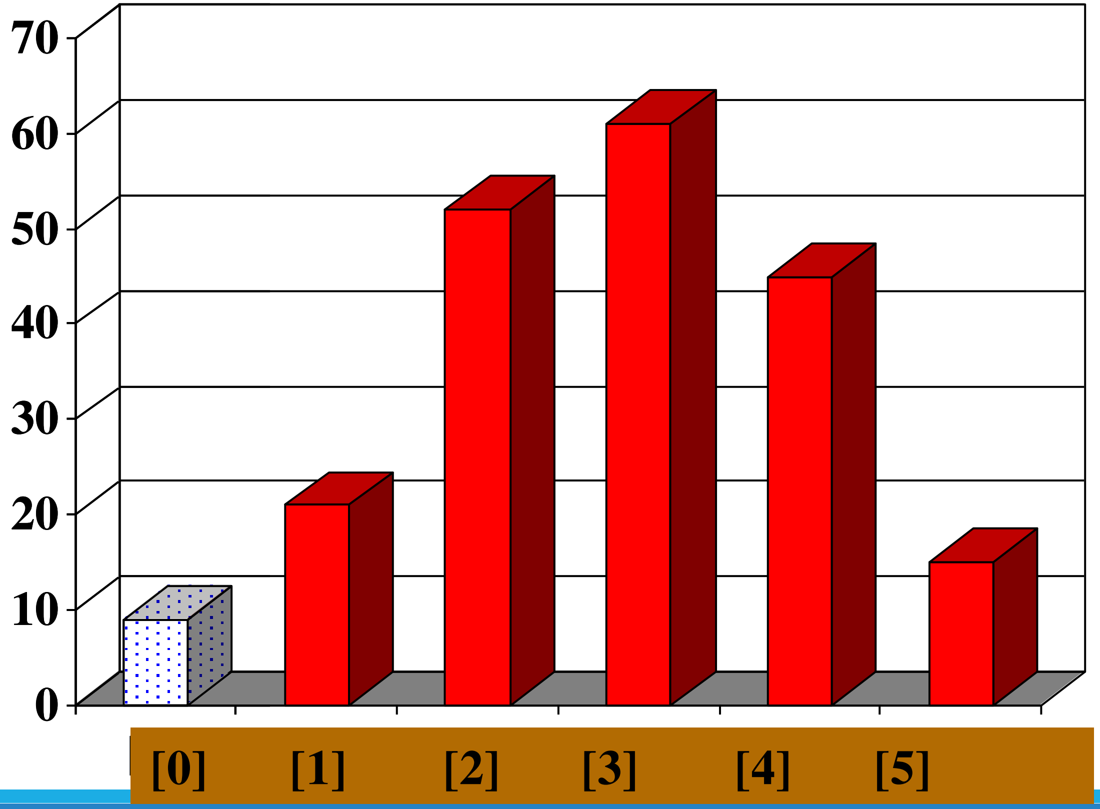
The Selection Sort Algorithm

Swap the smallest entry with the <u>first entry</u>.



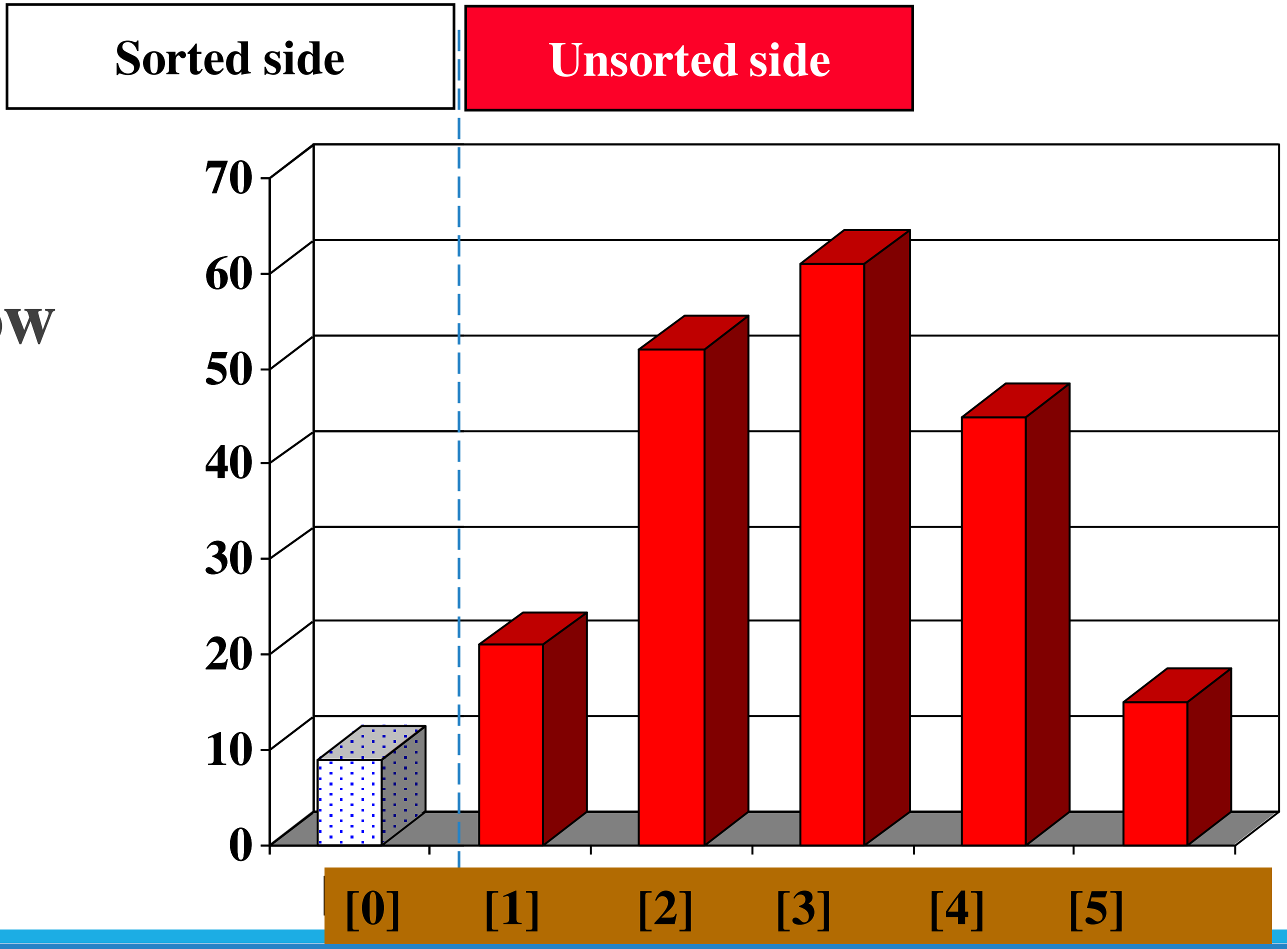
The Selection Sort Algorithm

Swap the smallest entry with the <u>first entry</u>.



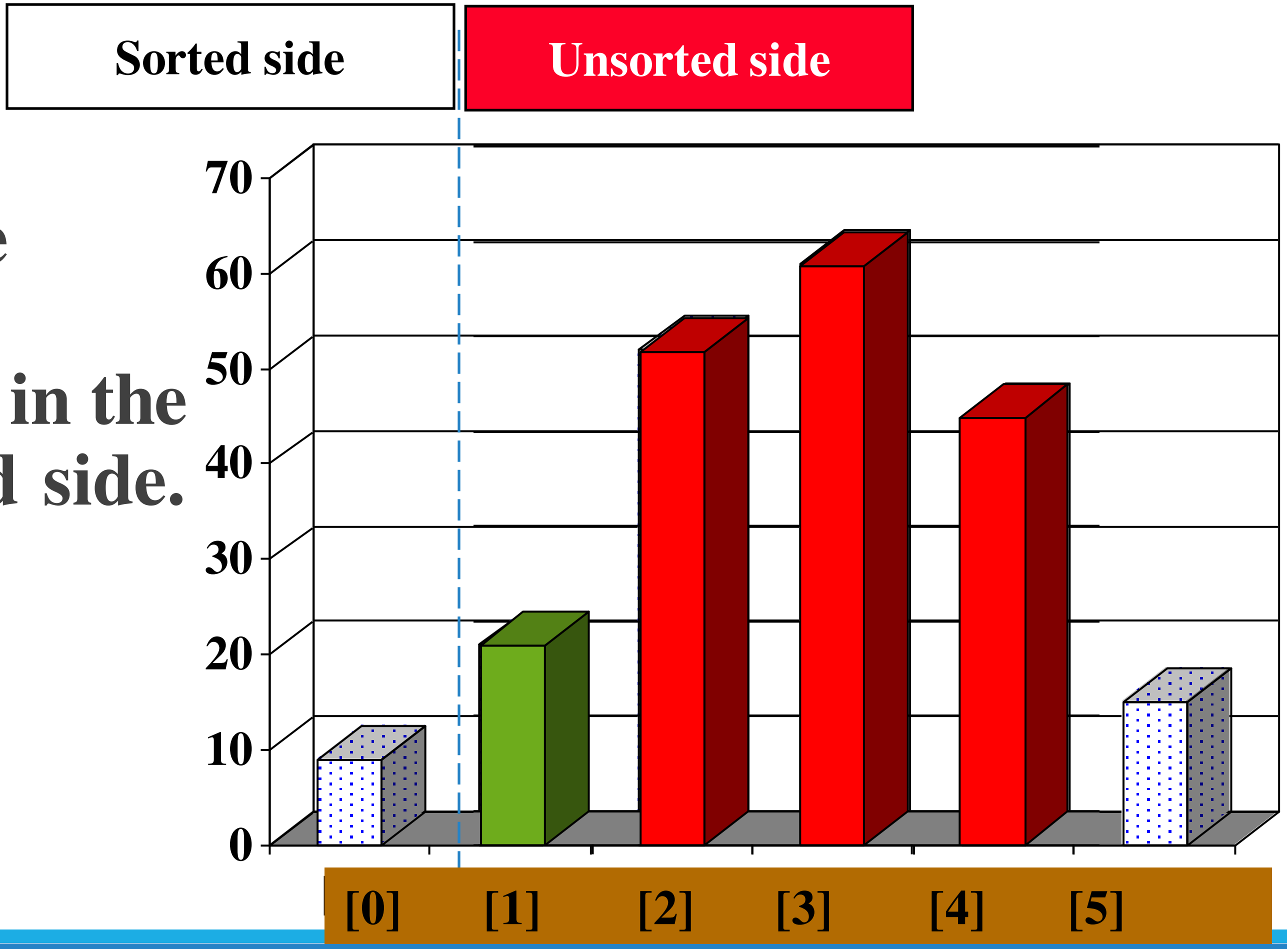
The Selection Sort Algorithm

Part of the array is now sorted.



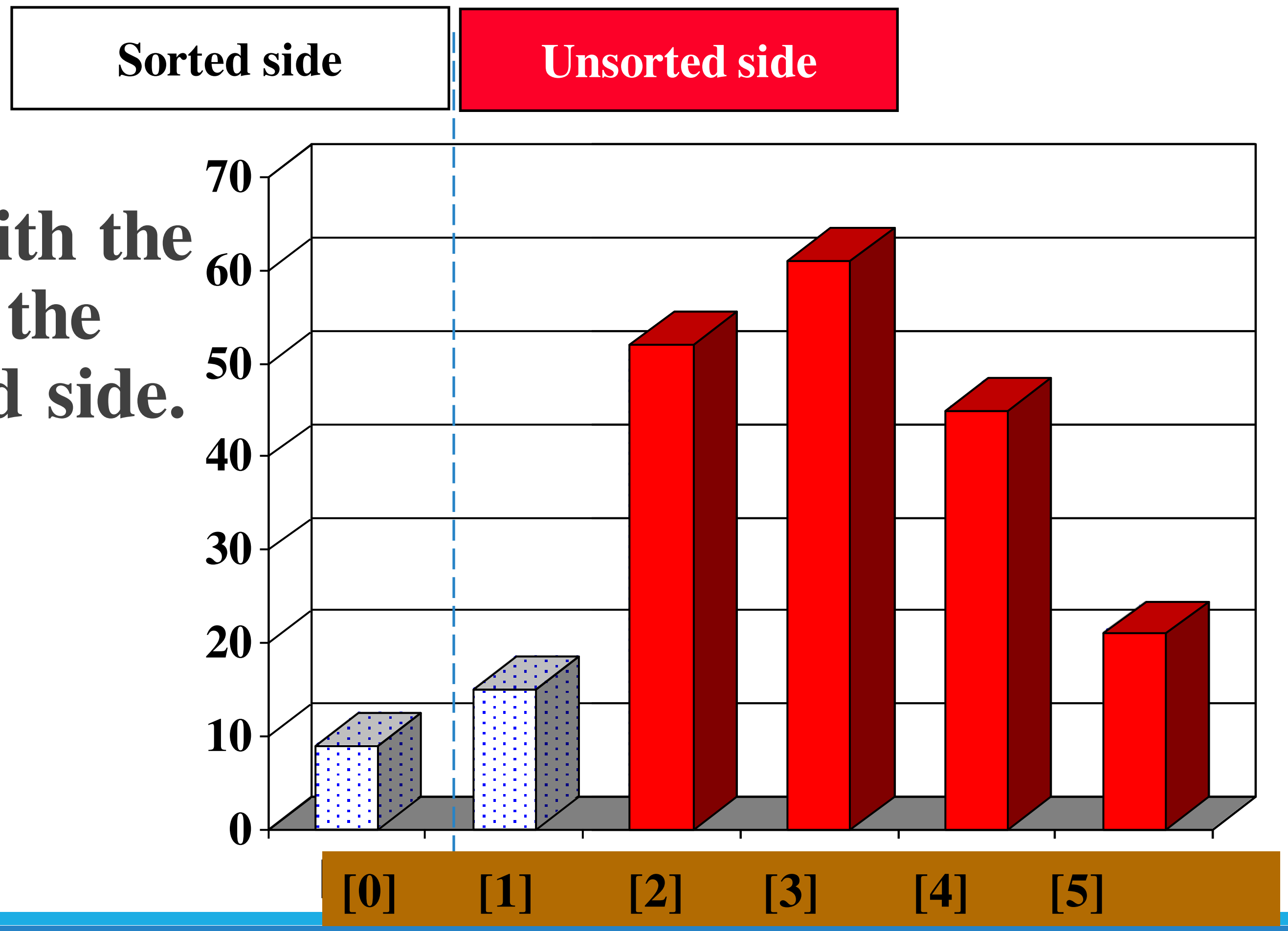
The Selection Sort Algorithm

Find the smallest element in the unsorted side.



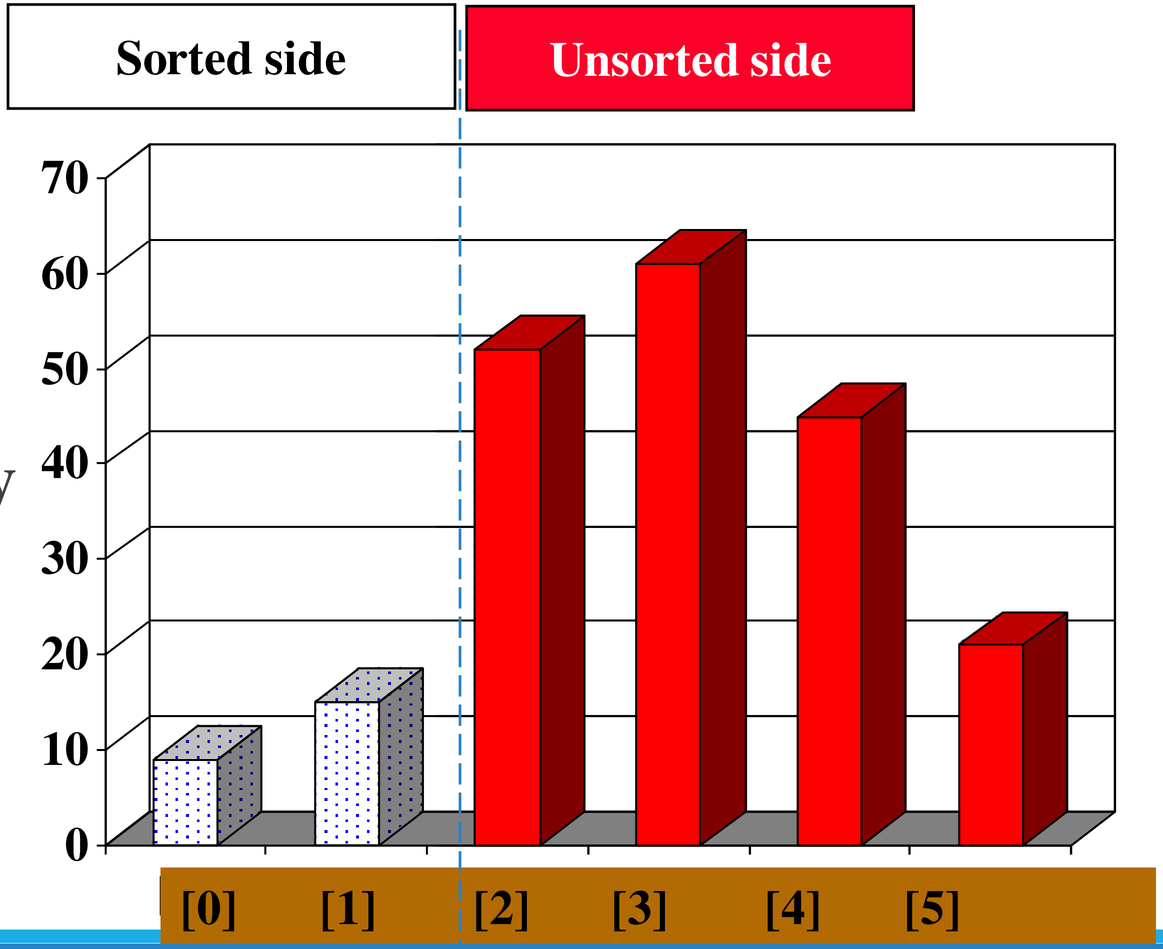
The Selection Sort Algorithm

Swap with the front of the unsorted side.



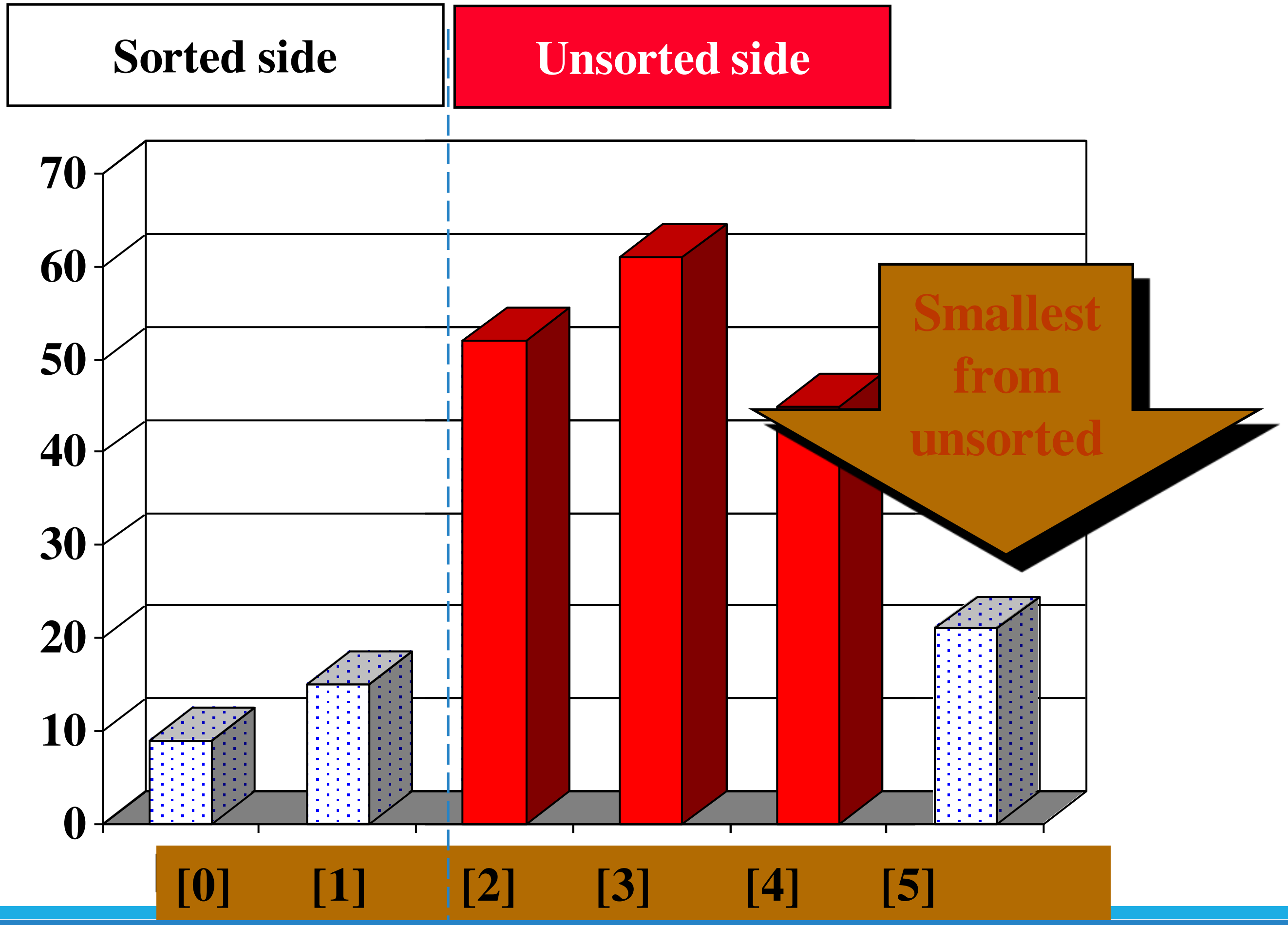
The Selection Sort Algorithm

We have increased the size of the sorted side by one element.



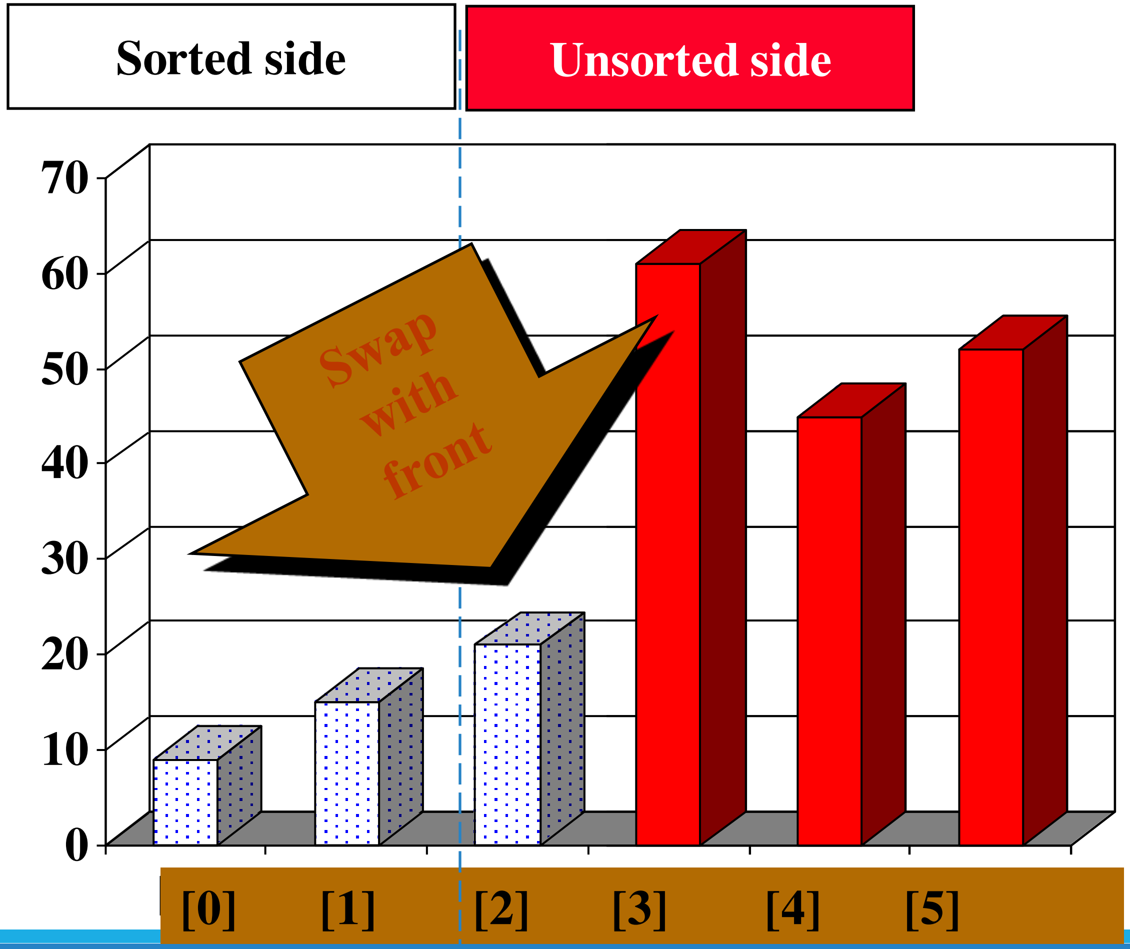
The Selection Sort Algorithm

The process continues...



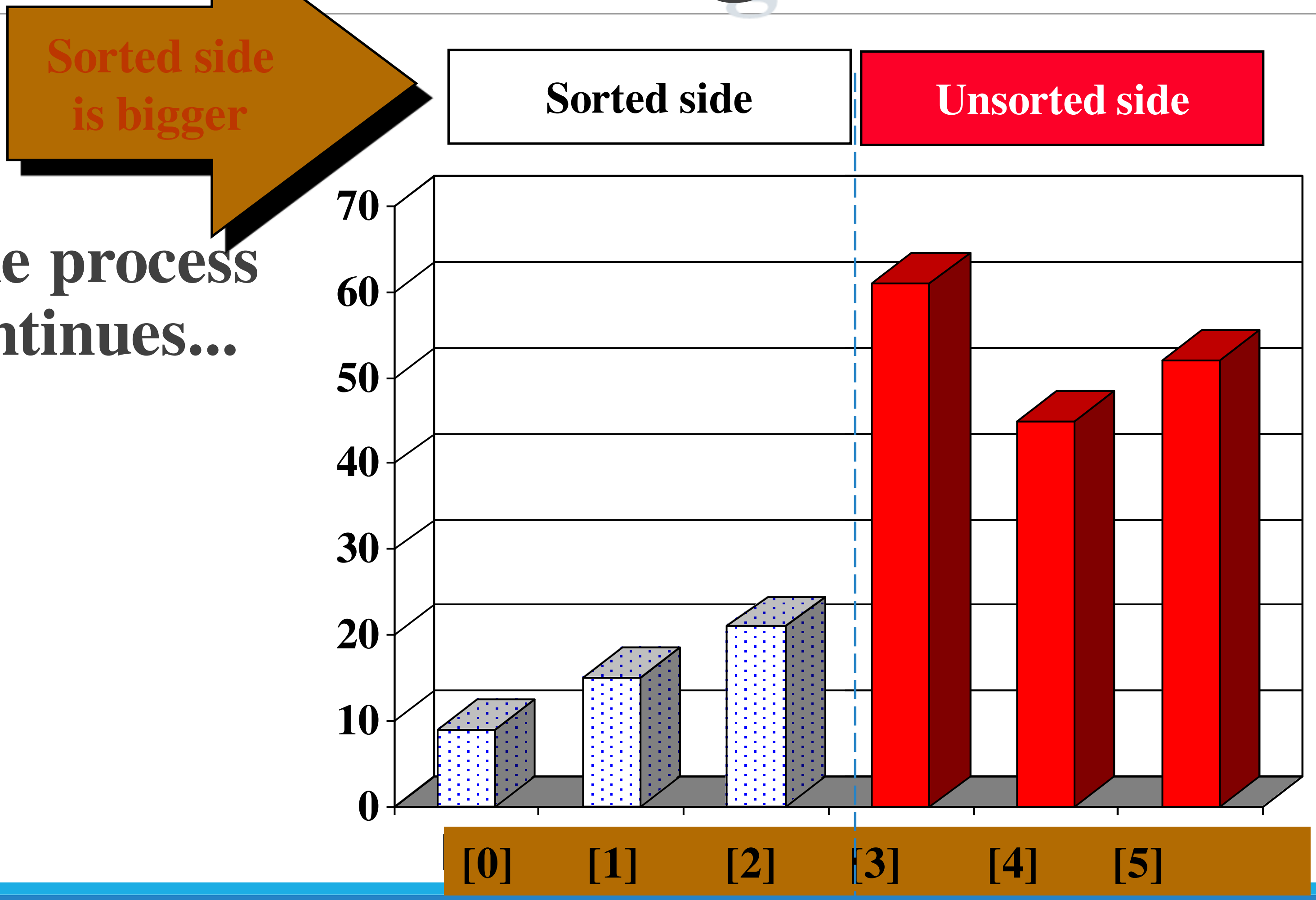
The Selection Sort Algorithm

The process continues...



The Selection Sort Algorithm

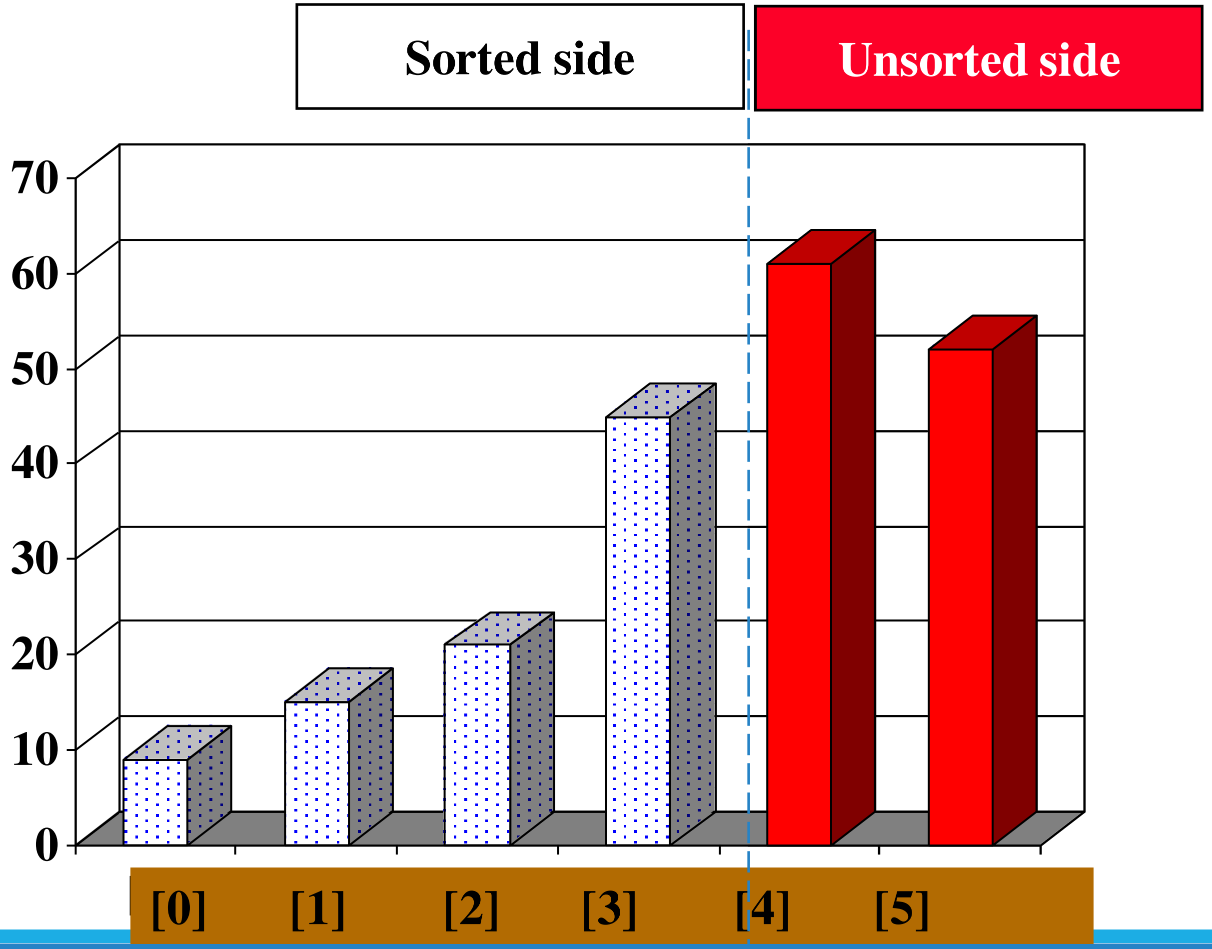
The process continues...



The Selection Sort Algorithm

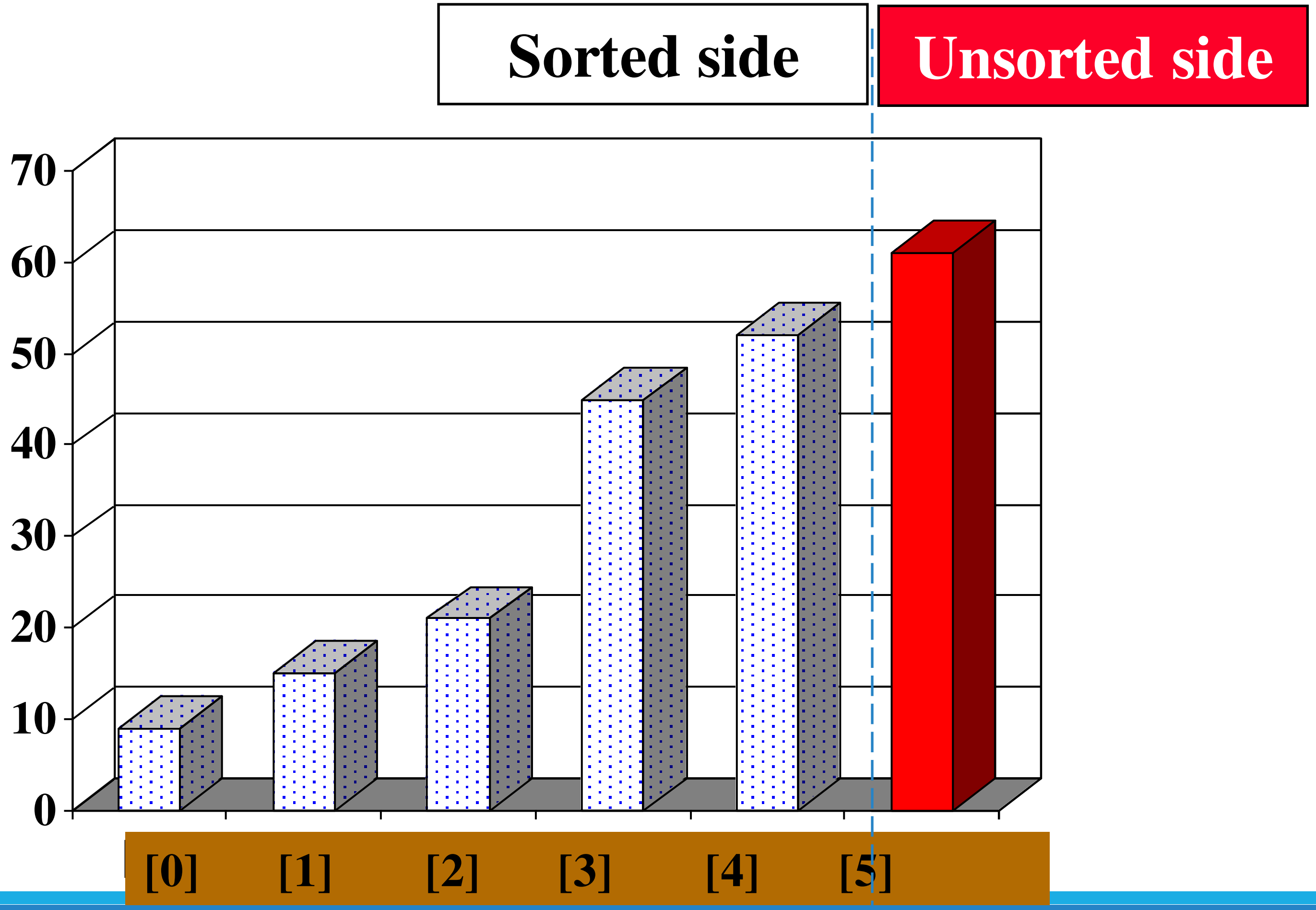
The process keeps adding one more number to the sorted side.

The sorted side has the smallest numbers, arranged from small to large.



The Selection Sort Algorithm

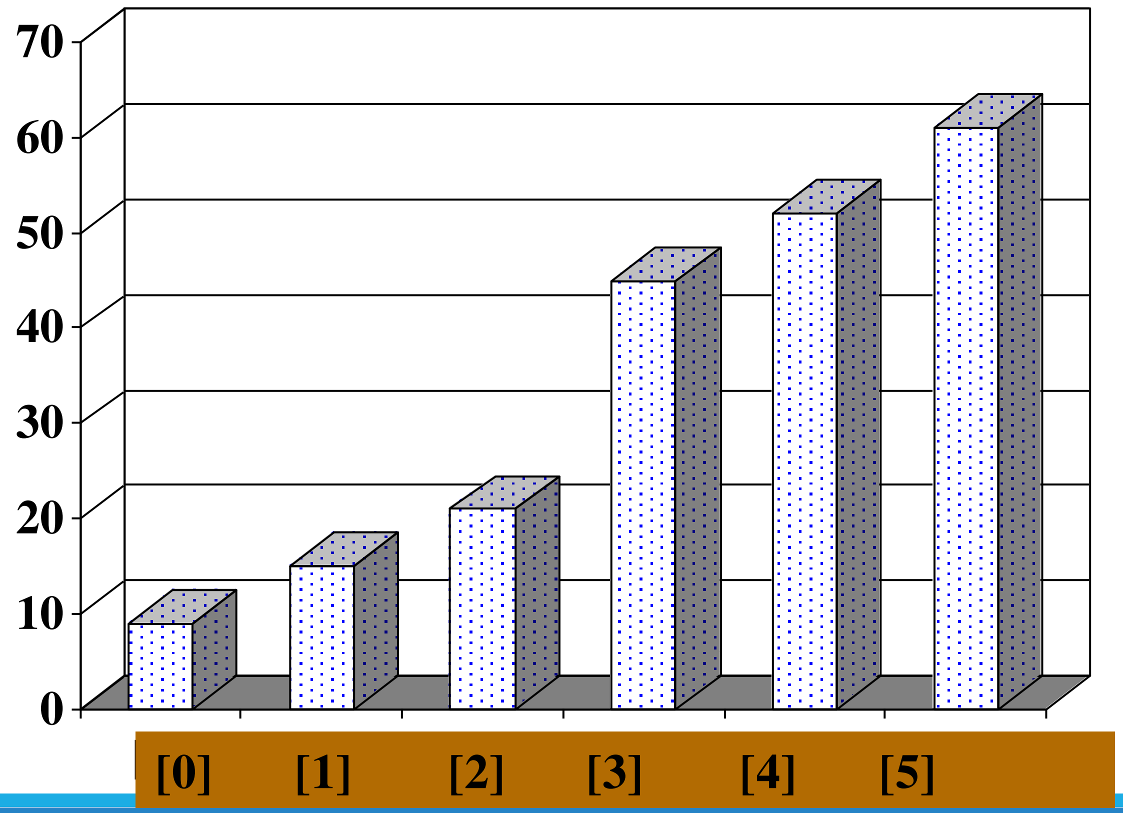
We can stop when the unsorted side has just one number, since that number must be the largest number.



The Selection Sort Algorithm

The array is now sorted.

We repeatedly <u>selected</u> the smallest element, and moved this element to the front of the unsorted side.



Selection Sort

```
void selection_sort(int arr[], int n)
{int i, j, min;
for (i = 0; i < n - 1; i++)
 {
 min = i;
 for (j = i+1; j < n; j++)
   { if (list[j] < list[min])  min = j;  }
 swap(arr[i],arr[min]);
 }
```

Tutorial

1. Write a C function to add two numbers using pointers
2. Write a C function to swap two numbers using pointers
3. Write a C macro to swap two numbers
4. Write a Complete C program to perform selection sort with macro for swap and sort().

References

1. http://mitra.ac.in
2. Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed“,Fundamentals of Data Structures in C”, Computer Science Press, 1992.
3. https://www.csie.ntu.edu.tw/~ds/ppt/ch1/chapter1.ppt
4. https://www.csie.ntu.edu.tw/~ds/ppt/ch7/chapter7.ppt

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

What is recursion?

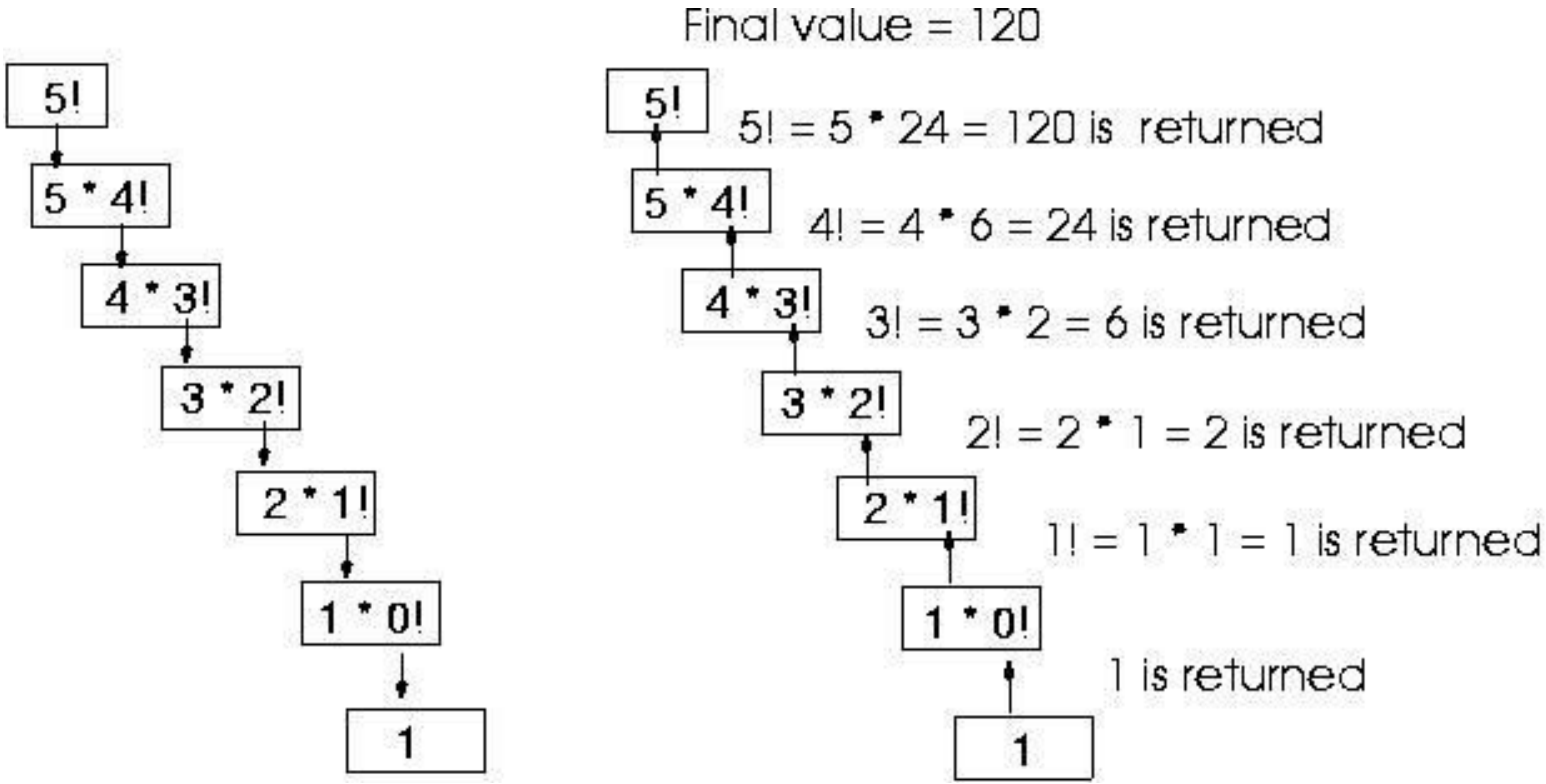
Sometimes, the best way to solve a problem is by solving a <u>smaller version</u> of the exact same problem first

Recursion is a technique that solves a problem by solving a <u>smaller problem</u> of the same type

Coding the factorial function

Recursive implementation

```
int Factorial(int n)
{
 if (n==0)
   return 1;
 else
   return n * Factorial(n-1);
}
```


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

Coding the factorial function (cont.)

Iterative implementation

```
int Factorial(int n)
{
 int fact = 1,count,n;

 for(count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

Ex. Binary search for 33.

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] $\leq$ value $\leq$ a[hi].

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi

Ex. Binary search for 33.

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo						↑ hi								

Ex. Binary search for 33.

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo			↑ mid			↑ hi								

Ex. Binary search for 33.

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo		↑ hi								

Ex. Binary search for 33.

Binary Search

Binary search. **Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.**

Invariant. **Algorithm maintains a[lo] ≤ value ≤ a[hi].**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo	↑ mid	↑ hi								

Ex. **Binary search for 33.**

Binary Search

Binary search. **Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.**

Invariant. **Algorithm maintains a[lo] ≤ value ≤ a[hi].**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo hi										

Ex. **Binary search for 33.**

Binary Search

Binary search. **Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.**

Invariant. **Algorithm maintains a[lo] ≤ value ≤ a[hi].**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

Ex. **Binary search for 33.**

Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] $\leq$ value $\leq$ a[hi].

Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑										
				lo hi mid										

Recursive binary search

```
void binary_search(int list[], int lo, int hi, int key)
{
int mid;

if (lo > hi)
{
printf("Key not found\n");
return;
}
mid = (lo + hi) / 2;
if (list[mid] == key)
{
printf("Key found\n");
}
else if (list[mid] > key)
{
binary_search(list, lo, mid - 1, key);
}
else if (list[mid] < key)
{
binary_search(list, mid + 1, hi, key);
}
}}
```

Exercise

Write Macro for comparing two variables using ternary operator(Conditional operator).

Define Macro : COMPARE

```
#define COMPARE(x,y) (((x)<(y))?-1 : ((x) ==(y)) ? 0:1)
```

Using COMPARE macro in Binary search

```
int binary_search(int list[], int lo, int hi, int key)
{
int mid;

if (lo > hi)
{
printf("Key not found\n");
return;
}
mid = (lo + hi) / 2;
switch(COMPARE(list[mid],key))
{
 case 0: {printf("Key found\n");
              return mid;}
case 1:  // (list[mid] > key)
{
return binary_search(list, lo, mid - 1, key);
}
case -1:// (list[mid] < key)
{
return binary_search(list, mid + 1, hi, key);
}
}}
```

Iterative Implementation of Binary Search with compare function

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
int middle;
 while (left<= right){
   middle= (left+ right)/2;
   switch(compare(list[middle], searchnum)){
     case -1: left= middle+ 1;
            break;
     case 0: return middle;
     case 1: right= middle- 1; break;
   } }
 return -1;}
```

```
int compare(int x, int y)
{
  if (x< y) return -1;
  else if (x== y) return 0;
  else return 1;
}
```

Recursive Implementation of Binary Search

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
int middle;
 while (left<= right){
   middle= (left+ right)/2;
   switch(compare(list[middle], searchnum)){
     case -1:return binsearch(list, searchnum, middle+1,
right);
     case 0: return middle;
     case 1: return binsearch(list, searchnum, left, middle-
1);
   }
 }
 return -1;}
```

Permutation

A PERMUTATION IS AN ARRANGEMENT IN WHICH ORDER MATTERS.

A B C DIFFERS FROM B C A

4 x 3 x 2 x 1 = 24 Permutations

ABCD	ABDC	ACBD
ACDB	ADBC	ADCB
BACD	BADC	BCAD
BCDA	BDAC	BDCA
CABD	CADB	CBAD
CBDA	CDAB	CDBA
DABC	DACB	DBAC
DBCA	DCAB	DCBA

Generalization

THERE ARE 4! WAYS TO
ARRANGE 4 ITEMS.

THERE ARE N! WAYS TO
ARRANGE N ITEMS.

Recursive Permutation generator

```
Void perm(char *list,int I,int n)
{
        int j,temp;
      ◦ If(i==n)
        {
                for{j=0;j<=n;j++)
                        printf("%",list;j);

        Printf(" ");
        }
```

Contd….

Else{ //list[i] to list[n] has more than one permutation , generate them recursively

```
for(j=i;j<=n;j++)
{
  swap(list[i],list[j],temp);
  perm(list,i+1,n);
   swap(list[i],list[j],temp);
}
}
}
```

Data Abstraction

Types of data

- All programming language provide at least minimal set of predefined data type, plus user defined types

Data types of C

- Char, int, float, and double
  - may be modified by short, long, and unsigned
- Array, struct, and pointer

Data Type

Definition

- **A *data type* is a collection of *objects* and a set of *operations* that act on those objects**

Example of "int"

- **Objects: 0, +1, -1, ..., Int_Max, Int_Min**
- **Operations: *arithmetic*(+, -, *, /, and %), *testing*(equality/inequality), *assigns*, *functions***

Define operations

- **Its *name*, possible *arguments* and *results* must be specified**

The design strategy for representation of objects

- ***Transparent* to the user**

Abstract Data Type

Definition

- **An *abstract data type*(*ADT*) is a *data type* that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operation.**

Why abstract data type ?

- **implementation-independent**

Classifying the Functions of a Data Type

Creator/constructor:

- Create a new instance of the designated type

Transformers

- Also create an instance of the designated type by using one or more other instances

Observers/reporters

- Provide information about an instance of the type, but they do not change the instance

Notes

- An ADT definition will include at least one function from each of these three categories

An Example of the ADT

structure Natural_Number is
objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (INT_MAX) on the computer
functions:
for all x, y is Nat_Number, TRUE, FALSE is Boolean and where . +, -, <, and == are the usual integer operations
Nat_NoZero() ::= 0
Boolean Is_Zero(x) ::= if (x) return FALSE

Contd….

```
Nat_No Add(x, y)          ::= if ((x+y)<= INT_MAX) return x+ y
                              else return INT_MAX
Boolean Equal(x, y)       ::= if (x== y) return TRUE
                              else return FALSE
Nat_No Successor(x)       ::= if (x== INT_MAX) return x
                              else return x+ 1
Nat_No Subtract(x, y)     ::= if (x< y) return 0
                              else return x-y
end Natural_Number
```

Tutorial

Write a C function to find sum of array elements using recursive function.

Define a ADT for complex numbers

References

1. https://www.cise.ufl.edu/class/cop3275fa16/lectures/Recursion.ppt
2. http://www.sanfoundry.com/c-program-binary-search-recursion/
3. https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

Data Abstraction

Types of data

- All programming language provide at least minimal set of predefined data type, plus user defined types

Data types of C

- Char, int, float, and double
  - may be modified by short, long, and unsigned
- Array, struct, and pointer

Data Type

Definition

- **A *data type* is a collection of *objects* and a set of *operations* that act on those objects**

Example of "int"

- **Objects: 0, +1, -1, ..., Int_Max, Int_Min**
- **Operations: *arithmetic*(+, -, *, /, and %), *testing*(equality/inequality), *assigns*, *functions***

Define operations

- **Its *name*, possible *arguments* and *results* must be specified**

The design strategy for representation of objects

- ***Transparent* to the user**

Abstract Data Type

Definition

- **An *abstract data type*(*ADT*) is a *data type* that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operation.#**

Why abstract data type ?

- **implementation-independent**

Classifying the Functions of a Data Type

Creator/constructor:

- Create a new instance of the designated type

Transformers

- Also create an instance of the designated type by using one or more other instances

Observers/reporters

- Provide information about an instance of the type, but they do not change the instance

Notes

- An ADT definition will include at least one function from each of these three categories

An Example of the ADT

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (INT_MAX) on the computer

functions:

for all x, y is Nat_Number, TRUE, FALSE is Boolean and where . +, -, <, and == are the usual integer operations

Nat_NoZero() ::= 0

Boolean Is_Zero(x) ::= if (x) return FALSE

Contd….

```
Nat_No Add(x, y)          ::= if ((x+y)<= INT_MAX) return x+ y
                               else return INT_MAX
Boolean Equal(x, y)       ::= if (x== y) return TRUE
                               else return FALSE
Nat_No Successor(x)       ::= if (x== INT_MAX) return x
                               else return x+ 1
Nat_No Subtract(x, y)     ::= if (x< y) return 0
                               else return x-y
end Natural_Number
```

Exercises

Write a C function to find sum of array elements using recursive function.

Define a ADT for complex numbers

Arrays

Array: a set of index and value

data structure

For each index, there is a value associated with that index.

representation (possible)

implemented by using consecutive memory.

Arrays- ADT

Structure *Array* is
objects: A set of pairs <*index*, *value*> 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, … , n-1} for one dimension,
{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions,
etc.

Functions:

for all A $\in$ Array, i $\in$ index, x $\in$ item, j, size $\in$ integer

Array Create(j, list) ::= return an array of j dimensions where list is a j-tuple whose ith element is the size of the ith dimension. Items are undefined.

Item Retrieve(A, i) ::= if (i $\in$ index) return the item associated with index value i in array A else return error

Array Store(A, i, x) ::= if (i in index) return an array that is identical to array A except the new pair <i, x> has been inserted else return error

end array

Arrays in C

int list[5], *plist[5];

list[5]: five integers

list[0], list[1], list[2], list[3], list[4]

***plist[5]: five pointers to integers**

plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

list[0]	**base address = α**
list[1]	**α + sizeof(int)**
list[2]	**α + 2*sizeof(int)**
list[3]	**α + 3*sizeof(int)**
list[4]	**α + 4*size(int)**

Arrays in C *(Continued)*

Compare int *list1 and int list2[5] in C.

Same:	**list1 and list2 are pointers.**
Difference:	**list2 reserves five locations.**

Notations:

list2 - a pointer to list2[0]
(list2 + i) - a pointer to list2[i] (&list2[i])
***(list2 + i) - list2[i]**

Example: 1-dimension array addressing

```
int one[] = {0, 1, 2, 3, 4};
```

Goal: print out address and value

```
void print1(int *ptr, int rows)
{
/* print out a one-dimensional array using a pointer */
        int i;
        printf(“Address Contents\n”);
        for (i=0; i < rows; i++)
                printf(“%8u%5d\n”, ptr+i, *(ptr+i));
        printf(“\n”);
}
```

call print1(&one[0], 5)

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

Dynamically Allocated Arrays

One dimensional arrays

```
int i,n,*list;
//read n
//if(n<1) error
MALLOC(list,n*sizeof(int))
```

Two Dimensional Arrays

```
int x[3][5]

int **pa;
int r,c; //read r and c
pa= create2d(r,c);
//read i and j
p[i][j]=60;
```

Creating array function

```
int **create2d(int r, int c)
{
  MALLOC(x,r*sizeof(*x));

  for(i=0;i<r;i++)
      {
            MALLOC(x[i],c*sizeof(**x));
      }
  return x;
}
```

Calloc()

To allocate n blocks of memory and initialize

References

1. https://www.cise.ufl.edu/class/cop3275fa16/lectures/Recursion.ppt
2. http://www.sanfoundry.com/c-program-binary-search-recursion/
3. https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt
4. https://www.csie.ntu.edu.tw/~ds/ppt/ch2/chapter2.PPT

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

Data Abstraction

Types of data

- All programming language provide at least minimal set of predefined data type, plus user defined types

Data types of C

- Char, int, float, and double
  - may be modified by short, long, and unsigned
- Array, struct, and pointer

Data Type

Definition

- **A *data type* is a collection of *objects* and a set of *operations* that act on those objects**

Example of "int"

- **Objects: 0, +1, -1, ..., Int_Max, Int_Min**
- **Operations: *arithmetic*(+, -, *, /, and %), *testing*(equality/inequality), *assigns*, *functions***

Define operations

- **Its *name*, possible *arguments* and *results* must be specified**

The design strategy for representation of objects

- ***Transparent* to the user**

Abstract Data Type

Definition

- **An *abstract data type*(*ADT*) is a *data type* that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operation.#**

Why abstract data type ?

- **implementation-independent**

Classifying the Functions of a Data Type

Creator/constructor:

- Create a new instance of the designated type

Transformers

- Also create an instance of the designated type by using one or more other instances

Observers/reporters

- Provide information about an instance of the type, but they do not change the instance

Notes

- An ADT definition will include at least one function from each of these three categories

An Example of the ADT

structure Natural_Number is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (INT_MAX) on the computer

functions:

for all x, y is Nat_Number, TRUE, FALSE is Boolean and where . +, -, <, and == are the usual integer operations

Nat_NoZero() ::= 0

Boolean Is_Zero(x) ::= if (x) return FALSE

Contd….

```
Nat_No Add(x, y)          ::= if ((x+y)<= INT_MAX) return x+ y
                              else return INT_MAX
Boolean Equal(x, y)       ::= if (x== y) return TRUE
                              else return FALSE
Nat_No Successor(x)       ::= if (x== INT_MAX) return x
                              else return x+ 1
Nat_No Subtract(x, y)     ::= if (x< y) return 0
                              else return x-y
end Natural_Number
```

Exercises

Write a C function to find sum of array elements using recursive function.

Define a ADT for complex numbers

Arrays

Array: a set of index and value

data structure

For each index, there is a value associated with that index.

representation (possible)

implemented by using consecutive memory.

Arrays- ADT

Structure *Array* is
objects: A set of pairs <*index*, *value*> 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, … , n-1} for one dimension,
{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions,
etc.

Functions:

for all A $\in$ Array, i $\in$ index, x $\in$ item, j, size $\in$ integer

Array Create(j, list) ::= return an array of j dimensions where list is a j-tuple whose ith element is the size of the ith dimension. Items are undefined.

Item Retrieve(A, i) ::= if (i $\in$ index) return the item associated with index value i in array A else return error

Array Store(A, i, x) ::= if (i in index) return an array that is identical to array A except the new pair <i, x> has been inserted else return error

end array

Arrays in C

int list[5], *plist[5];

list[5]: five integers

list[0], list[1], list[2], list[3], list[4]

***plist[5]: five pointers to integers**

plist[0], plist[1], plist[2], plist[3], plist[4]

implementation of 1-D array

list[0]	**base address = α**
list[1]	**α + sizeof(int)**
list[2]	**α + 2*sizeof(int)**
list[3]	**α + 3*sizeof(int)**
list[4]	**α + 4*size(int)**

Arrays in C *(Continued)*

Compare int *list1 and int list2[5] in C.

Same: list1 and list2 are pointers.
Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]
(list2 + i) - a pointer to list2[i] (&list2[i])
***(list2 + i) - list2[i]**

Example: 1-dimension array addressing

int one[] = {0, 1, 2, 3, 4};

Goal: print out address and value

```
void print1(int *ptr, int rows)
{
/* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}
```

Dynamically Allocated Arrays

One dimensional arrays

```
int i,n,*list;
//read n
//if(n<1) error
MALLOC(list,n*sizeof(int))
```

Two Dimensional Arrays

```
int x[3][5]

int **pa;
int r,c; //read r and c
pa= create2d(r,c);
//read i and j
p[i][j]=60;
```

Creating array function

```
int **create2d(int r, int c)
{
  MALLOC(x,r*sizeof(*x));

  for(i=0;i<r;i++)
      {
            MALLOC(x[i],c*sizeof(**x));
      }
  return x;
}
```

Calloc()

calloc(): To allocate n blocks of memory and initialize to 0.

```
#define CALLOC(p,n,s)
If(!((p)=calloc(n,s)))
{
  printf(stdeff,"no memory");
  exit(0);
}
```

Realloc()

realloc() : Used to resize memory space

realloc(p,newsize)

References

1. https://www.cise.ufl.edu/class/cop3275fa16/lectures/Recursion.ppt
2. http://www.sanfoundry.com/c-program-binary-search-recursion/
3. https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt
4. https://www.csie.ntu.edu.tw/~ds/ppt/ch2/chapter2.PPT

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

Arrays in C *(Continued)*

Compare int *list1 and int list2[5] in C.

Same: list1 and list2 are pointers.
Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]
(list2 + i) - a pointer to list2[i] (&list2[i])
***(list2 + i) - list2[i]**

Example: 1-dimension array addressing

int one[] = {0, 1, 2, 3, 4};

Goal: print out address and value

```
void print1(int *ptr, int rows)
{
/* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}
```

Dynamically Allocated Arrays

One dimensional arrays

```
int i,n,*list;
//read n
//if(n<1) error
MALLOC(list,n*sizeof(int))
```

Two Dimensional Arrays

```
int x[3][5]
or
int **x;
int r,c; //read r and c
x= create2d(r,c);
//read i and j
x[i][j]=60;
```

Creating array function

```
int **create2d(int r, int c)
{
   int **x;
   MALLOC(x,r*sizeof(*x));

   for(i=0;i<r;i++)
         {
                  MALLOC(x[i],c*sizeof(**x));
         }
   return x;
}
```

Calloc()

calloc(): To allocate n blocks of memory and initialize to 0.

```
#define CALLOC(p,n,s)
if(!((p)=calloc(n,s)))
{
  printf("no memory");
  exit(0);
}
```

Realloc()

realloc() : Used to resize memory space allocated for a pointer

realloc(p,newsize)

Structures (records)

```
struct {
        char name[10];
        int age;
        float salary;
        } person;

strcpy(person.name, “james”);
person.age=10;
person.salary=35000;
```

Structures: Exercises

How will you store the data in structure (after reading from user)?

How will you compare whether two structure variable contents are same or not?

Define structure for date.

Create structure data type

```
typedef struct human_being {
        char name[10];
        int age;
        float salary;
        };
or
typedef struct {
        char name[10];
        int age;
        float salary
        } human_being;

human_being person1, person2;
```

Unions

Similar to struct, but only one field is active.

Example: Add fields for male and female.

```
typedef struct gender_type {
        enum gender_field {female, male} gender;
        union {
                int children;
                int beard;
                } u;
        };
typedef struct human_being {
        char name[10];
        int age;        float salary;
        date dob;       gender_type gender_info;
        }
```

```
human_being person1, person2;
person1.gender_info.gender=male;
person1.gender_info.u.beard=FALSE;
```

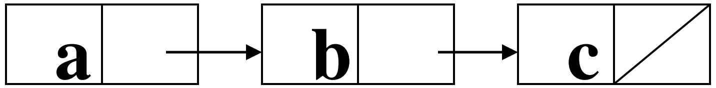
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef         struct list {
        char data;
        list *link;
        }

list item1, item2, item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node



References

1. https://www.cise.ufl.edu/class/cop3275fa16/lectures/Recursion.ppt
2. http://www.sanfoundry.com/c-program-binary-search-recursion/
3. https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt
4. https://www.csie.ntu.edu.tw/~ds/ppt/ch2/chapter2.PPT

Data Structures

UNIT-1

PREPARED BY

PARKAVI.A, BE,ME,(PH.D)

ASSISTANT PROFESSOR, CSE DEPARTMENT,

RAMAIAH INSTITUTE OF TECHNOLOGY,BANGALORE

Unions

Similar to struct, but only one field is active.

<u>Example</u>: Add fields for male and female.

```
typedef struct gender_type {
        enum gender_field {female, male} gender;
        union {
                int children;
                int beard;
                } u;
        };
typedef struct human_being {
        char name[10];
        int age;        float salary;
        date dob;       gender_type gender_info;
        }
```

```
human_being person1, person2;
person1.gender_info.gender=male;
person1.gender_info.u.beard=FALSE;
```

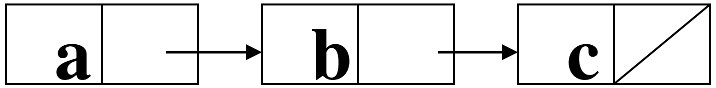
Self-Referential Structures

One or more of its components is a pointer to itself.

```
typedef struct list {
        char data;
        list *link;
        };

list item1, item2, item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node



Polynomial Addition

Polynomials $A(X)=3X^{20}+2X^5+4$, $B(X)=X^4+10X^3+3X^2+1$

$$p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$$

Structure ***Polynomial*** **is**

objects: a set of ordered pairs of $\langle e_i, a_i \rangle$ **where** a_i **in** ***Coefficients*** **and** e_i **in** ***Exponents*****,** e_i **are integers >= 0**

functions:

for all ***poly, poly1, poly2*** **are** ***Polynomial*****,** ***coef*** **stores** ***Coefficients*****,** ***expon stores Exponents***

Polynomial **Zero( )** ::= **return the polynomial,** $p(x) = 0$

Boolean **IsZero(*poly*)** ::= **if (*poly*) return** ***FALSE*** **else return** ***TRUE***

Contd….

Coefficient Coef(poly, expon) ::= if (expon is in poly) return its coefficient else return Zero

Exponent Lead_Exp(poly) ::= return the largest exponent in poly

Polynomial Attach(poly,coef, expon) ::= if (expon is in poly) return error else return the polynomial poly with the term <coef, expon> inserted

Polynomial Remove(poly, expon) ::= if (expon is in poly) return the polynomial poly with the term whose exponent is expon deleted
else return error

Polynomial SingleMult(poly, coef, expon) ::= return the polynomial poly • coef • x^{expon}

Polynomial Add(poly1, poly2) ::= return the polynomial poly1 +poly2

Polynomial Mult(poly1, poly2) ::= return the polynomial poly1 • poly2

Polynomial Addition : Implementation

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

Code for polynomial addition

```
/* d =a + b, where a, b, and d are polynomials */
d = Zero( )
while (! IsZero(a) && ! IsZero(b)) do {
  switch COMPARE (Lead_Exp(a), Lead_Exp(b))
 {
     case -1: d =
        Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
        b = Remove(b, Lead_Exp(b));
        break;
```

Contd…

```
case  0:
sum = Coef (a, Lead_Exp (a)) + Coef ( b, Lead_Exp(b));
        if (sum) {
           Attach (d, sum, Lead_Exp(a));
           a = Remove(a , Lead_Exp(a));
           b = Remove(b , Lead_Exp(b));
           }
         break;
```

Contd….
advantage: easy implementation
disadvantage: waste space when sparse

```
case 1: d =
        Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
        a = Remove(a, Lead_Exp(a));
      }
  }
insert any remaining terms of a or b into d
```

<u>*Program 2.4 :Initial version of *padd* function</u>

Data structure 2: Use one global array to store all polynomials

<u>Array representation of two polynomials</u>

$A(X)=2X^{1000}+1$

$B(X)=X^4+10X^3+3X^2+1$

	starta	*finisha*	*startb*		*finishb*	*avail*	
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	**1**	**2**	**3**	**4**	**5**	**6**

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

Storage requirements: start, finish, 2*(finish-start+1)
nonparse: twice as much as (1) when all the items are nonzero

```
MAX_TERMS 100 /* size of terms array */
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

Add two polynomials: D = A + B

```
void padd (int starta, int finisha, int startb, int finishb, int * startd, int *finishd)
{
/* add A(x) and B(x) to obtain D(x) */
    float coefficient;
   *startd = avail;
   while (starta <= finisha && startb <= finishb)
     switch (COMPARE(terms[starta].expon,
                                  terms[startb].expon)) {
      case -1: /* a expon < b expon */
             attach(terms[startb].coef, terms[startb].expon);
             startb++;
             break;
```

```
case  0: /* equal exponents */
                coefficient = terms[starta].coef + terms[startb].coef;
                if (coefficient)
                   attach (coefficient, terms[starta].expon);
                starta++;
                startb++;
                break;
case 1: /* a expon > b expon */
          attach(terms[starta].coef, terms[starta].expon);
          starta++;
}
```

```
/* add in remaining terms of  A(x) */
for( ; starta <= finisha; starta++)
     attach(terms[starta].coef, terms[starta].expon);
/* add in remaining terms of B(x) */
for( ; startb <= finishb; startb++)
    attach(terms[startb].coef, terms[startb].expon);
*finishd =avail -1;
}
```

```
void attach(float coefficient, int exponent)
{
/* add a new term to the polynomial */
   if (avail >= MAX_TERMS) {
      fprintf(stderr, "Too many terms in the polynomial\n");
      exit(1);
     }
     terms[avail].coef  = coefficient;
     terms[avail++].expon = exponent;
}
```

***Program :Function to add anew term**

References

1. https://www.cise.ufl.edu/class/cop3275fa16/lectures/Recursion.ppt
2. http://www.sanfoundry.com/c-program-binary-search-recursion/
3. https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt
4. https://www.csie.ntu.edu.tw/~ds/ppt/ch2/chapter2.PPT