RAMAIAH
Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Data Structures
Course Code: CS32
Credits: 3:1:1

Term: September – December 2020

Faculty:
Vandana S Sardar
Mamatha Jadhav V

Basic Concepts

The contents in this presentation are selected from
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C", Universities Press, 2008

Pointers

- *
  - dereferencing operator, indirection operator
- &
  - address operator
- int i, *pi;
- pi = &i;
- *pi = 10; or i = 10;

Dynamic Memory Allocation

```
int i, *pi;
float f, *pf;
pi = (int *) malloc (sizeof(int));
pf = (float *) malloc (sizeof(float));
*pi = 1024;
*pf = 3.14;
printf ("Integer = %d; float = %f\n", *pi, *pf);
free (pi);
free (pf);
```

Dynamic Memory Allocation

```
#define MALLOC(p, s)    \
if (!((p) = malloc(s))) {    \
    fprintf(stderr, "Insufficient memory");    \
    exit (EXIT_FAILURE);    \
}
……
int *pi;
……
MALLOC(pi, sizeof(int));
*pi = 1024;
```

Now memory can be initialized using following:
MALLOC(pi,sizeof(int));
MALLOC(pf,sizeof(float))

Dynamic Memory Allocation

DANGLING REFERENCE

• Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. This is called a *dangling reference*.

POINTERS CAN BE DANGEROUS

1) Set all pointers to NULL when they are not actually pointing to an object. This makes sure that you will not attempt to access an area of memory that is either

→ out of range of your program or

→ that does not contain a pointer reference to a legitimate object

2) Use explicit type casts when converting between pointer types.

pi=malloc(sizeof(int)); //assign to pi a pointer to int

pf=(float*)pi; //casts an 'int' pointer to a 'float' pointer

Algorithm Specification

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

Criteria:

- Input: Zero/more inputs
- Output: At least one output
- Definiteness: clear and unambiguous
- Finiteness: terminate after a finite number of steps
- Effectiveness: instruction is basic enough to be carried out

Algorithm Specification: Selection Sort

Selection Sort Algorithm:

for(i=0;i<n;i++) {

Examine list[i] to list[n-1] and suppose that the smallest integer is at list[min]

Interchange list[i] and list[min]

}

Algorithm Specification: swap function and macro

Function call: swap(&a,&b);

```
void swap(int *p,int *q) //both parameters are pointers to ints
{
int temp=*p; //declares temp and assigns to it the contents of what p points to
*p=*q; //stores what q points to into the location where p points
*q=temp; //places the contents temp in location pointed to by q
}
#define SWAP(x,y,t) ((t)=(x), (x)=(y),(y)=(t))
```

Algorithm Specification: Selection Sort function

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

Algorithm Specification: Binary Search

Searching a sorted list:

```
while(there are more integers to check)   {
   middle= (left+right)/2;
  if(searchnum<list[middle])
      right= middle-1;
   else if(searchnum==list[middle])
      return middle;
 else    left=middle+1;
}
```

Algorithm Specification: Compare function and macro

Comparison of Integers:

```
int compare(int x, iny y)
{  if(x<y)  return -1;
  else if (x==y)  return 0;
  else  return 1;
}
#define COMPARE(x,y) (((x)<(y))? -1: ((x)==(y))?0:1)
```

Algorithm Specification: 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: left= middle+ 1;
break;
case 0: return middle;
case 1: right= middle- 1;
}
}
return -1;
}
```

RAMAIAH
Institute of Technology

Algorithm Specification: Binary Search, Recursive function

```
int binsearch(int list[], int searchnum, int left, int right)
{
// search list[0]<= list[1]<=...<=list[n-1] for searchnum

int middle;

if (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;
}
```

Algorithm Specification: Permutation of a string

For list =”a” : Only one permutation a.

For list= “ab” “ Two permutations: ab, ba

For list =“abc” “: Six permutations

abc, acb, bac, bca, cab, cba

Function call:

perm(list,0,n-1);

Algorithm Specification: Perm function

```
void perm(char *list, int i,int n)
{
int j,temp;
if(i==n)      {
            for(j=0;j<=n;j++)
                    printf("%c",list[j]);
   printf("  ");
   }
   else
   {
   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

- The process of separating logical properties of data from implementation details of data.

Data Type

• A data type is a collection of objects and a set of operations that act on those objects.

• For e.g., data type 'int' consists of
→ objects {0,+1,-1,+2,-2. . . . }
→ operations such as arithmetic operators + - * /

ADT (ABSTRACT DATA TYPE)

- This is a data type that is organized in such a way that

→ specification of objects is separated from representation of objects
→ specification of operations on objects is separated from implementation of operations.

For example:
Specification: The specification of operations on objects consists of names of functions, type of arguments and return type. But, no information is given about how to implement in a programming language. So, specifications are implementation independent.

Implementation: The implementation of operations consists of a detailed algorithm using which we can code (i.e. functions) using any programming language(C or C++).

ADT (ABSTRACT DATA TYPE)

ADT definition contains 2 main sections:

→ Objects & → Functions

- Functions of a data type can be classified into

1) Constructor: These functions create a new instance of the designated type.

2) Transformers: These functions create an instance of the designated type, generally by using one or more other instances.

3) Reporters: These functions provide information about an instance of the type, but they do not change the instance.

RAMAIAH
Institute of Technology

Data Abstraction

ADT *NaturalNumber* is

objects: An ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer.

functions:

for all $x, y \in$ *NaturalNumber*; TRUE, FALSE $\in$ *Boolean*

and where +, −, <, ==, and = are the usual integer operations

Zero( ) : *NaturalNumber*	::=	0
IsZero(*x*) : *Boolean*	::=	**if** (x == 0) *IsZero* = **true**
		else *IsZero* = **false**
Add(*x*, *y*) : *NaturalNumber*	::=	**if** ($x + y$ <= MAXINT) *Add* = $x + y$
		else *Add* = MAXINT
Equal(*x*, *y*) : *Boolean*	::=	**if** (x == y) *Equal* = TRUE
		else *Equal* = FALSE
Successor(*x*) : *NaturalNumber*	::=	**if** (x == MAXINT) *Successor* = x
		else *Successor* = $x + 1$
Subtract(*x*, *y*) : *NaturalNumber*	::=	**if** ($x < y$) *Subtract* = 0
		else *Subtract* = $x - y$

end *NaturalNumber*

Arrays

ADT for array

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 ∈ Array, *i* ∈ *index*, x ∈ *item*, *j*, *size* ∈ integer

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

Item Retrieve(A, *i*) ::= **if** (*i* ∈ *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

One-dimensional array can be declared as follows:

int list[5]; *//array of 5 integers*

int *plist[5]; *//array of 5 pointers to integers*

- Compiler allocates 5 consecutive memory-locations for each of the variables 'list' and 'plist'.
- Address of first element list[0] is called base-address.
- Memory-address of list[i] can be computed by compiler as

  α+i*sizeof(int) where α =base address

Arrays in C: A simple Program

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main (void)
{
        for (i = 0; i < MAX_SIZE; i++)
        input[i] = i;
        answer = sum(input, MAX_SIZE);
        printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for (i = 0; i < n; i++)
                tempsum += list[i];
        return tempsum;
}
```

Arrays in C: Printing array

```
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");
}

void main()
{
    int one[] = {0, 1, 2, 3, 4};
    print1(&one[0], 5)
}
```

Dynamically Allocated Arrays

ONE-DIMENSIONAL ARRAYS:

When writing programs, sometimes we cannot reliably determine how large an array must be.

- A good solution to this problem is to

→ defer this decision to run-time &

→ allocate the array when we have a good estimate of required array-size

- Dynamic memory allocation can be performed as follows:

```
int i,n,*list;
printf("enter the number of numbers to generate");
scanf("%d",&n);
if(n<1)
{
        printf("improper value");
        exit(0);
}
MALLOC(list, n*sizeof(int));
```

Dynamically Allocated Arrays: 2D arrays

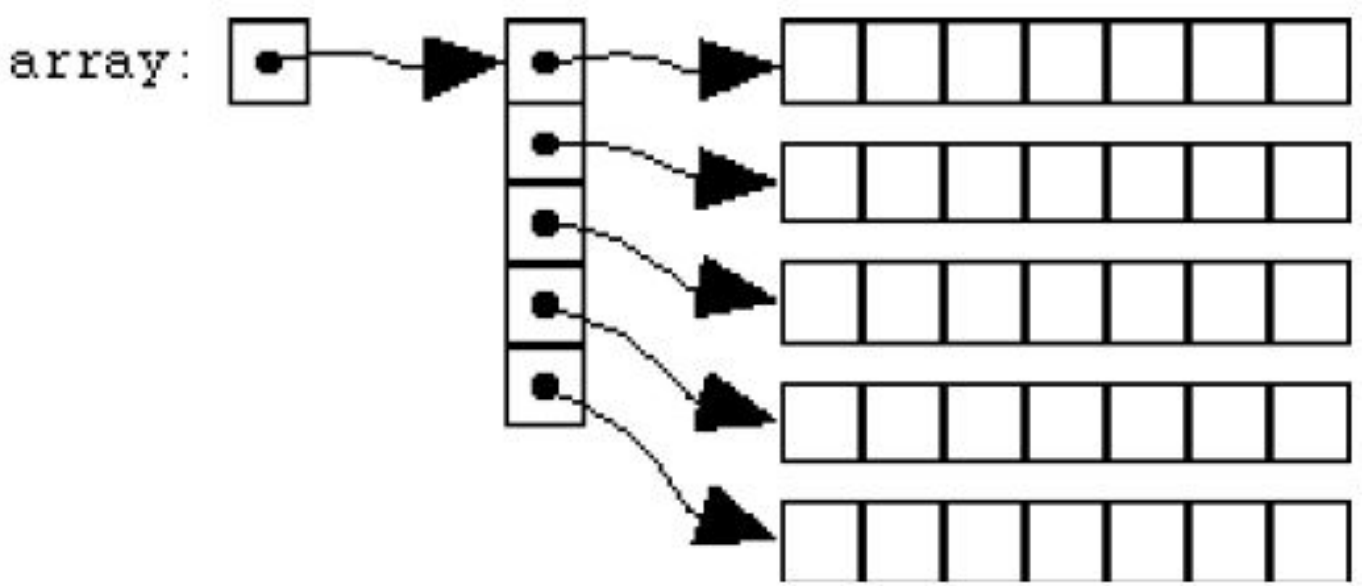
These are created by using the concept of array of arrays.

- A 2-dimensional array is represented as a 1-dimensional array in which each element has a pointer to a 1-dimensional array as shown below

```
int x[5][7];        //we create a 1-dimensional array x whose length is 5;
                    //each element of x is a 1-dimensional array whose length is 7.
```

- Address of x[i][j] = x[i]+j*sizeof(int)

Dynamically Allocated Arrays: 2D arrays



Array-of-arrays representation

Dynamically Allocated Arrays: 2D arrays

```
#include <stdlib.h>
int **array;
array = malloc(nrows * sizeof(int *));
if(array == NULL)
{
        printf("out of memory\n");
        exit or return
}
for(i = 0; i < nrows; i++)
{
        array[i] = malloc(ncolumns * sizeof(int));
        if(array[i] == NULL)
        {
                printf("out of memory\n");
                exit or return
        }
}
```

Dynamically Allocated Arrays: 2D arrays

```
int ** make2DArray(int rows, int cols)
{
int **x,i;
/* get memory for row pointers */
     MALLOC(x, rows * sizeof(*x));
/* get memory for each row  */
for(i = 0; i < rows; i++)
    MALLOC(x[i], cols*sizeof(**x));
}
```

Other dynamic allocation functions: calloc()

⟶ Allocates user-specified amount of memory &
⟶ Initializes the allocated memory to 0.

• On successful memory-allocation, it returns a pointer to the start of the new block and on failure, it returns the value NULL.

• Memory can be allocated using calloc as shown below:

```
int *x;
x=calloc(n, sizeof(int));        //where n=array size
```

Other dynamic allocation functions: calloc()

To create clean and readable programs, a CALLOC macro can be created as shown below:

```
#define CALLOC(p,n,s) \
if(!((p)=calloc(n,s))) \
{ \
fprintf(stderr,"insufficient memory"); \
exit(EXIT_FAILURE); \
}
```

Other dynamic allocation functions: realloc()

This function resizes memory previously allocated by either malloc or calloc.

For example,

realloc(p,s); *//this changes the size of memory-block pointed at by p to s.*

- When s>oldSize, the additional s-oldSize have an unspecified value and

when s<oldSize, the rightmost oldSize-s bytes of old block are freed.

- On successful resizing, it returns a pointer to the start of the new block and on failure, it returns the value NULL.

Other dynamic allocation functions: realloc()

- To create clean and readable programs, the REALLOC macro can be created as shown below

```
#define REALLOC(p,n,s) \
if(!((p)=realloc(n,s))) \
{ \
fprintf(stderr,"insufficient memory"); \
exit(EXIT_FAILURE); \
}
```

Structures

- This is collection of elements whose data types are different.

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

- Dot operator(.) is used to access a particular member of the structure.

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

- Variables can be declared as follows:

```
humanBeing person1,person2;
```

Structures

Structures cannot be directly checked for equality or inequality. So, we can write a function to do this:

```
if(humansEqual(person1,person2))
printf("two human beings are same");
else
printf("two human beings are different");
-----------------------------------------------------------------------------------------
int humansEqual(humanBeing person1,humanBeing person2)
{
if(strcmp(person1.name,person2.name))
return 0;

if(person1.age!=person2.salary)
return 0;

if(person.salary!=person2.salary)
return 0;

return 1;
}
```

Structures

```
typedef struct
{
int month;
int day;
int year;
}date;

typedef struct
{
char name[10];
int age;
float salary;
date dob;
}humanBeing;
```

Person1.dob.month=08
Person1.dob.day=07
Person1.dob.year=1992

Self-referential structures

• A self-referential structure is one in which one or more of its components is a pointer to itself.

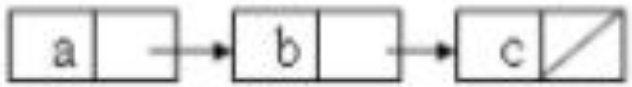
• These require dynamic storage management routines (malloc & free) to explicitly obtain and release memory.

```
typedef struct
{
char data;
struct list *link; //list is a pointer to a list structure
}list;
```

• Consider three structures and values assigned to their respective fields:

```
list item1,item2,item3;

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



Unions

This is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time.

```
typedef struct
{
enum tagField{female,male} gender;
typedef union
{
int children;
int beard;
}u;

}genderType;

typedef struct
{
char name[10];
int age;
float salary;
date dob;
genderType genderInfo;
}humanBeing;

humanBeing person1,person2;
```

Unions

We can assign values to person1 and person2 as:

person1.genderInfo.gender=male;

person1.genderInfo.u.beard=FALSE;

and

person2.genderInfo.gender=female;

person1.genderInfo.u.children=3;

Internal implementation of structures

- The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.

- Structures must begin and end on the same type of memory boundary. For ex, an even byte boundary (2, 4, 6 or 8).

Representation of Multidimensional arrays

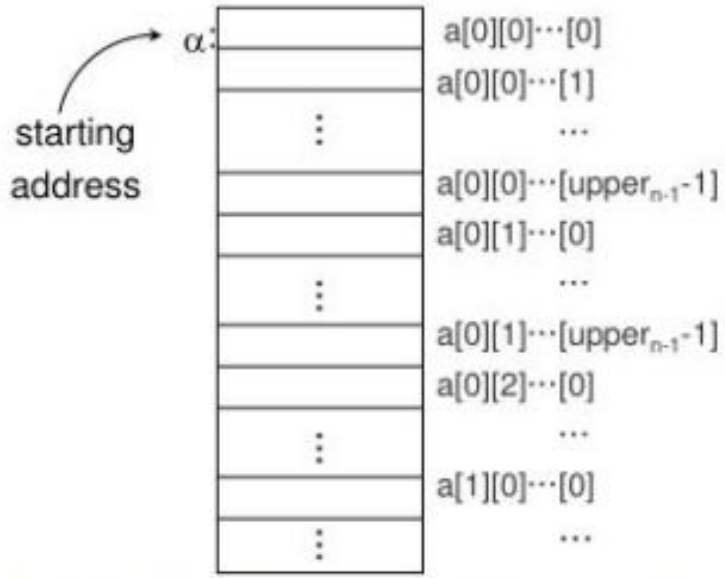
- how to state n-dimensional array into 1-dimensional array ?
- how to retrieve arbitrary element in $a[upper_0][upper_1]\cdots[upper_{n-1}]$
- **the number of elements in the array**

$$\prod_{i=0}^{n-1} upper_i$$

- **e.g.) a[10][10][10]**
  - **10*10*10 = 1000 (units)**

Representation of Multidimensional arrays

- Represent multidimensional array by
  - what order ?
    - **row-major-order**
  - store multidimensional array by rows



Representation of Multidimensional arrays

- How to retrieve
  - starting-address + offset-value
  - assume α: starting-address

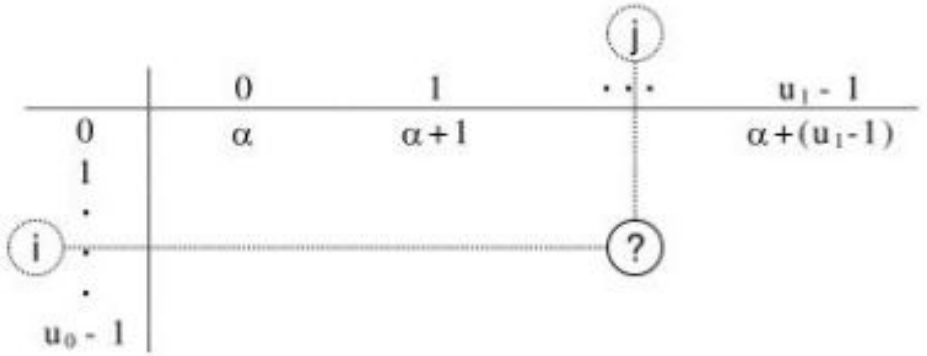
- **1-dimensional** array $a[u_0]$

a[0]	: α
a[1]	: $\alpha + 1$
⋮	⋮
$a[u_0-1]$	: $\alpha + (u_0 - 1)$

$a[i] = \alpha + i$

Representation of Multidimensional arrays

- **2-dimensional** array $a[u_0][u_1]$



- $a[i][j] = \alpha + i \cdot u_1 + j$

Representation of Multidimensional arrays

- **3-dimensional** array $a[u_0][u_1][u_2]$

$$a[i][j][k] = \alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2 + k$$
$$= \alpha + u_2[i \cdot u_1 + j] + k$$

- **General** case $a[u_0][u_1]\cdots[u_{n-1}]$

$$a[i_0][i_1]\cdots[i_{n-1}]$$
$$= \alpha + \sum_{j=0}^{n-1} i_j \cdot a_j \quad \begin{cases} a_j = \prod_{k=j+1}^{n-1} u_k \quad 0 < j < n-1 \\ a_{n-1} = 1 \end{cases}$$

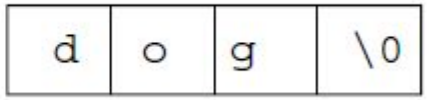
Strings

ADT *String* is
objects: a finite set of zero or more characters.
functions:
for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

String Null(m)	::=	**return** a string whose maximum length is m characters, but is initially set to *NULL*. We write *NULL* as "".
Integer Compare(s, t)	::=	**if** s equals t **return** 0 **else if** s precedes t **return** −1 **else return** +1
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 characters of s at positions $i, i + 1, \cdots, i + j - 1$. **else return** *NULL*.

Strings in C

```
#define MAX_SIZE 100
char s[MAX_SIZE] = {"dog"};
```



C String Functions

char *strcat(char *dest, char *src)

char *strncat(char *dest, char *src, int n)

int strcmp(char *str1, char *str2)

int strncmp(char *str1, char *str2, int n)

char *strcpy(char *dest, char *src)

char *strncpy(char *dest, char *src, int n)

C String Functions

size_t strlen(char *s)

char *strchr(char *s, int c)

char *strtok(char *s, char *delimiters) : return a token from s, token is surrounded by delimiters

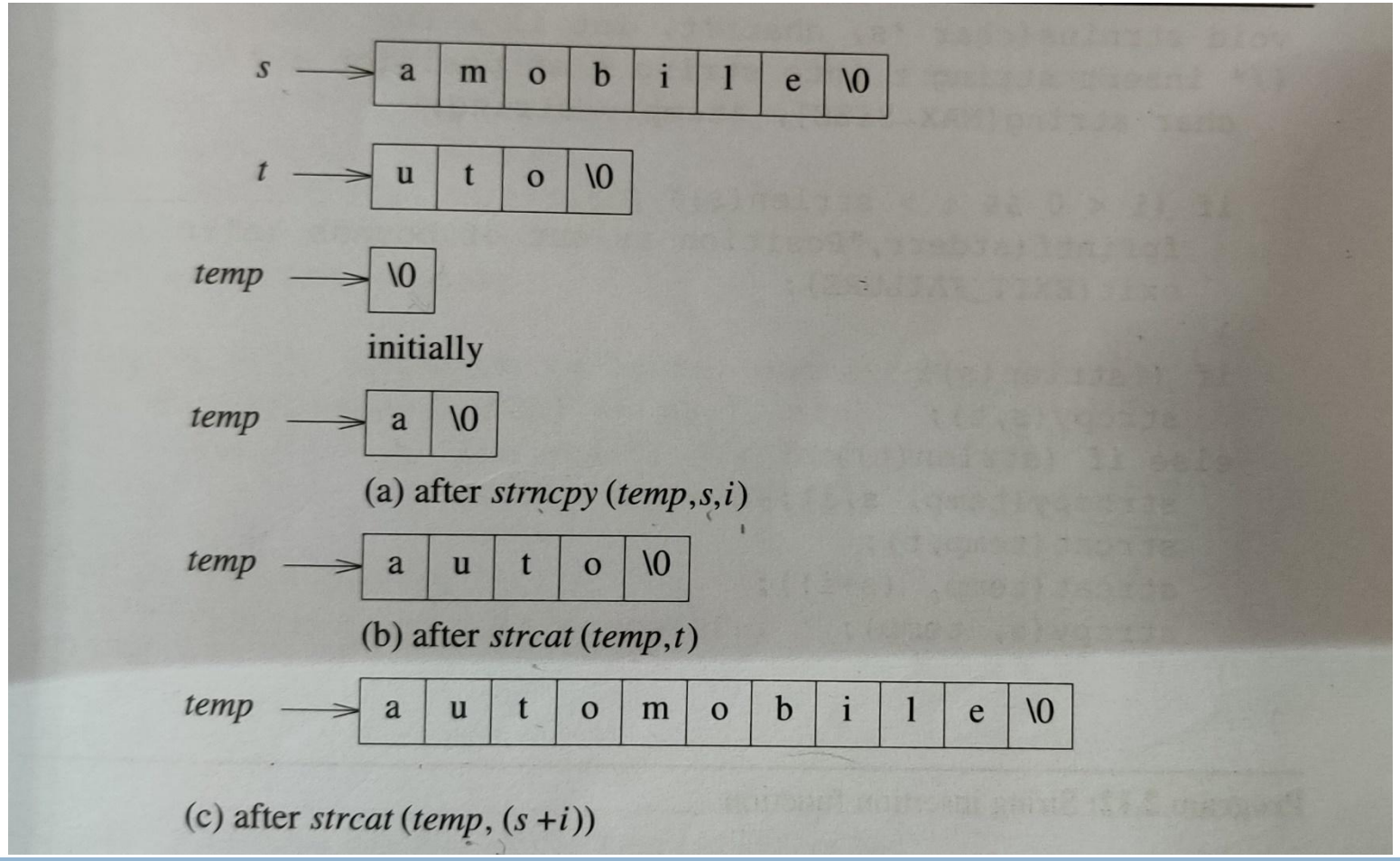
char *strstr(char *s, char *pat)

size_t strspn(char *s, char *spanset) : scan s for characters in spanset, return length of span

size_t strcspn(char *s, char *spanset): scan s for characters not in spanset, return length of span

char *strpbrk(char *s, char *spanset) : scan s for characters in spanset, return pointer to first occurrence of a character from spanset.

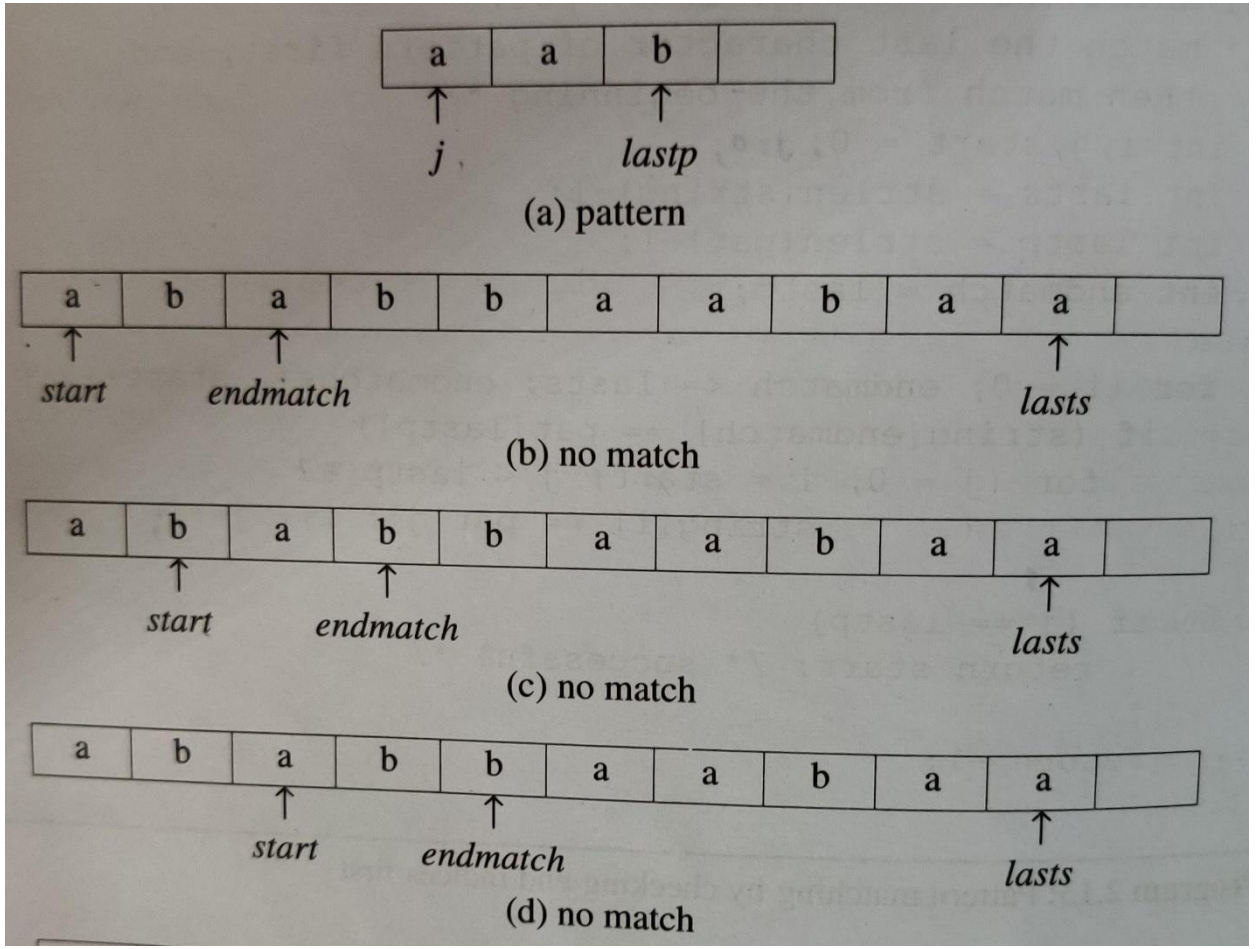
String Insertion Example



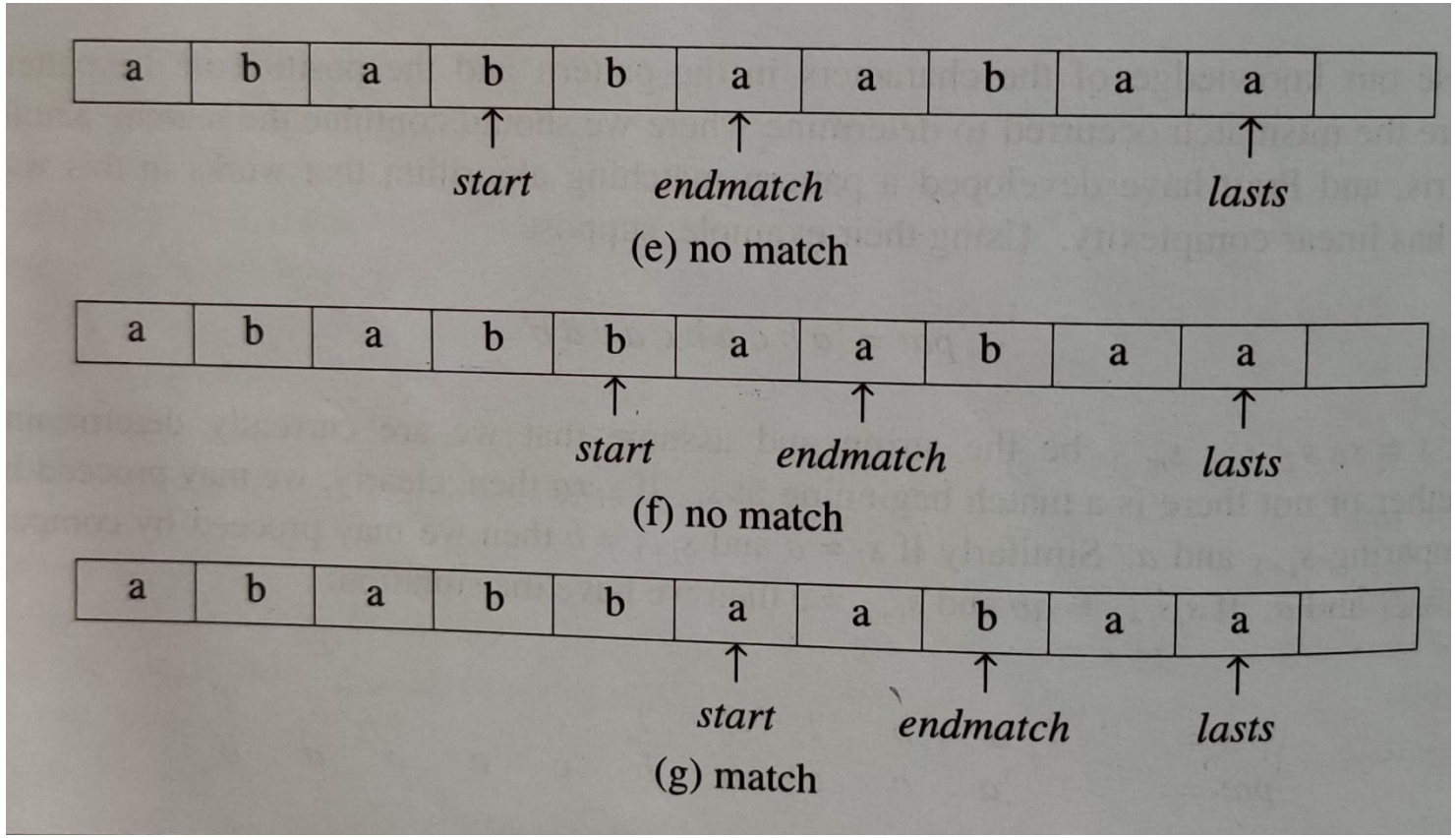
String Insertion Function

```
void stringins (char *s, char *t, int i)
{
char string[MAX_SIZE], *temp=string;
if (i<0 || i>strlen(s)) {
        fprintf (stderr, “position is out of bounds\n”);      exit(EXIT_FAILURE));        }
}
if (!strlen(s))
      strcpy(s,t);
else
if (strlen(t))  {
strncpy(temp, s, i);
strcat(temp, t);
strcat(temp, s+i);
strcpy(s, temp);
}
}
```

Pattern matching by checking end indices first



Pattern matching by checking end indices first



Pattern matching by checking end indices first

```
int nfind(char *string, char *pat)
{/* match the last character of pattern first, and
    then match from the beginning */
  int i,j,start = 0; j=0;
  int lasts = strlen(string)-1;
  int lastp = strlen(pat)-1;
  int endmatch = lastp;

  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; /* successful */
    }
    return -1;
}
```