DATA STRUCTURES AND APPLICATIONS

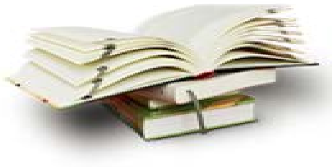
Module 1_3

Introduction

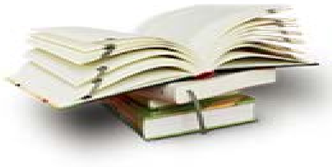
Dr. Pushpalatha K
Associate Professor
Sahyadri College of Engineering & Management

STRINGS

- Each programming languages contains a character set that is used to communicate with the computer. The character set include the following:
  - Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  - Digits: 0 1 2 3 4 5 6 7 8 9
  - Special characters: + - / * ( ) , . $ = ‘ _ (Blank space)
- **String**: A finite sequence S of zero or more Characters is called string.
- **Length**: The number of characters in a string is called length of string.
- **Empty or Null String**: The string with zero characters.



- **Concatenation:**
  - Let S1 and S2 be the strings.
  - The string consisting of the characters of S1 followed by the character S2 is called Concatenation of S1 and S2.
    - Ex: ‘THE’ || ‘END’ = ‘THEEND’
    - ‘THE’ || ‘ ‘|| ‘END’ = ‘THE END’
- **Substring: A string Y is called substring of a string S if there exist string X and Z such that S = X || Y || Z**
  - If X is an empty string, then Y is called an Initial substring of S, and Z is an empty string then Y is called a terminal substring of S.
    - Ex: ‘BE OR NOT’ is a substring of ‘TO BE OR NOT TO BE’
    - ‘THE’ is an initial substring of ‘THE END’



STRINGS IN C

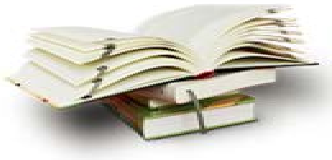
- **In C, the strings are represented as character arrays terminated with the null character \0.**
  - Ex:

```
#define MAX_SIZE 100 /* maximum size of string */
char i[MAX_SIZE] = {“ice”};   // char i[]={“ice”};
```

i[0]	i[1]	i[2]	i[3]
i	c	e	\0

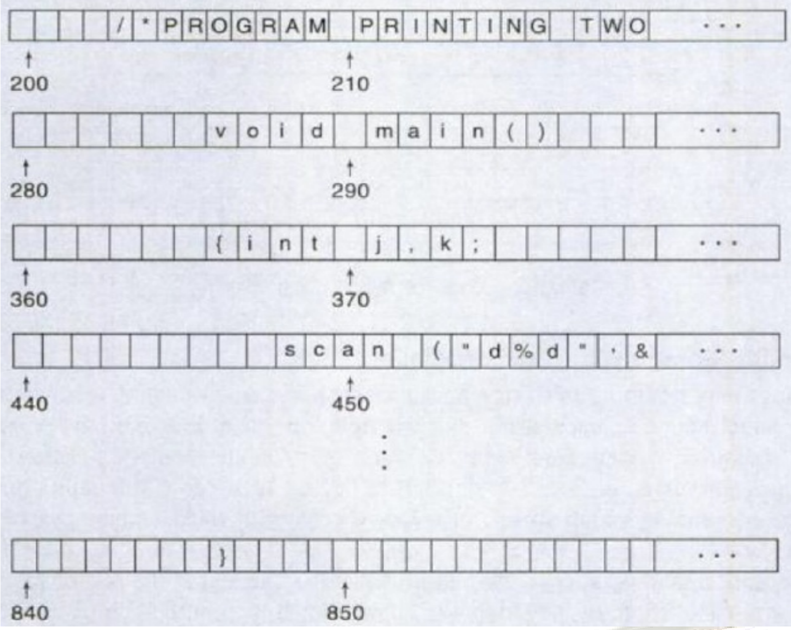
```
char c [MAX_SIZE] = {“cream”};  // char c [] = {“cream”};
```

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]
c	r	e	a	m	\0



STORING STRINGS

- Strings are stored in three types of structures
  - Fixed length structures
  - Variable length structures with fixed maximum
  - Linked structures
- Record Oriented Fixed length storage
  - In fixed length structures each line of print is viewed as a record, where all have the same length i.e., where each record accommodates the same number of characters
    - Ex: input is a program.

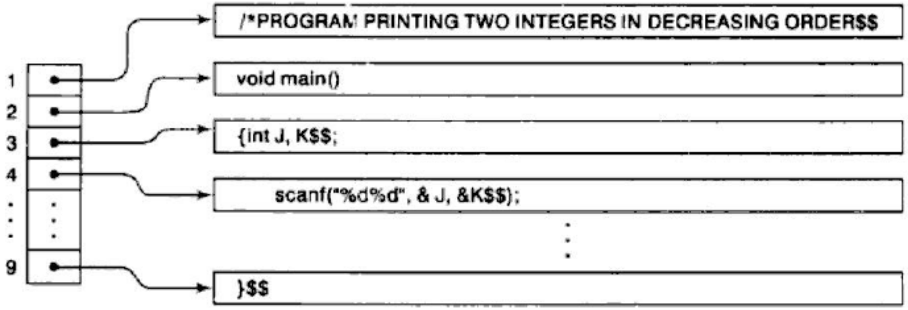


- **Advantages:**
  1. The ease of accessing data from any given record
  2. The ease of updating data in ny given record

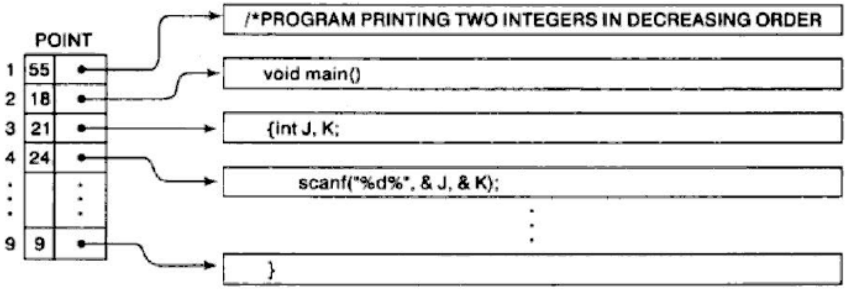
     (as long as the length of the new data does’nt exceed the record length)
- **The main disadvantages are**
  1. Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.
  2. Certain records may require more space than available
  3. When the correction consists of more or fewer characters than the original text, changing a misspelled word requires record to be changed

Variable length structures with fixed maximum

– The storage of variable-length strings in memory cells with fixed lengths can be done in two general ways
  - One can use a marker, such as two dollar signs ($$), to signal the end of the string
  - One can list the length of the string—as an additional item in the pointer array

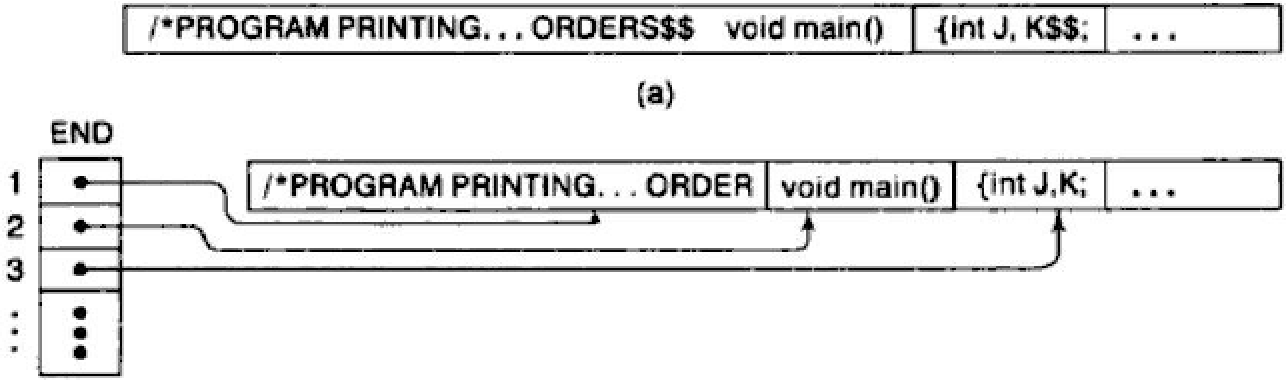


(a) Records with sentinels.



(b) Record whose lengths are listed.

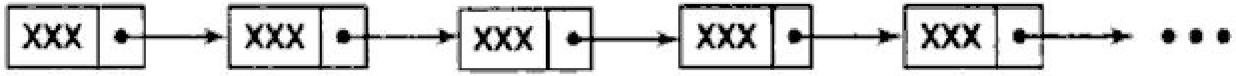
- In an other method strings can be stored one after another by using some separation marker, such as the two dollar sign ($$) or by using a pointer giving the location of the string.



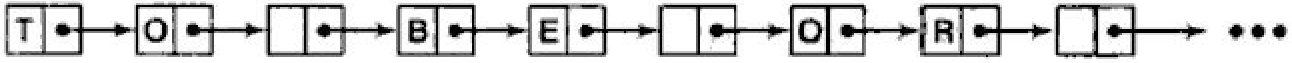
- **Advantages**
  - Saves space and are sometimes used in secondary memory when records are relatively permanent and require little changes.
- **Disadvantages**
  - These methods are usually inefficient when the strings and their lengths are frequently being changed.

Linked Storage

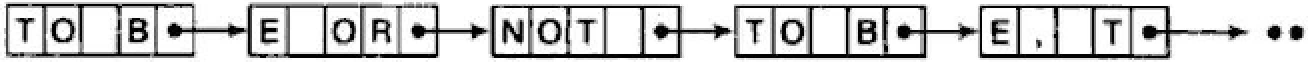
- In computers words are processed frequently.
  - Fixed length memory cells cannot be featsible for frequent processing of strings
- In most extensive word processing applications, strings are stored by means of linked lists.
- An one way linked list is a linearly ordered sequence of memory cells called nodes,
  - each node contains an item called a link, which points to the next node in the list, i.e., which consists the address of the next node.



- **Strings may be stored in linked list as follows:**
  - **Each memory cell is assigned one character or a fixed number of characters and a link contained in the cell gives the address of the cell containing the next character or group of character in the string.**



(a) One character per node.



(b) Four characters per node.

String as ADT(Abstract data type)

- **A string data type should include operations to**
  - Return n^{th} character of a string
  - Set the n^{th} character of a string
  - Find the length a string
  - Concatenate two strings
  - Copy two strings
  - Delete part of strings
  - Modify and compare the strings

GETCHAR(str,n)	Returns the n^{th} character in the string
PUTCHAR(str,n,c)	Sets the n^{th} character in the string to c
LENGTH(str)	Returns the number of characters in the string
POS(str1,str2)	Returns the position of the first occurrence of str2 found in str1, or 0 if no match
CONCAT(str1,str2)	Rerurns a new string consisting of characters in str1 followed by characters in str2
SUBSTRING(str1,i,m)	Returns a substring of length i starting at position i in string str
DELETE(str, i, ,m)	Deletes m characters from srr starting at position i
INSERT(strl, str2, i)	Changes str1into a new string with str2 inserted in position i
COMPARE(strl. str2)	Returns an integer indicating whether strl > str2

STRING OPERATIONS

- **Substrings**
  – groups of consecutive elements in a string (such as words phrases and sentences), called substrings
- **Accessing a substring from a given string requires :**
  – The name of the string or the string itself
  – The position of the first character of the substring in the given string
  – The length of the substring or the position of the last character of the substring.

Syntax: SUBSTRING (string, initial, length)

- Ex: SUBSTRING ('TO BE OR NOT TO BE’, 4, 7) = 'BE OR N’
- SUBSTRING ('THE END', 4, 4) = ' END'

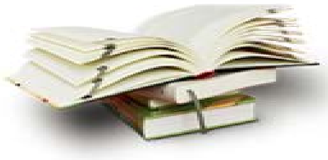
implementation of SUBSTRING function in C

```
# include <stdio. h>
#include <conio.h>
void main ()
{
        char S(80]={“SAHYADRI  COLLGE”);
        char *SUBSTR(char* , int, int);
        clrscr();
        printf ( "STRING = %S”, S);
        printf  (“\NSUBSTRING  (S,  4,  7)  =  %s",
             SUBSTR( S, 4,7) ) ;
        getch ();
}
```

```
char *SUBSTR(char  *str,int i,int j)
{
     int k,m=0;
     char resstr[80] ;
     for(k=i-1;k<=i+j-1-l;k++)
      {
         resstr[m]=str[k];
         m=m+1;
      }
    resstr[m]=‘\0’;
    return resstr;
}
```

Indexing

- **Indexing also called pattern matching, refers to finding the position where a string pattern P first appears in a given string text T.**
  - **Syntax: INDEX (text, pattern)**
    - **If the pattern P does not appears in the text T, then INDEX is assigned the value 0.**
    - **The arguments “text” and “pattern” can be either string constant or string variable.**
  - **Let T= “THE HOUSE IS BEHIND THE PARK”**
    - **INDEX(T, ”THE”) returns 1**
    - **INDEX(T, ”HORSE”) returns 0**
    - **INDEX(T, ” THE ”) returns 21**



implementation of INDEX function in C

```
# include<stdio.h>
# include <conio.h>
# include <string.h>
void main()
{
	 char T[100]={“THE HOUSE IS BEHIND THE PARK”};
	int INDEX ( char *,char * ) ;
	printf ("T = is'" ,T);
	printf("\n INDEX(T, ‘THE’) = %d", INDEX(T, "THE"));
	printf ( "\n\nINDEX (T, ‘HORSE ') = %d", INDEX (T, " HORSE " ));
	printf("\n INDEX(T, ‘ THE ’) = %d", INDEX(T, “ THE "));
}
```

```
int INDEX (char * str1, char * str2)
{
int  m, n;
int idx , flag;
for(m=0;m<strlen(str1);m++)
 {
      idx=m; flag=1;
      for(n=0;n<strlen(str2);n++)
      {
            if(str1[m+n]!=str2[n])
            { flag=0; break;}
      }
      if (flag==0) continue;
      else return(idx);
}
If (m==strlen(str1)
return(-1);
}
```

Concatenation

- **Let S1 and S2 be string.**
  - The concatenation of S1 and S2 which is denoted by S1 // S2, is the string consisting of the characters of S1 followed by the character of S2.
  - Let S1 = 'MARK' and S2= ‘TWAIN' then
    - $S_1 // S_2$ = ‘MARKTWAIN’
  - Concatenation is performed in C language using strcat function:

strcat (S1, S2);

  - Concatenates string S1 and S2 and stores the result in S1
    - strcat ( ) function is part of the string.h header file; hence it must be included at the time of pre- processing

Length

- The number of characters in a string is called its length.

  Syntax: LENGTH (string)

    - Ex: LENGTH (‘computer’) = 8

- String length is determined in C language using the strlen( ) function:

  X = strlen ("sunrise");

  – strlen function returns an integer value 7 and assigns it to the variable X

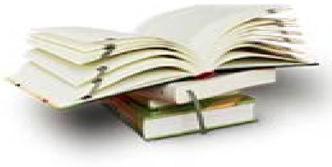
- strlen is also a part of string.h, hence the header file must be included at the time of pre-processing.

Pattern Matching Algorithm

- **Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T.**
  - **Two pattern matching algorithms**
- **First pattern matching Algorithm**
  - **This algorithm compares a** given **pattern P** with each of the substrings of T, moving from left to right, until a match is found.

$$W_K = \text{SUBSTRING}(T, K, \text{LENGTH}(P))$$

  - Where, W_K denote the substring of **T** having the **same length as P** and beginning with the **K^th^ character of T**.



STEPS

- First compare P, character by character, with the first substring, W_1.
  - If all the characters are the same, then $P = W_1$ and so P appears in T and INDEX (T, P) = 1.
- If that some character of P is not the same as the corresponding character of W_1. Then $P \neq W_1$
- Immediately move on to the next substring, W_2 That is, compare P with W2.
  - If $P \neq W_2$ then compare P with W_3 and so on.
- The process stops, When P is matched with some substring W_K and so P appears in T and INDEX(T,P) = K or When all the W_K'S with no match and hence P does not appear in T.
- The maximum value MAX of the subscript K is equal to LENGTH(T)LENGTH(P) +1.

Ex: If P is 4 char string, T is 20 char string

P = P[1]P[2]P[3]P[4]

T = T[1]T[2]T[3]T[4]T[5]T[6]….T[18]T[19]T[20]

P is compared with each of the 4 character substrings of T

MAX= 20- 4 + 1 = 17 ie total 17 substring of T

W1=T[1]T[2]T[3]T[4]

W2=T[2]T[3]T[4]T[5]

W17=T[17]T[18]T[19]T[20]

First pattern matching algorithm

P and T are strings with lengths R and S respectively and are stored as arrays with one character per element. *This algorithm finds the INDEX of P in T*

Step 1. [Initialize] Set K:=1 , MAX := S – R + 1

Step 2. Repeat Step 3 to 5 while K≤ MAX

Step 3. Repeat for L = 1 to R [Test each character of P]

If P[L] ≠ T [K+L-1], then Goto Step 5

[End of inner loop]

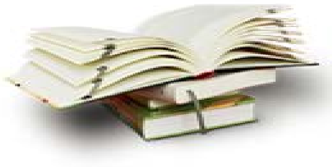
Step 4. [Success] Set INDEX:= K and Exit

Step 5. Set K:=K+1

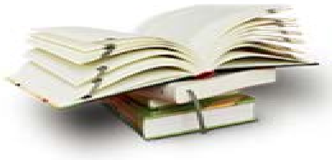
[End of Step 2 outer loop]

Step 6. [Failure] Set INDEX :=0

Step 7. Exit



- The complexity of this pattern matching algorithm is measured by the number C of comparisons between characters in the pattern P and characters of the text T.
- Let N_K is the number of comparisons that takes place in the inner loop when P is compared with W_K. Then,
  - $C = N_1 + N_2 + \ldots + N_L$
  - L is the position L in T where P first appears or L=MAX if P does not appear in T.
- Note: Characters denoted in lower case and exponents are used to denote repetition
  - Ex: $a^2b^3ab^2$ for aabbbabb
  - $(cd)^3$ for cdcdcd



- Use slow pattern matching algorithm to find number of comparisons C and also find INDEX of P in text T.

a) P = aaba and T = cdcd..cd = $(cd)^5$

Total substrings of length(P)=10-4+1=7

Total No. of comparisons = 1+1+1+1+1+1+1=7

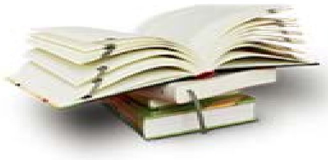
b) P = aaba and T = abababab….

c) P = aaab and T = aa….aaa= a^{20}

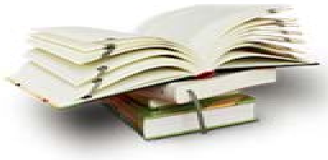
- Let p is an r-character string
- T is an s-character string
- Total data size of the algorithm
  - n=r+s;
- The complexity of this pattern matching algorithm is $O(n^2)$

Second Pattern Matching Algorithm

- The second pattern matching algorithm uses a table which is derived from a particular pattern P but is independent of the text T.
- For definiteness, suppose P = aaba
- This algorithm contains the table that is used for the pattern P = *aaba.*
- LET $T=T_1T_2T_3$....
  - Where T_i denotes the i^{th} character of the text T
- Suppose T=aa.., then T has one of following forms
  - T=aab…, T=aaa…, T=aax…
    - Where x is any character different from a or b



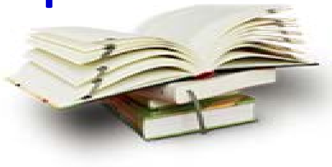
- The table is obtained as follows.
  - Let Q_i denote the initial substring of P of length I
    - hence Q_0 = L, Q_1 = a, Q_2 = a2, Q_3 = aab, Q_4 = aaba = P
      - Here Q_0 = A is the empty string
  - The rows of the table are labeled by these initial substrings of P, excluding P itself.
  - The columns of the table are labeled a, b and x,
    - x represents any character that doesn't appear in the pattern P.



- Let f be the function determined by the table let $f(Q_i, t)$ denote the entry in the table in row Q_i and column t (where t is any character).
  - This entry $f(Q_i, t)$ is defined to be the largest Q that appears as a terminal substring in the string Q_it the concatenation of Q_i and t.

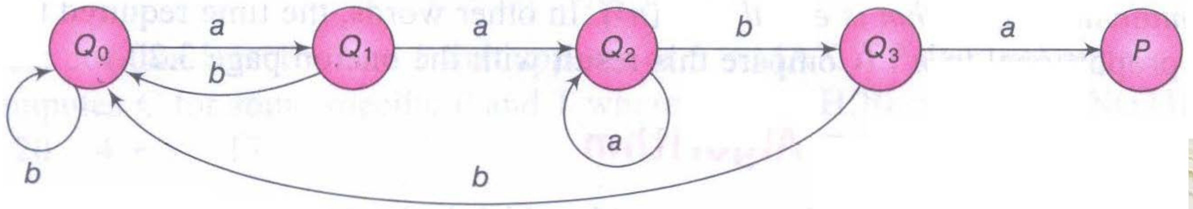
	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

- **Ex:**
  - a^2 is the largest Q that is a terminal substring of $Q_2a = a^3$
    - so $f(Q_2, a) = Q_2$
  - A is the largest Q that is a terminal substring of $Q_1b = ab$
    - so $f(Q_1, b) = Q_0$
  - a is the largest Q that is a terminal substring of $Q_0a = a$
    - so $f(Q_0, a) = Q_1$
  - A is the largest Q that is a terminal substring of $Q_3a = a^3bx$
    - so $f(Q_3, x) = Q_0$
- **Although $Q_1 = a$ is a terminal substring of $Q_2a = a^3$, f(Q2, a) = Q_2 because Q_2 is also a terminal substring of $Q_2a = a^3$ and Q2 is larger than Q1.**
- **We note that $f(Q_i, x) = Q_0$ for any Q, since x does not appear in the pattern P**
- **The column corresponding to x is usually omitted from the table.**



Pattern matching Graph

- The graph is obtained with the table as follows.
  - First, a node in the graph corresponding to each initial substring Q_i of P.
    - The Q's are called the states of the system, and Q_0 is called the initial state.
- Second, there is an arrow (a directed edge) in the graph corresponding to each entry in the table.
  - Specifically, if f(Qi, t) = Qj
    - then there is an arrow labeled by the character t from Qi to Qj
- For example, f(Q2, b) = Q3 so there is an arrow labeled b from Q2 to Q3



(b) Pattern matching graph

The second pattern matching algorithm for the pattern P = *aaba*

- Let $T = T_1\ T_2\ T_3\ ...\ T_N$ denote the n-character-string text which is searched for the pattern P.
- Beginning with the initial state Q_0 and using the text T, we will obtain a sequence of states S_1, S_2, S_3, ... as follows.
  - Let $S_1 = Q_0$ and read the first character T_1. The pair (S_1, T_1) yields a second state S_2; that is, $F(S_1, T_1) = S_2$, Read the next character T_2, The pair (S_2, T_2) yields a state S_3, and so on.
- There are two possibilities:
  - Some state S_K = P, the desired pattern. In this case, P does appear in T and its index is K - LENGTH(P).
  - No state S_1, S_2, ... , S_{N+1} is equal to P. In this case, P does not appear in T.

Algorithm: (PATTERN MATCHING)

The pattern matching table F(Q1, T) of a pattern P is in memory, and the input is an N-character string $T = T_1\ T_2\ T_3 \ldots\ldots T_N$. The algorithm finds the INDEX of P in T.

1. [Initialize] set K: =1 ans $S_1 = Q_0$
2. Repeat steps 3 to 5 while $S_K \neq P$ and $K \leq N$
3. Read T_K
4. Set $S_{K+1} := F(S_K, T_K)$ [finds next state]
5. Set K: = K + 1 [Updates counter]

   [End of step 2 loop]

6. [Successful ?]

   If $S_K = P$, then :

   INDEX = K – LENGTH (P)

   Else

   INDEX = 0

   [End of IF structure]

7. Exit.

The complexity is O(n)

Abstract data type String

- **ADT String is**

  Objects: a finite set of zero or more characters

  Functions:

  For all s, t∈String,I,j,m∈ non negative integers

String Null(m)	::=	return a string whose maximum length is m characters, but 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 Subset(s, i, j)	::=	if ((j>0) && (i+j-1)<Length(s)) return the string containing the characters of s at position i, i+1, ..., i+j-1. else return NULL.

C String functions

Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if $str1 < str2$; 0 if $str1 = str2$; > 0 if $str1 > str2$
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if $str1 < str2$; 0 if $str1 = str2$; > 1 if $str1 > str2$
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*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*

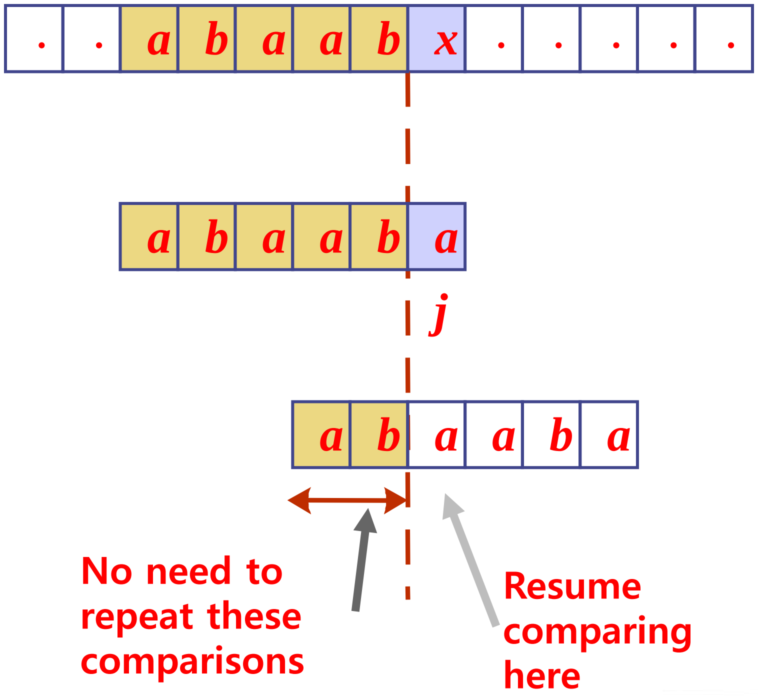
Knuth–Morris–Pratt algorithm

- The algorithm was conceived in 1974 by Donald Knuth and Vaughan Pratt, and independently by James H. Morris. The three published it jointly in 1977
- Problem Defination
  - Given a string ‘S’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘S’ and if ‘p’ does occur then returning position in ‘S’ where ‘p’ occurs.

Drawbacks of the O(mn) Approach

- if ‘m’ is the length of pattern ‘p’ and ‘n’ the length of string ‘S’, the matching time is of the order O(mn). This is a certainly a very slow running algorithm.
  - the elements of ‘S’ with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations.
  - These repetitive comparisons lead to the runtime of O(mn).

- Knuth-Morris-Pratt’s (KMP) algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.
- A matching time of O(n) is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

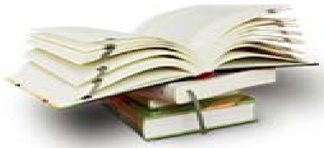


- The Knuth-Morris-Pratt (KMP) string searching algorithm keeps track of information gained from previous comparisons.
- A failure function (f) is computed that indicates how much of the last comparison can be reused if it fails.
- Specifically, f is defined to be the longest prefix of the pattern P[0,..,j] that is also a suffix of P[1,..,j]

Computing the Failure Function

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

j	0	1	2	3	4	5
P(i)	a	b	a	c	a	b
F(j)	-1	-1	0	-1	0	1



Algorithm *failureFunction*(*P*)

```
F[0] ← -1
j ← 1
i ← F[0]
while j < m
if P[j] = P[i+1]
    {we have matched i + 1 chars}
    F[j] ←  i + 1
    j ← j + 1
    i ← i + 1
else if  i > =0 then
    {use failure function to shift P}
    i ← F[i ]
else
    F[j] ← -1 { no match }
    j ← j + 1
```

Steps to compute failure function

1. F[0]=-1
2. j=1,i=-1, P[j]=b, P[i+1]=a,
   - F[j]=F[1]=-1, j=j+1
3. j=2, i=-1, P[j]=a, P[i+1]=a,
   - F[j]=i+1=0, i=i+1=0, j=j+1=3
4. j=3, i=0, P[j]=c, P[i+1]=b,
   - i=F[i-1]=F[0]=-1
5. j=3, i=-1, P[j]=c, P[i+1]=a,
   - F[j]=F[3]=-1, j=j+1=4
6. j=4, i=-1, P[j]=a, P[i+1]=a,
   - F[j]=i+1=0, i=i+1=0, j=j+1=5
7. j=5, i=0, P[j]=b, P[i+1]=b,
   - F[j]=i+1=1, i=i+1=1, j=j+1=6

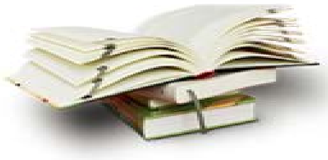
j	0	1	2	3	4	5
P(i)	a	b	a	c	a	b
F(j)	-1	-1	0	-1	0	1

KMP string matching algorithm

```
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);

}
```



Algorithm *KMPMatch(T, P)*

```
F ← failureFunction(P)
i ← 0
j ← 0
while j < n
if T[i] = P[j]
    if  j = j – 1
        return  i – j { match }
    else
        i ← i + 1
        j ← j + 1
else
    if  j > 0
        j ← F[j - 1]+1
    else
        i ← i + 1
return  –1 { no match }
```

Example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T	a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a	b	b
P	a	b	a	c	a	b														
					a	b	a	c	a	b										
						a	b	a	c	a	b									
										a	b	a	c	a	b					
											a	b	a	c	a	b