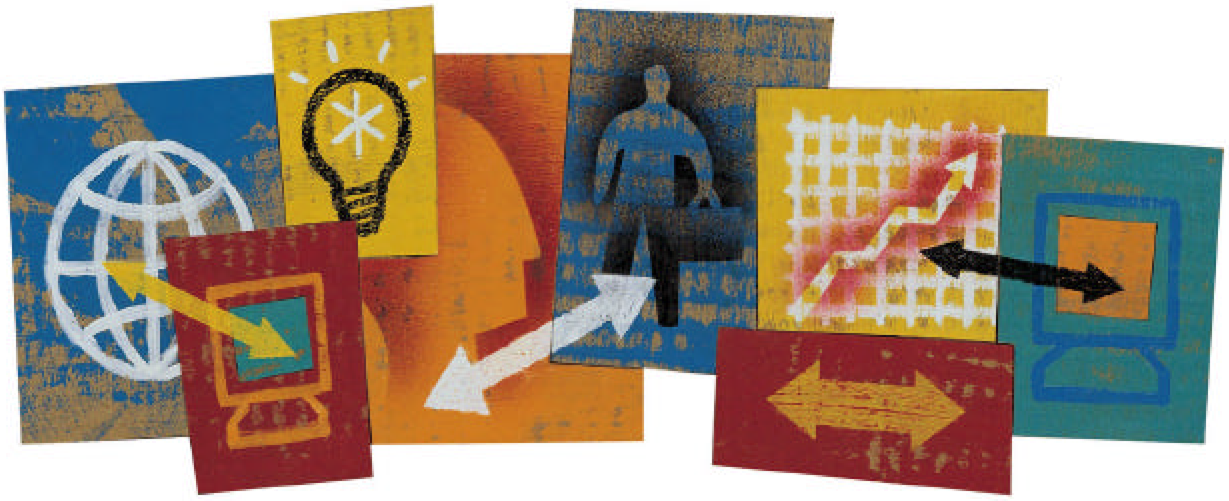


Data structures & C

Version: DATASTRUCTURE/HANDOUT/0704/1.0
Date: 08-07-04

Cognizant Technology Solutions

Cognizant Technology Solutions
500 Glen Pointe Center West
Teaneck, NJ 07666
Ph: 201-801-0233
www.cognizant.com

TABLE OF CONTENTS

Introduction

About this Module

This module provides participants with the basic knowledge and skills that are needed to understand the fundamentals of Data Structures and C.

Target Audience

Entry Level Trainees

Module Objectives

The main objective of this module is,

- To gain knowledge on the very basics of a Data structures
- How to implement them in a Programming Languages

After Completion of this module, the trainee will be able to:

- Explain the various Data structures in C
- Write programs using the data structures

Pre-requisite

The participants should know basic programming in C.

Chapter 1: Introduction To Data Structures

Learning Objectives

After completing this chapter, you will be able to:

- ✍✍ Appreciate the need of data Structures

Overview of Data Structure

Data Structure is about study of data and algorithms. This encompasses the study in the following areas:

- ✍✍ Machines that holds data / executing algorithms
- ✍✍ Languages for describing data manipulation / algorithms
- ✍✍ Foundation of algorithms
- ✍✍ Structures for representing data &
- ✍✍ Analysis of algorithms

A data type is a well-defined collection of data with a well-defined set of operations on it. A data structure is an actual implementation of a particular abstract data type. Abstraction can be thought of as a mechanism for suppressing irrelevant details while at the same time emphasizing relevant ones.

Data Structures & C

Data Structures is not specific to any language. It is more about writing efficient algorithms. It can be implemented in any language. For our purpose we have chosen to implement all the algorithms in 'C' language.

What is an algorithm?

An algorithm is a finite set of instructions, which accomplish a particular task. Every algorithm must satisfy the following criteria:

- ✍✍ Input – Zero or more quantities which are externally supplied
- ✍✍ Output – at least one quantity is produced

- Definiteness – Each instruction must be clear and unambiguous
- Finiteness – If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps
- Effectiveness – Every instruction must be sufficiently basic that a person using only pencil and paper can in principle carry it out. If is not enough that each operation be definite but it must also be feasible.

The logical or mathematical model of organization of data is called a data structure. A data structure describes not just a set of data but also how they are related. Data Structures and algorithms should be thought of as a unit, neither one makes sense without the other.

Types of Data Structures

There are several data structures that are available and the choice of an appropriate one depends on the requirement. Some of the data structures that are commonly used include:

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

Static vs. Dynamic Data Structures

A static data structure has a fixed size. This meaning is different from the meaning of the static modifier. Arrays are static; once you define the number of elements it can hold, the number doesn’t change. A dynamic data structure grows and shrinks at execution time as required by its contents. A dynamic data structure is implemented using linked lists.

SUMMARY

- An algorithm is a finite set of instructions, which accomplish a particular task.
- A data structure is an actual implementation of a particular abstract data type.
- There are several data structures that are available and the choice of an appropriate one depends on the requirement.
- A static data structure has a fixed size.
- A dynamic data structure is implemented using linked lists.

Test your Understanding

1. Which of the following data structures does OS use for process scheduling?
    a) Stack
    b) Queue
    c) Linked List
    d) Binary Tree

2. Non-linear data structures are,
    a) Stack & Queue
    b) Tree & Graphs
    c) Tree & Stack
    d) Tree & Linked List

Chapter 2: Introduction To Arrays

Learning Objective

After completing this chapter, you will be able to

✍✍ Give a brief overview of arrays

✍✍ Explain the memory layout of single and multi dimensional arrays

✍✍ List the merits and de-merits of arrays

Introduction to Arrays

The simplest type of data structure is a linear array and most often it is the only data structure that is provided in any programming language. An array can be defined as a collection of homogenous elements, in the form of index/value pairs, stored in consecutive memory locations. An array always has a predefined size and the elements of an array are referenced by means of an index / subscript.

Memory Organization for an array

If A is an array of 5 integer elements declared as:

int A[5] = {1, 2, 3, 4, 5};

Then the memory organization for this array is as shown below:

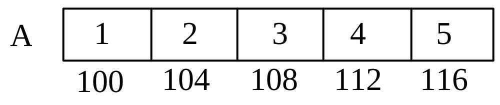


Fig 2-1 Memory organization of an Array

The elements of the array are always stored in consecutive memory locations. The elements are accessed by adding the index with the base address and this being a pointer arithmetic will always fetch the appropriate address depending on the type of data.

Multi Dimensional Arrays

An array can be of more than one dimension. There are no restrictions to the number of dimensions that we can have. But as the dimensions increase the memory requirements increase drastically which can result in shortage of memory. Hence a multidimensional array must be used with utmost care.

For example the following declaration

```
int a[50][50][50]
```

will result in occupying 50 * 50 * 50 = 1,25,000 * 4 = 5,00,000 bytes of memory.

Memory Organization

The memory arrangement for a multidimensional array is very similar to linear arrays. Only the way in which we access the data varies. i.e. if it is a 2 dimensional array for example, then it can be accessed by using 2 indexes one which represents the row and the other representing the column. If A is a 2 dimensional array defined like

```
int arr[3][3];
```

The maximum elements that this array can hold is 3 * 3 which is 9 elements and the elements of this array can be referred as

arr[0][0], arr[0][1], arr[0][2], arr[1][0], arr[1][1]….arr[2][2].

As array is used in all the programming languages we shall move on to other data structures.

Advantages

- Searching is faster as elements are in continuous memory locations
- Memory management is taken care of by the compiler itself

Limitations

- Number of elements must be known in advance
- Inserting and Deletions are costlier as it involves shifting the rest of the elements

SUMMARY

- The simplest type of data structure is a linear array
- An array can be defined as a collection of homogenous elements, in the form of index/value pairs, stored in consecutive memory locations. An array always has a predefined size and the elements of an array are referenced by means of an index / subscript.
- An array can be of more than one dimension. There are no restrictions to the number of dimensions that we can have.

Test your Understanding

1. Write a function to accept a string and a group of characters to search in the input string. If the input group of characters exists in the input string, display an appropriate message along with the starting position in the string.

2. Write a function that concatenates 2 input strings and concatenates into one large string and displays the result.

3. Using pointers swap two numbers entered by user.

Exercise 1a

4. An interesting word puzzle is acrostic. An extremely innovative acrostic is,

   ROTAS
   OPERA
   TENET
   AREPO
   SATOR

Write a C program to read in 5x5 acrostic and check whether it reads the same horizontally or vertically

Chapter 3: Introduction To Pointers

Learning Objective

After completing this topic, you will be able to:

- Understand and use pointer in programs
- Exemplify the fundamentals of pointers, to describe how to use pointers, and to list the advantages and limitations of pointers

Introduction to Pointers

The study of data structures in C involves extensive usage of pointers. So it is absolutely necessary to understand the basics of this most powerful feature of ‘C’ called pointers. Before getting into pointers it is necessary for anyone to understand some of the internal operations that happen in the system. Whenever a variable is declared what actually happens? For example if we declare a variable as,

```
int a = 10;
```

The compiler allocates memory for that variable and stores the value of that variable in that memory location. The 1000 here represents the memory location for the variable a.

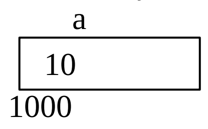


Fig 3-1 Memory Representation of a variable

Now use the following printf statement to see the output

```
printf(“ Value = %d” , a);
printf(“ Address of a = %u”, &a);
```

The first statement prints the value 10 while the second one prints the address of the variable a, in this case 1000.

Having understood this now let’s see what is a pointer?

- A pointer is nothing but a variable, which can hold the address of another variable.
- How to declare a variable as pointer?
- Just like any other declaration the pointer declaration will have the data type followed by the variable name with an * in between to denote that this is a pointer type variable.

```
i.e. int* p;
```

The above declaration means that p is a variable, which can hold the address of another integer variable. This means the following assignment is perfectly valid

```
p = &a;
```

This stores the address of the variable ‘a’ in the pointer ‘p’, 1000 in our example. i.e.

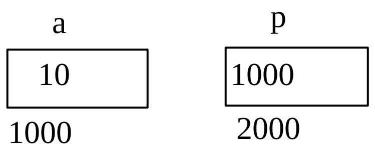


Fig 3-2 Memory Representation of a Pointer Variable

Remember the following assignment is *invalid*

```
int x;
x = &a;
```

An ordinary variable is capable of holding only values. It cannot hold addresses. So to make the above assignment valid, the declaration of x should be like

```
int* x;
```

Now execute the following printf statements and observe the results.

```
printf(“P = %u”, p);
printf(“&P = %u”, &p);
printf(“*p = %d”, *p);
```

- The first printf statement will print the address of a. In other words, it prints 1000. This is because whenever we refer a variable by just its name we actually refer to whatever is there in that location.
- The second printf statement prints the address of p, which is 2000 as per our sample. A pointer being a variable will also have an address on its own. &p refers to the address where p is stored in memory.
- The third printf statement prints the value that is present in the memory location 1000, which is 10. The * operator is called as the *de-referencing operator*.

De-referencing a Pointer

A **pointer** is a datatype whose value is used to refer to ("points to") another value stored elsewhere in the computer memory. Obtaining the value that a pointer refers to is called

dereferencing the pointer. In the above example the * operator is used to get the value of the pointer p.

Assigning a value through pointer

Whenever a value is assigned to a pointer the * operator has to be used so that it gets stored in the address pointed by p. Whenever an address is assigned do not use the * as we need to store an address only inside p and not at where p is pointing to.

```
*p = 50;
```

Observe the results of the following printf statements

```
printf(“P = %u”, p);

printf(“&P = %u”, &p);

printf(“*p = %d”, *p);

printf(“a = %d”, a);
```

The results of the first two printf statements are exactly similar to the previous ones. But the third printf statement prints the value 50. Surprisingly the last printf statement also prints 50 and not 10 which was assigned earlier to a. This is because *p refers to the place where p is pointing. In other words *p denotes that the value will be stored on the address pointed by p which is the same as the address of the variable a. Thus with the help of the pointer it is possible to change the value of the variable/memory location it is pointing to.

If p is a pointer and a is a variable defined as

```
int *p;

int a;
```

then the following assignments are invalid :

```
*p = &a;  // as *p can hold only a value and not an address
p = a;      // as P can hold only an address and not a value
```

Note: *(&a) is equal to a. [*(&a) = a] this means the content of the address of ‘a’ is nothing but ’a’ itself.

Points to remember:

If p is a pointer then,

- ✍✍ Whenever we refer the pointer as p we are referring to the address that p is pointing to.

- Whenever we refer the pointer as &p we are referring to the address of the pointer variable.
- Whenever we refer the pointer as *p we are referring to the value pointed by the address that the pointer contains.

If you remember and understand these rules, then you have understood pointers. Once understood properly, pointers are the easiest and efficient way of writing programs.

Having understood the basics now let's see one simple application of pointers. Consider the following example 1

```
void swap(int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}
void main()
{
  int a = 10, b = 20;
  swap(a, b);
  printf("a =  %d", a);
  printf("b = %d",  b);
}
```

What will be the output of the above program? One normally expects the output to be 20, 10 for 'a' and 'b' respectively after the swap. But don't get surprised if you see the output as

a = 10 & b = 20

This is because whenever a variable is passed to a function it is passed by value. This means that 'a' and 'b' will have 2 memory locations in memory. Let's say 100 and 200 respectively

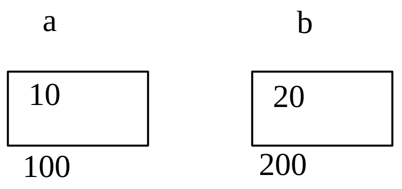


Fig 3-3 Memory representation of variables a and b

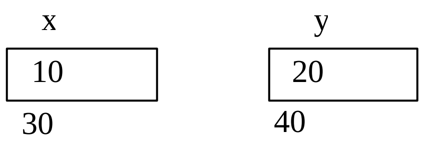


Fig 3-4 Memory representation of variables x and y before swap

When a method call is made, 2 new memory locations are created for x and y

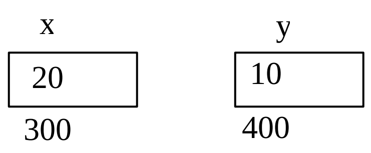


Fig 3-5 Memory representation of variables x and y after swap

and after the swapping logic, x and y will have the values 20 and 10 respectively. When the function is over the local variables (x & y) goes out of scope. Note that throughout we are **NOT** referring to the memory locations of 'a' and 'b' anywhere. So after the swap method the values still remain as it is.

Now let's see how we can make use of pointers to solve this issue. The following is the revised version of the swap program example 2

```
void swap(int* x, int* y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}
void main()
{
  int a = 10, b = 20;

  swap(&a, &b);

  printf("a =  %d", a);
  printf("b = %d",  b);
}
```

In the above example instead of passing the values we are passing the addresses of a and b respectively. We know that a normal variable cannot hold addresses. So the method arguments are modified as pointers so that they can be assigned addresses. Now let's see what happens. When the addresses of 'a' and 'b' are passed they are copied into the memory locations 300 and 400.

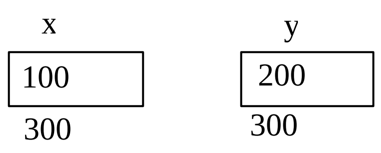


Fig 3-6 Memory representation of pointers x and y

Now the swapping logic is done. Note that we need to swap only the values and not the memory locations. So the * operator is used while doing the swapping. That is the value stored in address 100 and the value stored in address 200 are swapped through the pointer x and y. At the end of the swapping logic x and y will still be holding the addresses of a and b respectively. But the values are swapped in the original locations 100 and 200. When the method terminates the local variables are destroyed but since the swapping was carried out on the original memory locations, i.e. in the location of 'a' and 'b', the output of this program will be 20 and 10.

Note: When more than one value needs to be returned from a method, pointers are the only way through which it is possible. So from the above example let us look into the definition of Passing by value and passing by reference

Passing values

Passing values is known as **call by value.** You actually pass a copy of the variable to the function. If the function modifies the copy, the original remains unaltered. The example1 demonstrated **call by value**.

Passing pointers

This is known as **call by reference** and is an area worth spending some time on. We do not pass the data to the function; instead we pass a pointer to the data. This means that if the function alters the data, the original is altered. Example 2 demonstrated call by reference.

Pointers and Arrays

Having seen how to use pointer with variables let's see how pointers can be used with arrays. Just like a pointer can point to a variable, it can also point to an array. It really doesn't matter as to what it is pointing to as long as a valid address is assigned to a pointer. For example if arr is an array defined as

```
int arr[5] = {10, 20, 30, 40, 50};
```

and p is a pointer then this pointer can point to the array with the help of either one of the assignment statements.

```
int *p = &arr[0] or  int *p = arr;
```

Whenever an array is referred by its name we are actually referring to the address of the 0th element. This means that

```
arr = &arr[0];
```

Once the address is assigned to the pointer, execute the following printf statement and observe the result

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

The output of the above statement is the content of the 0th element, which is 10 according to our sample. We know that the elements of the array are stored in consecutive memory locations. So how do we print the next value in this array? That is done with the help of a concept called *pointer arithmetic*.

Pointer Arithmetic

Just like a normal variable, increment / decrement of a pointer variable is a valid operation. That is if we say P++ or P += 2 what does that it mean? Whenever a pointer variable is incremented the number of bytes that gets incremented depends on the data type. If it is an integer pointer and if we say P++, then the pointer moves by 2 bytes in a turbo C compile and 4 bytes in Microsoft Visual Studio. i.e. it points to the next valid address. Similarly if it is a character pointer, then incrementing the pointer will increment it by one byte. This is referred to as pointer arithmetic.

Having understood pointer arithmetic how do we print the subsequent elements from an array? There are various ways for doing it. One way is to increment the pointer and then print the value again. This we can repeat till the end of the array is reached i.e.

```
p++;
printf("%d", *p);
```

This time it prints the second element of the array, as the pointer is incremented. For printing subsequent elements the above 2 steps can be repeated. The more easier and elegant way of writing it is as follows:

```
for(i = 0;  i< 5; i++)
{
  printf("Value = %d\n", *(p+i));
}
```

The above loop will print the values of all the elements in the array. Here we keep the base address of the pointer as constant and adding the value of i to p (pointer arithmetic) and hence we will be referring to the elements of the array properly. It is also possible to write the above block as follows

```
for(i =0;  i< 5; i++)
{
  printf("Value = %d\n",  p[i]);
}
```

Yes. We can access a pointer just like an array. Similarly we can access an array like a pointer i.e. the following reference to the array is perfectly valid.

```
for(i =0;  i< 5; i++)
```

```
{
  printf(“Value = %d\n”,  *(arr + i));
}
```

In other words, arrays are nothing but constant pointers. The difference between arrays and pointers are:

- An array has a fixed base address whereas a pointer’s can point to different addresses at different points of time
- A pointer can be incremented / decremented where as an array cannot be incremented/decremented

Pointers and multi dimensional arrays

As the internal representation of a multi dimensional array is also linear. A pointer can point to an array of any dimension. Only the way in which we access the pointer varies according to the dimension. For example if arr is a2 dimensional array of 3 rows and 3 cols

```
int arr[3][3] = {1, 2, 3, 4, 5,  6, 7, 8, 9};
```

then the following assignment is valid to make

```
int* p = arr;
```

The elements can be accessed using the following loop

```
for(i =0;  i< 9; i++)
{
  printf(“Value = %d\n”, *(p+i));
}
```

Here is another example as to where a pointer can be helpful to us. Whenever we need to pass an array to a function and if we want to make the function generic for any number of elements this is how we can do it.

```
void displayMatrix(int** x, int iRow, int iCol)
{
  int i, j;
  for(i = 0; i < iRow; i++)
  {
    for(j = 0; j < iCol; j++)
    {

      printf(“x[%d][%d] = %d”, iRow + 1, iCol + 1, *( *(x + i) + j));

    }
    printf(“\n”);

  }

}

void main()
{
  int a[2][2] = {1, 2, 3, 4};
  int b[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  displayMatrix(a, 2, 2);
  displayMatrix(b, 2, 2);
}
```

Character Pointers

Just like an integer pointer a character pointer is a pointer, which is capable of holding the address of a character variable. Suppose if we have a character array declared as

```
char name[30] = {“Data Structures”};
```

and if we need to store the address of this array we need a character pointer declared as follows

```
char *p = NULL;
```

Once the pointer is declared we need to assign the address of the array to this pointer. We know that when an array is referenced by its name we are referring to the address of the 0th element i.e.

```
p = name;
```

will assign the address of the 0th element to p. Now issue the following printf statements and check the output

```
printf(“character output = %c\n”, *p);
printf(“String output = %s”, *p);
```

We know that when we refer to a pointer with the indirection operator we refer to the content of the address pointed by the pointer. The above printf statements will product the output as follows:

```
character output = D
String output = Data Structures
```

The reason for the output produced by the second printf statement is because of the %s format specifier, which will print the string till it encounters a ‘\0’ character.

Pointers can be used interchangeably with arrays as follows

```
int Length(char* str)
{
  int iLength = 0;
  for(iLength = 0; *(str + iLength) != ‘\0’; iLength++);
  return iLength;
}
void main()
{
   char str1[] = {“This is a String”};
   char str2[] = {“This is another string”};
   printf(“Length of Str1 = %d”, Length(str1));
   printf(“Length of Str2 = %d”, Length(str2));
}
```

Memory Allocation

So far we have been discussing about a pointer pointing to another variable or array. Can’t we directly make use of pointers? i.e.

```
int *p;
```

followed by

```
scanf(“%d”, p);
```

Note that there is no ‘&’ in the scanf for the pointer as referring a pointer by its name actually refers to the address it points to. The above statements are treated valid by the compiler and the code might execute properly as well. But this is really a dangerous programming. Because p is a pointer and we have not assigned anything to it, it might be having some value in it. Trying to store a value in that address can cause serious damage to the system, as we are not aware of what p will be pointing to.

So it is always a better programming practice to initialize any pointer declaration to NULL i.e.

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

This ensures that the pointer is not pointing to anywhere. Even after this we should not accept values from the user because there is no valid memory location for storing this value. In such cases before we start using the pointer we should allocate memory to it.

Malloc() is the function used for memory allocation in C. malloc() takes one argument that is the number of bytes to allocate. Suppose P is a pointer then

```
int *p;
p = (int *)malloc(sizeof(int));
```

The above assignment will allocate memory for holding an integer value. Remember whenever malloc is used for memory allocation the system allocates memory from the heap and not from the stack. So it is the responsibility of the user to free the memory that is allocated. For freeing the allocated memory the function used is,

```
free(<pointer variable>);
```

Allocating memory for arrays

```
int *p = (int*)malloc(10 * sizeof(int));
```

The above statement will allocate memory for 10 integer elements.

Points to Remember

- Always initialize a pointer to NULL.
- Allocate memory to a pointer before using it.

Two Dimensional Arrays

A two dimensional array can be represented using a pointer in any one of the following ways:

- Array of Pointers
- Pointer to Pointers

Array of pointers

Array of pointers is like any other array with all the elements in it being pointers. How to declare a variable as an array of pointers?

```
int *p[10];
```

The above declaration means that p is an array of 10 elements, each of which is a pointer, which can hold the address of another integer variable. Since we know that a pointer is capable of pointing to more than one element the above declaration actually means that the number of rows is always 10. But for every row the number of columns can vary depending on the memory allocation.

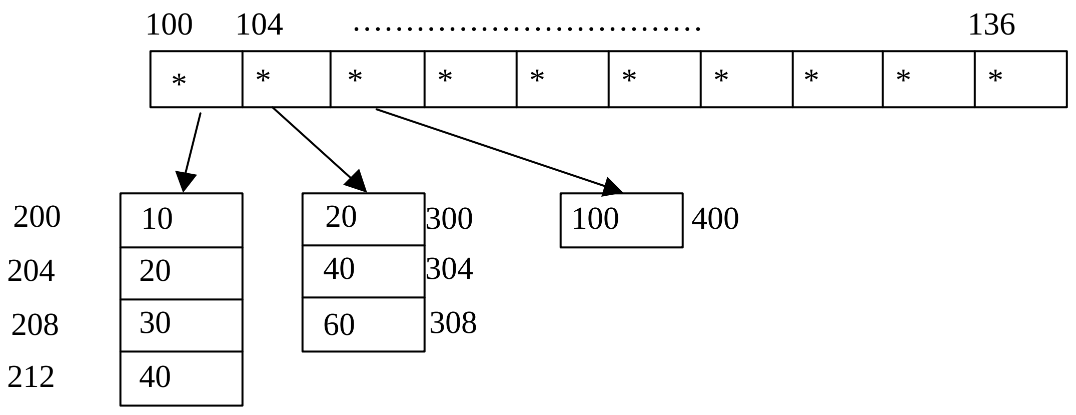


Fig 3-7 Memory Organization of Array of Pointers

Pointer To Pointers

Most often in real time situations the number of rows as well as columns might vary dynamically. This being the case the above approach cannot be used. In such situations we can make use of pointer to pointers.

A pointer to pointer is nothing but a pointer, which is capable of holding the address of another pointer variable in it. As discussed earlier a pointer is also a variable and hence it is quite possible to hold the address of the pointer also in another pointer. A pointer to pointer can be declared as,

```
int **p;
```

The above declaration means that p is a pointer, which can hold the address of another integer pointer (a pointer which can hold the address of an integer).

As such there is no restriction imposed by the compiler as to how many levels we can go about in using a pointer. i.e. the following declaration is perfectly valid

```
int *****p;
```

However beyond 3 levels if we make use of pointers then it will make the code highly complex and un-maintainable.

Pointer and structures

A pointer can be used with a structure just like any other data type. The only difference is in the way we access the data members of the structure. Suppose if we have a structure as

```
typedef struct player
{
  char name[30];
  int age;
}player;
player p1;
```

Then we can access the members as,

```
p1.name and p1.age
```

Suppose if we change the declaration as

```
Player* p1;
```

then for accessing the members of the structure we can follow either one of the following approaches

```
p1->name            or            (*p1).name
p1->age                           (*p1).age
```

Function Pointers

Just like variables have address functions also have address. This means that we can make use of a pointer to represent the address of a function as well. If we use a pointer to represent the address of an integer we declare it as int* p. Similarly if it is a character pointer we declare it as char* p. But if it is a function pointer how can we declare it.
Suppose we have a function as

```
void add(int x, int y)
{
  printf("Value = %d", x + y);
}
```

if we want to declare a pointer to represent this function we can declare it as

```
void (*p)(int x, int y);
```

The above declaration means that p is a function pointer that can point to any function whose return type is void and takes two integer arguments. The general syntax for a function pointer declaration will look like,

<return type> (<Pointer Variable>) (<data type> [variable name], <data type> [variable name]);

Variable names in the argument list are optional. It is only the data type that matters.
Having seen how we can declare a function pointer the next step is to assign the address of a function to this pointer. But how do we get the address of a function? Just like an array whenever we refer a function by its name we actually refer to the address of the function. This means that we can assign the function address as,

```
p = add;
```

Note that we are not specifying the function parenthesis for add. If we put a function parenthesis it means that we are calling a function. Now how can we use this function pointer? Wherever we can call add we can use this function pointer interchangeably

i.e. we can say,

```
printf(“Value = %d”, (*p)(10, 20));
```

Which will make a call to add and print the result. Since it is a pointer it can be made to point to different functions at different places. i.e. if we have another method sub declared as

```
void sub(int x., int y)
{
  printf(“Value = %d”, x - y);
}
```

Then we can say,

```
p = sub
```

and call it as

```
printf(“Value = %d”, (*p)(20, 10));
```

This time it will make a call to sub and print the result as 10.

Advantages

- It gives direct control over memory and thus we can play around with memory by all possible means. For example we can refer to any part of the hardware like keyboard, video memory, printer, etc directly.
- As working with pointers is like working with memory it will provide enhanced performance
- Pass by reference is possible only through the usage of pointers
- Useful while returning multiple values from a function
- Allocation and freeing of memory can be done wherever required and need not be done in advance

Limitations

- If memory allocations are not freed properly it can cause memory leakages
- If not used properly can make the program difficult to understand

What is Recursion?

Recursion is one of the wonderful features provided by C. A function making a call to itself is called as recursion. This can be helpful in a lot of situation making the logic compact and easy to understand. When a function is going to call itself how will this recursive call get terminated? There must always be one condition based on which the recursion should terminate, otherwise it will go in an infinite loop resulting in a stack overflow error.

Let's us write a recursive routing of a factorial program to understand recursion clearly.

```
int factorial(int iFact)
{
  if (iFact == 1)
  {
    return 1;
  }
  else
  {
    return iFact * factorial(iFact – 1);
  }
}
void main()
{
  printf("Factorial = %d", factorial(5));
}
```

If the above method is called with let's say 5 then for the first time, it will check whether the value is 1, since it is not it will make a call as 5 * factorial(4). Next time it will check whether the value is 1 again, since it is not again it will make a call as 4 * factorial(3) and so on.

When the fact value reaches 1, the recursion is terminated and it will return back to the called function which is again factorial and finally when it comes out of all the recursive loops the output will be returned back to the called function. Here are the sequences of calls :

```
5 * factorial(4)
4 * factorial(3)
3 * factorial(2)
2 * factorial(1)
```

when the function calls are returned the last call will return a value of 1 which results in 2 * 1 for the previous call and will return 2 to the next previous which will result in 3 * 2 which is 6 and so on and the data flow is as follows

```
2 * 1

3 * 2
4 * 6

5 * 24
```

The output will be printed as 120

Recursion is typically meant for programs where

- The problem itself is recursively defined, for example factorial
- The data structure that the algorithm operates is recursively defined, for example binary trees
- Most importantly whenever we need to describe a backtracking procedure (My personal opinion is it is the only place where we should use recursion and we can use effectively

and efficiently. In all the other places it may reduce the program complexity but will result in inefficient algorithms)

Even in these cases recursion should be used with great care as the memory requirements can increase drastically. For example using a recursive routine for a factorial is unnecessary as it involves a number of method calls and local variables getting created. This results in memory overhead and performance overhead.

Back Tracking Algorithms

Back tracking algorithms are those places where we need to repeat a process going back and forth. One famous example for back tracking is the eight-queen problem. The problem is to place 8 queens on a chessboard without clashing each other. If we do it normally on chessboard how we will go about doing it. First we can place up to 5 queens easily. When we are about to place the 6th queen we might not be having any place for it to be placed. In such situations we will take the 5th queen and place it on some other place and we will try to place the 6th one again. If we still couldn't locate a place then we will replace the 4th queen also on a different place and then we will be trying all possible permutation combinations. Similarly we will be trying going back and forth till we successfully place all the queens. Such kinds of problems are called as back tracking problems. It is on these places recursion can play its role wonderfully and make the algorithm look compact without much of memory or performance overhead.

SUMMARY

- A **pointer** is a datatype whose value is used to refer to ("points to") another value stored elsewhere in the computer memory. Obtaining the value that a pointer refers to is called **dereferencing** the pointer.
- A two dimensional array can be represented using a pointer in any one of the following ways:
  - Array of Pointers
  - Pointer to Pointers
- A pointer can be used with a structure just like any other data type. The only difference is in the way we access the data members of the structure.
- A function making a call to itself, is called as recursion.
- Back tracking algorithms are those places where we need to repeat a process going back and forth. One famous example for back tracking is the eight-queen problem.

Test your Understanding

1. Write a program to compress any given string such that that multiple blanks present should be eliminated. The compressed string should be stored in another string.

Exercise 1b

2. Write a program to sort an array of 10 integers using an array of pointers.

Chapter 4: Stacks

Learning Objective

After completing this chapter, you will be able to:

✍✍ Understand and write programs to create stacks

Introduction To Stacks

A stack is an ordered list in which items are inserted and removed at only one end called the TOP. There are only 2 operations that are possible on a stack. They are the ‘Push’ and the ‘Pop’ operations. A Push operation inserts a value into the stack and the Pop operation retrieves the value from the stack and removes it from the stack as well.

✍✍ An example for a stack is a stack of plates arranged on a table. This means that the last item to be added is the first item to be removed. Hence a stack is also called as **Last-In-First-Out List** or **LIFO** list.

To understand the usage of a stack let us see what does the system does when a method call is executed:

✍✍ Whenever a method call is executed the compiler needs to know the return address for it to resume from the place where it left off in the calling method i.e. let’s take the following example

```
void func1()
{
  printf(“I am in function 1”);
  func2();
  printf(“I am again in function 1”);
}

void func2()
{
  printf(“I am in function 2”);
  func3();
  printf(“I am again in function 2”);
}


void func3()
```

```
{
   printf(“I am in function 3”);
}
```

If the above method calls are executed by the compiler, after printing the line “I am in function 1”, the compiler will transfer the method call to func2(). From function 2 it prints the line “I am in function 2” and after that it switches to func3. Now it prints “I am in function 3” from within func3(). Once that is over where should the function call return? Should it return to func2() or func1() or to anywhere else? How does the system identify that it has to go to func2() after finishing func3()?

This is done with the help of a stack. Before entering into function 2, the system pushes the address of where it has to return in to a stack and then makes a call to func2(). Similarly before entering into function 3, the system pushes the return address into the stack to which it will return after exiting from func3(). Once this is done, it then makes a call to func3(). Now, when we take the value from a stack, the last value pushed will be first one that will be popped back and hence the system returns exactly to the line after where it had left off and continues further. i.e. it prints the line “I am again in function 2”. Now again the system needs to know where it has to return. So it will issue another pop which will return the position where it will have to resume in func1() which eventually will print “I am again in func1” and terminates the function.

Graphical Representation

A graphical representation of a stack is shown below:

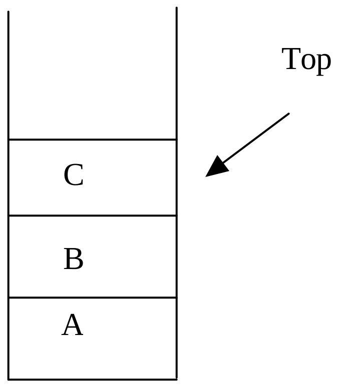


Fig 4-1 Graphical Representation of a Stack

Memory Organization of a Stack

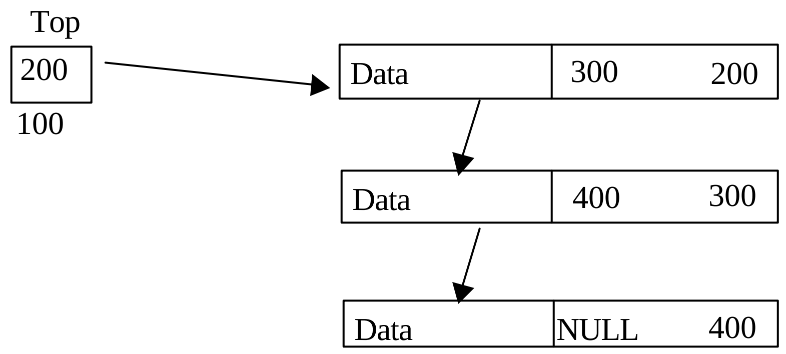


Fig 4-2 Memory organization of a Stack

The memory organization of stack is very similar to that of a linear list

Programmatic representation of a stack

```
typedef struct node
{
  int iData;
  struct node* pNext;
}node;
node* pTop = NULL;
```

The node of a stack is very similar to the node in a list. The difference is only in the way the data is organized. As discussed there are only 2 operations permitted on a stack: the Push and the Pop

Push Operation

A push operation is for inserting an element in to a stack. Elements are always inserted in the beginning of a stack.

Steps for the push operation

- Create a new node
- Make the new node's next point to the node pointed by Top
- Make the top point to the new node

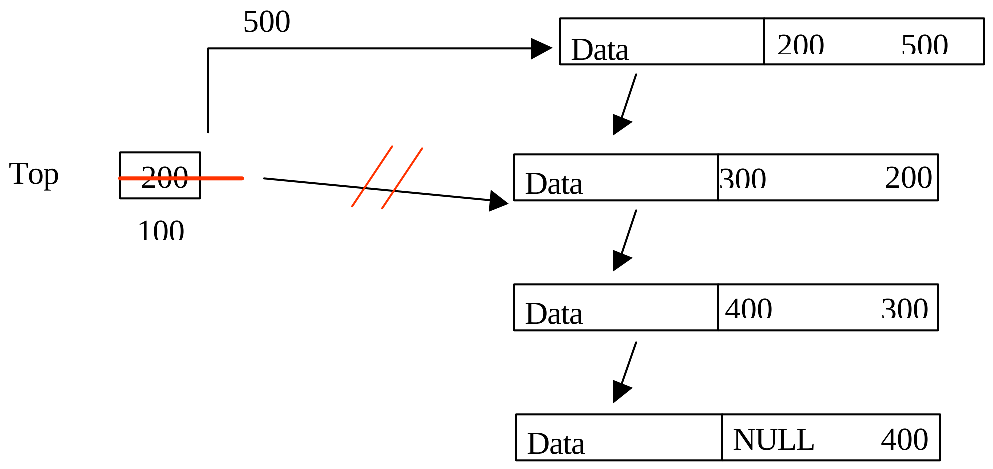


Fig 7-3 Push Operation of a Stack

```
void push(int iData)
{
  node* pNewNode = (node *)malloc(sizeof(node));
  pNewNode->iData = iData;
  pNewNode->pNext = pTop;
  pTop = pNewNode;
}
```

If you had a careful look over this code this is nothing but equal to inserting an element at the end of a list.

Pop Operation

A pop operation is basically used for retrieving a value from the stack. This also removes the element from the Stack.

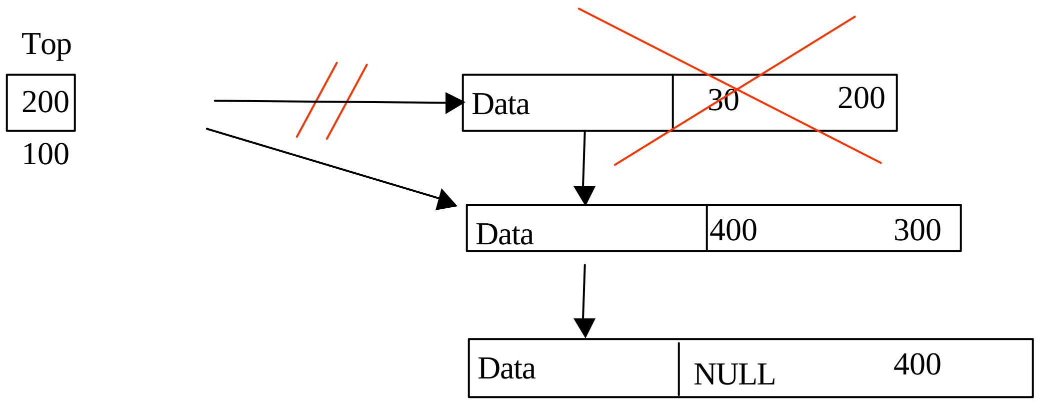


Fig 7-4 Pop Operation

```
int pop()
{
  int iData;
  node* pTempNode = NULL;
  if (pTop == NULL)
  {
    iData = -1;
    printf(“Stack Empty”);
  }
  else
  {
    pTempNode = pTop;
    iData = pTop->iData;
    pTop = pTop->pNext;
    free(pTempNode);
  }
  return iData;
}
```

Application of Stacks

Evaluating Expressions

In normal practice any arithmetic expression is written in such a way that the operator is placed between its operands. For example

(A + B) * (C + D)

Such kind of expressions is called as infix notation.

Polish notation refers to the notation in which the operator is placed before the operands. For example the above expression can be written as

*+AB+CD

The idea is, whenever we write expressions in this notation, parenthesis are not required for determining the priority of the expressions. Let us see the steps involved in the conversion

(A+B)*(C+D) = (+AB)*(+CD) = *+AB+CD

Reverse polish notation is exactly the opposite of polish notation i.e. the operator is always placed after the operands. For example, the above expression can be written in Reverse Polish Notation as

(A+B) * (C+D) = (AB+) * (CD+) = AB+CD+*

Whenever an expression is evaluated by the system it usually performs it by means of 2 steps. First it converts any expression into prefix or postfix notation and then it evaluates the expression as it makes the job much simpler.

Converting an infix expression to postfix or prefix expression makes use of stacks extensively. The following is the algorithm for doing this conversion. A complete program requires a lot more than what is described in the algorithm.

```
PostFixExpression ConvertToPolishNotation(InfixExpression)
{
    1. Push “(“ onto the STACK and add “)” to the end of the InfixExpression
    2. Scan the InfixExpression from left to right and repeat steps 3 to 6 for each element of
       InfixExpression until the stack is empty
    3. If an operand is encountered, add it to Postfix Expression
    4. If a left parenthesis is encountered, push it to STACK
    5. If an operator XX is encountered
        a. Pop from STACK repeatedly and add it to Postfix expression which has the same
           / higher precedence than XX.
        b. Add XX to STACK
6. If a right parenthesis is encountered, then
        a. Pop from STACK repeatedly and add it to Postfix Expression until a left
           parenthesis is encountered.
        b. Remove the left parenthesis
7. Exit
}
```

SUMMARY

- ✍✍ A stack is also called as Last-In-First-Out List or LIFO list.
- ✍✍ A stack is an ordered list in which items are inserted and removed at only one end called the TOP.
- ✍✍ There are only 2 operations permitted on a stack: the Push and the Pop
  - o A push operation is for inserting an element in to a stack. Elements are always inserted in the beginning of a stack.
  - o A pop operation is basically used for retrieving a value from the stack. This also removes the element from the Stack

Test your Understanding

1. Write a function called stack copy that copies the contents of a stack into another. The function must have two arguments for the source stack and destination stack.

Exercise 2

2. Write a new function to concatenate the contents of one stack of integers upon top another. Test the function by creating two stacks printing them the concatenating them and then printing them again.

Chapter 5: Queues

Learning Objective

After completing this chapter, you will be able to:

✍✍ Understand and write programs to create queues

Overview

A Queue is an ordered list in which all insertions can take place at one end called the rear and all deletions take place at the other end called the front. The two operations that are possible in a queue are Insertion and Deletion.

A real time example for a queue is people standing in a queue for billing on a shop. The first person in the queue will be the first person to get the service. Similarly the first element inserted in the queue will be the first one that will be retrieved and hence a queue is also called as First In First Out or FIFO list.

Graphical Representation

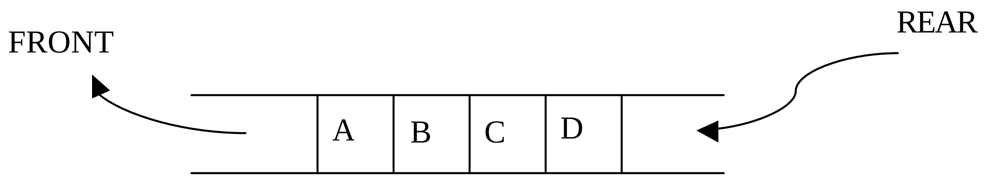


Fig 5-1 Graphical Representation of a Queue

Memory Organization of a Queue

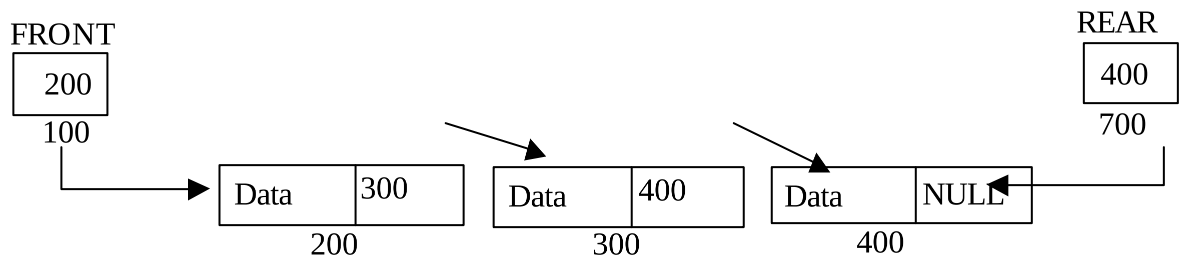


Fig 5-2 Memory Organization of a Queue

Programmatic representation of a Queue

The node of a queue will have 2 parts. The first part contains the data and the second part contains the address of the next node. This is pretty much similar to a linear list representation. But in a queue insertions and deletions are going to occur on two different ends. If we have only one pointer called START then for every insertion we need to traverse the complete queue as insertions are always on the end and hence will be time consuming.

To avoid this we are going to have 2 pointers to represent a queue, one pointer is called the FRONT which will always be pointing to the first element and is used for deletions, and the other pointer called REAR which will always be pointing to the last element in the queue and is used in insertions.

```
typedef struct node
{
  int iData;
  struct node* pNext;
}node;
node* pFRONT = NULL, *pREAR = NULL;
```

When the queue is empty, both FRONT and REAR will be pointing to NULL. When there is only one element in the queue, then both FRONT and REAR will be pointing to the same element.

Steps for inserting an element into a Queue

Inserting into a queue can happen only at the REAR end.

1. Create a new node
2. Make the next of new node as NULL as it is the last element always
3. If the queue is empty, then make FRONT point to new node

4. Otherwise make the previous node's next (the node pointed by REAR is always the previous node) point to this node
5. Make Rear point to new node

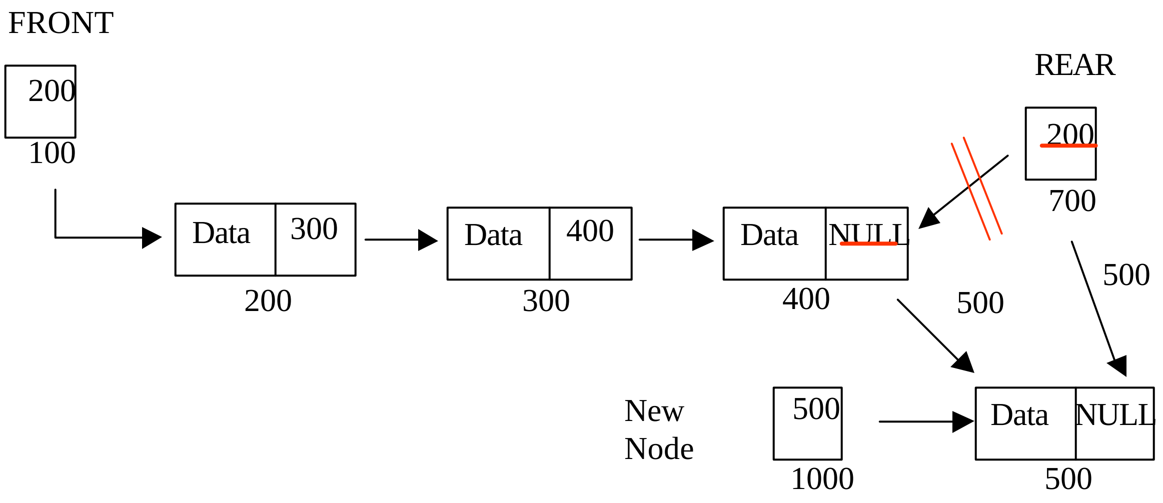


Fig 8-3 Inserting a node in a queue

```
void QInsert(int iData)
{
  node* pTempNode = getNode(iData);
  if ( pFRONT == NULL )
  {
    //If the queue is empty
    pFRONT = pTempNode;
  }
  else
  {
    pREAR->pNext = pTempNode;
  }
  pREAR = pTempNode;
}
```

Steps for deleting an element from the queue

The deletion is the only way through which we can retrieve values from a queue. Along with retrieving the value this will remove the entry from the queue. Deletions always happen in the FRONT end.

1. Store the node pointed by FRONT in a temporary pointer
2. Make Front as Front's next
3. Delete the node pointed by temporary pointer
4. If all the nodes are deleted from the queue make the FRONT and REAR as NULL

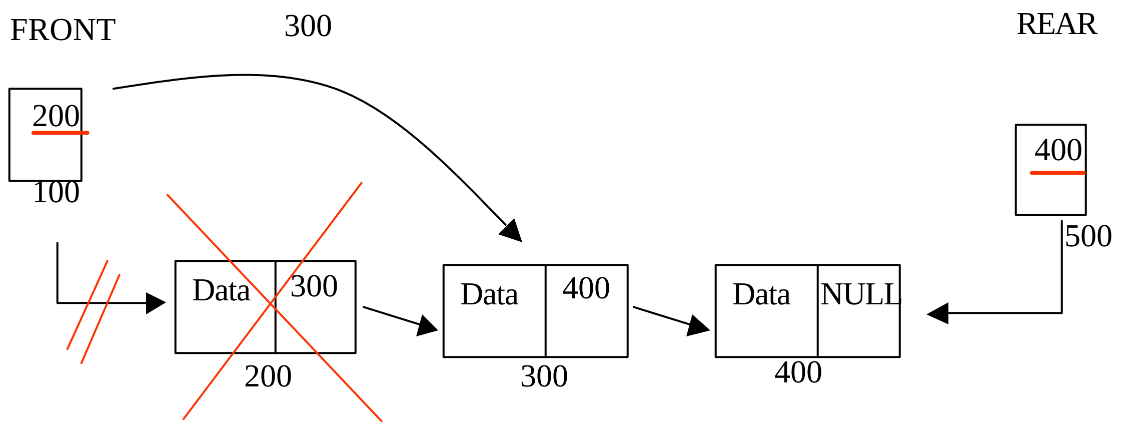


Fig 8-4 Deleting a node in a queue

```
int QDelete()
{
  int iData = -1;
  node* pDelNode = NULL;

  if (pFRONT == NULL)
  {
    printf("Queue Empty");
  }
  else
  {
    iData = pFRONT->iData;
    pDelNode = pFRONT;
    pFRONT = pFRONT->pNext;
    if (pFRONT == NULL)
    {
       pREAR = NULL;
    }
    free(pTempNode);
```

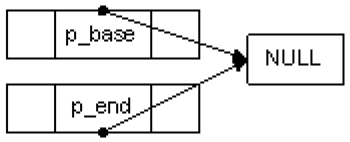
```
        }
        return iData;
    }
```

Deque

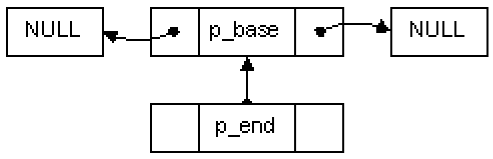
A Deque is a queue in which insertions and deletions can happen at both ends of a queue. A **deque**, or *double-ended queue* is a data structure, which unites the properties of a queue and a stack. Like the stack, items can be *pushed* into the deque; once inserted into the deque the last item pushed in may be extracted from one side (*popped*, as in a stack), and the first item pushed in may be pulled out of the other side (as in a queue).

Implementation of Deque

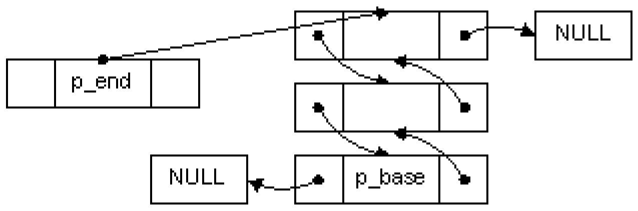
The push (insert/assign) and pop operation is done at both the end that is start and end of the deque. The following pictures show how a deque is formed based on this change in algorithm. Initially the base and end pointer will be pointing to NULL or 0 (zero).



We define two pointers, p_base and p_end to keep track of front and back of the deque. Initially when the deque object is created, both p_base and p_end would point to NULL.



When the first node is created, p_base assumes the position and p_end starts pointing to p_base. The next and previous pointer of p_base assumes NULL or 0(zero).



Further insertions happen at p_end.

Priority Queues

A priority queue is the one in which each element will have a priority associated with it. The element with the highest priority is the one that will be processed/deleted first. If two or more nodes have the same priority then they will be processed in the same order as they were entered in to the queue.

Application of Queues

The most common occurrence of a queue in computer applications is for scheduling of print jobs. For example if several people are giving print requests, all the request get queued up in the printer and is processed on first come first serve basis.

Priority queues are used for job scheduling by the operating system. The operating system will assign a priority for every process running currently in the system. The jobs with the highest priory are the ones which are taken up by the operating system for processing first and once all the jobs in that priority are over it will try to find the job with the next priority and so on.

SUMMARY

- A Queue is an ordered list in which all insertions can take place at one end called the rear and all deletions take place at the other end called the front. The two operations that are possible in a queue are Insertion and Deletion.
- Inserting into a queue can happen only at the REAR end.
- The deletion is the only way through which we can retrieve values from a queue.
- A Deque is a queue in which insertions and deletions can happen at both ends of a queue. A **deque**, or *double-ended queue* is a data structure, which unites the properties of a queue and a stack.
- A priority queue is the one in which each element will have a priority associated with it. The element with the highest priority is the one that will be processed/deleted first.
- Priority queues are used for job scheduling by the operating system.

Test your Understanding

1. Write an application that copies the contents of one queue into another.

2. Write an application called queue to stack that creates a stack from a queue. At the end of the algorithm the queue is to be unchanged, the front of the queue is to be the top of the stack and rear of the queue is to be the base.

Exercise 3

3. Given a queue of numbers, write an application that deletes all negative integers without affecting the other numbers in the queue.

Chapter 6: Linked Lists

Learning Objective

After completing this chapter, you will be able to:

✍✍ Understand and write programs to create linked list

Evolution of Linked Lists

In real time systems most often the number of elements will not be known in advance. The major drawback of an array is that the number of elements must be known in advance. Hence an alternative approach was required. This give rise to the concept called linked lists.

A linear list is a linear collection of data elements called nodes, where the linear order is given by means of pointers. The key here is that every node will have two parts: first part contains the information/data and the second part contains the link/address of the next node in the list. Memory is allocated for every node when it is actually required and will be freed when not needed.

What is a Structure in C?

Aggregate data types built using elements of other types.

Example:

```
struct Time {
              int hour;
              int minute;
              int second;
 };
```

Self-referential structure

Self-referential structures contain pointers within the structs that refer to another identical structure. Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

Graphical Representation of Linked List

A graphical representation of a node will look like

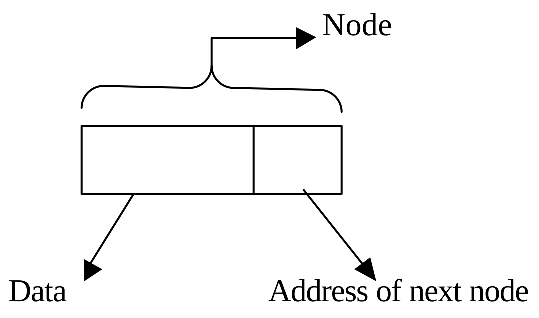


Fig 6-1 Graphical Representation of a Node

This means that for traversing through the list we need to know the starting point always. This is achieved by having a START pointer always pointing to the first element in the list.

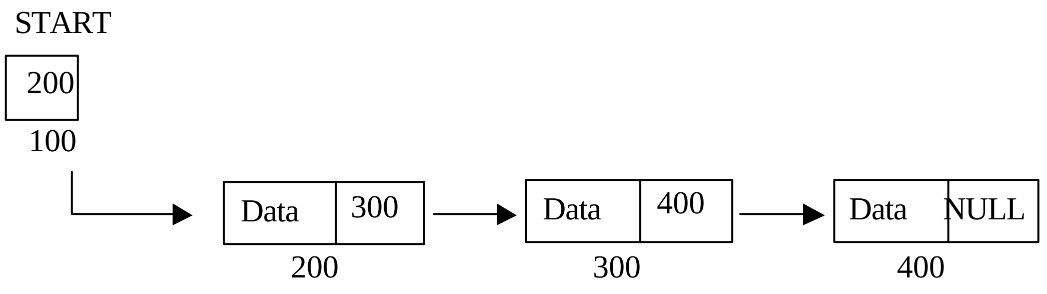


Fig 6-2 Memory organization of a Linked List

Programmatic representation of a node

```
typedef struct node
{
   int   iData;
   struct node* pNext;
}node;
node* pSTART = NULL;
```

The node structure contains the data and the pointer for the next node. The address of the next node must again be of type node and hence we have a pointer to the node within node itself. Such kind of structures which refers to itself are called as Re-Directions or Self Referential Structures.

Inserting a node into a list

Insertion logic varies depending on where in the list the element is going to be inserted, first, middle or at the end. In all situations it is as simple as making the pointer point to different nodes and hence insertion at any place will be much faster.

Steps for inserting an element in the beginning of a list

1. Create a new node
2. Make the next part of new node point to the node pointed by START
3. Make the START pointer point to new node

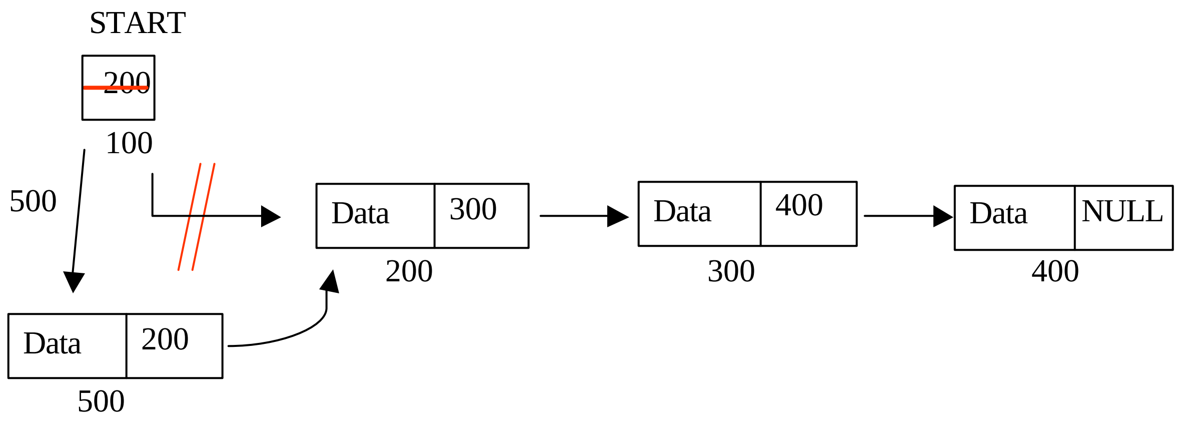


Fig 6-3 Inserting a node in the beginning of a list

```
void add_begin(int iData)
{
  node* pNewNode = (node*)malloc(sizeof(node));
  pNewNode->iData = iData;
  pNewNode->pNext = pSTART;
  pSTART = pNewNode;
}
```

Steps for inserting an element in the middle

1. Create a new node
2. Make the next part of new node point to the next of previous node

3. Make the previous node point to new node

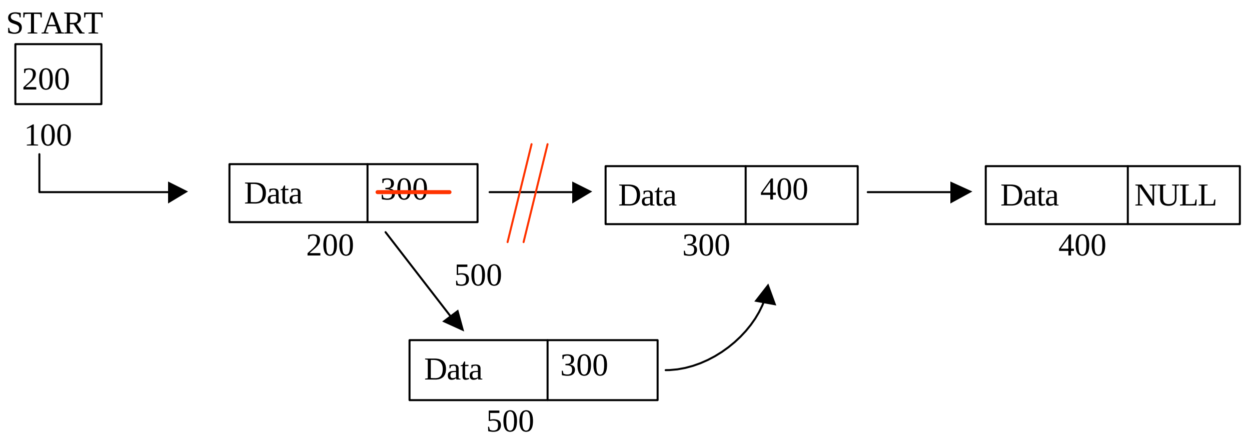


Fig 6-4 Inserting a node in the middle of a list

```
void add_Middle(int iData, int iLoc)
{
  int iPos = 1;
  node* pTempNode = NULL;
  node* pNewNode  = NULL;
  if (iLoc == 1)
  {
    add_begin(iData);
  }
  else
  {
    for (pTempNode = pSTART; pTempNode->pNext != NULL && iPos < iLoc;
                         pTempNode = pTempNode->pNext, iPos++);
    if (iPos == iLoc)
    {
      pNewNode = (node*)malloc(sizeof(node));
      pNewNode->iData = iData;
      pNewNode->pNext  = pTempNode->pNext;
      pTempNode->pNext = pNewNode;
    }
  }
}
```

Steps for inserting an element in the end

1. Create a new node
2. Make the next of new node point to NULL
3. Make the previous node point to new node

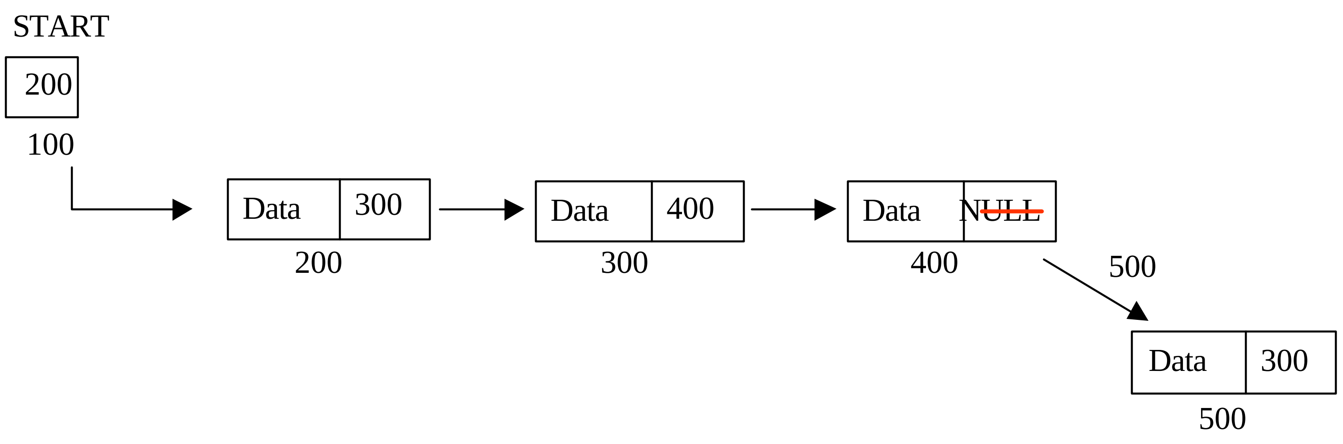


Fig 6-5 Inserting a node at the end

```
void add_End(int iData)
{
  int iPos = 0;
  node* pTempNode = NULL;
  node* pNewNode  = NULL;

  for (pTempNode = pSTART; pTempNode->pNext != NULL;
                        pTempNode = pTempNode->pNext);

  pNewNode = (node*)malloc(sizeof(node));
  pNewNode->iData = iData;
  pNewNode->pNext = NULL;
  pTempNode->pNext = pNewNode;
}
```

Steps for deleting a node in the beginning

The next step is deleting a node from a list. Again deletion logic varies depending on from where the node is getting deleted, beginning, middle or at the end.

1. Store the node to be deleted in a temporary pointer
2. Make START point to next node in the list

3. Delete the node pointed by temporary pointer

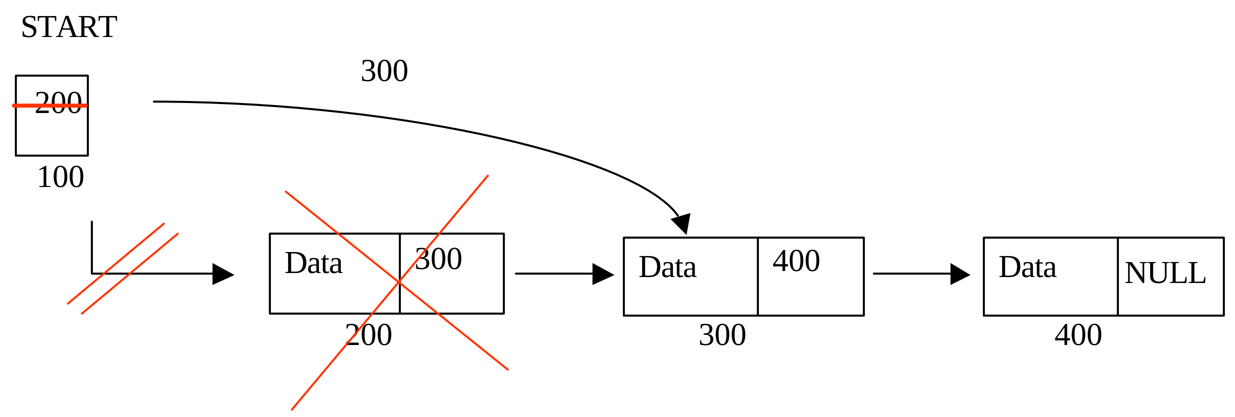


Fig 6-6 Deleting a node from the beginning

Steps for deleting a node from the middle / end

1. Store the node to be deleted in a temporary pointer
2. Make the previous node’s next point to the next of the node that is being deleted
3. Delete the node pointed by temporary pointer

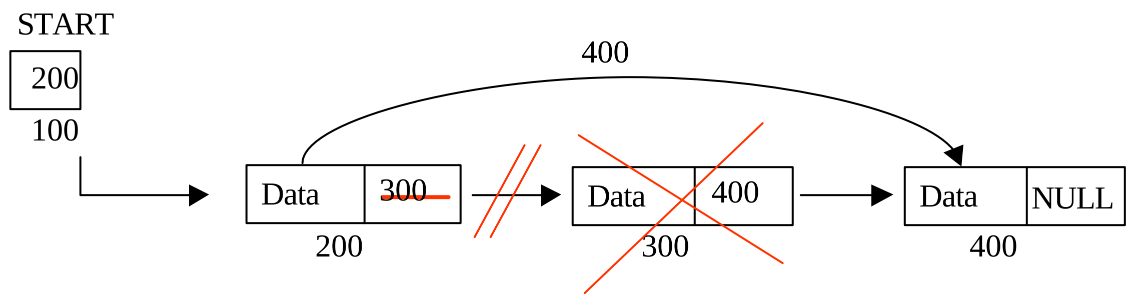


Fig 5-7 Deleting a node from the middle

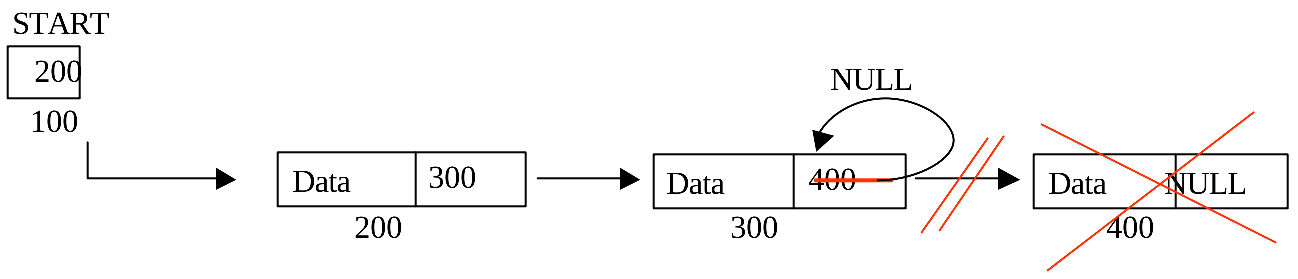


Fig 5-8 Deleting a node at the end

```
void delete(int iData)
{
  node* pTempNode = NULL;
```

```
    node* pDelNode  = NULL;
    node* pPrevNode = NULL;

    for(pTempNode = pPrevNode = pSTART; pTempNode != NULL;
                                    pPrevNode = pTempNode, pTempNode =
                                    pTempNode->pNext)
    {
      if (iData == pTempNode->iData)
      {
        pDelNode = pTempNode;
        pPrevNode->pNext = pDelNode->pNext;
        if (pDelNode == pSTART)
        {
          pSTART = pSTART->pNext;
        }
        free(pDelNode);
      }
    }
  }
```

Applications of Linked Lists

A linked list is used typically in computer applications for memory allocations. The system keeps tracks of all the memory available in the system with the help of a list. This area of memory is called as the memory pool. As and when the user requests for memory the system allocates memory to the user from this available pool. Once allocated these blocks will be deleted from the available list. Once the user frees the memory again the memory will be added to the list. In general linked lists are used in almost all places whenever a collection of elements are required and when the number of elements in the collection is not known in advance.

Advantages

- The number of elements in the linked lists need not be known in advance. As and when required elements can be added to a list
- Insertions and deletions are much faster in a linked list as there is no physical movement of elements of a list.

Limitations

- Searching a list is always sequential and hence it is a time consuming operation
- Traversal is always possible in only one direction.

Circular Linked Lists

Circular linked lists are very similar to a linear list except that the last node will be pointing to the first node again instead of NULL. So care should be taken while doing the traversal to avoid infinite loops.

A graphical representation of a circular linked list is as follows

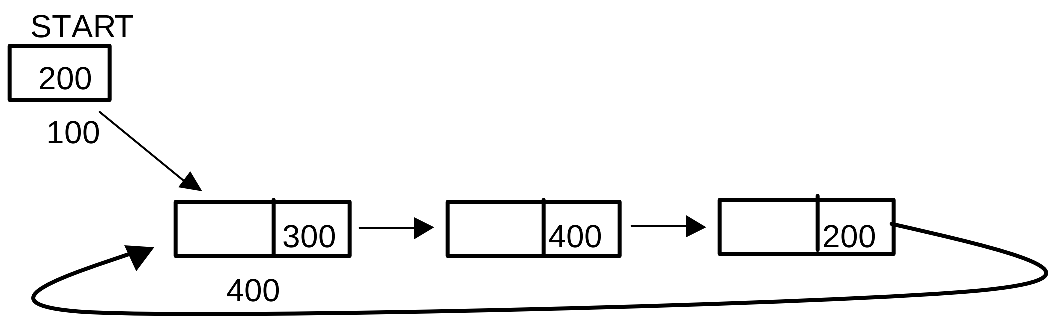


Fig 6-9 Memory Representation of a Circular Linked List

Assuming that *someNode* is some node in the list, this code iterates through that list starting with *someNode*:

Forwards

```
node := someNode
do
    <do something with node.value>
    node := node.next
while node not someNode
```

Backwards

```
node := someNode
do
    <do something with node.value>
    node := node.prev
while node not someNode
```

SUMMARY

✍✍ A linear list is a linear collection of data elements called nodes, where the linear order is given by means of pointers. The key here is that every node will have two parts: first part contains the information/data and the second part contains the link/address of the next node in the list

- ✍✍ Self-referential structures contain pointers within the structs that refer to another *identical structure.* Linked lists are the most basic self-referential structures.
- ✍✍ A linked list is used typically in computer applications for memory allocations.
- ✍✍ Circular linked lists are very similar to a linear list except that the last node will be pointing to the first node again instead of NULL.

Test your Understanding

1. Write code to add nodes, delete nodes and display node contents etc in a linked list.

3. Write an application that appends two lined lists together.

Exercise 4

2. Write code to implement a linked list where each node is a structure consisting of name, roll no, marks obtained in three subjects. Use the information in the nodes to give grades.

[Above 80% : A+, 60% - 79% : A, 50% - 59% : B, less than 50% : C]

Chapter 7: Doubly Linked Lists

Learning Objective

After completing this chapter, you will be able to:

✍✍ Understand and write programs to create double linked list

Introduction To Doubly Linked Lists

With the linked list traversal is possible in only one direction. This limitation is overcome with the help of doubly linked lists. A doubly linked list is a list in which each node will have 3 parts: one part containing the information / data, one part containing the pointer/address of next node and one part containing the pointer / address of previous node. In addition to the START pointer, which contains the first node there will also be a TAIL pointer pointing to the last node in the list.

Memory Organization Of A Doubly Linked List

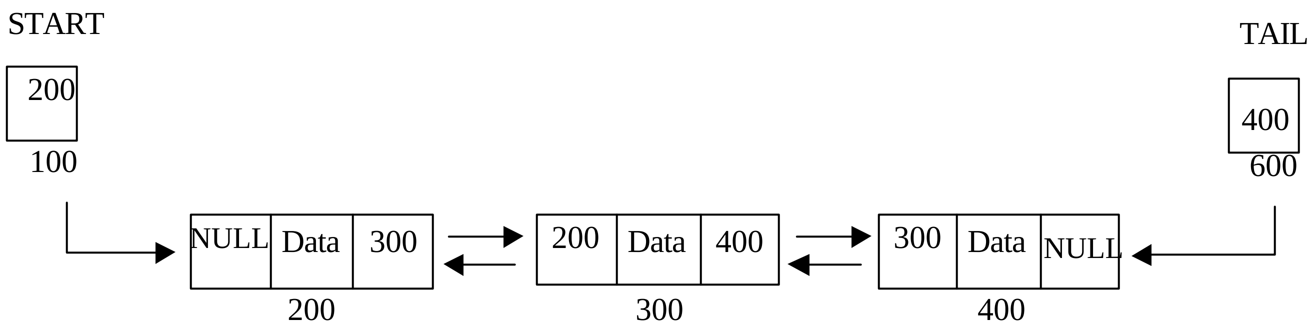


Fig 7-1 Memory organization of a Doubly Linked List

Programmatic Representation of a node in Doubly Linked Lists

```
typedef struct node
{
  int iData;
  struct node* pNext;
  struct node* pPrev;
}node;
node* pSTART = NULL, *pTAIL = NULL;
```

The node of a doubly linked list will have 2 pointers apart from the data, one for pointing to the previous element and the other for pointing to the next element in the list. There will also be a TAIL pointer pointing to the last node in the list. It is possible for us to traverse from the beginning of a list till the end or from the end of the list to the beginning. The end of the list is identified with the help of a NULL value in the next / previous parts.

Inserting an element in a doubly linked list

Inserting an element in a doubly linked list involves just changing the pointers unlike arrays where every element after the inserted element needs to be shifted once. Hence inserting an element in a doubly linked list is much faster than inserting a node in an array.

Steps for inserting in the beginning of a doubly linked list

1. Create a new node
2. Make the next part of new node equal to START
3. Make the previous part of new node equal to NULL
4. Make the previous part of the node pointed by START to new node
5. Make START point to new node

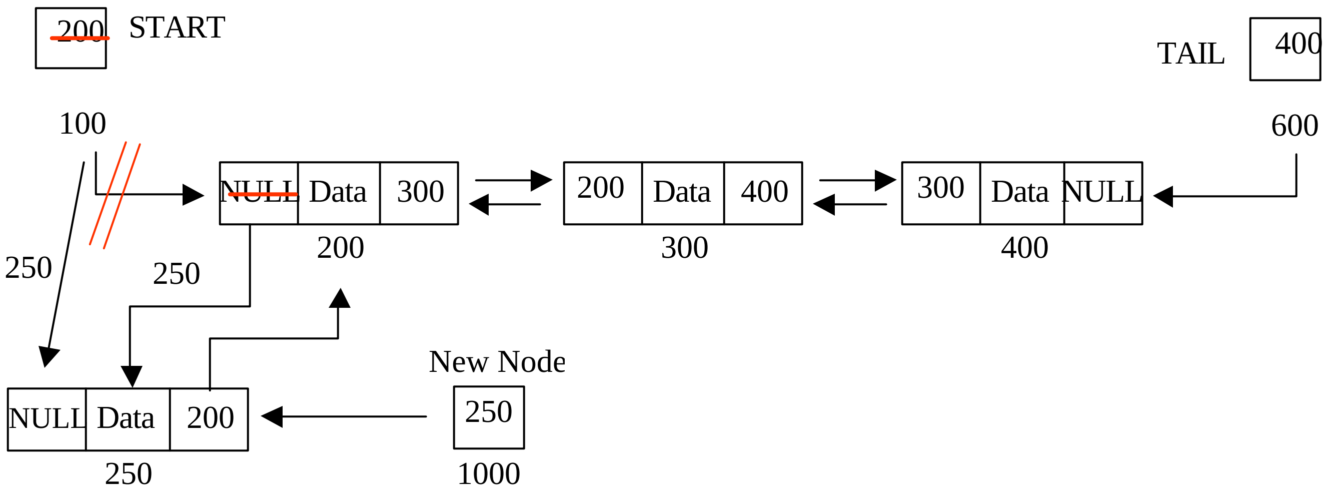


Fig 7-2 Inserting a node in the beginning of a Doubly Linked List

```
node* getNode(int iData)
{
  node* pNewNode = (node *)malloc(sizeof(node));
  pNewNode->iData = iData;
  pNewNode->pNext = NULL;
  pNewNode->pPrev = NULL;
}
void addHead(int iData)
{
  node* pNewNode = getNode(iData);
  pNewNode->pNext = pSTART;
  pSTART->pPrev   = pNewNode;
  pSTART          = pNewNode;
}
```

Steps for inserting an element in the middle of a doubly linked list

1. Create a new node
2. Make the next part of new node equal to next part of the previous node
3. Make the previous part of new node equal to previous part of next node
4. Make the next part of previous node point to new node
5. Make the previous part of next node point to new node

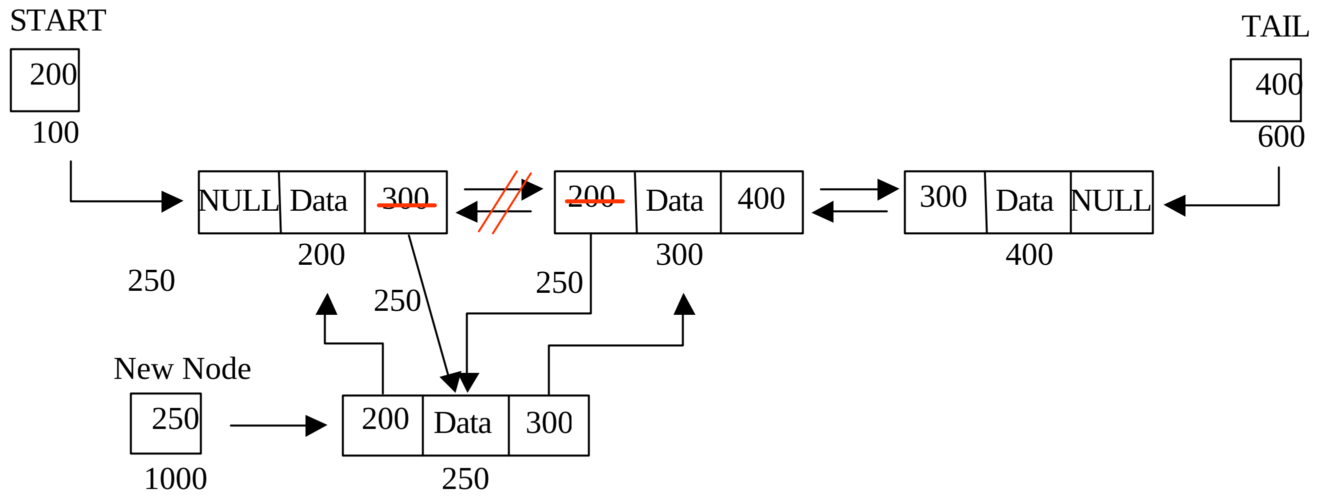


Fig 7-3 Inserting a node in the middle of a Doubly Linked List

```
void insertAt(int iLoc, int iData)
{
  node* pNewNode  = NULL;
  node* pTempNode = NULL;
  int iPos = 1;
  if (iLoc == 1)
  {
    addHead(iData);
  }
  else
  {
    for(pTempNode = pSTART; pTempNode->pNext != NULL && iLoc < iPos;
                            pTempNode = pTempNode->pNext, iPos++);
      if (iLoc == iPos)
    {
      pNewNode = getNode(iData);
      pNewNode->pNext = pTempNode->pNext;
      pNewNode->pPrev = pTempNode;

      if (pTempNode->pNext != NULL)
      {
        pTempNode->pNext->pPrev = pNewNode;
      }
      else  // If it is the last node
      {
        pTAIL = pNewNode;
      }
      pTempNode->pNext = pNewNode;
    }
    else
    {      printf(“Invalid Position”);
    }
  }
}
```

Steps for inserting an element at the end of a doubly linked list

1. Create a new node
2. Make the next part of the new node equal to NULL
3. Make the previous part of the new node equal to TAIL
4. Make the next part of the previous node equal to new node
5. Make TAIL equal to new node

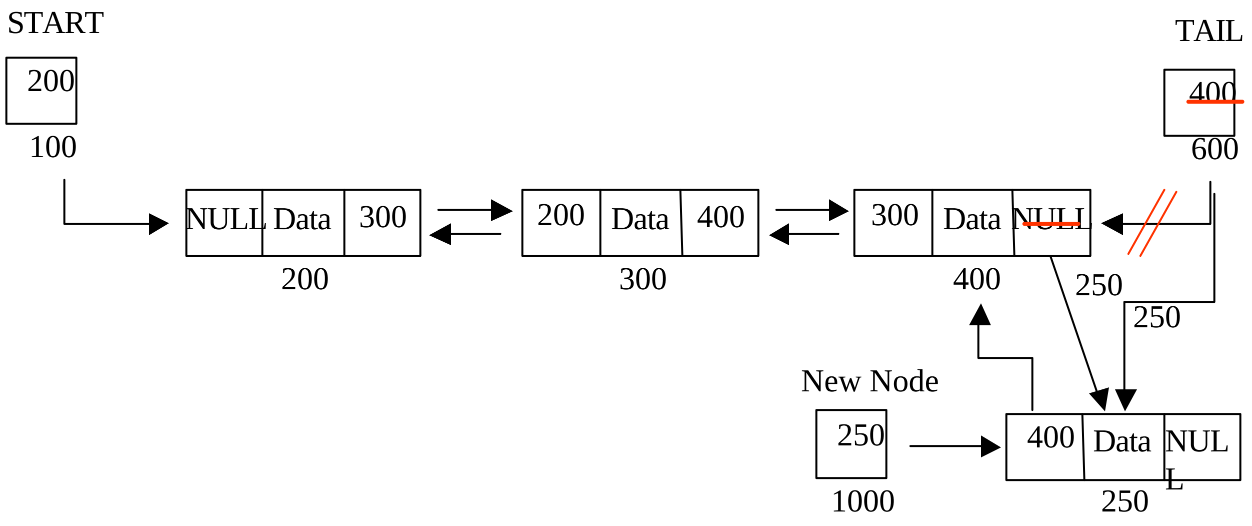


Fig 7-4 Inserting a node at the end of a Doubly Linked List

```
void addTail(int iData)
{
  node* pNewNode  = NULL;
  node* pTempNode = NULL;
  pNewNode = getNode(iData);
  pNewNode->pPrev  = pTAIL;
  pTail->pNext = pNewNode;
  pTAIL = pNewNode;
}
```

Deleting an element from a doubly linked list

Deleting a node from a list is as simple as changing the links. Hence deleting a node from a list is much faster when compared to arrays. Like insertion the deletion logic also varies depending on from where in the list we are going to delete the node.

Deletion in the beginning of a doubly linked list

1. Make the temporary pointer point to the node to be deleted
2. Make the START point to the next node of START
3. Make the previous of the next node equal to previous of the node to be deleted
4. Delete the node pointed to by temporary pointer

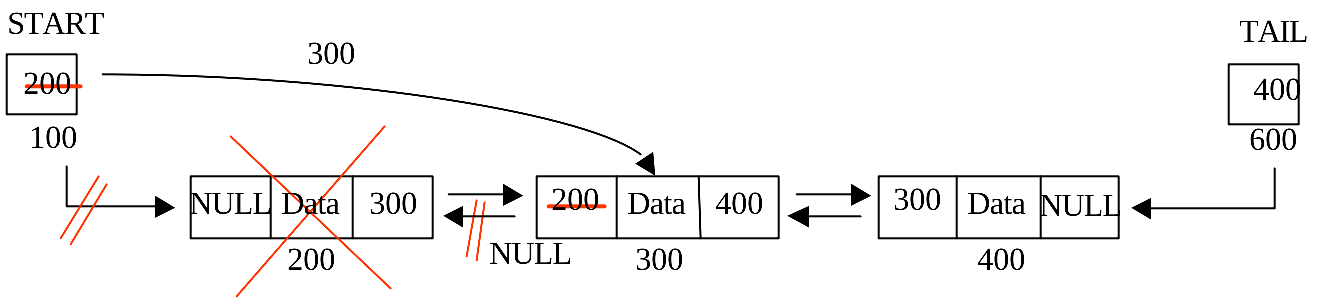


Fig 7-5 Deleting a node from the beginning of a Doubly Linked List

```
void removeHead()
{
  node* pDelNode = NULL;
  pDelNode = pSTART;
  pSTART = pSTART->pNext;
  if (pDelNode == pTAIL)  // If it is the last element in the list
  {
    pTAIL = NULL;
  }
  else
  {
    pSTART->pPrev = NULL;
  }
  free(pTempNode);
}
```

Deletion in the middle of a doubly linked list

1. Make the temporary pointer point to the node to be deleted
2. Make the next part of the previous node equal to next of the node to be deleted
3. Make the previous part of the next node equal to previous part of the node to be deleted
4. Delete the node pointed to by temporary pointer

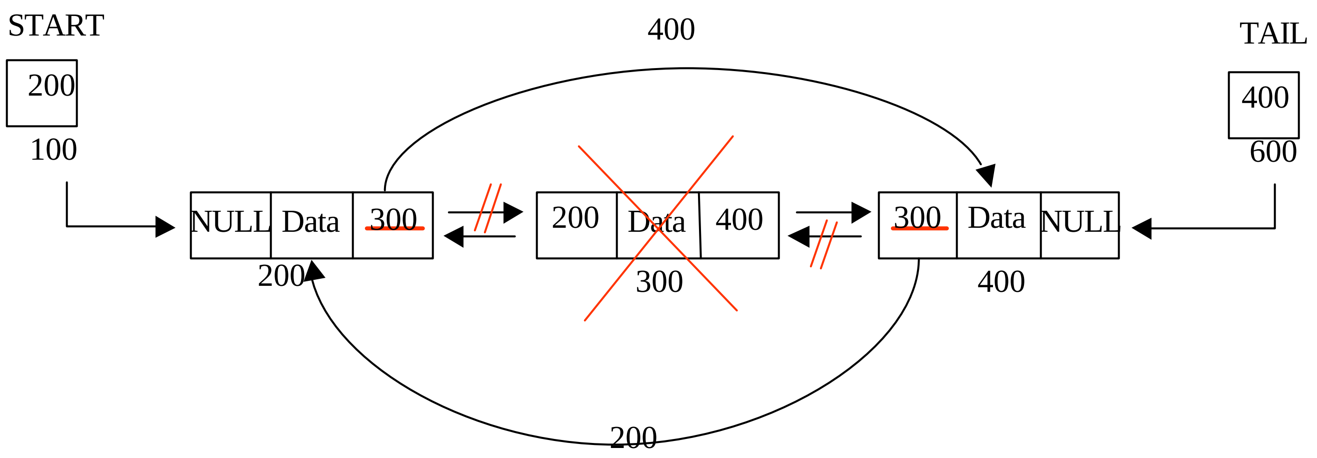


Fig 7-6 Deleting a node from the middle of a Doubly Linked List

```
void deleteAt(int iLoc)
{
  node* pTempNode = NULL;
  node* pDelNode    = NULL;
  int iPos = 1;

  if (iLoc == 1)
  {
    removeHead();
  }
  else
  {
    for (pTempNode = pSTART; iPos < iLoc && pTempNode->pNext != NULL;
                        pTempNode = pTempNode->pNext, iPos++);
```

```
        if (iLoc == iPos)
        {
          pDelNode = pTempNode->pNext;
          pTempNode->pNext = pDelNode->pNext;

          if (pDelNode == pTAIL)
          {
            pTAIL = pTAIL->pNext;
          }
          else
          {
            pDelNode->pNext->pPrev = pTempNode;
          }
          free(pDelNode);
        }
      }
    }
```

Deletion at the end of a doubly linked list

1. Make the temporary pointer point to the node to be deleted
2. Make the next part of the previous node equal to next of the node to be deleted
3. Make the TAIL equal to the previous part of the node to be deleted
4. Delete the node pointed to by temporary pointer

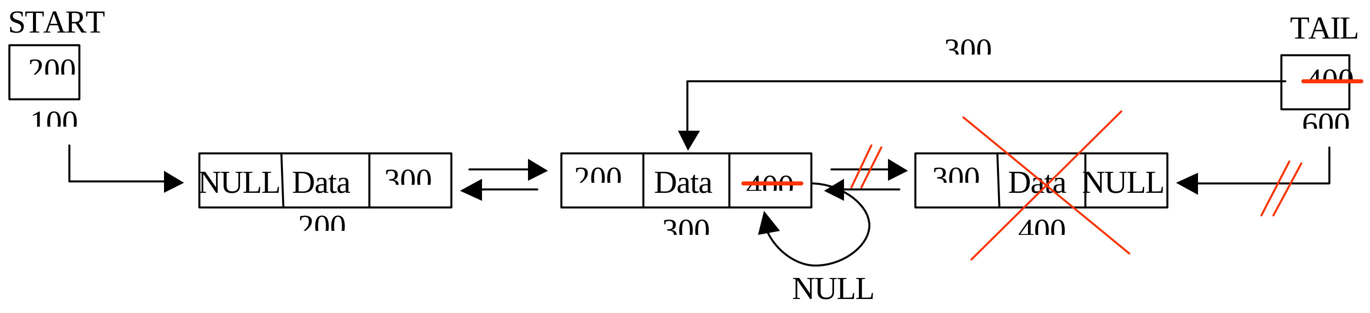


Fig 7-7 Deleting a node from the end of a Doubly Linked List

```
void removeTail()
{
  struct node* pTempNode = NULL;
  struct node* pDelNode  = NULL;
  if (pSTART == NULL)
  {
    printf(“List is Empty”);
  }
  else
  {
    for (pTempNode = pSTART; pTempNode->pNext != NULL;
                          pTempNode = pTempNode->pNext);

    pDelNode = pTempNode->pNext;
    pTempNode->pNext = NULL;
    pTAIL = pTAIL->pNext;
    if (pTAIL == NULL)
    {
      pSTART = NULL;
    }
    free(pDelNode);
  }
}
```

Traversing a doubly linked list

A doubly linked list can be traversed in both directions.

```
void displayFromStart()
{
  node* pTempNode = NULL;
  for(pTempNode = pSTART; pTempNode != NULL; pTempNode = pTempNode-
>pNext)
  {
    printf(“Data = %d\n”, pTempNode->iData);
  }
}
void displayFromTail()
{
  node* pTempNode = NULL;
```

```
 for(pTempNode = pTAIL; pTempNode != NULL; pTempNode = pTempNode->pPrev)
 {
   printf(“Data = %d\n”, pTempNode->iData);
 }
}
```

Application of Doubly Linked Lists

✍✍ A doubly linked list can be used in all the places where we use a linear list if the traversal happens frequently in both the directions.

Advantages

✍✍ The number of elements in a doubly linked list need not be known in advance. As and when required elements can be added to a list

✍✍ Insertions and deletions are much faster as there is no physical movement of elements of a list.

✍✍ Traversal is possible in both the directions unlike a linear list where the traversal is possible in only one direction

Limitations

✍✍ Takes up additional memory for storing the previous pointer in each node

✍✍ Makes the insertion and deletion logic a bit complex

SUMMARY

✍✍ A doubly linked list is a list in which each node will have 3 parts: one part containing the information / data, one part containing the pointer/address of next node and one part containing the pointer / address of previous node.

Test your Understanding

1. Write a program to insert, delete, traverse and display elements of a doubly linked list.

2. Use a circular linked list to solve the Josephus problem

Exercise 5

3. Write a program to implement doubly linked lists of integers and add a function called traverse backward.

Chapter 8: Trees

Learning Objective

After completing this chapter, you will be able to:

- Understand and write programs to create Tree and B-Tree

Introduction To Trees

A tree is a finite set of one or more nodes such that

i) There is a specially designated node called the root

ii) The remaining nodes are partitioned into n >= 0 disjoint sets T1, ..., Tn where each of these sets is a tree. T1, ..., Tn are called sub trees of the root

Graphical Representation

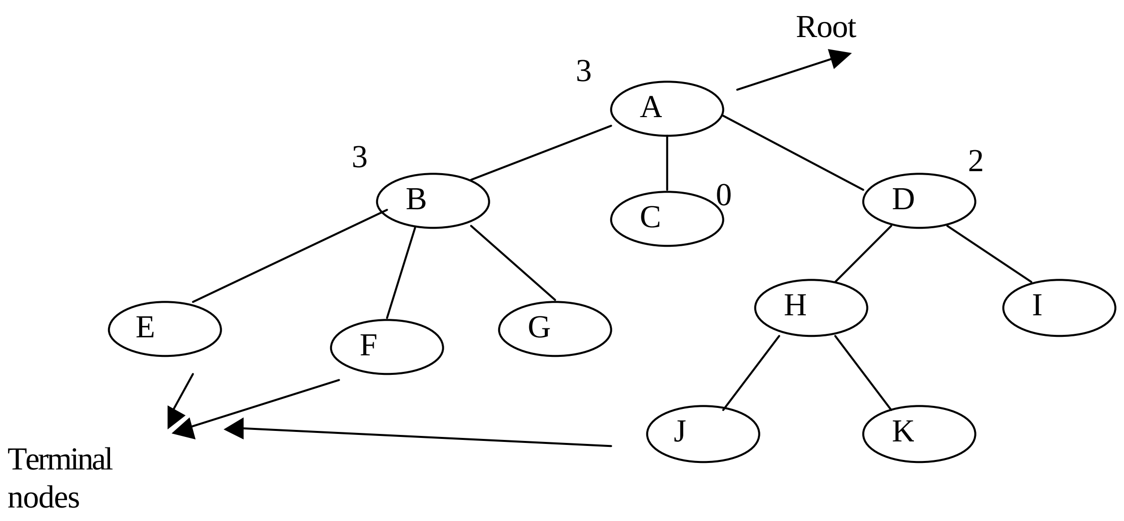


Fig 8-1 Graphical Representation of a Tree

A tree is a non-linear data structure. This structure is mainly used to represent data containing a hierarchical relationship between elements like family tree, organization chart etc.

Some of the commonly used terms and their definitions with respect to a tree are as follows:

Definitions

Degree of a node

The number of sub trees of a node is called its degree. In the diagram shown above the degree of A is 3, B is 3, C is 0 , D is 2, ...

Terminal or Leaf Nodes

Nodes that have degree 0 are called as Terminal or Leaf nodes. In other words the nodes that does not have any sub trees are called Terminal or Leaf nodes

Siblings

The children of the same parent are called as siblings

Level

The level of the root node is 1. If a node is at level L then its children are at level L + 1.

Height or Depth of a Tree

The height or depth of a tree is the maximum level of any node in the tree.

Forest

A forest is a set of n>= disjoint trees.

Introduction to Binary Trees

A binary tree T is defined as a finite set of elements, called nodes, such that:

- T is empty or
- T contains a distinguished node R, called the root of T, and the remaining nodes of T forms an ordered pair of disjoint binary trees T1 and T2.
- If T does contain a root R, then the two trees T1 and T2 are called the left and right sub trees of R

Graphical Representation

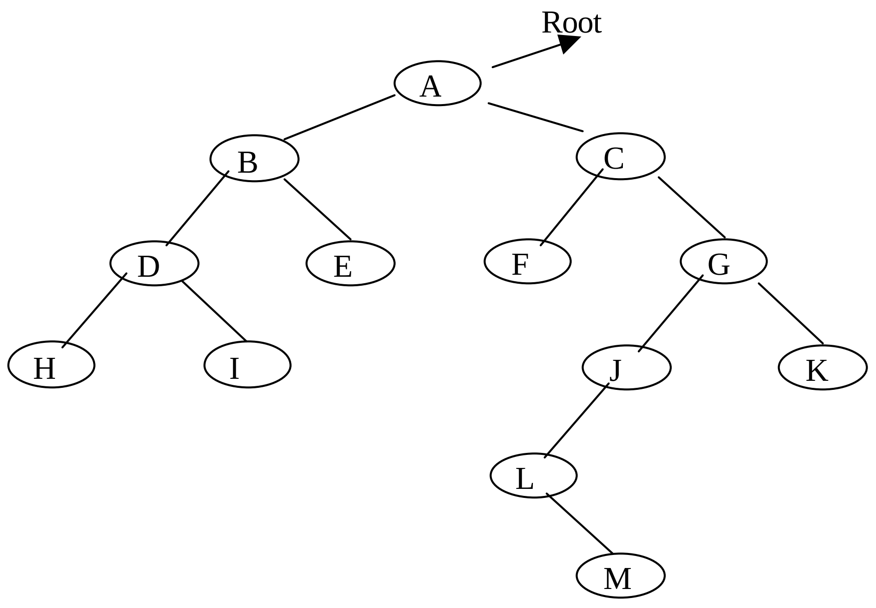


Fig 10-1 Graphical Representation of a Binary Tree

A binary tree is said to be complete if all its levels except the last level have the maximum number of possible nodes and if all the nodes at the last level appear as far left as possible.

Binary Search Trees

A binary tree T is called a binary search tree if each node of T has the property: The value at N is greater than every value in the left sub tree of N and is less than every value in the right sub tree of N.

Memory Organization of a binary tree

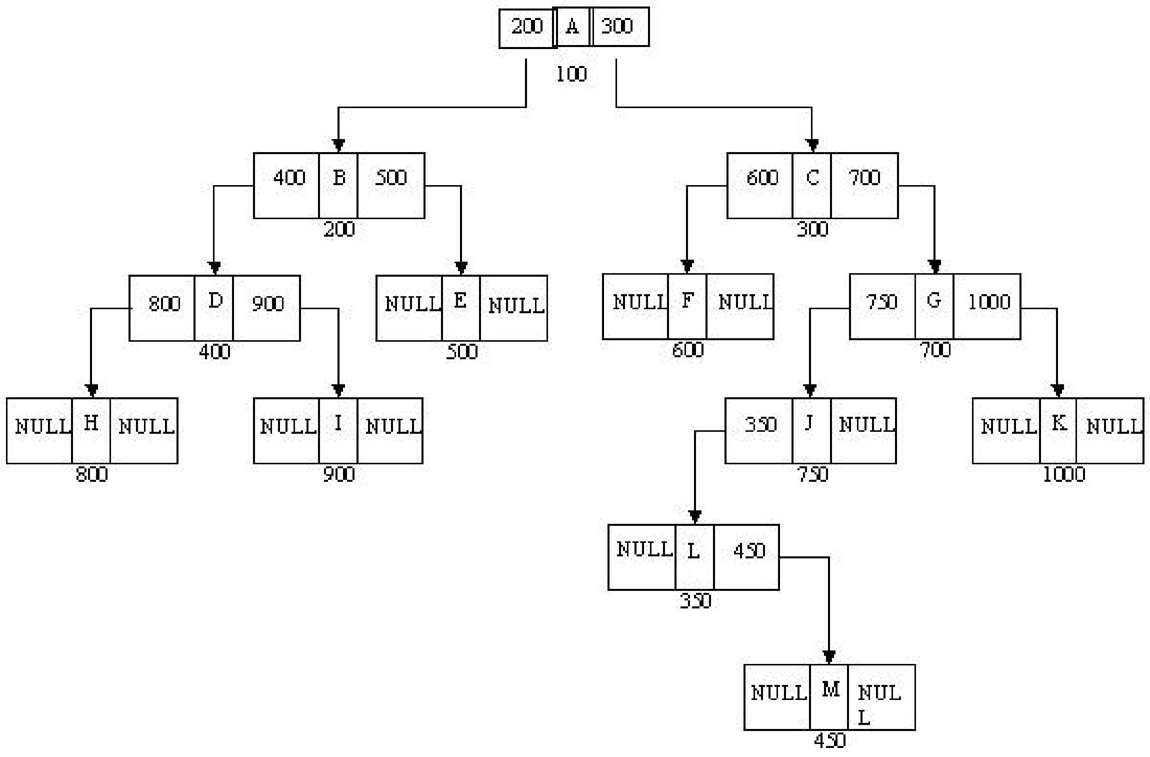


Fig 10-2 Memory Organization of a Binary Tree

Programmatic Representation of a Binary Tree

```
typedef struct node
{
  int iData;
  struct node* pLeft;
  struct node* pRight;
}node;
```

```
node* pROOT = NULL;
```

Inserting an element in a binary search tree

Inserting an element in a binary search tree involves creating a new node and re-organizing the parent nodes link to accommodate the new node. Typically insertions will always be on a terminal node. We will never be inserting a node in the middle of a binary search tree.

Steps for inserting an element in a binary search tree

1. If the tree is empty, then make the root node point to this node
2. If the tree contains nodes then compare the data with node’s data and take appropriate path till the terminal node is reached
3. If data < node’s data then take the left sub tree otherwise take the right sub tree
4. When there are no more nodes add the node to the appropriate place of the parent node.

```
node* getNode(int iData)
{
    node* pNewNode = (node *)malloc(sizeof(node));
    pNewNode->iData     = iData;
    pNewNode->m_pLeft  = NULL;
    pNewNode->m_pRight = NULL;
    return pNewNode;
}
void  insert(node* pParent, int iData)
{
  if (pParent == NULL)
  {
    pParent = getNode(iData);
    if (pRoot == NULL)
    {
      pRoot = pParent;
    }
  }
  else
  {
    if (iData < pParant->iData)
    {
       insert(pParent->pLeft, iData);
    }
```

```
                else
                {
                  insert(pParent->pRight, iData);
                }
} } // end of function …
```

Deleting an element from a binary tree

The logic for deleting an element from a binary tree varies depending on from where we are going to delete the node, whether it is a leaf node, a node with a single child or a node in the middle.

- If the node being deleted is a terminal node then deletion is a straightforward procedure of just making the parent node point to NULL.
- If the node being deleted has only one child then the parent node will have to point to the child node of the node being deleted.
- If the node being deleted has both the children then we first need to find the inorder successor of the node being deleted and replace the node being deleted with this node. (The inorder successor of a node can be obtained by taking the right node of the current node and traversing in the left till we reach the left most nodes.)

Steps for deleting an element from a binary search tree

- If the node N has no children, then change the location of this node in the parent node to NULL.
- If the node N has exactly one child, then change the location of this node, in the parent, to child of N.
- If the node N has 2 children, then
    a. Store N in a temporary pointer
    b. Determine the inorder successor of N (an inorder successor is obtained by moving to the right node and finding the left most node of that node)
    c. Make the left of this node point to left of N
    d. Make the right of this node point to right of N
    e. Make the parent of N point to this node.
    f. Delete the node pointed by temporary pointer

```
node* delete(node* pParent, int iData)
{
  node* pTempNode1, *pTempNode2;

  if (pROOT == NULL)
  {
    printf(“Tree is Empty”);
    return NULL;
  }
  else
  {
    if (iData == pParent->iData)
    {
      if (pParent->pLeft == pParent->pRight) // if it is a terminal node
      {
         free(pParent);
         return NULL;
      }
      else if (pParent->pLeft == NULL) // if there is only right child
      {
         pTempNode1 = pParent->pRight;
         free(pParent);
         return pTempNode1;
      }
      else if (pParent->pRight == NULL) // if there is only left child
      {
         pTempNode1 = pParent->pLeft;
         free(pParent);
         return pTempNode1;
      }
      else  //if it has both the child
      {
        pTempNode2 = pParent->pRight;
         // Get the inorder successor of pParent
        for (pTempNode1 = pParent->pRight; pTempNode1->pLeft != NULL;
                                                          pTempNode1 = pTempNode1-
>pLeft);
        pTempNode1->pLeft = pParent->pLeft;
        free(pParent);
        return pTempNode2;
      }
    }
```

```
        if (iData < pParent->iData)
        {
          pParent->pLeft = delete(pParent->pLeft, iData);
        }
        else
        {
          pParent->pRight = delete(pParent->pRight, iData);
        }
        return pParent;
      }
    } // end of function delete …
```

Binary Tree Traversal

A binary tree can be traversed in 3 ways namely

- Preorder Traversal / NLR Traversal
  - a) Process the root
  - b) Traverse the left sub tree in preorder
  - c) Traverse the right sub tree in preorder
- Inorder Traversal / LNR Traversal
  - a) Traverse the left sub tree in inorder
  - b) Process the root
  - c) Traverse the right sub tree in inorder
- Postorder Traversal / LRN Traversal
  - a) Traverse the left sub tree in postorder
  - b) Traverse the right sub tree in postorder
  - c) Process the root

From the definition itself it is very clear that the implementation of this traversal algorithm involves recursion.

Example:

```
      9
    /   \
   4     3
  / \     \
 8   2     5
```

For in-order traversal, start at the root and first traverse each node's left branch, then the node and finally the node's right branch. This is a recursive process since each left and right branch is a tree in its own right. For example, with the tree of Example above, in-order traversal produces the sequence of values 8 4 2 9 3 5.

For post-order traversal, start at the root and first access each node's left branch, then the node's right branch and finally the node itself. For the tree of Example, the corresponding sequence is 8 2 4 5 3 9.

In the case of pre-order traversal, start at the root and access the node itself, its left branch and finally its right branch. This leads to the sequence 9 4 8 2 3 5 for the tree of Example.

Code Snippet for Inorder Traversal

```
void InOrder(struct node* pParent)
{
  if ( pParent != NULL )
  {
    InOrder(pParent->pLeft);
    printf("Data = %d\n", pParent->iData );
    InOrder(parent->pRight);
  }
}
```

Code Snippet for Preorder Traversal

```
void PreOrder(struct node* pParent)
{
  if ( pParent != NULL )
  {
    printf("Data = %d\n", pParent->iData );
    PreOrder(pParent->pLeft);
    PreOrder(parent->pRight);
  }
}
```

Code Snippet for Postorder Traversal

```
void PostOrder(struct node* pParent)
{
  if ( pParent != NULL )
  {
    PostOrder(pParent->pLeft);
```

```
        PostOrder(parent->pRight);
        printf("Data = %d\n", pParent->iData );
      }
    }
```

Application of trees

A tree can be used in a wide variety of applications, which includes set representations, decision-making, game trees, etc. Within the context of programming language execution, compilers utilize tree structures to obtain forms of an arithmetic expression, which can be evaluated efficiently. The in-order traversal of the binary tree for an arithmetic expression produces the infix form of the expression, while the pre-order and post-order traversal lead to the prefix and postfix (reverse Polish) forms of the expression respectively.

The advantage of reverse Polish notation is that arithmetic expressions can be represented in a way, which can be simply read from left to right without the need for parentheses. For example, consider the expression a * (b + c). This expression can be represented by the binary tree in the following way,

```
   *
  / \
 a   +
    / \
   b   c
```

If we perform a post-order traversal in which we traverse the left branch of the tree, the right branch, followed by the node, we immediately recover the reverse Polish form of the expression, namely abc+*. This expression can then be evaluated using a stack. Starting from the left of the expression, each time an operand (one of the numerical values a, b or c in this example) is found, it is placed on the top of the stack.

When an operator (* or +) is read, the top two elements are removed form the stack and the appropriate operator applied to them and the result placed on the stack. When the complete expression has been evaluated, the stack will contain a single item, which is the result of evaluating the expression.

SUMMARY

- A tree is a finite set of one or more nodes such that,
  - There is a specially designated node called the root
  - The remaining nodes are partitioned into n >= 0 disjoint sets T1, ..., Tn where each of these sets is a tree. T1, ..., Tn are called sub trees of the root
- A binary tree T is called a binary search tree if each node of T has the property: The value at N is greater than every value in the left sub tree of N and is less than every value in the right sub tree of N.
- A binary tree can be traversed in 3 ways namely

- ✍✍ Preorder Traversal / NLR Traversal
- ✍✍ Inorder Traversal / LNR Traversal
- ✍✍ Postorder Traversal / LRN Traversal

Test your Understanding

1. Write code that counts the number of nodes in a given binary tree.
2. Write an algorithm that determines given a binary tree whether it’s a complete binary tree or not.
3. Write code to traverse a tree in preorder.
4. Write code to traverse a tree in post-order.

Exercise 6

5. Write code to create a BST and functions to insert and delete nodes into and from the BST.

Chapter 9: Introduction To Graphs

Learning Objective

After completing this chapter, you will be able to**:**

- Understand and write programs to create graphs

Introduction To Graph Theory

A graph G consists of two things:

a) A set of elements called vertices / nodes / points V.

b) A set of edges E, such that each edge e in E is identified with a unique pair [u,v] of nodes in V, denoted by e = [u, v]

A graph can also be represented as G =(V, E)

Before getting into the details of a Graph let’s see few of the commonly used terms with respect to graph and their definitions:

Adjacent Nodes

If e = [u, v] then the nodes u and v are called endpoints of e and u and v are said to be adjacent nodes or neighbors.

Definitions

Degree of a node

The degree of a node u is the number of edges containing u. If deg(u) = 0, then the node is called as an isolated node.

Path

A Path P of length n from node u to v is defined as a sequence of n + 1 nodes. P = (V0, V1,....Vn)

Closed Path

A path P is said to be closed if V0 = Vn

Simple Path

The path P is said to be simple, if all the nodes are distinct, with the exception that v0 may equal v1, that is, P is simple if the nodes v0, v1, ... vn-1 are distinct and the nodes v0, v1, ...vn are distinct.

Cycle

A cycle is a closed simple path.

Connected Graph

A graph G is connected if and only if there is a simple path between any two nodes in G.
Complete Graph

A graph G is said to be complete if every node u in G is adjacent to every other node v in G.
A graph G is said to be labeled, if its edges are assigned data. A graph is said to be weighted if each edge e in G is assigned a non-negative numerical value called the weight or length of e.

Multiple edges

Distinct edges e and e' are called multiple edges if they connect the same end points. i.e. if e = [u, v] and e' = [u, v].

Loops

An edge e is called a loop if it has identical endpoints. i.e. e = [u, u]

Multi Graph

A graph, which contains multiple edges or Loops, is called as Multigraph M.

Directed Graphs

A directed graph G is the same as a Multigraph except that each edge e in G is assigned a direction.

Out Degree

The number of edges beginning at a node u is the outdegree of that node and the number of edges ending at the node e is the indegree of e. A node is called a Source if it has positive outdegree and zero indegree. A node is called as a Sink if it has a positive indegree and zero outdegree.

Memory Organization of a Graph

A graph is represented in memory with the help of a linear list of all the nodes in the graph called as the node list. The adjacent nodes for every node are again maintained by means of a list, which is called as the adjacency list. Consider the following graph:

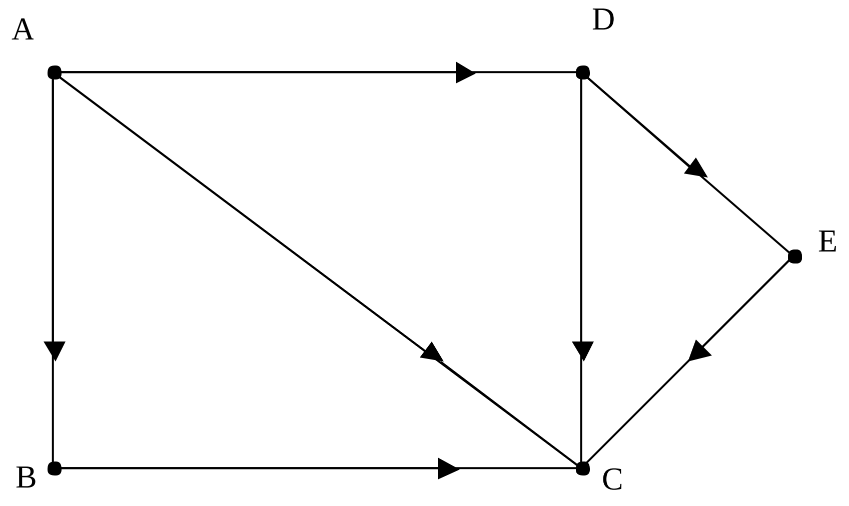


Fig 11-1 Graphical Representation of Graph

First prepare an adjacency list for every node which is as shown in the table:

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

Table 11-1 Adjacency List for the graph

The memory representation for the above graph is shown below

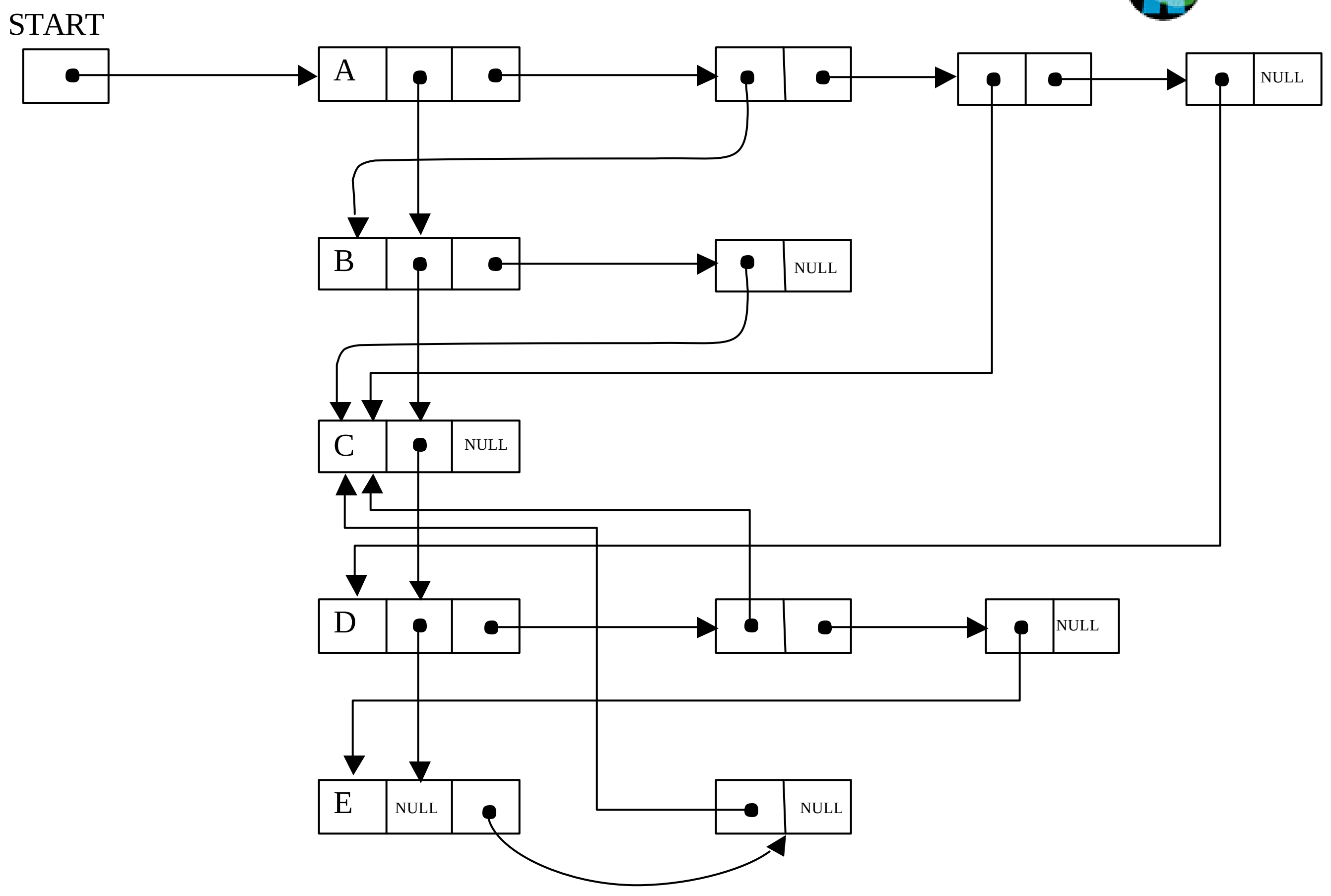


Fig 11-2 Memory Organization of a Graph

Structural Representation of a Graph

```
typedef struct listNode
{
  struct node* pData;
  struct listNode* pNext;
}listNode;
```

The above structure is for the adjacency list. The data part represents a pointer to a node in the node list and the next part contains the link for the subsequent node in the adjacency list

```
typedef struct node
{
  int iData;
  struct node* pNext;
```

```
        struct listNode* pAdjNode;
    }node;
```

The above structure is for the node list. It contains 3 parts. The first part contains the data, the second part contains a pointer to the next node in the node list and the third part contains a pointer to the adjacency list.

Inserting into graph

We can insert either a node or an edge in a graph. Inserting a node involves inserting an element in the node list and the adjacent part has to be set NULL as this will be an isolated node. Inserting an edge involves inserting an element in to the adjacency list of all the nodes, which are adjacent to that node.

Deleting from a graph

Deleting an element from a graph again involves deleting an edge in which case the adjacency list of all the nodes adjacent to this node will have to be deleted and the node as such will have to be deleted from the node list. Deleting an edge involves deleting an element from the adjacency list of the node that is getting deleted.

Traversing a graph

There are 2 ways by means of which a graph can be traversed namely:

- **Breadth First Search**

  Check the starting node A. The check all the neighbors of A. Then examine the neighbors of neighbors of A and so on.

- **Depth First Search**

  Check the starting node A. Then search each node N along the path P, which begins at node A. i.e., search neighbor of A, then neighbor of neighbor of A and so on. Once we reach the dead end then we backtrack on P until we can continue along another path P’ and so on.

Application of Graphs

Graphs are used in a wide variety of applications. A graph can be used by an airlines for maintaining the flight information such as the various flights, their routes and the distance between the places, etc. With the help of such a graph one will be easily able to find out whether there is any flight for a particular destination and if there are several routes to reach that destination which one is having the optimum distance or cost, etc.

SUMMARY

- A graph G consists of two things:
  - o A set of elements called vertices / nodes / points V.

- o A set of edges E, such that each edge e in E is identified with a unique pair [u,v] of nodes in V, denoted by e = [u, v]

✍✍ A graph can also be represented as G =(V, E)

✍✍ There are 2 ways by means of which a graph can be traversed namely:

- o Breadth First Search
- o Depth First Search

Test your Understanding

1. Write a program for depth first traversal of a graph.

2. Write a program for breadth first traversal of a graph.

Exercise 7

3. Write an algorithm followed by the code that determines if a node is disjoint.

GLOSSARY

Appendix A

[Code Samples]

Matrix Addition using Pointers

```
#include <stdio.h>
#include <malloc.h>
void accept(int ***x, int iRow, int iCol)
{
  int i, j;

  *x = (int *)malloc(iRow * sizeof(int));

  for(i = 0; i < iRow; i++)
  {
    *(*x + i) = (int *)malloc(iCol * sizeof(int));

    for(j = 0; j < iCol; j++)
    {
      scanf(“%d”, (*(*x + i) + j));
    }
  }
}

void display(int **x, int iRow, int iCol)
{
  int i, j;

  for(i = 0; i < iRow; i++)
  {
    for(j = 0; j < iCol; j++)
    {
      printf(“%d \t”, *(*(x + i) + j));
    }
    printf(“\n”);
  }
}

int** add(int **x, int **y, int iRow, int iCol)
{
  int i, j;
  int**z;
  z = (int *)malloc(iRow * sizeof(int));
  for(i = 0; i < iRow; i++)
  {
    *(z + i) = (int *)malloc(iCol * sizeof(int));
    for(j = 0; j < iCol; j++)
    {
      *(*(z + i) + j) = *(*(x + i) + j) + *(*(y + i) + j);
    }
  }
  return z;
} // end of function add ...
void main()
{
  int **a, **b, **c;

  int iRow, iCol;


  printf(“Enter No of Rows : “);

  scanf(“%d”, &iRow);


  printf(“Enter No of Cols : “);

  scanf(“%d”, &iCol);


  printf(“Enter values for Matrix A :\n”);
  accept(&a, iRow, iCol);
  printf(“Enter values for Matrix B :\n”);
  accept(&b, iRow, iCol);
  c = add(a, b, iRow, iCol);
  printf(“Added Matrix is :\n”);
  display(c, iRow, iCol);
} // end of main...
```

A Complete Linked List Example

```
#include <stdio.h>
#include <malloc.h>
typedef struct node

{

   int   iData;

   struct node* pNext;

}node;

node* pSTART = NULL;

node* getNode()
{
  int iData;
  node* pNewNode = NULL;
  printf("Enter a number : ");
  scanf("%d", &iData);
  pNewNode = (node*)malloc(sizeof(node));
  pNewNode->iData = iData;
  pNewNode->pNext = NULL;
  return NULL;
}
void addHead()
```

```
{
  node* pNewNode = getNode();
  pNewNode->pNext = pSTART;
  pSTART = pNewNode;
}
```

void insertAt(int iLoc)

```
{
  int iPos = 1;
  node* pTempNode = NULL;
  node* pNewNode  = NULL;
  if (iLoc == 1)
  {
    addHead();
  }
  else
  {
    for (pTempNode = pSTART; pTempNode->pNext != NULL && iPos < iLoc;
                         pTempNode = pTempNode->pNext, iPos++);
    if (iPos == iLoc)
    {
      pNewNode = getNode();
      pNewNode->pNext  = pTempNode->pNext;
      pTempNode->pNext = pNewNode;
    }
  }
}
```

void addTail()

```
{
  int iPos = 0;
  node* pTempNode = NULL;
  node* pNewNode  = NULL;
  for (pTempNode = pSTART; pTempNode->pNext != NULL;
                         pTempNode = pTempNode->pNext);
  pNewNode = getNode();
  pTempNode->pNext = pNewNode;
```

```
}
void delete(int iData)
{
  node* pTempNode = NULL;
  node* pDelNode  = NULL;
  node* pPrevNode = NULL;
  for(pTempNode = pPrevNode = pSTART; pTempNode != NULL;
                                   pPrevNode = pTempNode,
                                   pTempNode = pTempNode->pNext)
  {
    if (iData == pTempNode->iData)
    {
      pDelNode = pTempNode;
      pPrevNode->pNext = pDelNode->pNext;
      if (pDelNode == pSTART)
      {
        pSTART = pSTART->pNext;
      }
      free(pDelNode);
    }
  }
}
void display()
{
  node* pTempNode = NULL;
  for(pTempNode = pSTART; pTempNode != NULL; pTempNode = pTempNode-
>next )
  {
    printf("Data = %d\n", pTempNode->iData);
  }
}
void main()
{
  int iChoice;
  int iLoc;
  int iData;
  printf(“1. Enter 1 To add a node in the beginning \n”);
  printf(“2. Enter 2 To add a node in the middle \n”);
  printf(“3. Enter 3 To add a node in the end \n”);
  printf(“4. Enter 4 To delete a node in the list \n”);
  printf(“5. Enter 5 To display the list \n”);
  fflush(stdin);
  scanf(“%d”, &iChoice);
  if (iChoice == 1)
  {
    addHead();
    display();
  }
  else if (iChoice == 2)
  {
    printf(“Enter the location to insert “);
    scanf(“%d”, &iLoc);
    insertAt(iLoc);
    display();
  }
  else if (iChoice == 3)
  {
    addTail();
    display();
  }
  else if (iChoice == 4)
  {
    printf(“Enter the element to Delete “);
    scanf(“%d”, &iData);
    remove(iData);
    display();
  }
  else if (iChoice == 5)
  {
    display();
  }
}
```

REFERENCES

WEBSITES

http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/ds_ToC.html

BOOKS

- Data Structures and Algorithms, Niklaus Wirth
- Data Structure using C and C++, LANGSAM, AUGESTEIN, TANENBAUM

STUDENT NOTES: