OXFORD
HIGHER EDUCATION

Data Structures Using C

Reema Thareja, Assistant Professor
Institute of Information Technology and Management
GGS IP University

OXFORD
HIGHER EDUCATION

CHAPTER 4

INTRODUCTION TO DATA STRUCTURES

INTRODUCTION

- A *data structure* is nothing but an arrangement of data either in computer's memory or on the disk storage.
- Some common examples of data structure includes arrays, linked lists, queues, stacks, binary trees, and hash tables.
- Data structures are widely applied in areas like:

  a. Compiler design
  b. Operating system
  c. Statistical analysis package
  d. DBMS
  e. Numerical analysis
  f. Simulation
  g. Artificial Intelligence
  h. Graphics

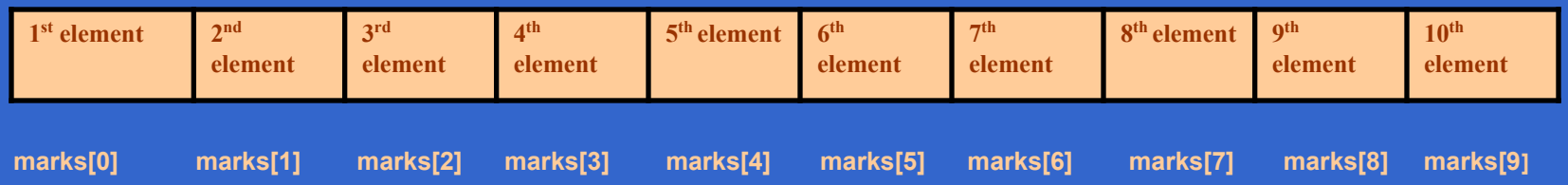
DIFFERENT TYPES OF

DATA STRUCTURES

Arrays

- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript).
- Arrays are declared using the following syntax.

  type name[size];



The limitations with arrays include:

Arrays are of fixed size.

Data elements are stored in continuous memory locations which may not be available always.

Adding and removing of elements is problematic because of shifting the elements from their positions.

Linked List

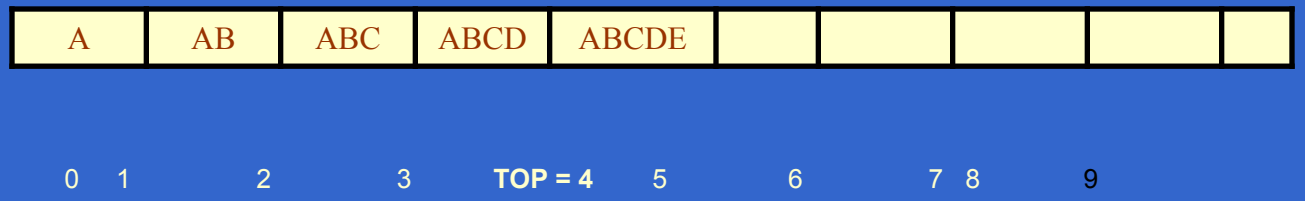
- Linked list is a very flexible dynamic data structure in which elements can be added to or deleted from anywhere at will.
- In contrast to using static arrays, a programmer need not worry about how many elements will be stored in the linked list. This feature enables the programmers to write robust programs which require much less maintenance.
- In a linked list, each element (is called a node) is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list every node contains two types of information:

  The data stored in the node and

  A pointer or link to the next node in the list
- *Advantage:* Provides quick insert, and delete operations
- *Disadvantage:* Slower search operation and requires more memory space

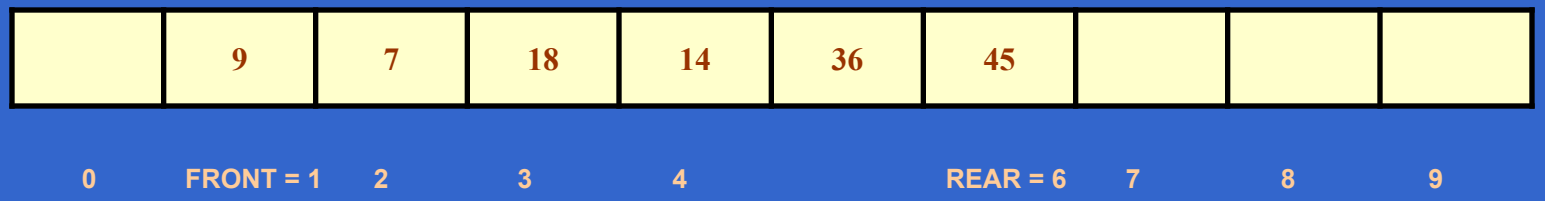
Stack

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it. Top is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted. There is another variable MAX which will be used to store the maximum number of elements that the stack can store.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX, then the stack is full.
- is full.
- If TOP = -1, it indicates that there is no element in the stack.
- *Advantage:* Last-in, first-out access (LIFO)
- *Disadvantage:* Slow access to other elements



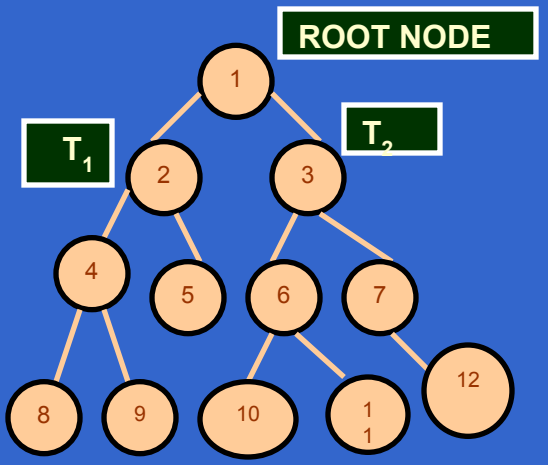
Queue

- A queue is a FIFO (First In First Out) data structure in which the element that was inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the rear and removed from the other one end called front. Like stacks, queues can be implemented either using arrays or linked lists.
- When Rear = MAX – 1, where MAX is the size of the queue that is, MAX specifies the maximum number of elements in the queue.
- If front = -1 and rear = -1, this means there is no element in the queue.
- *Advantage:* Provides first-in, first-out data access
- *Disadvantage:* Slow access to other items



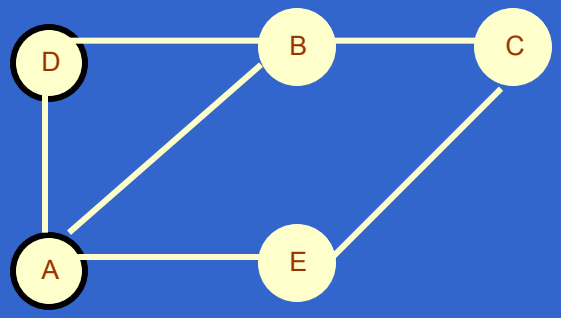
Tree

- A binary tree is a data structure which is defined as a collection of elements called nodes.
- Every node contains a "left" pointer, a "right" pointer, and a data element.
- Every binary tree has a root element pointed by a "root" pointer. The root element is the topmost node in the tree.
- If root = NULL, then it means the tree is empty.
- *Advantages:* Provides quick search, insert and delete operations
- *Disadvantage:* Complicated deletion algorithm



Graph

- A graph is a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can be represented.
- While in trees, the nodes can have many children but only one parent, a graph on the other hand relaxes all such kinds of restrictions.
- unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with any other node in the graph. When two nodes are connected via an edge, the two nodes are known as neighbors.
- *Advantages:* Best models real-world situations
- *Disadvantages:* Some algorithms are slow and very complex



ABSTRACT DATA TYPE

- **An *Abstract Data Type* (ADT)** is the way at which we look at a data structure, focusing on what it does and ignoring how it does its job.
- For example, stack and queue are perfect examples of an abstract data type. We can implement both these ADTs using an array or a linked list. This demonstrates the "abstract" nature of stacks and queues.
- In C, an Abstract Data Type can be a structure considered without regard to its implementation. It can be thought of as a "description" of the data in the structure with a list of operations that can be performed on the data within that structure.
- The end user is not concerned about the details of how the methods carry out their tasks. They are only aware of the methods that are available to them and are concerned only about calling those methods and getting back the results but not HOW they work.

CODE	DESCRIPTION
typedef struct stack_Rep stack_Rep;	instance representation record
typedef stack_Rep *stack_T;	pointer to a stack instance
typedef void *stack_Item;	value to be stored on stack
stack_T create_stack(void);	Create an empty stack instance
void push(stack_T st, stack_Item v)	Add an item at the top of the stack
stack_Item pop(stack_T st);	Remove the top item from the stack and return it
int is_empty(stack_T st);	Check whether stack is empty

Stack.h to be used in abstract stack implementation

```
#include <stack.h>

stack_T s = create_stack();

 int val = 09;

 s = push(t, &val);

void *v = pop(s);

if (is_empty(s))

{ … }
```

Implementation of abstract stack using stack.h

ALGORITHM

- The typical meaning of "algorithm" is a formally defined procedure for performing some calculation.
- An algorithm provides a blueprint to write a program to solve a particular problem.
- It is considered to be an effective procedure for solving a problem in finite number of steps. That is, a well-defined algorithm always provides an answer and is guaranteed to terminate.
- Algorithms are mainly used to achieve **software re-use**. Once we have an idea or a blueprint of a solution, we can implement in any high level language like C, C++, Java, so on and so forth.

- **Write an algorithm to find whether a number is even or odd**
- Step 1: Input the first number as A
- Step 2: IF A%2 =0
- Then Print "EVEN"
- ELSE
- PRINT "ODD"
- Step 3: END

TIME AND SPACE COMPLEXITY OF ALGORITHM

- To analyze an algorithm means determining the amount of resources (such as time and storage) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time complexity and space complexity.
- The time Complexity of an algorithm is basically the running time of the program as a function of the input size. On similar grounds, space complexity of an algorithm is the amount of computer memory required during the program execution, as a function of the input size
- Time complexity of an algorithm depends on the number of machine instructions in which a program executes. This number is primarily dependant on the size of the program's input and the algorithm used.
- The space needed by a program depends on:

  Fixed part, that varies with problem to problem. It includes space needed for storing instructions, constants, variables and structured variables (like arrays, structures)

  Variable part, that varies from program to program. It includes space needed for recursion stack, and for structured variables that are allocated space dynamically during the run-time of the program.

Calculating Algorithm Efficiency

- If a function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.
- If an algorithm contains certain loops or recursive functions then the efficiency of that algorithm may vary depending on the number of loops and the running time of each loop in the algorithm.
- The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed. So, if n is the number of elements, then the efficiency can be stated as

  f(n) = efficiency

Linear loops

```
for(i=0;i<n;i++)
        statement block
```

f(n) = efficiency

Logarithmic Loops

```
for(i=1;i<n;i*=2)
        statement block;
```

f(n) = log n f(n) = log n

```
for(i=0;i<n;i/=2)
        statement block;
```

Calculating Algorithm Efficiency contd.

Nested Loops

- Total no. of iterations = no. of iterations in inner loop * no. of iterations in outer loop
- In case of nested loops, we will analyze the efficiency of the algorithm based on whether it's a linear logarithmic, quadratic or dependant quadratic nested loop.

Linear logarithmic

```
for(i=0;i<n;i++)
        for(j=1; j<n;j*=2)
                statement block;
```

f(n)= n log n

Quadratic Loop

```
for(i=0;i<n;i++)
        for(j=1; j<n;j++)
                statement block;
```

*f(n) = n * n*

Dependent Quadratic

```
for(i=0;i<n;i++)
        for(j=1; j<i;j++)
                statement block;
```

f(n) = n (n + 1)/2

OR f(n) = n (n + 1)/2

BIG OH NOTATION

- Big-Oh notation where the "O" stands for "order of" is concerned with what happens for very large values of n.
- For example, if a sorting algorithm performs n^2 operations to sort just n elements, then that algorithm would be described as an $O(n^2)$ algorithm.
- When expressing complexity using Big Oh notation, constant multipliers are ignored. So a $O(4n)$ algorithm is equivalent to $O(n)$, which is how it should be written.
- If f(n) and g(n) are functions defined on positive integer number n, then

  $f(n) = O(g(n))$
- That is, f of n is big oh of g of n if and only if there exists positive constants c and n, such that

  $f(n) \leq Cg(n) \leq n$
- This means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). Hence, g provides an upper bound.

Categories of Algorithms

- Constant time algorithm that have running time complexity given as O(1)
- Linear time algorithm that have running time complexity given as O(n)
- Logarithmic time algorithm that have running time complexity given as O(log n)
- Polynomial time algorithm that have running time complexity given as O(nk) where k>1
- Exponential time algorithm that have running time complexity given as $O(2^n)$

n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096

Number of operations for different functions of n

NP- COMPLETE

- In computational complexity theory, the complexity class NP-C (Non-deterministic Polynomial time- Complete), is a class of problems that exhibits two properties:
- A set of problems is said to be NP is any given solution to the problem can be verified quickly in polynomial time.
- If the problem can be solved quickly in polynomial time, then it implies that every problem in *NP* can be solved in polynomial time
- Even if a given solution to a problem can be verified quickly, there is no known efficient way to locate a solution in the first place. Therefore, it is not wrong to say that solutions to NP-complete problems may not even be known.
- A problem p in NP is also in NPC if and only if every other problem in NP can be transformed into p in polynomial time.
- if any single problem in NP-complete can be solved quickly, then every problem in NP can also be quickly solved.