LOVELY
PROFESSIONAL
UNIVERSITY

www.lpude.in

DIRECTORATE OF DISTANCE EDUCATION

FUNDAMENTALS OF DATA STRUCTURES

Produced & Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Directorate of Distance Education
Lovely Professional University
Phagwara

Directorate of Distance Education

LPU is reaching out to the masses by providing an intellectual learning environment that is academically rich with the most affordable fee structure. Supported by the largest University[1] in the country, LPU, the Directorate of Distance Education (DDE) is bridging the gap between education and the education seekers at a fast pace, through the usage of technology which significantly extends the reach and quality of education. DDE aims at making Distance Education a credible and valued mode of learning by providing education without a compromise.

DDE is a young and dynamic wing of the University, filled with energy, enthusiasm, compassion and concern. Its team strives hard to meet the demands of the industry, to ensure quality in curriculum, teaching methodology, examination and evaluation system, and to provide the best of student services to its students. DDE is proud of its values, by virtue of which, it ensures to make an impact on the education system and its learners.

Through affordable education, online resources and a network of Study Centres, DDE intends to reach the unreached.

[1] in terms of no. of students in a single campus

SYLLABUS

Fundamentals of Data Structures

Objectives:

The objective of the course it to:

- Solve problems using data structures such as linear lists, stacks, queues, hash tables, binary trees, heaps, tournament trees, binary search trees, and graphs and writing programs for these solutions.
- Solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking, and branch and bound and writing programs for these solutions.
- Develop proficiency in the specification, representation, and implementation of Data Types and Data Structures. .

S. No.	Description
1.	**Introduction & Overview:** Concept of data type, definition and brief description of various data structures.
2.	Operations on data structures, algorithm complexity, Big O notation, recursion, some illustrative examples of recursive functions.
3.	**Arrays:** Linear and multi-dimensional arrays and their representation Pointers, Array Pointers, Records and Record Structures, Representation of Records in Memory; Parallel Arrays.
4.	**Arrays:** Operations on arrays, sparse matrices and their storage.
5.	**Linked Lists:** Linear linked list, operations on linear linked list, double linked list.
6.	**Stacks:** Sequential and linked representations, operations on stacks, multi stacks **Stacks:** application of stacks such as parenthesis checker, evaluation of postfix expressions
7.	**Queues**: Sequential representation of queue, linear queue, circular queue, operations and applications, linked representation of a queue.
8.	**Introduction to Trees:** Binary Tree Representation, Traversal.
9.	**Sorting:** Insertion Sort, Selection Sort, Merge Sort, Radix Sort, Hashing.
10.	**Searching:** Linear and Binary Search.

CONTENTS

Unit 1: Data Structures

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of data structure
- Explain various data types
- Discuss various data structures

Introduction

A data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More typically, a data structure is meant to be an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. Thus, it is possible to perform all necessary operations on any data structure. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days.

1.1 Overview of Data Structure

A data structure is a scheme for organizing data in the memory of a computer. A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks.

*Example:*B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs. The way in which the data is organized affects the performance of a program for different tasks. Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address — a bit string that can be itself stored in memory and manipulated by the program.

> *Notes* The record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself.

Data may be organized in many different ways: the logical or mathematical model of a particular organization of data is called data structure. Data model depends on two things. First, it much be rich enough in structure to mirror the actual relationship of the data in the real world. On other hand, the structure should be simple to execute the process the data when necessary.

Data are also organized into more complex types of structures. The study of such data structure, which forms the subject matter of the text, includes the following three steps:

1. Logical or mathematical description of the structure.
2. Implementation of the structure on a computer.
3. Quantitative analysis of the structure, which include determining the amount of memory needed to store the structure and the time required to process the structure.

Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported.

Example: of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.

2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure:

- Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data items be deleted?
- Are all data items processed in some well-defined order, or is search for specific data items allowed?

Typically, interspersing insertions with other operations, allowing deletion, and supporting search for data items all require more complex representations.

Self Assessment Questions

Fill in the blanks:

1. A .... is a scheme for organizing data in the memory of a computer.
2. .... on certain key operations normally drive the data structure selection process.

1.2 Concept of Data Type

A data type is a method of interpreting a pattern of bits.

C uses data types to describe various kinds of data such as integers, floating-point numbers, characters, etc . In C, an object refers to a memory location where its content represents a value. If you assign an object a name, that object becomes a variable. A data type determines the number of bytes to be allocated to the variable and valid operations can be performed on it.

C provides you with various data types that can be classified into the following groups:

- Basic type includes standard and extended integer types.
- Enumerated types contains real and complex floating-point types
- Derived types include pointer types, array types, structure types, union types and function types.
- Type void

Did u know? Function type depicts the interface of a function. It specifies types of parameters and return type of the function.

C data types can be also classified differently into the following groups:

- Arithmetic types include basic types and enumerated types
- Scalar types include arithmetic types and pointer types
- Aggregate types include array types and structure types.

1.2.1 Integer Types

Integer types include signed and unsigned integer types. These are discussed as below.

C signed integer types

C provides you with five signed integer types. Each integer type has several synonyms.

Table 1.1 illustrates the first five integer types with their corresponding synonyms:

Table 1.1: first five integer types

Integer Types	Synonyms	Notes
signed char		
int	Signed, signed int	
short	Short int, signed short, signed short int	
long	Long int, signed long, signed long int	
long	Long long int, signed long long,	Available
long	Signed long long int	Since C99

C unsigned integer types

For each signed integer, C also provides the corresponding unsigned integer type that has the same memory size as the signed integer type.

Table 1.2 illustrates the unsigned integer type:

Table 1.2: Unsigned Integer Types

Signed Integer Types	Unsigned Integer Types
char	unsigned char
int	unsigned int
short	unsigned short
long	unsigned long
long long	unsigned long long

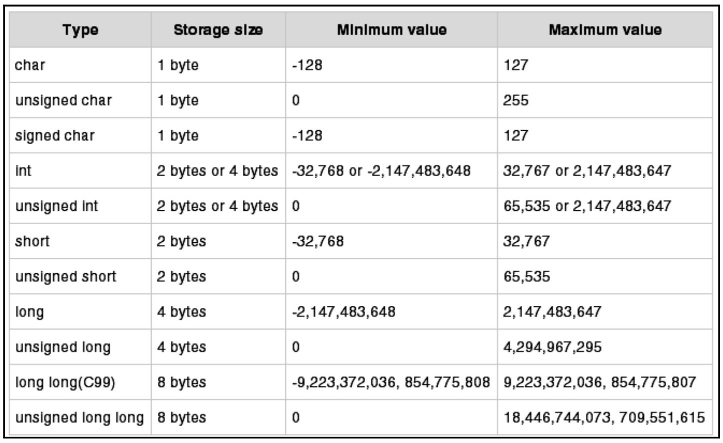
C integer types value ranges

C defines exactly minimum storage size of each integer type e.g., short takes at least two byes, long takes at least 4 bytes. Regardless of the C's implementation, the size of integer types must follows the order below:

sizeof(short) < sizeof(int) < sizeof(long) < sizeof(long long)

Table 1.3 gives you the common sizes of the integer types in C:

Table 1.3: Common sizes of the integer types in C



Type	Storage size	Minimum value	Maximum value
char	1 byte	-128	127
unsigned char	1 byte	0	255
signed char	1 byte	-128	127
int	2 bytes or 4 bytes	-32,768 or -2,147,483,648	32,767 or 2,147,483,647
unsigned int	2 bytes or 4 bytes	0	65,535 or 2,147,483,647
short	2 bytes	-32,768	32,767
unsigned short	2 bytes	0	65,535
long	4 bytes	-2,147,483,648	2,147,483,647
unsigned long	4 bytes	0	4,294,967,295
long long(C99)	8 bytes	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807
unsigned long long	8 bytes	0	18,446,744,073, 709,551,615

The value ranges of integer types can be found in the limits.h header file. This header file contains the macros that define minimum and maximum values of each integer type e.g., INT_MIN, INT_MAX for minimum and maximum size of the integer.

Task Compare and contrast C signed and unsigned integer types.

1.2.2 C Float Types

C provides various floating-point types that represents non-integer number with a decimal point at any position.

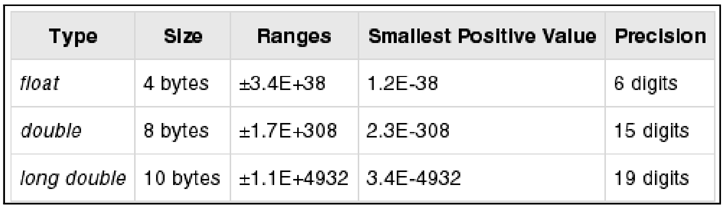
Example: with integer types, you only can have numbers 1, 2, 10, 200… however with floating-point type, you can have 1.0, 2.5, 100.25 and so on.

There are three standard floating-point types in C:

- float: for numbers with single precision.
- double: for numbers with double precision.
- long double: for numbers with extended precision.

Table 1.4 illustrates the technical attributes of various floating-point types in C. It is important to notice that this is only the minimal requirement for storage size defined by C.

Table 1.4: Technical attributes of various floating-point types in C



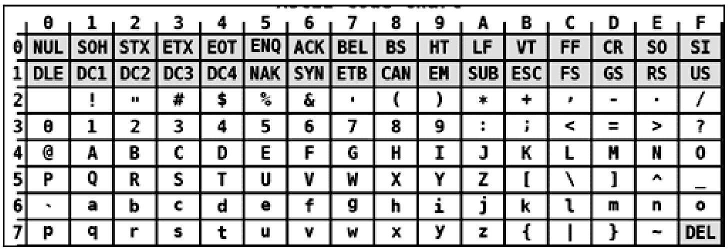
Type	Size	Ranges	Smallest Positive Value	Precision
float	4 bytes	±3.4E+38	1.2E-38	6 digits
double	8 bytes	±1.7E+308	2.3E-308	15 digits
long double	10 bytes	±1.1E+4932	3.4E-4932	19 digits

1.2.3 C Character Types

C uses char type to store characters and letters. However, the char type is integer type because underneath C stores integer numbers instead of characters.

In order to represent characters, the computer has to map each integer with a corresponding character using a numerical code. The most common numerical code is ASCII, which stands for American Standard Code for Information Interchange. Table 1.5 illustrates the ASCII code:

Table 1.5: ASCII Code Chart



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Example: the integer number 65 represents a character A in upper case.

In C, the char type has 1-byte unit of memory so it is more than enough to hold the ASCII codes. Besides ASCII code, there are various numerical codes available such as extended ASCII codes. Unfortunately, many character sets have more than 127 even 255 values. Therefore, to fulfill those needs, the Unicode was created to represent various available character sets. Unicode currently has over 40,000 characters.

1.2.4 C Enum

Enum defines enumeration types, to enhance your code readability.

C provides developers with special types called enumerated types or enum to declare symbolic names that represent integer constants. Its main purpose is to enhance the readability of the code. An enumeration consists of a group of symbolic integers.

Example: we can declare an enumeration for colors as follows:

```
enum color {red, green, blue};
```

In the example given above:

- enum is the keyword to declare a new enumeration type
- color is the tag name that you can use later as a type name.

Caution The tag name must be unique within its scope. This tag name is also optional so you can omit it.

- Inside the curly brackets {} is a set of named integers. This set is also known as enumeration constants, enumeration sets, enumerators or just members.

Notes

By default, the first enumeration member in the set has value 0. The next member has value of the first one plus 1. So in this case red = 0, green = 1 and blue = 2. You can explicitly assign an enumeration member to another value using assignment operator (=) when you define the enumeration type.

The enumeration members must follow the rules:

- An enumeration set may have duplicate constant values. For instance, you can assign 0 with two enumeration members.
- The name of enumeration member must be unique within its scope. It means its name should not be the same as other member or variable name.

If you declare a variable with enumeration type, the value of that value must be one of the values of the enumeration members.

Example: you can declare a variable called favorite_color as a variable of the color type.

```
enum color favorite_color = green;
```

Self Assessment Questions

Fill in the blanks:

3. A .... determines the number of bytes to be allocated to the variable.
4. The value ranges of integer types can be found in the .... header file.
5. .... types represents non-integer number with a decimal point at any position.
6. .... type is used to store characters and letters.
7. In C, the char type has .... unit of memory.
8. C provides developers with special types called .... to declare symbolic names that represent integer constants.
9. By default, the first enumeration member in the set has value ....

1.3 Description of Various Data Structures

Data structure are classified into two type such as linear or non-linear.

- ***Linear:*** A data structure is said to be linear if its elements form a sequence. The elements of linear data structure represents by means of sequential memory locations. The other way is to have the linear relationship between the elements represented by means of pointers or links. Ex- Array and Link List.
- ***Non-linear:*** A data structure is said to be non-linear if its elements a hierarchical relationship between elements such as trees and graphs. All elements assign the memory as random form and you can fetch data elements through random access process.

In this section, we will discuss the brief description of various data structures such as arrays, stack, queues, etc.

1.3.1 Arrays

The simplest type of data structure is a linear (or one dimensional) array. By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3, …n. If we choose the name A for the array, then the elements of A are denoted by subscript notation:

a1, a2, a3, ……., an

or, by the parenthesis notation:

A(1), A(2), A(3),……., A(N)

or, by the bracket notation:

A[1], a[2], A[3],…….., A[N]

Regardless of the notation, the number K in A[K] is called a subscript and A[K] is called a subscripted variable.

Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript. A two-dimensional array is a collection of similar data elements where each element is referenced by two subscripts.

Did u know? Such arrays are called matrices in mathematics, and tables in business applications.

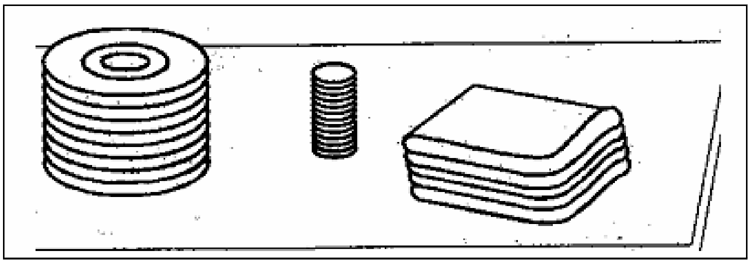
1.3.2 Stack

A stack is a linear structure in which items may be added or removed only at one end. The stack is a common data structure for representing things that need to maintained in a particular order. For instance, when a function calls another function, which in turn calls a third function, it's important that the third function return back to the second function rather than the first. One way to think about this implementation is to think of functions as being stacked on top of each other; the last one added to the stack is the first one taken off. In this way, the data structure itself enforces the proper order of calls.

Conceptually, a stack is simple: a data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack. Consequently, a stack is said to have "first in last out" behavior (or "last in, first out"). The first item added to a stack will be the last item removed from a stack.

Example: Figure 1.1 pictures three everyday examples of such a structure: a stack of dishes, a stack of pennies and a stack of folded towels.

Figure 1.1: A stack of dishes, a stack of pennies and a stack of folded towels



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Notes

Stacks are also called last-in first-out (LIFO) lists. Other names used for stacks are "piles" and "push-down lists. Stack has many important applications in computer science.

1.3.3 Queues

A queue, also called a first-in-first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list. The features of a Queue are similar to the features of any queue of customers at a counter, at a bus stop, at railway reservation counter etc. A queue can be implemented using arrays or linked lists. A queue can be represented as a circular queue. This representation saves space when compared to the linear queue. Finally, there are special cases of queues called Dequeues which allow insertion and deletion of elements at both the end.

Figure 1.2: queue of people waiting at a bus stop

Figure 1.2 pictures a queue of people waiting at a bus stop.

Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

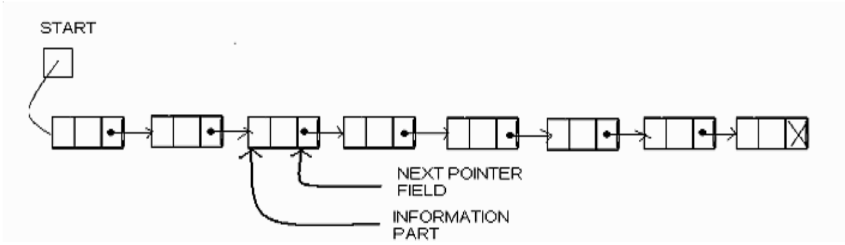
Task Analyze various applications of queues.

1.3.4 Linked List

A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list.

Figure 1.3 is a schematic diagram of a linked list with 6 nodes, Each node is pictured with two parts.

Figure 1.3: linked list with 6 nodes



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Notes

The left part represents the information part of the node, which may contain an entire record of data items (e.g., NAME, ADDRESS,...). The right part represents the Next pointer field of the node, and there is an arrow drawn from it to the next node in the list. This follows the usual practice of drawing an arrow from a field to a node when the address of the node appears in the given field.

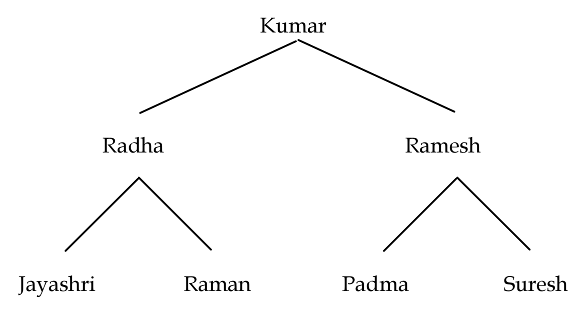
Notes The pointer of the last node contains special value, called the null pointer, which is any invalid address.

1.3.5 Tree

A tree is an acyclic, connected graph. A tree contains no loops or cycles. The concept of tree is one of the most fundamental and useful concepts in computer science. Trees have many variations, implementations and applications. Trees find their use in applications such as compiler construction, database design, windows, operating system programs, etc. A tree structures is one in which items of data are related by edges.

Example: A very common example is the ancestor tree as given in Figure 1.4. This tree shows the ancestors of KUMAR. His parents are RAMESH and RADHA. RAMESH's parents are PADMA and SURESH who are also grand parents of KUMAR (on father's side); RADHA's parents are JAYASHRI and RAMAN who are also grand parents of KUMAR (on mother's side).

Figure 1.4: A Family Tree



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

1.3.6 Graph

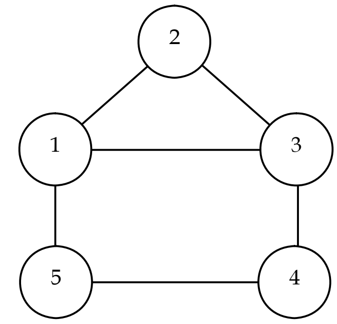
All the data structures (Arrays, Lists, Stacks, and Queues) except Graphs are linear data structures. Graphs are classified in the non-linear category of data structures. A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices). The notation used is as follows:

Graph G = (V,E)

Let we consider graph of figure 1.5.

Notes

Figure 1.5: A graph



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

From the above graph, we may observe that the set of vertices for the graph is V = {1,2,3,4,5}, and the set of edges for the graph is E = {(1,2), (1,5), (1,3), (5,4), (4,3), (2,3)}. The elements of E are always a pair of elements.

⚠ *Caution* The relationship between pairs of these elements is not necessarily hierarchical in nature.

Self Assessment Questions

Fill in the blanks:

10. A data structure is said to be .... if its elements form a sequence.
11. A .... array is a collection of similar data elements where each element is referenced by two subscripts.
12. A .... is a linear structure in which items may be added or removed only at one end.
13. A special queue known as .... allows insertion and deletion of elements at both the end.
14. A linked list, or one-way list, is a linear collection of data elements, called ...., where the linear order is given by means of pointers.
15. A .... structures is one in which items of data are related by edges.

Notes

Case Study

Counting Candy

The image in Figure 11.5 shows a simple counting puzzle. The task is to count how many of each different type of candy there are in the image.

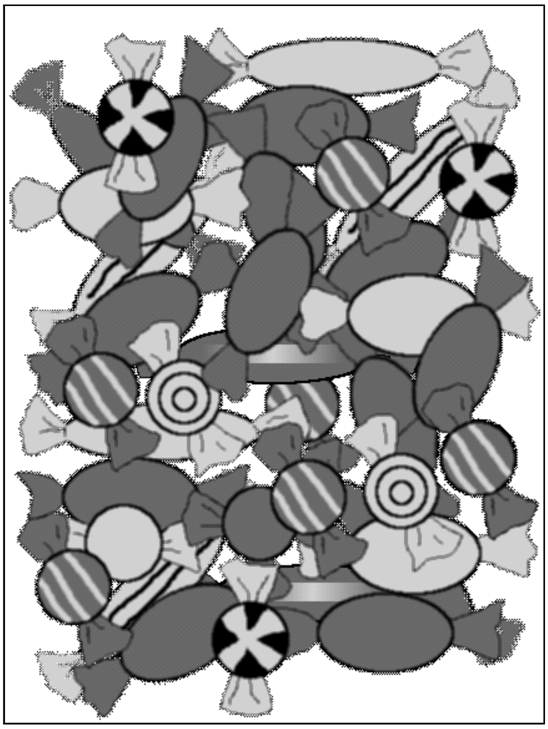


Figure 11.5: A counting puzzle. How many candies of each different shape are there? (round, oval, and long). How many candies have a pattern? How many candies are dark and how many are light?

Source: http://statmath.wu.ac.at/courses/data-analysis/itdtHTML/node80.html

Our task is to record the different shapes (round, oval, or long), the different shades (light or dark), and whether there is a pattern on the candies that we can see in the image. How can this information be entered into R?

One way to enter data in R is to use the scan() function. This allows us to type data into R separated by spaces. Once we have entered all of the data, we enter an empty line to indicate to R that we have finished. We will use this to enter the different shapes:

```
> shapeNames <- scan(what="character")

1: round oval long

4:

Read 3 items

> shapeNames

[1] "round" "oval"  "long"
```

Another way to enter data is using the c() function. We will use this to enter the possible patterns and shades:

```
> patternNames <- c("pattern", "plain")
> patternNames
[1] "pattern" "plain"
> shadeNames <- c("light", "dark")
> shadeNames
[1] "light" "dark"
```

All we have done so far is enter the possible values of these symbols.

1.4 Summary

- A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data.
- A data type is a method of interpreting a pattern of bits.
- By a linear array, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3, …n.
- The stack is a common data structure for representing things that need to maintained in a particular order.
- A queue, also called a first-in-first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list.
- A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.
- A tree is an acyclic, connected graph which contains no loops or cycles.
- A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices).

1.5 Keywords

Data structure: A data structure is a scheme for organizing data in the memory of a computer.

Data type: A data type is a method of interpreting a pattern of bits.

Array: Array is a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3, …n.

Stack: A stack is a linear structure in which items may be added or removed only at one end.

Queue: A queue, also called a first-in-first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list.

Linked list: A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

Tree: A tree is an acyclic, connected graph which contains no loops or cycles.

Graph: A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices).

1.6 Review Questions

1. Explain the concept of data structure with example.
2. Discuss how to select a data structure to solve a problem.
3. What are data types? Classify various C data types.
4. What are the different floating-point types in C? Also discuss the minimal requirement for storage size.
5. Make distinction between character types and enumeration types.
6. Differentiate between linear and non-linear data structure.
7. Discuss the concept of stacks with example.
8. Illustrate the difference between LIFO and FIFO system.
9. Discuss linked list in brief with example.
10. Graphs are classified in the non-linear category of data structures. Comment.

Answers: Self Assessment

1. data structure
2. Resource constraints
3. data type
4. limits.h
5. Floating-point
6. Char
7. 1-byte
8. enumerated types or enum
9. 0
10. Linear
11. two-dimensional
12. Stack
13. dequeue
14. Nodes
15. tree

1.7 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.roseindia.net/tutorial/datastructure

http://www.cprograms.in/

http://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/CONCEPT/data_types.html

http://www.asic-world.com/scripting/data_types_c.html

Unit 2: Data Structure Operations and Algorithms Complexity

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss various operations on data structures
- Explain algorithm complexity
- Discuss Big O notation

Introduction

Data are processed by means of certain operations which appearing in the data structure. Data has situation on depends largely on the frequency with which specific operations are performed. An essential aspect to data structures is algorithms. Data structures are implemented using algorithms. In this unit, we will introduce some of the most frequently used operations. Also we will discuss the concept of algorithm complexity.

2.1 Operations on Data Structures

We may perform the following operations on any linear structure, whether it is an array or a linked list.

- ***Traversal:*** By means of traversal operation, we can process each element in the list.
- ***Search:*** By means of this operation, we can find the location of the element with a given value or the record with a given key.

- ***Insertion:*** Insertion operation is used for adding a new element to the list.
- ***Deletion:*** Deletion operation is used to remove an element from the list.
- ***Sorting:*** This operation is used for arranging the elements in some type of order.
- ***Merging:*** By means of merging operation, we can combine two lists into a single list.

Notes Depending upon the relative frequency, we may perform these operations with a particular any one of the Linear structure.

Self Assessment Questions

Fill in the blanks:

1. .... operation is used for adding a new element to the list.

2. .... operation is used for arranging the elements in some type of order.

2.2 Algorithm Complexity

An algorithm is a clearly specified set of simple instructions to be followed to solve a problem. An algorithm is a procedure that you can write as a C function or program, or any other language. Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine how much in the way of resources, such as time or space, the algorithm will require. An algorithm that solves a problem but requires a year is hardly of any use.

Caution An algorithm that requires a gigabyte of main memory is not (currently) useful.

In order to analyze algorithms in a formal framework, we need a model of computation. Our model is basically a normal computer, in which instructions are executed sequentially. Our model has the standard repertoire of simple instructions, such as addition, multiplication, comparison, and assignment, but, unlike real computers, it takes exactly one time unit to do anything (simple). To be reasonable, we will assume that, like a modern computer, our model has fixed size (say 32-bit) integers and that there are no fancy operations, such as matrix inversion or sorting, that clearly cannot be done in one time unit. We also assume infinite memory.

This model clearly has some weaknesses. Obviously, in real life, not all operations take exactly the same time. In particular, in our model one disk read counts the same as an addition, even though the addition is typically several orders of magnitude faster. Also, by assuming infinite memory, we never worry about page faulting, which can be a real problem, especially for efficient algorithms. This can be a major problem in many applications.

The most important resource to analyze is generally the running time. Several factors affect the running time of a program. Some, such as the compiler and computer used, are obviously beyond the scope of any theoretical model, so, although they are important, we cannot deal with them here. The other main factors are the algorithm used and the input to the algorithm.

An algorithm states explicitly how the data will be manipulated. Some algorithms are more efficient than others.

Did u know? It is preferred to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.

The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm:

- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take. We try to keep this idea of time separate from "wall clock" time, since many factors unrelated to the algorithm itself can affect the real time (like the language used, type of computing hardware, proficiency of the programmer, optimization in the compiler, etc.). It turns out that, if we chose the units wisely, all of the other stuff doesn't matter and we can get an independent measure of the efficiency of the algorithm.
- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its space complexity is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

  We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

There is often a time-space-tradeoff involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

Example: We might say "this algorithm takes n^2 time," where n is the number of items in the input. Or we might say "this algorithm takes constant extra space," because the amount of extra memory needed doesn't vary with the number of items processed.

The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or nondeterminism. Amount of computer memory required during the program execution, as a function of the input size.

A small amount of space, deterministic machines can simulate nondeterministic machines, where as in time complexity, time increase exponentially in this case. A nondeterministic TM using O(n) space can be changed to a deterministic TM using only $O^{2(n)}$ space.

Task Analyze the difference between time and space complexity.

Self Assessment Questions

Fill in the blanks:

3. An .... is a clearly specified set of simple instructions to be followed to solve a problem.

4. In order to analyze algorithms in a formal framework, we need a .... of computation.

5. The............................ of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

6. .... is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

7. .... is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

2.3 Big O Notation

When solving a computer science problem there will usually be more than just one solution. These solutions will often be in the form of different algorithms, and you will generally want to compare the algorithms to see which one is more efficient. This is where Big O analysis helps – it gives us some basis for measuring the efficiency of an algorithm

"Big O" refers to a way of rating the efficiency of an algorithm. It is only a rough estimate of the actual running time of the algorithm, but it will give you an idea of the performance relative to the size of the input data.

The motivation behind Big-O notation is that we need some way to measure the efficiency of programs but there are so many differences between computers that we can't use real time to measure program efficiency. Therefore we use a more abstract concept to measure algorithm efficiency.

An algorithm is said to be of order O(expression), or simply of order expression (where expression is some function of n, like n^2 and n is the size of the data) if there exist numbers p, q and r so that the running time always lies below between p.expression+q for n > r. Generally expression is made as simple and as small as possible.

Definition: Let f(n) and g(n) be functions, where n is a positive integer. We write f(n) = O(g(n)) if and only if there exists a real number c and positive integer n0 satisfying 0 <= f(n) <= cg(n) for all n >= n0.

We say, "f of n is big oh of g of n." We might also say or write f(n) is in O(g(n)), because we can think of O as a set of functions all with the same property. But we won't often do that in Data Structures.

This means that, for example, that functions like $n^2 + n$, $4n^2 - n \log n + 12$, n2/5 - 100n, n log n, 50n, and so forth are all $O(n^2)$.

Every function f(n) bounded above by some constant multiple g(n) for all values of n greater than a certain value is O(g(n)).

Example:

- Show $3n^2 + 4n - 2 = O(n^2)$.

  We need to find c and n_0 such that:

$3n^2 + 4n - 2 \leq = cn^2$ for all $n \geq n_0$.

Divide both sides by n^2, getting:

$3 + 4/n - 2/n^2 \leq c$ for all $n \geq n_0$.

If we choose n_0 equal to 1, then we need a value of c such that:

$3 + 4 - 2 \leq c$

We can set c equal to 6. Now we have:

$3n^2 + 4n - 2 \leq 6n^2$ for all $n \geq 1$.

- Show $n^3 != O(n^2)$. Let's assume to the contrary that

  $n^3 = O(n^2)$

  Then there must exist constants c and n0 such that

  $n^3 \leq cn^2$ for all $n \geq n0$.

  Dividing by n^2, we get:

  $n \leq c$ for all $n \geq n0$.

But this is not possible; we can never choose a constant c large enough that n will never exceed it, since n can grow without bound. Thus, the original assumption, that n3 = O(n2), be wrong so n3 != O(n2).

Big O gives us a formal way of expressing asymptotic upper bounds, a way of bounding from above the growth of a function.

> *Notes* Knowing where a function falls within the big-O hierarchy allows us to compare it quickly with other functions and gives us an idea of which algorithm has the best time performance.

2.3.1 Growth Rate Functions

As we know Big O notation is a convenient way of describing the growth rate of a function and hence the time complexity of an algorithm.

Let n be the size of the input and f (n), g(n) be positive functions of n.

The time efficiency of almost all of the algorithms can be characterized by only a few growth rate functions:

O(l) - constant time

This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

Example: (assuming a reasonable implementation of the task):

- Push and Pop operations for a stack (containing n elements);
- Insert and Remove operations for a queue.

1O(n) - linear time

This means that the algorithm requires a number of steps proportional to the size of the task.

Example: (assuming a reasonable implementation of the task):

- Traversal of a list (a linked list or an array) with n elements;
- Finding the maximum or minimum element in a list, or sequential search in an unsorted list of n elements;
- Traversal of a tree with n nodes;
- Calculating iteratively n-factorial; finding iteratively the nth Fibonacci number.

O(n2) - quadratic time

The number of operations is proportional to the size of the task squared.

Example:

- Some more simplistic sorting algorithms, for instance a selection sort of n elements;
- Comparing two two-dimensional arrays of size n by n;
- Finding duplicates in an unsorted list of n elements (implemented with two nested loops).

O(log n) - logarithmic time

Example:

- Binary search in a sorted list of n elements;
- Insert and Find operations for a binary search tree with n nodes;
- Insert and Remove operations for a heap with n nodes.

O(n log n) - "n log n " time

It includes more advanced sorting algorithms.

For example, quicksort, mergesort

O(an) (a > 1) - exponential time

Example:

- Recursive Fibonacci implementation
- Towers of Hanoi
- Generating all permutations of n symbols

The best time in the above list is obviously constant time, and the worst is exponential time which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n.

Notes

Did u know? Polynomial growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth.

Order of asymptotic behavior of the functions from the above list:

Using the "<" sign informally, we can say that

O(l) < O(log n) < O(n) < O(n log n) < O(n2) < O(n3) < O(an)

Task Compare and contrast quadratic time and logarithmic time.

2.3.2 Properties of Big O

The definition of big O is pretty ugly to have to work with all the time, kind of like the "limit" definition of a derivative in Calculus. Here are some helpful theorems you can use to simplify big O calculations:

- Any kth degree polynomial is O(nk).
- a nk = O(nk) for any a > 0.
- Big O is transitive. That is, if f(n) = O(g(n)) and g(n) is O(h(n)), then f(n) = O(h(n)).
- logan = O(logb n) for any a, b > 1. This practically means that we don't care, asymptotically, what base we take our logarithms to. (We said asymptotically. In a few cases, it does matter.)
- Big O of a sum of functions is big O of the largest function. How do you know which one is the largest? The one that all the others are big O of. One consequence of this is, if f(n) = O(h(n)) and g(n) is O(h(n)), thenf(n) + g(n) = O(h(n)).
- f(n) = O(g(n)) is true if limn->infinityf(n)/g(n) is a constant.

2.3.3 Lower Bounds and Tight Bounds

Big O only gives you an upper bound on a function, i.e., if we ignore constant factors and let n get big enough, we know some function will never exceed some other function. But this can give us too much freedom.

For instance, the time for selection sort is easily O(n3), because n2 is O(n3). But we know that O(n2) is a more meaningful upper bound. What we need is to be able to describe a lower bound, a function that always grows more slowly than f(n), and a tight bound, a function that grows at about the same rate as f(n). Now let us look at a different (and probably easier to understand) way to approach this.

Big Omega is for lower bounds what big O is for upper bounds:

Definition: Let f(n) and g(n) be functions, where n is a positive integer. We write f(n) = Ω(g(n)) if and only if g(n) = O(f(n)). We say "f of n is omega of g of n."

This means g is a lower bound for f; after a certain value of n, and without regard to multiplicative constants, f will never go below g.

Finally, theta notation combines upper bounds with lower bounds to get tight bounds:

Definition: Let f(n) and g(n) be functions, where n is a positive integer. We write f(n) = Θ(g(n)) if and only if g(n) = O(f(n)). and g(n) = Ω(f(n)). We say "f of n is theta of g of n."

2.3.4 More Properties

- The first four properties listed above for big O are also true for Omega and Theta.
- Replace O with Ω and "largest" with "smallest" in the fifth property for big O and it remains true.
- f(n) = Ω(g(n)) is true if limn->infinityg(n)/f(n) is a constant.
- f(n) = Θ(g(n)) is true if limn->infinityf(n)/g(n) is a non-zero constant.
- nk = O((1+ε) n)) for any positive k and ε. That is, any polynomial is bound from above by any exponential. So any algorithm that runs in polynomial time is (eventually, for large enough value of n) preferable to any algorithm that runs in exponential time.
- (log n)ε = O(n k) for any positive k and ε. That means a logarithm to any power grows more slowly than a polynomial (even things like square root, 100th root, etc.)

Caution An algorithm that runs in logarithmic time is (eventually) preferable to an algorithm that runs in polynomial (or indeed exponential, from above) time.

Self Assessment Questions

Fill in the blanks:

8. "........................" refers to a way of rating the efficiency of an algorithm.
9. Every function f(n) bounded above by some constant multiple g(n) for all values of n greater than a certain value is ....
10. Big O gives us a formal way of expressing asymptotic upper bounds.
11. A function "........................" means that the algorithm requires the same fixed number of steps regardless of the size of the task.
12. A function "........................" means that the algorithm requires a number of steps proportional to the size of the task.
13. In case of "........................" function, the number of operations is proportional to the size of the task squared.

14. "........................" ***time*** includes more advanced sorting algorithms.

15. Big O is ...., that is, if f(n) = O(g(n)) and g(n) is O(h(n)), then f(n) = O(h(n)).

Case Study

Algorithm Analysis

Here we show how to use the big-Oh notation to analyze three algorithms that solve the same problem but have different running times. The problem we focus on is one that is reportedly often used as a job interview question by major software and Internet companies—the maximum subarray problem. In this problem, we are given an array of positive and negative integers and asked to find the subarray whose elements have the largest sum. That is, given A = [a1, a2, . . . , an], find the indices j and k that maximize the sum

Contd...

$s_{j,k} = a_j + a_{j+1} + \ldots + a_k = {}_{i=j}^{k} a_i$

Or, to put it another way, if we use A[j : k]to denote the subarray of A from index j to index k, then the maximum subarray problem is to find the subarray A[j : k] that maximizes the sum of its values. In addition to being considered a good problem for testing the thinking skills of prospective employees, the maximum subarray problem also has applications in pattern analysis in digitized images. An example of this problem is shown in Figure a.

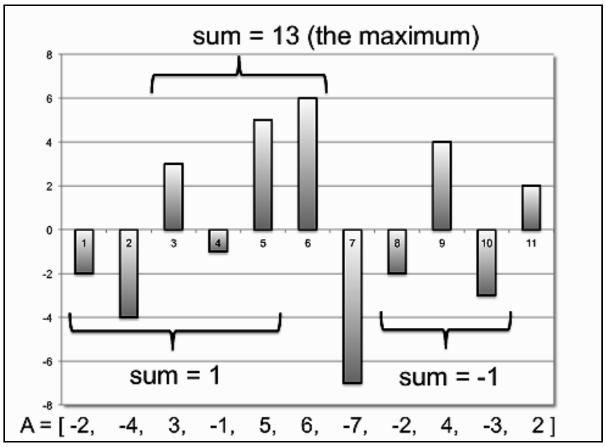


Figure a: An instance of the maximum subarray problem

In this case, the maximum subarray is A[3 : 6], that is, the maximum sum is $s_{3,6}$

A First Solution to the Maximum Subarray Problem

Our first algorithm for the maximum subarray problem, which we call MaxsubSlow, is shown in Algorithm given below. It computes the maximum of every possible subarray summation, $s_{j,k}$, of A separately.

Algorithm MaxsubSlow(A):

Input: An n-element array A of numbers, indexed from 1 to n.

Output: The subarray summation value such that A[j] + · · · + A[k] is maximized.

```
for j ← 1 to n do
    m ← 0 // the maximum found so far
    for k ← j to n do
        s ← 0 // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return max
```

It isn't hard to see that the MaxsubSlow algorithm is correct. This algorithm calculates the partial sum, $s_{j,k}$, of every possible subarray, by adding up the values in the subarray from a_j to a_k. Moreover, for every such subarray sum, it compares that sum to a running maximum

Contd...

and if the new value is greater than the old, it updates that maximum to the new value. In the end, this will be maximum subarray sum.

Incidentally, both the calculating of subarray summations and the computing of the maximum so far are examples of the accumulator design pattern, where we incrementally accumulate values into a single variable to compute a sum or maximum (or minimum). This is a pattern that is used in a lot of algorithms, but in this case it is not being used in the most efficient way possible.

Analyzing the running time of the MaxsubSlow algorithm is easy. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the inner-most loop, for index i, will iterate at most n times. Thus, the running time of the MaxsubSlow algorithm is O(n^3). Unfortunately, in spite of its use of the accumulator design pattern, giving theMaxsubSlow algorithm as a solution to the maximum subarray problem would be a bad idea during a job interview. This is a slow algorithm for the maximum subarray problem.

An Improved Maximum Subarray Algorithm

We can design an improved algorithm for the maximum subarray problem by observing that we are wasting a lot of time by recomputing all the subarray summations from scratch in the inner loop of the MaxsubSlow algorithm. There is a much more efficient way to calculate these summations. The crucial insight is to consider all the prefix sums, which are the sums of the first t integers in A for t = 1,2, . . . , n. That is, consider each prefix sum, S_t, which is defined as

$$s_t = a_1 + a_2 + ... + a_t = \sum_{i=1}^{t} a_i$$

If we are given all such prefix sums, then we can compute any subarray summation, $s_{j,k}$, in constant time using the formula:

$$s_{j,k} = S_k - S_{j-1}$$

where we use the notational convention that S0 = 0. To see this, note that

$$\begin{aligned} S_k - S_{j-1} &= \sum_{i=1}^{k} a_i - \sum_{i=1}^{j-1} a_i \\ &= \sum_{i=j}^{k} a_i = s_{j,k}, \end{aligned}$$

where we use the notational convention that $\sum_{i=1}^{0} a_i = 0$.

We can incorporate the above observations into an improved algorithm for the maximum subarray problem, called MaxsubFaster, which we show in Algorithm given below.

Algorithm MaxsubFaster(A):

Input: An n-element array A of numbers, indexed from 1 to n.

Output: The subarray summation value such that A[j] + · · · + A[k] is maximized.

```
S0 ← 0 // the initial prefix sum
for i ← 1 to n do
Si ← Si″1 + A[i]
```

Contd...

```
for j ← 1 to n do
m ← 0 // the maximum found so far
for k ← j to n do
if Sk “ Sj”1 > m then
m ← Sk “ Sj”1
return max
```

Analyzing the MaxsubFaster Algorithm

The correctness of the MaxsubFaster algorithm follows along the same arguments as for the MaxsubSlow algorithm, but it is much faster. In particular, the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the steps inside that loop will only take O(1) time in each iteration. Thus, the total running time of the MaxsubFaster algorithm is $O(n_2)$, which improves the running time of the MaxsubSlow algorithm by a linear factor. True story: a former student of one of the authors gave this very algorithm during a job interview for a major software company, when asked about the maximum subarray problem, correctly observing that this algorithm beats the running time of the naive $O(n^3)$-time algorithm by a linear factor. Sadly, this student did not get a job offer, however, and one possible reason could have been because there is an even better solution to the maximum subarray problem, which the student didn't give.

A Linear-Time Maximum Subarray Algorithm

We can improve the running time for solving the maximum subarray further by applying the intuition behind the prex summations idea to the computation of the maximum itself. That is, what if, instead of computing a partial sum, S_t , for t = 1,2, . . . , n, of the values of the subarray from a_1 to a_t , we compute a "partial maximum," M_t , which is the maximum summation of a subarray of A[1 : t] that ends at index t?

Such a denition is an interesting idea, but it is not quite right, because it doesn't include the boundary case where we wouldn't want any subarray that ends at t, in the event that all such subarrays sum up to a negative number. So, recalling our notation of letting $s_{j,k}$ denote the partial sum of the values in A[j : k], let us dene

$$M_t = \max\left\{0, \max_{j \le t}\{S_{j,t}\}\right\}$$

Note that if we know all the M_t values, for t = 1,2, . . . , n, then the solution to the maximum subarray problem would simply be the maximum of all these values. So let us consider how we could compute these Mt values.

The crucial observation is that, for t ≥ 2, if we have a maximum subarray that ends at t, and it has a positive sum, then it is either A[t : t] or it is made up of the maximum subarray that ends at t “ 1 plus A[t]. If this were not the case, then we could make an even bigger subarray by swapping out the one we chose to end at t “ 1 with the maximum one that ends at t “ 1, which would contradict the fact that we have the maximum subarray that ends at t. In addition, if taking the value of maximum subarray that ends at t “ 1 and adding A[t] makes this sum no longer be positive, then M_t = 0, for there is no subarray that ends at t with a positive summation. In other words, we can dene M_0 = 0 as a boundary condition, and use the following formula to compute M_t , for t = 1,2, . . . , n:

$$M_t = \max\{0, M_{t-1} + A[t]\}$$

Contd...

Therefore, we can solve the maximum subarray problem using the algorithm, MaxsubFastest, shown in Algorithm given below.

Algorithm MaxsubFastest(A):

Input: An n-element array A of numbers, indexed from 1 to n.

Output: The subarray summation value such that A[j] + · · · + A[k] is maximized.

```
M0 ← 0 // the initial prex maximum
for t ← 1 to n do
Mt ← max{0, Mt”1 + A[t]}
m ← 0 // the maximum found so far
for t ← 1 to n do
m ← max{m, Mt}
return m
```

Analyzing the MaxsubFastest Algorithm

The MaxsubFastest algorithm consists of two loops, which each iterate exactly n times and take O(1) time in each iteration. Thus, the total running time of the MaxsubFastest algorithm is O(n). Incidentally, in addition to using the accumulator pattern, to calculate the M_t and m variables based on previous values of these variables, it also can be viewed as a simple application of the dynamic programming technique. Given all these positive aspects of this algorithm, even though we can’t guarantee that a prospective employee will get a job offer by describing the MaxsubFastest algorithm when asked about the maximum subarray problem, we can at least guarantee that this is the way to nail this question.

Source: http://www.ics.uci.edu/~goodrich/teach/cs161/notes/MaxSubarray.pdf

2.4 Summary

- We can use various operations on data structures such as traversing, insertion, deletion, sorting, merging, etc.
- An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.
- The most important resource to analyze is generally the running time. Several factors affect the running time of a program.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
- The difference between space complexity and time complexity is that space can be reused. Space complexity is not affected by determinism or nondeterminism.
- “Big O” refers to a way of rating the efficiency of an algorithm. It is only a rough estimate of the actual running time of the algorithm, but it will give you an idea of the performance relative to the size of the input data.

2.5 Keywords

Insertion Operation: Insertion operation is used for adding a new element to the list.

Algorithm: An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.

Algorithm Complexity: The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.

Time complexity : Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

Space complexity : Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Big O: "Big O" refers to a way of rating the efficiency of an algorithm.

Constant time: This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

Linear time: This means that the algorithm requires a number of steps proportional to the size of the task.

2.6 Review Questions

1. Discuss the use of various data structure operations.
2. What is an algorithm? Discuss the process of analyzing algorithms.
3. Illustrate the use of running time in analyzing algorithm.
4. Explain the concept of algorithm complexity.
5. Describe the main complexity measures of the efficiency of an algorithm.
6. There is often a time-space-tradeoff involved in a problem. Comment.
7. What is Big O notation? Discuss with examples.
8. Define some theorems which can be used to simplify big O calculations.
9. Discuss the asymptotic behavior of the growth rate functions.
10. Discuss the various properties of Big O notation.

Answers: Self Assessment

1. Insertion
2. Sorting
3. algorithm
4. Model
5. complexity
6. Time complexity
7. Space complexity
8. Big O
9. O(g(n))
10. constant time
11. linear time
12. quadratic time
13. *n log n*
14. *n log n*

15. transitive

2.7 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.leda-tutorial.org/en/unofficial/ch02s02s03.html

http://www.computing.dcu.ie/~nstroppa/teaching/ca313_introduction.pdf

http://www.xpode.com/ShowArticle.aspx?Articleid=87

http://www.slideshare.net/NavtarSidhuBrar/data-structure-and-its-types-7577762

Notes

Unit 3: Recursion

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of recursion
- Explain recursive functions
- Discuss examples of recursive functions

Introduction

Sometimes a problem is too difficult or too complex to solve because it is too big. If the problem can be broken down into smaller versions of itself, we may be able to find a way to solve one of these smaller versions and then be able to build up to a solution to the entire problem. This is the idea behind recursion; recursive algorithms break down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results. Recursion turns out to be a wonderful technique for dealing with many interesting problems. Solutions written recursively are often simple. Recursive solutions are also often much easier to conceive of and code than their iterative counterparts. In this unit, we will discuss the concept of recursion and recursive functions.

3.1 Concept of Recursion

A recursive definition is defined in terms of itself. Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition so that successive repetitions are processed up to the critical step where the condition is met at which time the rest of each repetition is processed from the last one called to the first.

In computer programming, a recursion is programming that is recursive, and recursive has two related meanings:

- A recursive procedure or routine is one that has the ability to call itself. This usually means that it has the capability to save the condition it was in or the particular process it is serving when it calls itself (otherwise, any variable values that have been developed in executing the code are overlaid by the next iteration or go-through).

Did u know? Typically, this is done by saving values in registers or data area stacks before calling itself or at the beginning of the sequence where it has just been reentered.

- A recursive expression is a function, algorithm, or sequence of instructions (typically, an IF, THEN, ELSE sequence) that loops back to the beginning of itself until it detects that some condition has been satisfied.

3.1.1 Advantages of Recursion

- Recursion is more elegant and requires few variables which make program clean.
- Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.
- Recursion may provide a simpler solution than an iterative one. Sometimes iterative solutions lend themselves to complex algorithms
- It may lead to reduction in code size

3.1.2 Disadvantages of Recursion

- It is hard to think the logic of a recursive function.
- It is also difficult to debug the code containing recursion.
- It may require large amounts of memory.
    - ❖ For each call to the function more memory is allocated for local variables

Caution "Call stack" continues to grow for each call; may lead to stack overflow.

3.1.3 Significance of Recursion

Given that recursion is in general less efficient, why would we use it? There are two situations where recursion is the best solution:

- The problem is much more clearly solved using recursion: there are many problems where the recursive solution is clearer, cleaner, and much more understandable.

Caution As long as the efficiency is not the primary concern, or if the efficiencies of the various solutions are comparable, then you should use the recursive solution.

- Some problems are much easier to solve through recursion: there are some problems which do not have an easy iterative solution. Here you should use recursion.

Example: The Towers of Hanoi problem is an example of a problem where an iterative solution would be very difficult. We'll look at Towers of Hanoi in a later section of this guide.

Task Analyze the applications of recursion.

Self Assessment Questions

State true or false:

1. Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition.
2. A recursive expression is a function, algorithm, or sequence of instructions that loops back to the beginning of itself until it detects that some condition has been satisfied.
3. Recursion may lead to increase in code size.
4. As long as the efficiency is not the primary concern, or if the efficiencies of the various solutions are comparable, then you should use the recursive solution.
5. Problems are difficult to solve through recursion.
6. Recursion may require large amounts of memory.

3.2 Recursive function

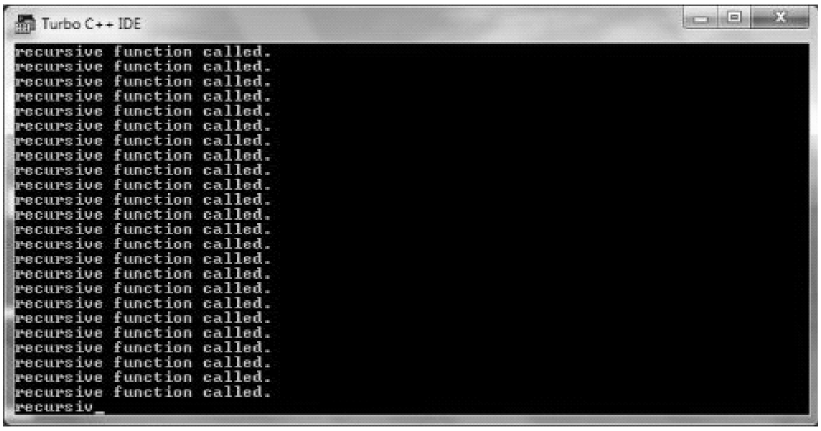
A recursive function is a function that calls itself during its execution. This enables the function to repeat itself several times, outputting the result and the end of each iteration. That is, a function is called "recursive" if a statement within body of that function calls the same function.

Example: look at below code:

```
void main()
{
printf("recursive function called.\n");
main();
}
```

When you will run this program it will print message "recursive function called." indefinitely.

Recursive function example output is given as below.



```
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursive function called.
recursiv_
```

Source: http://rajkishor09.hubpages.com/hub/C-Programming-Recursive-Function

Notes If you are using Turbo C/C++ compiler then you need to press Ctrl + Break key to break this in definite loop.

Let us consider another example:

Example:

Below is an example of a recursive function.

```
function Count (integer N)
  if (N <= 0) return "Must be a Positive Integer";
  if (N > 9) return "Counting Completed";
  else return Count (N+1);
end function
```

The function Count() above uses recursion to count from any number between 1 and 9, to the number 10. For example, Count(1) would return 2,3,4,5,6,7,8,9,10. Count(7) would return 8,9,10. The result could be used as a roundabout way to subtract the number from 10.

Recursive functions are common in computer science because they allow programmers to write efficient programs using a minimal amount of code. The downside is that they can cause infinite loops and other unexpected results if not written properly.

Example: in the example above, the function is terminated if the number is 0 or less or greater than 9.

Notes If proper cases are not included in the function to stop the execution, the recursion will repeat forever, causing the program to crash, or worse yet, hang the entire computer system.

Before we move to another example lets have attributes of "recursive function":-

- A recursive function is a function which calls itself.
- The speed of a recursive program is slower because of stack overheads. (This attribute is evident if you run above C program.)
- A recursive function must have recursive conditions, terminating conditions, and recursive expressions.

Did u know? Every recursive function must be provided with a way to end the recursion.

Example 1: Calculating factorial value using recursion

To understand how recursion works lets have another popular example of recursion. In this example we will calculate the factorial of n numbers. The factorial of n numbers is expressed as a series of repetitive multiplication as shown below:

Factorial of n = n(n-1)(n-2)……1.

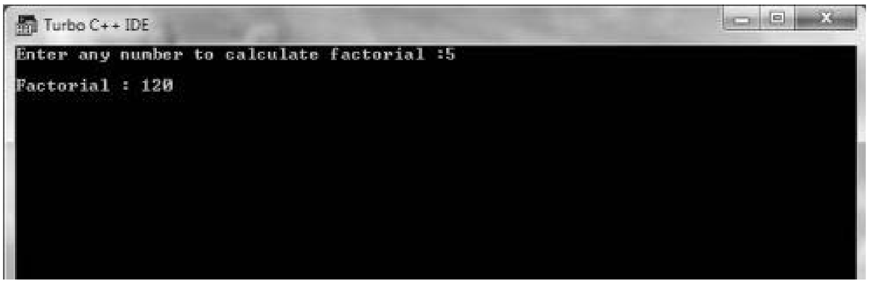
Notes

Example: Factiorial of 5 = 5 × 4 × 3 × 2 × 1

= 120

```
1      #include<stdio.h>
2      #include<conio.h>
3
4      int factorial(int);
5
6      int factorial (int i)
7      {
8       int f;
9       if(i==1)
10      return 1;
11      else
12      f = i* factorial (i-1);
13     return f;
14     }
15
16     void main()
17     {
18      int x;
19      clrscr();
20      printf("Enter any number to calculate factorial :");
21      scanf("%d",&x);
22      printf("\nFactorial : %d", factorial (x));
23      getch();
24     }
```

The output is shown as below:



```
Enter any number to calculate factorial :5

Factorial : 120
```

Source: http://rajkishor09.hubpages.com/hub/C-Programming-Recursive-Function

So from **line no. 6 – 14** is a user defined recursive function "**factorial**" that calculates factorial of any given number. This function accepts integer type argument/parameter and return integer value.

In **line no. 9** we are checking that whether value of i is equal to 1 or not; i is an integer variable which contains value passed from main function i.e. value of integer variable x. If user enters 1 then the factorial of 1 will be 1. If user enters any value greater than 1 like 5 then it will execute statement in **line no. 12** to calculate factorial of 5. This line is extremely important because in this line we implemented recursion logic.

Let’s see how **line no. 12** exactly works. Suppose value of i=5, since i is not equal to 1, the statement:

```
f = i* factorial (i-1);
```

will be executed with i=5 i.e.

```
f = 5* factorial (5-1);
```

will be evaluated. As you can see this statement again calls factorial function with value i-1 which will return value:

```
4*factorial(4-1);
```

This recursive calling continues until value of i is equal to 1 and when i is equal to 1 it returns 1 and execution of this function stops. We can review the series of recursive call as follow:

```
f = 5* factorial (5-1);
f = 5*4* factorial (4-1);
f = 5*4*3* factorial (3-1);
f = 5*4*3*2* factorial (2-1);
f = 5*4*3*2*1;
f = 120;
```

Example 2: Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include <stdio.h>
int sum(int n);
int main(){
  int num,add;
  printf(“Enter a positive integer:\n”);
  scanf(“%d”,&num);
  add=sum(num);
  printf(“sum=%d”,add);
}
int sum(int n){
  if(n==0)
  return n;
  else
  return n+sum(n-1); /*self call to function sum() */
}
```

The output is given as below:

```
Enter a positive integer:
5
15
```

In, this simple C program, sum() function is invoked from the same function. If n is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, n is 5 initially. Then, during next function calls, 4 is passed to function and the

Notes

value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
```

In this example when, n is equal to 0, there is no recursive call and recursion ends.

Example 3: C program to Reverse a Sentence Using Recusrion

This program takes a sentence from user and reverses that sentence using recursion. This program does not use string to reverse the sentence or store the sentence.

Code to reverse a sentence using recursion is given below.

```
/* Example to reverse a sentence entered by user without using strings.
*/
#include <stdio.h>
void Reverse();
int main()
{
 printf(“Enter a sentence: “);
 Reverse();
 return 0;
}
void Reverse()
{
 char c;
 scanf(“%c”,&c);
 if( c != ‘\n’)
 {
 Reverse();
 printf(“%c”,c);
 }
}
```

The output is given as below:

Enter a sentence: margorp emosewa

awesome program

This program prints "Enter a sentence: " then, Reverse() function is called. This function stores the first letter entered by user and stores in variable c. If that variable is other than '\n' [enter character] then, again Reverse()function is called. Don't assume this Reverse() function and the Reverse() function before is same although they both have same name. Also, the variables are also different, i.e., c variable in both functions are also different. Then, the second character is stored in variable c of second Reverse function. This process goes on until user enters '\n'. When, user enters '\n', the last function Reverse() function returns to second last Reverse() function and prints the last character. Second last Reverse() function returns to the third last Reverse() function and prints second last character. This process goes on and the final output will be the reversed sentence.

Example 4: C Program to Find H.C.F Using Recursion

This program takes two positive integers from user and calculates HCF or GCD using recursion.

Code to calculate H.C.F using recursion is given below:

```
/* Example to calculate GCD or HCF using recursive function. */
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
 int n1, n2;
 printf("Enter two positive integers: ");
 scanf("%d%d", &n1, &n2);
 printf("H.C.F of %d and %d = %d", n1, n2, hcf(n1,n2));
 return 0;
}
int hcf(int n1, int n2)
{
 if (n2!=0)
 return hcf(n2, n1%n2);
 else
 return n1;
}
```

The output is given as below:

Enter two positive integers: 366

60

H.C.F of 366 and 60 = 6

Task Illustrate the prime number program in c using recursion.

Self Assessment Questions

Fill in the blanks:

7. A .... is a function that calls itself during its execution.

8. If you are using Turbo C/C++ compiler then you need to press Ctrl + Break key to break in .... loop.

9. The function .... uses recursion to count from any number to any other number.

10. Recursive functions can cause .... loops and other unexpected results if not written properly.

11. If proper cases are not included in the function to stop the......................., the recursion will repeat forever.

12. The speed of a recursive program is slower because of ....

13. Every recursive function must be provided with a way to .... the recursion.

14. Positive integers are known as ....

15. .... function stores the first letter entered by user and stores in variable .

Case Study

Recursive Functions

We're going to implement a recursive function in MIPS. Recursion is one of those programming concepts that seem to scare students. One reason is that recursive functions can be difficult to trace properly without knowing something about how a stack works.

"Classic" recursion attempts to solve a "big" problem by solving smaller versions of the problem, then using the solutions to the smaller versions of the problem to solve the big problem.

"Classic" recursion is a divide-and-conquer method. For example, consider merge sorting. The idea behind merge sorting is to divide an array into two halves, and sort each half. Then, you "merge" the two sorted halves to sort the entire list.

Thus, to solve the big problem of sorting an array, solve the smaller problem of sorting part of an array, and use the solution of the smaller sorted array to solve the big problem (by merging).

How do you solve the smaller version of the problem? You break that smaller problem into even smaller problems, and solve those.

Eventually, you get to the smallest sized problem, which is called the *base case*. This can be solved without using recursion.

Recursion allows you to express solutions to a problem very compactly and elegantly. This is why people like using recursion.

However, recursion can use a lot of stack, and if you're not careful, you can overflow the stack.

For recursion to be successful, the problem needs to have a recursive substructure. This is a fancy term that says that to solve a big problem (say of size N), you should be able to solve a smaller problem (of smaller size) in exactly the same way, except the problem size is smaller.

As an analogy, you might have to, say, sweep the floor. Sweeping half the floor is the same as sweeping the entire floor, except you do it on a smaller area. A problem that is solved recursively generally needs to have this property.

Many folks think recursive functions are implemented in some weird and unusual way in assembly language. They can't believe it's implemented just like any other function.

The biggest misconception is that a function calls a smaller and smaller version, reaches the base case, then quits. For example, you might call **fact(3)**, which calls **fact(2)**, which calls **fact(1)**, which finally calls **fact(0)**, the base case.

Some programmers think the function stops there, right at the base case, and then jumps back to the initial recursive call. But you know that's not true.

Or do you? Suppose we have function **f()**, which calls function **g()**, and **g()** calls **h()**.

What happens when we're done with **h()**? We return back to **g()**. And what happens when we're done with **g()**? We return back to **f()**.

Recursive functions behave that way too! Thus, **fact(3)** calls **fact(2)** which calls **fact(1)** which calls **fact(0)**, the base case. We call this phase the winding of the stack.

Once **fact(0)** is done, it goes back to **fact(1)**, just like it would for non-recursive functions. When **fact(1)** is done, it goes back to **fact(2)**. When **fact(2)** is done, it goes back to **fact(3)**. When **fact(3)** is done, it goes back to whoever called **fact(3)**. We call this the "unwinding" of the stack.

During the winding of the stack, you are making progress towards the base case. Basically, you're trying to get to the base case, solve the base case, and slowly grow your solution back as you go though the unwinding part of the recursion. Thus, winding heads to the solution of the base case, while unwinding typically grows the solution from base case back to the original call.

An Example

Let's solve the following recursive function, written in C. This sums all elements of an array.

```
int sum( int arr[], int size ) {
 if ( size == 0 )
 return 0 ;
 else
 return sum( arr, size - 1 ) + arr[ size - 1 ] ;
}
```

A MIPS Translation

We assume **arr** is in **$a0** and **size** is in **$a1**.

We have to decide what to save to the stack. We have two choices. Either save **size - 1**, from which we can compute **arr[size - 1]**, or save **arr[size - 1]**. Let's opt to save **size - 1** on the stack.

Contd...

Notes

We must also save the return address, **$ra** since there is a function call. It's usually easy to tell whether to save the return address to the stack. If there's a function call, then save it.

```
sum:  addi $sp, $sp, -8          # Adjust sp
      addi $t0, $a0, -1          # Compute size - 1
      sw $t0, 0($sp)             # Save size - 1 to stack
      sw $ra, 4($sp)             # Save return address
      bne $a0, $zero, ELSE       # branch ! ( size == 0 )
      li $v0, 0                  # Set return value to 0
      addi $sp, $sp, 8           # Adjust sp
      jr $ra                     # Return
ELSE: move $a1, $t0              # update second arg
      jal sum
      lw $t0, 0($sp)             # Restore size - 1 from stack
      li $t7, 4                  # t7 = 4
      mult $t0, t7               # Multiple size - 1 by 4
      mflo $t1                   # Put result in t1
      add $t1, $t1, $a0          # Compute & arr[ size - 1 ]
      lw $t2, 0($t1)             # t2 = arr[ size - 1 ]
      add $v0, $v0, $t2          # retval = $v0 + arr[size - 1]
      lw $ra, 4($sp)             # restore return address from stack
      addi $sp, $sp, 8           # Adjust sp
      jr $ra                     # Return
```

Notice that we could have tested the if-else statement right away, and not used the stack. However, by adjusting the stack right away, it makes sure that we deal with the stack. So that's why we often have the saving of registers on the stack at the beginning even if the base case might not require any adjustment of the stack.

There's nothing wrong to avoid moving the stack for the base case, however.

Notice in the ELSE, we copy **size - 1**, which is in **$t0** to **$a0**. We didn't retrieve it from the stack. Why not? Because up to this point, we've made no subroutine call. Thus, **$t0** still contains **size - 1**.

A Second Example

Let's solve the following variation of a recursive function, written in C. This sums only the odd elements of an array.

```
int sumOdd( int arr[], int size ) {
 if ( size == 0 )
 return 0 ;
 else if ( arr[ size - 1 ] % 2 == 1 )
 return sumOdd( arr, size - 1 ) + arr[ size - 1 ] ;
 else
 return sumOdd( arr, size - 1 ) ;
}
```

Contd...

A MIPS Translation

Usually, the hard part is to decide what registers to save. **arr** is stored in **$a0**, and that this value really doesn't change throughout. **size** may need to be saved, though **size - 1** appears to be more useful. Since we make calls to **sumOdd**, we need to save **$ra**.

So, let's save **size - 1** and **$ra** to the stack. It turns out we also need to save **arr[size - 1]** to the stack too.

```
sumOdd: addi  $sp,  $sp,  -12      #  Adjust  sp
        addi  $t0,  $a0,  -1       #  Compute  size  -  1
        sw  $t0,  0($sp)           #  Save  size  -  1  to  stack
        sw  $ra,  4($sp)           #  Save  return  address
        bne  $a0,  $zero,  ELSE2   #  branch  !(  size  ==  0  )
        li  $v0,  0                #  Set  return  value  to  0
        addi  $sp,  $sp,  12       #  Adjust  sp
        jr  $ra                    #  Return
ELSE2:  li  $t7,  4                #  t7  =  4
        multi  $t0,  t7            #  Multiple  size  -  1  by  4
        mflo  $t1                  #  Put  result  in  t1
        add  $t1,  $t1,  $a0       #  Compute  &  arr[  size  -  1  ]
        lw  $t2,  0($t1)           #  t2  =  arr[  size  -  1  ]
        andi  $t3,  $t2,  1        #  is  arr[  size  -  1  ]  odd
        beq  $t3,  $zero,  ELSE3   #  branch  if  even
        sw  $t2,  8($sp)           #  save  arr[  size  -  1  ]  on  stack
        move  $a1,  $t0            #  update  second  arg
        jal  sumOdd
        lw  $t2,  8($sp)           #  restore  arr[  size  -  1  ]  from  stack
        add  $v0,  $v0,  $t2       #  update  return  value
        lw  $ra,  4($sp)           #  restore  return  address  from  stack
        addi  $sp,  $sp,  12       #  Adjust  sp
        jr  $ra                    #  Return
ELSE3:  move  $a1,  $t0            #  update  second  arg
        jal  sumOdd
        lw  $ra,  4($sp)           #  restore  return  address  from  stack
        addi  $sp,  $sp,  12       #  Adjust  sp
        jr  $ra                    #  Return
```

As it turns out, we did't have to save **size - 1**. So we can leave that off the stack if you want. As a rule of thumb, you may end up deciding later on what to save and what not to save. A compiler, of course, has to determine this without being too clever. Sometimes a compiler might save too much to the stack, just so it's easier to generate code.

A few comments:

- In **ELSE2**, we use **mult**. The result is placed in two registers HI and LO. Since the number is likely to be quite small, we can assume the result is in LO, and HI contains all 0's. We can access the low word from LO, using **mflo** (move from LO).

Contd...

Notes

- Note that using **mult** to multiply by 4 is generally a slow instruction compared to the doubling a value twice, or logical left shifting by 2 bits.
- In **ELSE2** we need to access **arr[size - 1]**. This then has to be placed on the stack because we'll need to use the value after the **jal** call.
- We use **andi** (bitwise and immediate) to test for even or odd. Essentially we mask all but the least significant bit and test if that value is 0 or 1. Fortunately, this trick works for UB and 2C representations.
- **ELSE3** is quite short because we don't access the array elements. We don't even need to worry about **$v0** because the call to **jal sumOdd** fills it in for us. All we have to do is to restore the return address and jump back.
- Array access tends to bloat the length of recursive function code in assembly language.

Conclusion

If you were to translate a plain C function, which had calls to helper functions, you'd find the code to be very similar to the recursive function call.

The main difference is **jal** does not call the same function. It calls a different one. However, it's all pretty much the same after that. The best way to see this is to write a non-recursive function in C, with helper functions, then translate it back.

Source: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/case_rec.html

3.3 Summary

- Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition.
- Recursive algorithms break down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results.
- Recursion may provide a simpler solution than an iterative one. S
- A recursive function is a function that calls itself during its execution.
- A function is called "recursive" if a statement within body of that function calls the same function.
- Recursive functions are common in computer science because they allow programmers to write efficient programs using a minimal amount of code.
- The speed of a recursive program is slower because of stack overheads.
- If proper cases are not included in the function to stop the execution, the recursion will repeat forever, causing the program to crash, or worse yet, hang the entire computer system.

3.4 Keywords

Recursion: Recursion is a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself in a step having a termination condition.

Recursive function: A recursive function is a function that calls itself during its execution.

Notes

Algorithm: An algorithm is a set of instructions, sometimes called a procedure or a function, that is used to perform a certain task.

Recursive procedure: A recursive procedure or routine is one that has the ability to call itself.

Recursive expression: A recursive expression is a function, algorithm, or sequence of instructions that loops back to the beginning of itself until it detects that some condition has been satisfied.

Iteration: Iteration is a problem-solving or computational method in which a succession of approximations used to achieve a desired degree of accuracy.

3.5 Review Questions

1. Explain the concept of recursion with example.
2. Discuss the advantages and disadvantages of recursion.
3. Discuss the situation where recursion provides the best solution.
4. What is a recursive function? Discuss with example.
5. If proper cases are not included in the function to stop the execution, the recursion will repeat forever. Comment.
6. Write a C program to calculate the power of a number using recursion.
7. Write a C program to find size of int, float, double and char of your system.
8. Write a C program to check whether a number is positive or negative or zero.
9. Write a C program to Count Number of Digits of an Integer.
10. Write a recursive C program to find the maximum value in an array.

Answers: Self Assessment

1. True
2. True
3. False
4. True
5. False
6. True
7. recursive function
8. Definite
9. Count()
10. Infinite
11. execution
12. stack overheads
13. end
14. natural numbers
15. Reverse

3.6 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Notes

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://gd.tuwien.ac.at/languages/c/programming-bbrown/c_053.htm

http://www.indiastudychannel.com/resources/92585-Recursion-C-Language.aspx

http://pages.cs.wisc.edu/~calvin/cs110/RECURSION.html

http://www.programiz.com/c-programming/examples/hcf-recursion

Notes

Unit 4: Arrays

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of linear and multi-dimensional arrays
- Explain the representation of linear arrays
- Discuss the representation of multi-dimensional arrays

Introduction

A data structure is the way data is stored in the machine and the functions used to access that data. An easy way to think of a data structure is a collection of related data items. An array is a data structure that is a collection of variables of one type that are accessed through a common name. A specific element is accessed by an index. A C-style array is aggregate data type. Array can hold multiple values of a single type. Elements are referenced by the array name and an ordinal index. Each element is a value, index pair, the what, where duo. Indexing begins at zero. The array forms a contiguous list in memory. The name of the array holds the address of the first array element. We specify the array size at compile time, often with a named constant.

4.1 Linear Arrays

The simplest form of array is a one-dimensional array that may be defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations.

Example: an array may contain all integers or all characters or any other data type, but may not contain a mix of data types.

Notes

The general form for declaring a single dimensional array is:

```
data_type  array_name[expression];
```

where data_type represents data type of the array. That is, integer, char, float etc. array_name is the name of array and expression which indicates the number of elements in the array.

Example: consider the following C declaration:

```
int  a[100];
```

It declares an array of 100 integers.

The amount of storage required to hold an array is directly related to its type and size. For a single dimension array, the total size in bytes required for the array is computed as shown below.

Memory required (in bytes) = size of (data type) X length of array

The first array index value is referred to as its lower bound and in C it is always 0 and the maximum index value is called its upper bound.

Did u know? The number of elements in the array, called its range is given by upper bound-lower bound.

We store values in the arrays during program execution. Let us now see the process of initializing an array while declaring it.

```
  int  a[4]  =  {34,60,93,2};
int  b[]  =  {2,3,4,5};
float  c[]  =  {-4,6,81,"  60};
```

We conclude the following facts from these examples:

- If the array is initialized at the time of declaration, then the dimension of the array is optional.
- Till the array elements are not given any specific values, they contain garbage values.

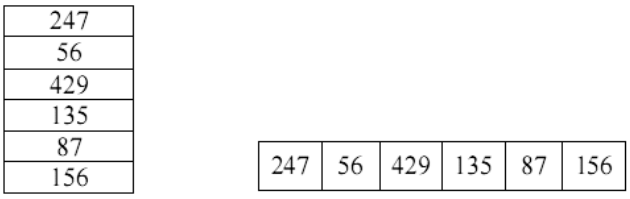
Example: Let us consider the following figure 4.1.

a) Array DATA be a 6-element linear array of integers such that

```
DATA[1]=247,  DATA[2]=56,  DATA[3]=429,  DATA[4]=135,  DATA[5]=87,
DATA[6]=156.
```

The following figures 4.1 and 4.2 depict the array DATA.

Figure 4.1 and Figure 4.2



247
56
429
135
87
156

247	56	429	135	87	156

Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Let us now consider the other figure 4.2.

b) An automobile company uses an array AUTO to record the number of automobiles sold each year from 1932 through 1984.

Rather than beginning the index set with 1, it is more useful to begin the index set with 1932, so we may learn that,

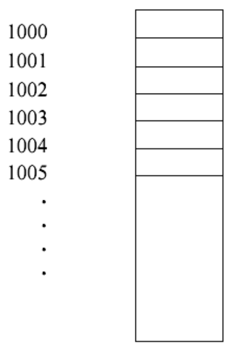
AUTO[K] = number of automobiles sold in the year K

Then, LB = 1932 is the lower bound and UB=1984 is the upper bound of AUTO.

4.1.1 Representation of Linear Arrays

Consider LA be a linear array in the memory of the computer. As we know that the memory of the computer is simply a sequence of addressed location as pictured in figure 4.3 as given below.

Figure 4.3: Computer Memory



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Let us use the following notation when calculate the address of any element in linear arrays in memory,

LOC(LA[K]) = address of the element LA[K] of the array LA.

The elements LA are stored in successive memory cells.

Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, which is denoted by:

Base(LA)

and called the base address of LA. Using this address Base(LA), the computer calculates the address of any element of LA by the following formula:

LOC(LA[K]) = Base(LA) + w (K- lower bound) (a)

where w is the number of words per memory cell for the array LA.

Let us observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K.

Notes Given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Notes

Example: Consider the previous example of array AUTO, which records the number of automobiles sold each year from 1932 through 1984. Array AUTO appears in memory is pictured in Figure 4.4. Assume, Base(AUTO) = 200 and w = 4 words per memory cell for AUTO.

Then the base addresses of following arrays are,

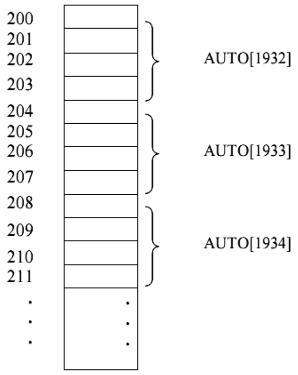
LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, ...

Let us find the address of the array element for the year K = 1965. It can be obtained by using Equation (1.4b):

LOC(AUTO[1965]) = Base(AUTO) + w(1965 – lower bound) = 200 + 4(1965-1932) = 332

Again, we emphasize that the contents of this element can be obtained without scanning any other element in array AUTO.

Figure 4.4: Array AUTO appears in Memory



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Task Analyze the uses of arrays.

4.1.2 Traversing Linear Arrays

Consider let A be a collection of data elements stored in the memory of the computer. Suppose we want to either print the contents of each element of A or to count the number of elements of A with a given property. This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of A exactly once.

The following algorithm is used to traversing a linear array LA.

As we know already, here, LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

```
1.  [Initialize counter] Set K:= LB.
2.  Repeat Steps 3 and 4 while K< UB
3.  [Visit element] Apply PROCESS to LA[K]
```

```
4.  [Increase counter] Set K:= K + 1
    [End of Step 2 loop]
5.  Exit.
```

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop. This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

```
1.  Repeat for K = LB to UB:
        Apply PROCESS to LA[K].
    [End of loop]
2.  Exit.
```

Example: Again consider the previous one [example 4.2] array AUTO, which records the number of automobiles sold each year from 1932 through 1984. Each of the following algorithms carry out the given operation involves traversing AUTO.

(a) Find the number NUM of years during which more than 300 automobiles were sold.

```
1.  [Initialization Step] Set NUM := 0.
2.  Repeat for K = 1932 to 1984:
        If AUTO[K] > 300, then: Set NUM := NUM + 1
    [End of loop]
3.  Return
```

(b) Print each year and the number of automobiles sold in that year.

```
1.  Repeat for K = 1932 to 1984:
        Write: K, AUTO[K].
    [End loop]
2.  Return.
```

Self Assessment Questions

Fill in the blanks:

1. The array forms a .... list in memory.
2. .... array that may be defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations.
3. The amount of .... required to hold an array is directly related to its type and size.
4. The first array index value is referred to as its ....
5. Expression indicates the number of .... in the array.

4.2 Multidimensional Arrays

Suppose that you are writing a chess-playing program. A chessboard is an 8-by-8 grid. What data structure would you use to represent it? You could use an array that has a chessboard-like

Notes

structure, i.e. a two-dimensional array, to store the positions of the chess pieces. Two-dimensional arrays use two indices to pinpoint an individual element of the array. This is very similar to what is called "algebraic notation", commonly used in chess circles to record games and chess problems.

In principle, there is no limit to the number of subscripts (or dimensions) an array can have. Arrays with more than one dimension are called multi- dimensional arrays.

While humans cannot easily visualize objects with more than three dimensions, representing multi-dimensional arrays presents no problem to computers.

In practice, however, the amount of memory in a computer tends to place limits on the size of an array. A simple four-dimensional array of double-precision numbers, merely twenty elements wide in each dimension, takes up 20^4 * 8, or 1,280,000 bytes of memory - about a megabyte.

Example: you have ten rows and ten columns, for a total of 100 elements. It's really no big deal. The first number in brackets is the number of rows, the second number in brackets is the number of columns. So, the upper left corner of any grid would be element [0][0]. The element to its right would be [0][1], and so on. Here is a little illustration to help.

Figure 4.5: Multi-dimensional array

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Three-dimensional arrays (and higher) are stored in the same way as the two-dimensional ones.

Notes They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest (then the next-to-last, and so on).

4.2.1 Two-Dimensional Arrays

A two-dimensional m x n array is a collection of m .n data elements such that each element is specified by a pair of integers (such as J, K), called subscripts, with the following property that,

$$I \leq J \leq m \text{ and } I \leq K \leq n$$

The element of A with first subscripts j and second subscript k will be denoted by

$$A\ J,K \text{ or } A[J, K]$$

Two-dimensional arrays are called matrices in mathematics and tables in business applications; hence two-dimensional arrays are called matrix arrays.

You can declare an array of two dimensions as follows:

```
datatype  array_name[size1][size2];
```

In the above example, variable_type is the name of some type of variable, such as int. Also, size1 and size2 are the sizes of the array's first and second dimensions, respectively.

Here is an example of defining an 8-by-8 array of integers, similar to a chessboard. Remember, because C arrays are zero-based, the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same: a two-dimensional array of 64 elements.

```
int  chessboard [8][8];
```

To pinpoint an element in this grid, simply supply the indices in both dimensions.

If you have an m x n array, it will have m * n elements and will require m*n*element size bytes of storage. To allocate storage for an array you must reserve this amount of memory. The elements of a two-dimensional array are stored row wise. If table is declared as:

```
int table [ 2 ] [ 3 ] = { 1,2,3,4,5,6 };
```

It means that element

```
table [0][0] = 1;

table [0][1] = 2;

table [0][2] = 3;

table [1][0] = 4;

table [1][1] = 5;

table [1][2] = 6;
```

The neutral order in which the initial values are assigned can be altered by including the groups in { } inside main enclosing brackets, like the following initialization as above:

```
int  table [2][3] = {{1,2,3}, {4,5,6}};
```

The value within innermost braces will be assigned to those array elements whose last subscript changes most rapidly. If there are few remaining values in the row, they will be assigned zeros. The number of values cannot exceed the defined row size.

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

It assigns values as

```
table [0][0] = 1;

table [0][1] = 2;

table [0][2] = 3;

table [1][0] = 4;

table [1][1] = 0;

table [1][2] = 0
```

Remember that, C language performs no error checking on array bounds. If you define an array with 50 elements and you attempt to access element 50 (the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task to check that all attempts to access or write to arrays are done only at valid array indexes.

⚠ *Caution* Writing or reading past the end of arrays is a common programming bug and is hard to isolate.

Let we understand that there is a standard way of drawing a two–dimensional m x n array A where the elements of A form a rectangular array with m rows and n columns where the element A[J, K] appears in row J and column K.

Notes

Did u know? A row is a horizontal list of elements and a column is a vertical list of elements.

In the following figure 4.6, we may observe that two-dimensional array A has 3 rows and 4 columns. Let we emphasize that each row contains those elements with the same first subscript, and each column contains those elements with the same second subscript.

Figure 4.6: Two-dimensional 3 x 4 array A

$$\begin{array}{cc|cccc|}
 & & 1 & 2 & 3 & 4 \\
 & 1 & A[1,1] & A[1,2] & A[1,3] & A[1,4] \\
\text{Rows} & 2 & A[2,1] & A[2,2] & A[2,3] & A[2,4] \\
 & 3 & A[3,1] & A[3,2] & A[3,3] & A[3,4]
\end{array}$$

Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Example:

Let we go through this example,

Let each student in a class of 25 students is given 4 tests. Assume the students are numbered from 1 to 25, the test scores can be assigned to a 25 x 4 matrix array SCORE as pictured in figure 4.7.

Thus, SCORE[K, L] contains the Kth student's score on the Lth test. In particular, the second row of an array,

```
SCORE [2, 1],    SCORE[2, 2]       SCORE[2, 3]         SCORE[2, 4]
```

contains the four test scores of the second student.

Let A is a two–dimensional m x n array. The first dimension of A contains the index set1,……., m with lower bound 1 and upper bound m; and the second dimension of A contains the index set 1,2,….. n, with lower bound 1 and upper bound n. The length of a dimension is the number of integers in its index set. The pair of lengths m x n (read "m by n") is called the size of the array. Let we find the length of a given dimension (i.e., the number of integers in its index set) by obtain from the formula,

Length = upper bound – lower bound + 1

Figure 4.7: Array SCORE

Student	Test 1	Test 2	Test 3	Test 4
1	84	73	88	81
2	95	100	88	96
3	72	66	77	72
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
25	78	82	70	85

Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

4.2.2 Representation of Two-Dimensional Arrays

Let A be a two-dimensional m x n array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of m.n sequential memory locations. If they are being stored in sequence, then how are they sequenced? Is it that the elements are stored row wise or column wise? Again, it depends on the operating system. Specifically, the programming languages will store the array A in either,

- Column by column, called column-major order, or
- Row by row, called row-major order.

Row Major Representation

The first method of representing a two-dimensional array in memory is the row major representation. Under this representation, the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set, and so forth.

The schematic of row major representation of an Array is shown in Figure 4.8.

Let us consider the following two-dimensional array:

```
a b c d
e f g h
i j k l
```

To make its equivalent row major representation, we perform the following process:

Move the elements of the second row starting from the first element to the memory location adjacent to the last element of the first row. When this step is applied to all the rows except for the first row, you have a single row of elements. This is the Row major representation.

By application of above mentioned process, we get {a, b, c, d, e, f, g, h, i, j, k, l }

Figure 4.8: Schematic of a Row major representation of an Array

Row 0	**Row 1**	**Row 2**	**.....**	**Row i**		

Column Major Representation

The second method of representing a two-dimensional array in memory is the column major representation. Under this representation, the first column of the array occupies the first set of the memory locations reserved for the array. The second column occupies the next set and so forth. The schematic of a column major representation is shown in Figure 4.9.

Consider the following two-dimensional array:

```
a b c d
e f g h
i j k l
```

To make its equivalent column major representation, we perform the following process:

Transpose the elements of the array. Then, the representation will be same as that of the row major representation.

Notes

Figure 4.9: Schematic of a Column major representation of an Array

Col 0	**Col 1**	**Col 2**	**.....**	**Col i**		

Task Compare and contrast row major representation and row major representation.

Self Assessment Questions

Fill in the blanks:

6. Arrays with more than one dimension are called .... arrays.
7. A simple four-dimensional array of .... numbers, merely twenty elements wide in each dimension, takes up 20^4 * 8, or 1,280,000 bytes of memory - about a megabyte.
8. .... arrays (and higher) are stored in the same way as the two-dimensional ones.
9. Two-dimensional arrays are called .... in mathematics.
10. If you have an m x n array, it will have .... elements and will require m*n*element size bytes of storage.
11. A row is a horizontal list of elements and a .... is a vertical list of elements.
12. The.............................. of a dimension is the number of integers in its index set.
13. The pair of lengths m x n (read "m by n") is called the.............................. of the array.
14. In the .... representation, the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set, and so forth.
15. In the .... representation in which the first column of the array occupies the first set of the memory locations reserved for the array.

Case Study

Case:

Write a C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.

```
#include <stdio.h>
int main(){
  float a[2][2], b[2][2], c[2][2];
  int i,j;
  printf(“Enter the elements of 1st matrix\n”);
```

/* Reading two dimensional Array with the help of two for loop. If there was an array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.*/

```
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
  printf(“Enter a%d%d: “,i+1,j+1);
```

Contd...

```
  scanf(“%f”,&a[i][j]);
  }
  printf(“Enter the elements of 2nd matrix\n”);
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
  printf(“Enter b%d%d: “,i+1,j+1);
  scanf(“%f”,&b[i][j]);
  }
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
/* Writing the elements of multidimensional array using loop. */
  c[i][j]=a[i][j]+b[i][j]; /* Sum of corresponding elements of two
arrays. */
  }
  printf(“\nSum Of Matrix:”);
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
  printf(“%.1f\t”,c[i][j]);
  if(j==1) /* To display matrix sum in order. */
  printf(“\n”);
  }
return 0;
}
Ouput:
Enter the elements of 1st matrix
Enter a11: 2;
Enter a12: 0.5;
Enter a21: -1.1;
Enter a22: 2;
Enter the elements of 2nd matrix
Enter b11: 0.2;
Enter b12: 0;
Enter b21: 0.23;
Enter b22: 23;

Sum Of Matrix:
2.2 0.5
-0.9 25.0
```

Question

Write a program to demonstrate two dimensional array.

Source: http://www.programiz.com/c-programming/c-multi-dimensional-arrays

Notes

4.3 Summary

- An array is a data structure that is a collection of variables of one type that are accessed through a common name.
- One-dimensional array is defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations.
- The first array index value is referred to as its lower bound and in C it is always 0 and the maximum index value is called its upper bound.
- Arrays with more than one dimension are called multi- dimensional arrays. Three-dimensional arrays (and higher) are stored in the same way as the two-dimensional ones.
- A two-dimensional m x n array is a collection of m .n data elements such that each element is specified by a pair of integers (such as J, K), called subscripts.
- Two-dimensional arrays are called matrices in mathematics and tables in business applications; hence two-dimensional arrays are called matrix arrays.
- The first method of representing a two-dimensional array in memory is the row major representation in which the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set, and so forth.
- The second method of representing a two-dimensional array in memory is the column major representation in which the first column of the array occupies the first set of the memory locations reserved for the array.

4.4 Keywords

Array: An array is a data structure that is a collection of variables of one type that are accessed through a common name.

One-dimensional array: One-dimensional array is defined as a finite ordered set of homogeneous elements, which is stored in contiguous memory locations.

Multi- dimensional arrays: Arrays with more than one dimension are called multi- dimensional arrays.

Row major representation: In the row major representation, the first row of the array occupies the first set of the memory location reserved for the array, the second row occupies the next set, and so forth.

Column major representation: In the column major representation in which the first column of the array occupies the first set of the memory locations reserved for the array.

Data type: A data type in a programming language is a set of data with values having predefined characteristics.

Array Indexing: Array indexing refers to any use of the square brackets ([]) to index array values.

Double precision : Double precision is a computer number format that occupies two adjacent storage locations in computer memory.

4.5 Review Questions

1. What is one-dimensional array? Discuss with example.
2. Illustrate the declaration of single dimensional array in C.

3. Explain the representation of linear array with example.
4. Describe the concept of traversing Linear Arrays. Give example.
5. Discuss multi-dimensional arrays with example.
6. What is two-dimensional array? Also discuss the declaration of two-dimensional array in c with example.
7. Describe the various methods used to represent two-dimensional arrays.
8. Make distinction between one dimensional arrays and two dimensional array.
9. Three-dimensional arrays (and higher) are stored in the same way as the two-dimensional ones. Comment.
10. Write a program illustrating the implementation of two-dimensional arrays in c.

Answers: Self Assessment

1. contiguous
2. One-dimensional
3. storage
4. lower bound
5. elements
6. multi- dimensional
7. double-precision
8. Three-dimensional
9. matrices
10. m * n
11. column
12. Length
13. size
14. row major
15. column major

4.6 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://rajkishor09.hubpages.com/hub/Array-in-C-programming—Programmers-view

http://www.eskimo.com/~scs/cclass/int/sx9.html

http://www.cprogrammingexpert.com/C/Tutorial/t_dimensional_array.aspx

http://www.eng.iastate.edu/efmd/cmultarray.html

Unit 5: Pointers

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of pointers
- Explain array of Pointers
- Discuss records and record structures
- Explain representation of records in memory, parallel arrays

Introduction

A pointer is nothing but a variable that contains an address of a location in memory. Pointers are used everywhere in C, and if you have a good understanding of them C should not pose a problem. C pointers are basically the same as Pascal pointers except they are used much more freely in C. A pointer is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address. Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That address points to a value. There is the pointer and the value pointed to. This fact can be a little confusing until you get used to it. In this unit, we will discuss the concept of pointers and array of pointers. We will also discuss the concept of record and record structures.

5.1 Concept of Pointers

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type. Simply stated, a pointer is an address. Instead of being a variable, it is a pointer to a variable stored somewhere in the address space of the program. It is always best to use an example, so examine the next program [POINTER.C] which has some pointers in it.

The following two rules are very important when using pointers an must be thoroughly understood. They may be somewhat confusing to you at first but we need to state the definitions before we can use them. Take your time, and the whole thing will clear up very quickly.

1. A variable name with an ampersand (&) in front of it defines the address of the variable and therefore points to the variable.
2. A pointer with a star in front of it refers to the value of the variable pointed to by the pointer

The ***unary*** or ***monadic*** operator **&** gives the "address of a variable".

The ***indirection*** or dereference operator ***** gives the "contents of an object ***pointed to*** by a pointer".

A pointer must be defined to point to some type of variable. Following a proper definition, it cannot be used to point to any other type of variable or it will result in a type incompatibility error. In the same manner that a float type of variable cannot be added to an int type variable, a pointer to a float variable cannot be used to point to an integer variable.

To declare a pointer to a variable do:

```
int  *pointer;
```

Caution We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

Consider the effect of the following code:

```
int x = 1, y = 2;
                int *ip;
                ip = &x;
y = *ip;
x = ip;
                *ip = 3;
```

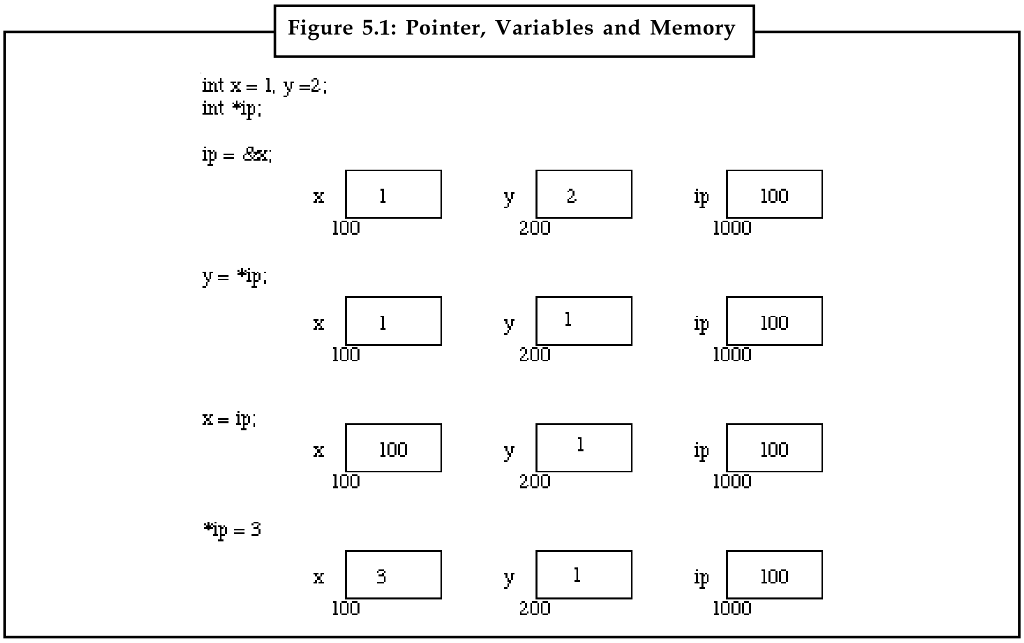
It is worth considering what is going on at the ***machine level*** in memory to fully understand how pointer work.

Notes A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is new.

Example: Consider Fig. 5.1. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000.

Notes

Figure 5.1: Pointer, Variables and Memory



```
int x = 1, y =2;
int *ip;

ip = &x;
```

```
y = *ip;
```

```
x = ip;
```

```
*ip = 3
```

Source: http://www.cs.cf.ac.uk/Dave/C/node10.html

Now the assignments x = 1 and y = 2 obviously load these values into the variables. ip is declared to be apointer to an integer and is assigned to the address of x (&x). So ip gets loaded with the value 100.

Next y gets assigned to the contents of ip. In this example ip currently points to memory location 100 — the location of x. So y gets assigned to the values of x — which is 1. We have already seen that C is not too fussy about assigning values of different type. Thus it is perfectly legal (although not all that common) to assign the current value of ip to x. The value of ip at this instant is 100.

Finally we can assign a value to the contents of a pointer (*ip).

When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

```
int *ip;
      *ip = 100;
```

will generate an error (program crash!!).

The correct use is:

```
int *ip;
               int x;
               ip = &x;
               *ip = 100;
```

We can do integer arithmetic on a pointer:

```
float *flp, *flq;
               *flp = *flp + 10;
               ++*flp;
               (*flp)++;
               flq = flp;
```

Notes

Did u know? A pointer to any variable type is an address in memory — which is an integer address. A pointer is definitely NOT an integer.

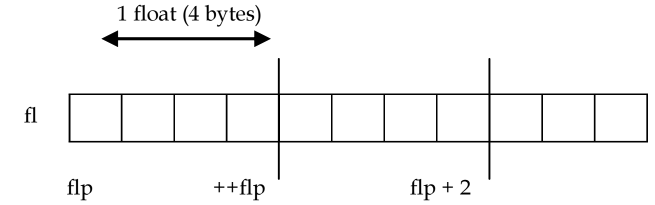
The reason we associate a pointer to a data type is so that it knows how many bytes the data is stored in. When we increment a pointer we increase the pointer by one "block" memory.

So for a character pointer ++ch_ptr adds 1 byte to the address.

For an integer or float ++ip or ++flp adds 4 bytes to the address.

Consider a float variable (fl) and a pointer to a float (flp) as shown in Fig 5.2.

Figure 5.2: Pointer Arithmetic



Source: http://www.cs.cf.ac.uk/Dave/C/node10.html

Assume that flp points to fl then if we increment the pointer (++flp) it moves to the position shown 4 bytes on. If on the other hand we added 2 to the pointer then it moves 2 float positions i.e 8 bytes as shown in the Figure.

Task Compare and contrast & operator and dereference operator.

5.1.1 Pointer and Functions

Let us now examine the close relationship between pointers and functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once to function has finished. Other languages do this (e.g. var parameters in PASCAL). C uses pointers explicitly to do this. Other languages mask the fact that pointers also underpin the implementation of this.

The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

The usual function call:

```
swap(a, b) WON'T WORK.
```

Pointers provide the solution: Pass the address of the variables to the functions and access address of function.

Notes

Thus our function call in our program would look like this:

```
swap(&a, &b)
```

The Code to swap is fairly straightforward:

```
void swap(int *px, int *py)
              { int temp;
                            temp = *px;
                            /* contents of pointer */

                            *px = *py;
                            *py = temp;
              }
```

We can return pointer from functions. A common example is when passing back structures.

Example:

```
typedef struct {float x,y,z;} COORD;
              main()
              { COORD p1, *coord_fn();
                                  /* declare fn to return ptr of
                                  COORD type */
                                          ....
                                  p1 = *coord_fn(...);
                            /* assign contents of address returned */
                                          ....
                            }
  COORD *coord_fn(...)
              { COORD p;
                                          .....
                                          p = ....;
                                          /* assign structure values */

                                          return &p;
                                          /* return address of p */
                            }
```

Here we return a pointer whose contents are immediately unwrapped into a variable. We must do this straight away as the variable we pointed to was local to a function that has now finished. This means that the address space is free and can be overwritten. It will not have been overwritten straight after the function has quit though so this is perfectly safe.

5.1.2 Pointers and Arrays

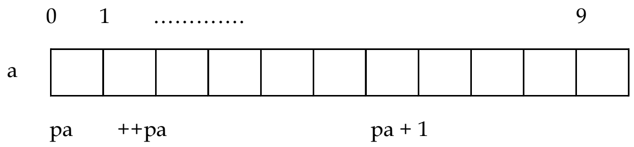
Pointers and arrays are very closely linked in C.

Consider of array elements arranged in consecutive memory locations.

Consider the following:

```
 int a[10], x;
                int *pa;

                pa = &a[0]; /* pa pointer to address of a[0] */

                x = *pa;
                /* x = contents of pa (a[0] in this case) */
```

Figure 5.3: Arrays and Pointers



Source: http://www.cs.cf.ac.uk/Dave/C/node10.html

To get somewhere in the array (Figure 5.3) using a pointer we could do:

pa + i ≡ a[i]

There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its link between arrays and pointers.

For example we can just type

```
pa = a;
```

instead of

```
pa = &a[0]
```

and

a[i] can be written as *(a + i).

i.e. `&a[i]   a + i.`

We also express pointer addressing like this:

```
pa[i]   *(pa + i).
```

However pointers and arrays are different:

A pointer is a variable. We can do pa = a and pa++.

An Array is not a variable. a = pa and a++ ARE ILLEGAL.

We can now understand how arrays are passed to functions. When an array is passed to a function what is actually passed is its initial elements location in memory. So:

```
strlen(s)  ≡  strlen(&s[0])
```

This is why we declare the function:

```
int strlen(char s[]);
```

Notes

An equivalent declaration is : int strlen(char *s); since char s[] ≡ char *s.

strlen() is a standard library function that returns the length of a string. Let's look at how we may write a function:

```
int strlen(char *s)
               { char *p = s;

                              while (*p != '\0);
                              p++;
                              return p-s;
               }
```

Now lets write a function to copy a string to another string. strcpy() is a standard library function that does this.

```
void strcpy(char *s, char *t)
               { while ( (*s++ = *t++) != '\0);}
```

This uses pointers and assignment by value.

Self Assessment Questions

Fill in the blanks:

1. A .... is a variable which contains the address in memory of another variable.
2. The .... operator * gives the "contents of an object ***pointed to*** by a pointer".
3. When we increment a pointer we increase the pointer by one "..................... " memory.
4. When an array is passed to a function what is actually passed is its initial elements location in ....
5. .... is a standard library function that returns the length of a string.

5.2 Arrays of Pointers

We can have arrays of pointers since pointers are variables.

An array of pointers can be declared as :

```
<type> *<name>[<number-of-elements];
```

Example:

```
char *ptr[3];
```

The above line declares an array of three character pointers.

Caution Text can't be moved or compared in a single operation.

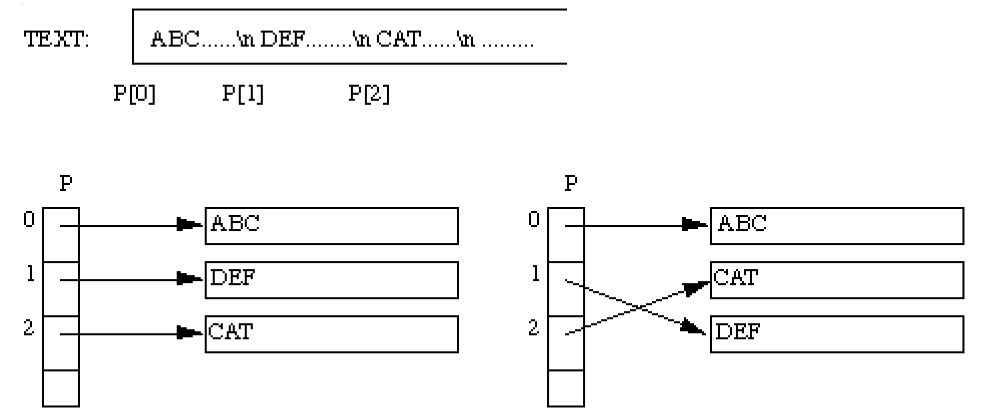
Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

How can we do this?. Let us discuss with the example given below.

Example:

- Store lines end-to-end in one big char array (Fig.5.4). n will delimit lines.
- Store pointers in a different array where each pointer points to 1st char of each new line.
- Compare two lines using strcmp() standard library function.
- If 2 lines are out of order — swap pointer in pointer array (not text).

Figure 5.4: Arrays of Pointers (String Sorting Example)



This eliminates:

- complicated storage management.
- high overheads of moving lines.

Example:

Lets take a working example :

```
#include<stdio.h>
int main(void)
{
 char *p1 = “Himanshu”;
 char *p2 = “Arora”;
 char *p3 = “India”;
 char *arr[3];
 arr[0] = p1;
 arr[1] = p2;
 arr[2] = p3;
 printf(“\n p1 = [%s] \n”,p1);
 printf(“\n p2 = [%s] \n”,p2);
 printf(“\n p3 = [%s] \n”,p3);
```

Notes

```
 printf("\n arr[0] = [%s] \n",arr[0]);
 printf("\n arr[1] = [%s] \n",arr[1]);
 printf("\n arr[2] = [%s] \n",arr[2]);
 return 0;
}
```

In the above code, we took three pointers pointing to three strings. Then we declared an array that can contain three pointers. We assigned the pointers 'p12, 'p22 and 'p32 to the 0,1 and 2 index of array. Let's see the output :

```
$ ./arrayofptr
 p1 = [Himanshu]
 p2 = [Arora]
 p3 = [India]
 arr[0] = [Himanshu]
 arr[1] = [Arora]
 arr[2] = [India]
```

So we see that array now holds the address of strings.

5.2.1 Multidimensional Arrays and Pointers

We should think of multidimensional arrays in a different way in C:

A 2D array is really a 1D array, each of whose elements is itself an array. Hence

```
 a[n][m] notation.
```

Array elements are stored row by row.

When we pass a 2D array to a function we must specify the number of columns — the number of rows is irrelevant.

The reason for this is pointers again. C needs to know how many columns in order that it can jump from row to row in memory.

Consider int a[5][35] to be passed in a function:

We can do:

```
 f(int a[][35]) {.....}
```

or even:

```
 f(int (*a)[35]) {.....}
```

We need parenthesis (*a) since [] have a higher precedence than *

So:

`int (*a)[35];` declares a pointer to an array of 35 ints.

`int *a[35];` declares an array of 35 pointers to ints.

Now lets look at the (subtle) difference between pointers and arrays. Strings are a common application of this.

Consider:

```
 char *name[10];
 char Aname[10][20];
```

We can legally do name[3][4] and Aname[3][4] in C.

However

- Aname is a true 200 element 2D char array.
- access elements via

  20*row + col + base_address

  in memory.
- name has 10 pointer elements.

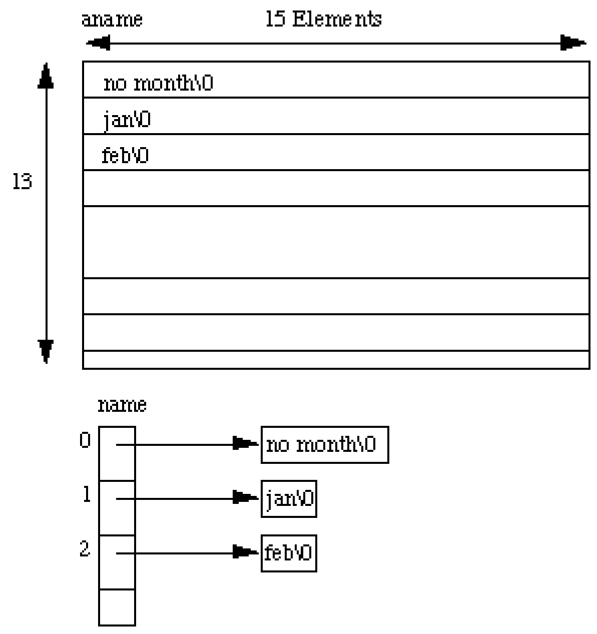
Did u know? If each pointer in name is set to point to a 20 element array then and only then will 200 chars be set aside (+ 10 elements).

The advantage of the latter is that each pointer can point to arrays be of different length.

Consider:

```
char *name[] = { "no month", "jan",
                     "feb", ... };
          char Aname[][15] = { "no month", "jan",
                     "feb", ... };
```

Figure 5.5: 2D Arrays and Arrays of Pointers



Notes

5.2.2 Static Initialisation of Pointer Arrays

Initialisation of arrays of pointers is an ideal application for an internal static array.

```
some_fn()
               { static char *months = { "no month",
                                          "jan", "feb",
                                          ...};
                            }
```

static reserves a private permanent bit of memory.

Task Write a C program to read through an array of any type using pointers.

Self Assessment Questions

Fill in the blanks:

6. .... are a data representation that will cope efficiently and conveniently with variable length text lines.

7. A 2D array is really a .... array, each of whose elements is itself an array.

8. C needs to know how many .... in order that it can jump from row to row in memory.

9. Initialisation of arrays of pointers is an ideal application for an internal .... array.

10. .... elements are stored row by row.

5.3 Records and Record Structures

As we studied already that the collections of data are frequently organized into a hierarchy of field, records and files. Specifically a record is a collection of related data items, each of which is called a field or attribute, and a file is a collection of similar records. Each data item itself may be a group item composed of sub items; those items which are indecomposable are called elementary items or atoms or scalars. The names given to the various data items are called identifiers.

As we know that a record is a collection of data items, it differs from a linear array in the following way,

(a) A record may be a collection of non-homogeneous data; i.e., the data items in a record may have different data types.

(b) The data items in a record are indexed by attribute names, so there may not be a natural ordering of its elements.

Under the relationship of group item to sub item, the data items in a record form a hierarchical structure which can be described by means of "level" numbers. Let we understand through the following examples.

Example:

Consider a hospital keeps a record on each newborn baby who contains the following data items: Name, Sex, Birthday, Father, and Mother. Assume that Birthday is a group item with sub

Notes

items Month, Day and Year, and Father and Mother are group items, each with sub items Name and Age.

Let we describe the structure of the above record. (Name appears three times and age appears twice in the structure.)

1 Newborn

2 Name

2 Sex

2 Birthday

3 Month

3 Day

3 Year

2 Father

3 Name

3 Age

2 Mother

3 Name

3 Age

The number to the left of each identifier is called a level number. Observe that each group item is followed by its sub items, and the level of the sub items is 1 more than the level of the group item. Furthermore, an item is a group if and only if it is immediately followed by an item with a greater level number.

Some of the identifiers in a record structure may also refer to arrays of elements. In fact, the first line of the above structure is replaced by

1 Newborn(20)

This will indicate a file of 20 records, and the usual subscript notation will be used to distinguish between different records in the file. That is, we will write to

Newborn1, Newborn2, Newborn3, . . .

or

Newborn[1], Newborn[2], Newborn[3],. . .

to denote different records in the file.

Example:

A class of student records may be organized as follows:

1 Student(20)

2 Name

3 Last

3 First

3 MI (Middle Initial)

Notes

2 Test(3)

2 Final

2 Grade

The identifier Student(20) indicates that there are 20 students. The identifier Test (3) indicates that there are three tests per student. Observe that there are 8 elementary items per Student, since test is counted three times. Altogether, there are 160 elementary items in the entire Student structure.

5.3.1 Indexing Items in a Record

Suppose we want to access some data item in a record. In some cases, we cannot simply write the data name of the item since the same name may appear in different places in the record.

> *Notes* Accordingly, in order to specify a particular item, we may have to qualify the name by using appropriate group item names in the structure. This qualification is indicated by using decimal points to separate group items from sub items.

Example:

(a) Consider the record structure Newborn in the example of newborn given above. Sex and year need no qualification, since each refers to a unique item in the structure. On the other hand, suppose we want to refer to the age of the father. This can be done by writing

Newborn.Father.Age   or simply   Father.Age

The first reference is said to be fully qualified. Sometimes one adds qualifying identifiers for clarity.

(b) Suppose the first line in the record structure in the example of newborn given above is replaced by

1 Newborn(20)

That is, Newborn is defined to be a file with 20 records. Then every item automatically becomes a 20-element array. Some languages allow the sex of the sixth newborn to be referenced by writing

Newborn.Sex[6]   or simple   Sex[6]

Analogously, the age of the father of the sixth newborn may be referenced by writing

Newborn. Father. Age[6]   or simply   Father.Age[6]

(c) Consider the record structure Student in the student example given above. Since Student is declared to be a file with 20 students, all items automatically become 20-element arrays. Furthermore, Test becomes a two-dimensional array. In particular, the second test of the sixth student may be referenced by writing

Student. Test[6, 2]   or simply   Test[6, 2]

The order of the subscripts corresponds to the order of the qualifying identifiers. For example,

Test[3,1]

does not refer to the third test of the first student, but to the first test of the third student.

5.3.2 Representation Of Records In Memory; Parallel Arrays

If the records may contain non-homogeneous data, then the elements of a record cannot be stored in an array. Some programming languages, such as PL/1, Pascal and COBOL, do have record structures built into the language.

Example:

Consider the record structure Newborn in example of newborn. One can store such a record in PL/1 following declaration, which defines a data aggregate called a structure:

```
DECLARE    1 NEWBORN,
                  2 NAME CHAR(20),
                  2 SEX CHAR(1),
                  2 BIRTHDAY,
                          3 MONTH FIXED,
                          3 DAY FIXED,
                          3 YEA R FIXED,
                  2 FATHER,
                          3 NAME CHAR(20),
                          3 AGE FIXED,
                  2 MOTHER
                          3 NAME CHAR(20),
                          3 AGE FIXED;
```

Let we observe that the variables SEX and YEAR are unique; hence references to them need not be qualified. On the other hand, AGE is not unique. Accordingly, we should use like that

FATHER.AGE   or   MOTHER.AGE

when we want to reference the father's age or the mother's age.

If a programming language does not have available the hierarchical structures that are label inPL/1, Pascal and COBOL. Assuming the record contains nonhomogeneous data, the record may have to be stored in individual variables, one for each of its elementary data items.

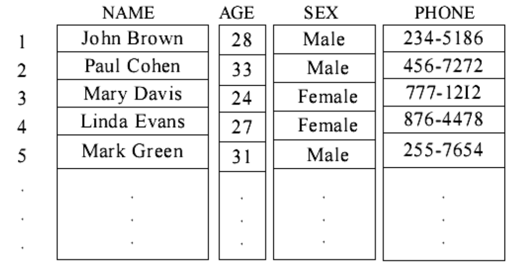
On the other hand, if we want to store an entire file of records, note that all data elements belonging to the same identifier do have the same type. Such a file may be stored in memory as a collection of parallel arrays; that is, where elements in the different arrays with the same subscript belong to the same record. To understand this parallel arrays, let we go through the following example.

Example:

Consider a membership list contains the name, age, sex and telephone number of each member. We can store the file in four parallel arrays, NAME, AGE, SEX and PHONE, as pictured in Figure 5.6 ; that is, for a given subscript K, the elements NAME[K], AGE[K], SEX[K] and PHONE[K] belong to the same record.

Notes

Figure 5.6: Four parallel arrays, NAME, AGE, SEX and PHONE



	NAME	AGE	SEX	PHONE
1	John Brown	28	Male	234-5186
2	Paul Cohen	33	Male	456-7272
3	Mary Davis	24	Female	777-1212
4	Linda Evans	27	Female	876-4478
5	Mark Green	31	Male	255-7654
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

Example:

Consider again the Newborn record in previous example of newborn. One can store a file of such records in nine linear arrays, such as

NAME, SEX, MONTH, DAY, YEAR, FATHERNAME, FATHERAGE, MOTHERNAME, MOTHERAGE

one array is for each elementary data item. Here we must use different variablenames for the name and age of the father and mother, which was not necessary in the previous example. Again, we assume that the arrays are parallel, i.e., that for a fixed subscript K, the elements

NAME[K], SEX[K], MONTH[K],….., MOTHERAGE[K]

belong to the same record.

Records with Variable Lengths

Consider an elementary school keeps a record for each student which contains the following data: Name, Telephone Number, Father, Mother, Siblings. Here Father, Mother, Siblings contain, respectively, the names of the student's father, mother, and brothers or sisters attending the same school. Three such records may be as follows.

Adams, John;	345-6677;	Richard;	Mary;	Jame,William,Donald
Bailey, Susan;	222-1234;	XXXX;	Sheela;	XXXX
Sami, Mohammed;	567-3344;	Abdul;	Fathima;	Aliya

Here, XXXX means that the parent has died or is not living with the student, or that the student has no sibling at the school. From the above records, we could learn that they are in variable-lengths, since the data element Siblings can contain zero or more names.

Self Assessment Questions

Fill in the blanks:

11. A .... is a collection of related data items, each of which is called a field or attribute.

12. The names given to the various data items are called ....

13. The data items in a record are indexed by .... names.

14. If the records may contain .... data, then the elements of a record cannot be stored in an array.

15. In ...., elements in the different arrays with the same subscript belong to the same record.

5.4 Summary

- A pointer is a variable which contains the address in memory of another variable.
- A pointer must be defined to point to some type of variable. It cannot be used to point to any other type of variable or it will result in a type incompatibility error.
- When an array is passed to a function what is actually passed is its initial elements location in memory.
- Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.
- When we pass a 2D array to a function we must specify the number of columns — the number of rows is irrelevant.
- A record is a collection of related data items, each of which is called a field or attribute, and a file is a collection of similar records.
- In order to specify a particular item, we may have to qualify the name by using appropriate group item names in the structure.
- In parallel arrays, elements in the different arrays with the same subscript belong to the same record.

5.5 Keywords

Pointer: A pointer is a variable which contains the address in memory of another variable.

& operator: The unary or monadic operator & gives the "address of a variable".

Dereference operator : The indirection or dereference operator * gives the "contents of an object pointed to by a pointer".

strlen(): strlen() is a standard library function that returns the length of a string.

Arrays of Pointers : Arrays of Pointers are a data representation that will cope efficiently and conveniently with variable length text lines.

Strings: Strings are defined as arrays of characters.

Record: A record is a collection of related data items, each of which is called a field or attribute, and a file is a collection of similar records.

Parallel arrays: In parallel arrays, elements in the different arrays with the same subscript belong to the same record.

5.6 Review Questions

1. Illustrate the concept of pointers with example.
2. Describe the relationship between pointers and functions with example.
3. Explain how arrays are passed to functions.
4. What do you mean by array of pointers? Discuss its declaration with example.
5. Illustrate the Static Initialisation of Pointer Arrays.

Notes

6. What is a record. Differentiate it with a linear array.
7. Discuss the concept of Indexing Items in a Record with example.
8. Describe the representation of records in memory.
9. Discuss the concept of parallel arrays with example.
10. Make distinction between pointers and arrays.

Answers: Self Assessment

1. pointer
2. *indirection* or dereference
3. block
4. Memory
5. strlen()
6. Arrays of Pointers
7. 1D
8. Columns
9. static
10. Array
11. record
12. Identifiers
13. attribute
14. non-homogeneous
15. parallel arrays

5.7 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.stanford.edu/~fringer/teaching/operating_systems_03/handouts/lecture9.pdf

http://www.cs.cornell.edu/courses/CS2022/2011sp/lectures/lect04.pdf

http://euklid.mi.uni-koeln.de/c/mirror/mickey.lcsc.edu/%257Esteve/c9.html

http://www.cprogramming.com/tutorial/c/lesson6.html

Unit 6: Operations on Arrays and Sparse Matrices

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss operations on arrays
- Explain the concept of sparse matrices
- Discuss storage of sparse matrices

Introduction

An array in C Programing Language can be defined as number of memory locations, each of which can store the same data type and which can be references through the same variable name.

An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, number of chairs in home, or salaries of 300 employees or ages of 25 students. Thus an array is a collection of similar elements. These similar elements could be all integers or all floats or all characters etc. Usually, the array of characters is called a "string", where as an array of integers or floats is called simply an array. In this unit, we will discuss operations on arrays. Also, we will discuss the concept of sparse matrices.

6.1 Operations on Array

The array is a homogeneous structure, i.e. the elements of an array are of the same type. It is infinite; it has a specified number of elements. An array is ordered; there is an ordering of elements in it as zeroth, first, second etc. Following set of operations are defined for this structure.

- Creating an array
- Initialising an array
- Storing an element
- Retrieving an element
- Inserting an element

Notes

- Deleting an element
- Merging arrays

Let us now write a function that deletes an element from an array. The function in the listing given below takes the name of the array(which is actually a pointer to an int), the index of the element to be removed and the index of the last element as arguments.

Caution A function which deletes an element removes the element and returns the index of the last element.

A function to delete an element from an array is given below.

```
int delete_element(int *list, int last_index, int index)
{
int i;
for(i = index; i < last_index; i ++)
list [i]=list [i+1];
return (last_index-1);
}
```

Let us now see how to insert an element in an array. Array insertion does not mean increasing size of array.

Example: consider an array a[10] having three elements in it initially and a[0] = 1, a[1] = 2 and a[2] = 3 and you want to insert a number 45 at location 1 i.e. a[0] = 45, so we have to move elements one step below so after insertion a[1] = 1 which was a[0] initially, and a[2] = 2 and a[3] = 3.

A program to insert an element in an array is given as below:

```
#include <stdio.h>

int main()
{
int array[100], position, c, n, value;

printf("Enter number of elements in array\n");
scanf("%d", &n);
printf("Enter %d elements\n", n);
for (c = 0; c < n; c++)
scanf("%d", &array[c]);
printf("Enter the location where you wish to insert an element\n");
scanf("%d", &position);

printf("Enter the value to insert\n");
scanf("%d", &value);
for (c = n - 1; c >= position - 1; c--)
array[c+1] = array[c];

array[position-1] = value;

printf("Resultant array is\n");

for (c = 0; c <= n; c++)
printf("%d\n", array[c]);

return 0;
}
```

Now we will see how to merge two arrays in C. This is shown in the following program.

Example: In this example, there are four functions, that is, one for reading the array element, second for writing on console and third for sorting of both array and last for merge of two array into one.

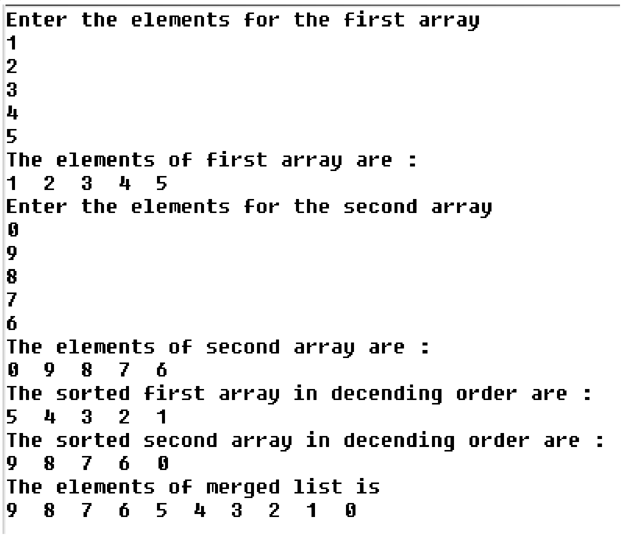
```
#include<stdio.h>
#include<conio.h>
void main()
{
 void read(int *,int);
 void display(int *,int);
 void sort(int *,int);
 void mergelist(int *,int *,int *,int);
 int a[5],b[5],c[10];
 clrscr();
 printf("Enter the elements for the first array \n");
 read(a,5);
 printf("The elements of first array are : \n");
 display(a,5);
 printf("Enter the elements for the second array \n");
 read(b,5);
 printf("The elements of second array are : \n");
 display(b,5);
 sort(a,5);
 printf("The sorted first array in decending order are :\n");
 display(a,5);
 sort(b,5);
      printf("The sorted second array in decending order are :\n");
 display(b,5);
 mergelist(a,b,c,5);
```

Notes

```
 printf(“The elements of merged list is \n”);
 display(c,10);
 getch();
 }
 void read(int c[],int i)
 {
     int j;
  for(j=0;j<i;j++)
     scanf(“%d”,&c[j]);
 }
 void display(int d[],int i)
 {
  int j;
     for(j=0;j<i;j++)
     printf(“%d “,d[j]);
     printf(“\n”);
 }
void sort(int arr[] ,int k)
 {
 int temp;
 int i,j;
 for(i=0;i<k;i++)
  {
     for(j=0;j<k-i-1;j++)
      {
      if(arr[j]<arr[j+1])
       {
          temp=arr[j];
          arr[j]=arr[j+1];
          arr[j+1]=temp;
       }
      }
  }
  }
 void mergelist(int a[],int b[],int c[],int k)
 {
     int ptra=0,ptrb=0,ptrc=0;
     while(ptra<k && ptrb<k)
      {
          if(a[ptra] < b[ptrb])
           {
            c[ptrc]=b[ptrb];
```

```
                        ptrb++;
                    }
            else
                    {
                      c[ptrc]=a[ptra];
                     ptra++;
                    }
              ptrc++;
              }
        while(ptra<k)
              {
                      c[ptrc]=a[ptra];
                      ptra++;ptrc++;
                    }
        while(ptrb<k)
              {
                      c[ptrc]=b[ptrb];
                      ptrb++;  ptrc++;

                    }
   }
```

The output is shown as below:



```
Enter the elements for the first array
1
2
3
4
5
The elements of first array are :
1  2  3  4  5
Enter the elements for the second array
0
9
8
7
6
The elements of second array are :
0  9  8  7  6
The sorted first array in decending order are :
5  4  3  2  1
The sorted second array in decending order are :
9  8  7  6  0
The elements of merged list is
9  8  7  6  5  4  3  2  1  0
```

Tasks Write a program that creates an array consisting of elements 3,4, 5, 6 and 7; removes 4 from the array; and inserts 11 after 7 in the array.

Notes

Self Assessment Questions

State True or False:

1. A function which deletes an element removes the element and returns the index of the last element.

2. Array insertion mean increasing size of array.

6.2 Sparse Matrices and their Storage

Matrices with good number of zero entries are called sparse matrices. A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements. Sparse matrix is very useful in engineering field, when solving the partial differentiation equations.

Figure 6.1: (a) Triangular Matrix (b) Tridiagonal Matrix

$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix}$$

(a)

$$\begin{pmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & -7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(b)

A triangular matrix is a square matrix in which all the elements either above or below the main diagonal are zero. Triangular matrices are sparse matrices. A tridiagonal matrix is a square matrix in which all the elements except for the main diagonal, diagonals on the immediate upper and lower side are zeroes. Tridiagonal matrices are also sparse matrices.

Let us consider a sparse matrix from storage point of view. Suppose that the entire sparse matrix is stored. Then, a considerable amount of memory which stores the matrix consists of zeroes. This is nothing but wastage of memory. In real life applications, such wastage may count to megabytes. So, an efficient method of storing sparse matrices has to be looked into.

Example: Figure 6.2 shows a sparse matrix of order 7 × 6.

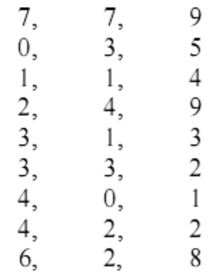
Figure 6.2: Representation of a sparse matrix of order 7 × 6

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	5	0	0
1	0	4	0	0	0	0
2	0	0	0	0	9	0
3	0	3	0	2	0	0
4	1	0	2	0	0	0
5	0	0	0	0	0	0
6	0	0	8	0	0	0

A common way of representing non zero elements of a sparse matrix is the 3-tuple form. The first row of sparse matrix always specifies the number of rows, number of columns and number of non zero elements in the matrix. In the example given above, the number 7 represents the total number of rows sparse matrix. Similarly, the number 6 represents the total number of columns in the matrix. The number 8 represents the total number of non zero elements in the matrix. Each non zero element is stored from the second row, with the 1st and 2nd elements of the row, indicating the row number and columnnumber respectively in which the element is present in the original matrix. The 3rdelement in this row stores the actual value of the non zero element.

Example: the 3- tuple representation of the matrix of Figure 6.2 is shown in Figure 6.3.

Figure 6.3: 3-tuple representation of Figure 6.2



7,	7,	9
0,	3,	5
1,	1,	4
2,	4,	9
3,	1,	3
3,	3,	2
4,	0,	1
4,	2,	2
6,	2,	8

The following program accepts a matrix as input, which is sparse and prints the corresponding 3-tuple representations.

The following program accepts a matrix as input and prints the 3-tuple representation of it.

```
#include<stdio.h>
void main()
{
  int a[5][5],rows,columns,i,j;
printf("enter the order of the matrix. The order should be less than 5 ×
5:\n");
  scanf("%d %d",&rows,&columns);
printf("Enter the elements of the matrix:\n");
for(i=0;i<rows;i++)
  for(j=0;j<columns;j++)
  { scanf("%d",&a[i][j]);
  }
  printf("The 3-tuple representation of the matrix is:\n");
for(i=0;i<rows;i++)
  for(j=0;j<columns;j++)
  {
  if (a[i][j]!=0)
  {
```

Notes

```
printf("%d  %d  %d\n",  (i+1),(j+1),a[i][j]);
  }
  }
}
```

Output:

```
enter the order of the matrix. The order should be less than 5 × 5:
          3        3
Enter the elements of the matrix:
          1        2        3
          0        1        0
          0        0        4
The 3-tuple representation of the matrix is:
          1        1        1
          1        2        2
          1        3        3
          2        2        1
          3        3        4
```

The program initially prompted for the order of the input matrix with a warning that the order should not be greater than 5 × 5. After accepting the order, it prompts for the elements of the matrix. After accepting the matrix, it checks each element of the matrix for a non zero. If the element is non zero, then it prints the row number and column number of that element along with its value.

> *Task* Compare and contrast triangular matrix and tridiagonal matrix.

6.2.1 Sparse Matrix Storage Formats

The efficiency of most of the iterative methods is determined primarily by the performance of the matrix-vector product and therefore on the storage scheme used for the matrix.

> *Did u know?* Often, the storage scheme used arises naturally from the specific application problem.

There are many methods for storing the data such as compressed row and column storage, block compressed row storage, diagonal storage, jagged diagonal storage, and skyline storage. These are discussed as below.

Compressed Row Storage

The compressed row and column storage formats are the most general: they make absolutely no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary

Notes

elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioner solve.

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix A, we create three vectors: one for floating point numbers (val) and the other two for integers (col_ind, row_ptr). The val vector stores the values of the nonzero elements of the matrix A as they are traversed in a row-wise fashion. The col_ind vector stores the column indexes of the elements in the val vector. That is, if val(k) = $a_{i,j}$, then col_ind(k) = j. The row_ptr vector stores the locations in the val vector that start a row; that is, if val(k) = $a_{i,j}$, then row_ptr(i) ≤ k < row_ptr(i + 1). By convention, we define row_ptr(n + 1) = nnz + 1, where nnz is the number of nonzeros in the matrix A. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only 2nnz + n + 1 storage locations.

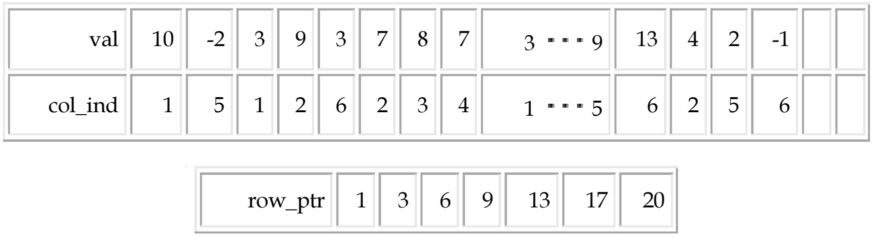
As an example, consider the nonsymmetric matrix A defined by

Figure 6.4: nonsymmetric Matrix A

$$A = \begin{array}{cccccc} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{array}$$

The CRS format for this matrix is then specified by the arrays {val, col_ind, row_ptr} given below:

Figure 6.5: CRS format for matrix A given in figure 6.4



val	10	-2	3	9	3	7	8	7	3 ··· 9	13	4	2	-1		
col_ind	1	5	1	2	6	2	3	4	1 ··· 5	6	2	5	6		

row_ptr	1	3	6	9	13	17	20

If the matrix A is symmetric, we need only store the upper (or lower) triangular portion of the matrix.

Notes The tradeoff is a more complicated algorithm with a somewhat different pattern of data access.

Compressed Column Storage

Analogous to CRS, there is compressed column storage (CCS), which is also called the Harwell-Boeing sparse matrix format. The CCS format is identical to the CRS format except that the columns of are stored (traversed) instead of the rows. In other words, the CCS format is the CRS format for A^T.

The CCS format is specified by the 3 arrays {val, row_ind, col_ptr}, where row_ind stores the row indices of each nonzero, and col_ptr stores the index of the elements in val which start a column of A.

Example: The CCS format for the matrix A is given by

Figure 6.6: CCS format for the matrix given in figure 6.4

val	10	3	3	9	7	8	4	8	8 ▪▪▪ 9	2	3	13	-1		
row_ind	1	2	4	2	3	5	6	3	4 ▪▪▪ 5	6	2	5	6		

col_ptr	1	4	8	10	13	17	20

Block Compressed Row Storage

If the sparse matrix A is composed of square dense blocks of nonzeros in some regular pattern, we can modify the CRS (or CCS) format to exploit such block patterns. Block matrices typically arise from the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point. We then partition the matrix in small blocks with a size equal to the number of degrees of freedom and treat each block as a dense matrix, even though it may have some zeros.

If n_b is the dimension of each block and *nnzb* is the number of nonzero blocks in the $n \times n$ matrix A, then the total storage needed is nnz = nnzb × n_b^2 . The block dimension n_d of A is then defined by $n_d = n/n_b$.

Similar to the CRS format, we require three arrays for the BCRS format: a rectangular array for floating point numbers (val(1 : *nnzb*, 1 : n_b 1 : $n_{b,}$)) which stores the nonzero blocks in (block) row-wise fashion, an integer array (col_ind(1 : *nnzb*)) which stores the actual column indices in the original matrix A of the (1, 1) elements of the nonzero blocks, and a pointer array (row_blk(1 : n_d + 1)) whose entries point to the beginning of each block row in val(:,:,:) and col_ind(:).

Caution The savings in storage locations and reduction in time spent doing indirect addressing for block compressed row storage (BCRS) over CRS can be significant for matrices with a large n_b.

Compressed Diagonal Storage

If the matrix A is banded with bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage scheme by storing subdiagonals of the matrix in consecutive locations. Not only can we eliminate the vector identifying the column and row, but we can pack the nonzero elements in such a way as to make the matrix-vector product more efficient. This storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.

Notes

We say that the matrix $A = (a_{i,j})$ is banded if there are nonnegative constants p, q, called the left and right halfbandwidth, such that $a_{i,j} \neq 0$ only if $i - p \leq j \leq i + q$. In this case, we can allocate for the matrix an array val(1:n,-p:q). The declaration with reversed dimensions (-p:q,n) corresponds to the LINPACK band format, which, unlike compressed diagonal storage (CDS), does not allow for an efficiently vectorizable matrix-vector multiplication if $p + q$ is small.

Usually, band formats involve storing some zeros. The CDS format may even contain some array elements that do not correspond to matrix elements at all.

Example: Consider the nonsymmetric matrix A defined by

Figure 6.7: Nonsymmetric Matrix

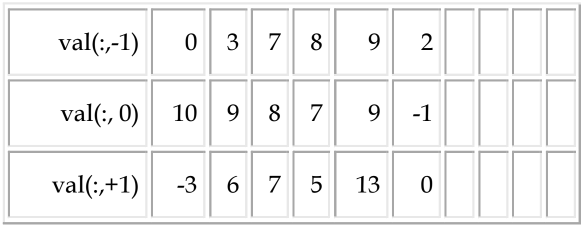
$$A = \begin{matrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{matrix}$$

Using the CDS format, we store this matrix A in an array of dimension (6,-1:1) using the mapping

$$\text{val}\ (i, j) = a_{i,i+j}$$

Hence, the rows of the val(:,:) array are

Figure 6.8: Rows of the val(:,:) array



val(:,-1)	0	3	7	8	9	2				
val(:, 0)	10	9	8	7	9	-1				
val(:,+1)	-3	6	7	5	13	0				

Notice the two zeros corresponding to nonexisting matrix elements.

Jagged Diagonal Storage

The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors. Like the CDS format, it gives a vector length of essentially the same size as the matrix.

Did u know? It is more space-efficient than CDS at the cost of a gather/scatter operation.

A simplified form of JDS, called ITPACK storage or Purdue storage, can be described as follows.

Example: For the following nonsymmetric matrix, all elements are shifted left after which the columns are stored consecutively.

Figure 6.9: Elements are shifted left

$$\begin{array}{cccccc} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{array} \rightarrow \begin{array}{cccc} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{array}$$

All rows are padded with zeros on the right to give them equal length. Corresponding to the array of matrix elements val(:,:), an array of column indices,col_ind(:,:), is also stored:

Figure 6.10: an array of column indices,col_ind(:,:)

val(:, 1)	10	9	3	6	9	5
val(:, 2)	–3	6	8	7	13	–1
val(:, 3)	1	–2	7	5	0	0
val(:, 4)	0	0	0	4	0	0

col_ind(:, 1)	1	2	1	2	5	5
col_ind(:, 2)	2	3	3	4	6	6
col_ind(:, 3)	4	5	4	5	0	0
col_ind(:, 4)	0	0	0	6	0	0

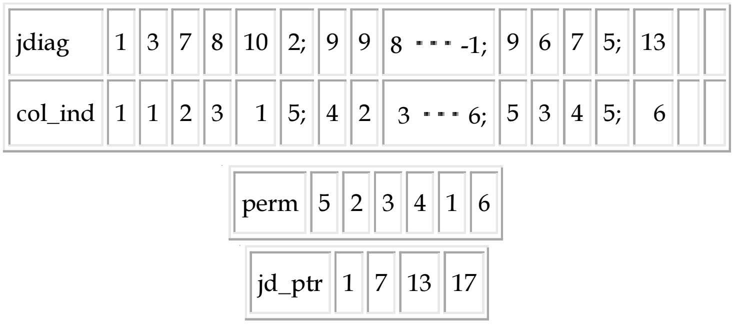
It is clear that the padding zeros in this structure may be a disadvantage, especially if the bandwidth of the matrix varies strongly. Therefore, in the CRS format, we reorder the rows of the matrix decreasingly according to the number of nonzeros per row. The compressed and permuted diagonals are then stored in a linear array. The new data structure is called jagged diagonals.

Specifically, we store the (dense) vector of all the first elements in val, col_ind from each row, together with an integer vector containing the column indices of the corresponding elements. This is followed by the second jagged diagonal consisting of the elements in the second positions from the left. We continue to construct more and more of these jagged diagonals (whose length decreases).

The number of jagged diagonals is equal to the number of nonzeros in the first row, i.e., the largest number of nonzeros in any row of A. The data structure to represent the $n \times n$ matrix A therefore consists of a permutation array (perm(1:n)), which reorders the rows, a floating point array (jdiag(:)) containing the jagged diagonals in succession, an integer array (col_ind(:)) containing the corresponding column indices, and finally a pointer array (jd_ptr(:)) whose elements point to the beginning of each jagged diagonal.

The JDS format for the above matrix in using the linear arrays {perm, jdiag, col_ind, jd_ptr} is given below (jagged diagonals are separated by semicolons):

Figure 6.11: JDS format for the matrix A given in figure 6.7



jdiag	1	3	7	8	10	2;	9	9	8 ··· -1;	9	6	7	5;	13		
col_ind	1	1	2	3	1	5;	4	2	3 ··· 6;	5	3	4	5;	6		

perm	5	2	3	4	1	6

jd_ptr	1	7	13	17

Skyline Storage

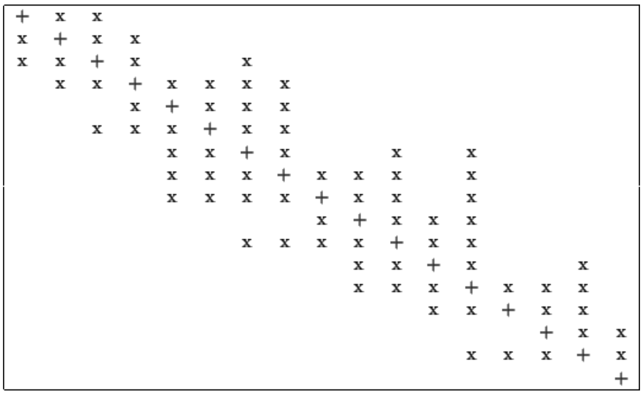
The final storage scheme we consider is for skyline matrices, which are also called variable band or profile matrices. It is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods.

Notes A major advantage of solving linear systems having skyline coefficient matrices is that when pivoting is not necessary, the skyline structure is preserved during Gaussian elimination.

If the matrix is symmetric, we only store its lower triangular part. A straightforward approach in storing the elements of a skyline matrix is to place all the rows (in order) into a floating point array (val(:)), and then keep an integer array (row_ptr(:)) whose elements point to the beginning of each row. The column indices of the nonzeros stored in val(:) are easily derived and are not stored.

For a nonsymmetric skyline matrix such as the one illustrated in Figure 6.12, we store the lower triangular elements in skyline storage (SKS) format and store the upper triangular elements in a column-oriented SKS format (transpose stored in row-wise SKS format). These two separated substructures can be linked in a variety of ways. One approach is to store each row of the lower triangular part and each column of the upper triangular part contiguously into the floating point array (val(:)). An additional pointer is then needed to determine where the diagonal elements, which separate the lower triangular elements from the upper triangular elements, are located.

Figure 6.12: Profile of a nonsymmetric skyline or variable-band matrix.



Self Assessment Questions

Fill in the blanks:

1. Matrices with good number of zero entries are called sparse matrices.
2. A .... is a square matrix in which all the elements either above or below the main diagonal are zero.
3. A .... matrix is a square matrix in which all the elements except for the main diagonal, diagonals on the immediate upper and lower side are zeroes.
4. The .... format puts the subsequent nonzeros of the matrix rows in contiguous memory locations.
5. The compressed column storage (CCS) is also known as the .... sparse matrix format.
6. .... matrices typically arise from the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point.
7. The .... format can be useful for the implementation of iterative methods on parallel and vector processors.
8. .... storage is considered as a simplified form of JDS.
9. The number of jagged diagonals is equal to the number of .... in the first row.
10. .... matrices is mostly of importance in direct solution methods.
11. The CCS format is identical to the CRS format except that the .... of are stored (traversed) instead of the rows.
12. Like the CDS format, JD format gives a .... length of essentially the same size as the matrix.
13. A common way of representing non zero elements of a sparse matrix is the .... form.

Case Study

Case:

Write a C Program to delete duplicate elements in an array.

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[20],i,j,k,n;
clrscr();
printf("nEnter array size : ");
scanf("%d",&n);
printf("nAccept Numbers : ",n);
for(i=0;i<n;i++)
```

Contd...

```
  scanf("%d",&a[i]);
clrscr();
printf("nOriginal array is : ");
for(i=0;i<n;i++)
  printf(" %d",a[i]);
printf("nUpdated array is : ");
for(i=0;i<n;i++)
{
   for(j=i+1;j<n;)
  {
     if(a[j]==a[i])
    {
        for(k=j;k<n;k++)
          a[k]=a[k+1];
       n--;
    }
    else
       j++;
  }
}
for(i=0;i<n;i++)
   printf("%d ",a[i]);
getch();
}
```

Output:

```
Enter array size : 5
Accept Numbers : 1 2 2 3 4
Original array is : 1 2 2 3 4
Updated array is : 1 2 3 4
```

Question

Write a program to retrieve an element from an array.

Source: http://www.c4learn.com/c-programs/to-delete-duplicate-elements-in-array.html

6.3 Summary

- Operations on arrays include Creating an array initializing an array, inserting an element, deleting an element, merging arrays, etc.
- A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements.
- A triangular matrix is a square matrix in which all the elements either above or below the main diagonal are zero. Triangular matrices are sparse matrices.

Notes

- The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations.
- Compressed column storage (CCS), also called the Harwell-Boeing sparse matrix format is identical to the CRS format except that the columns of are stored (traversed) instead of the rows.
- Block matrices typically arise from the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point.
- Compressed diagonal storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.
- The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors.
- Skyline matrices, also called variable band or profile matrices is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods.

6.4 Keywords

Sparse matrices: A sparse matrix is a matrix that allows special techniques to take advantage of the large number of zero elements.

***Triangular matrices**:* A triangular matrix is a square matrix in which all the elements either above or below the main diagonal are zero.

CRS: The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations.

CCS: Compressed column storage (CCS), also called the Harwell-Boeing sparse matrix format is identical to the CRS format except that the columns of are stored (traversed) instead of the rows.

Block matrices: Block matrices typically arise from the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point.

CDS: Compressed diagonal storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.

JDS: The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors.

SKS: Skyline matrices, also called variable band or profile matrices is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods.

6.5 Review Questions

1. Illustrate with example how to delete an element form an arrays.
2. Elucidate the insert operation on arrays with example.
3. What is a sparse matrix? Discuss its usage with example.
4. Illustrate the concept of representing non zero elements of a sparse matrix.
5. Discuss various storage schemes used for the matrix.
6. Make distinction between Compressed Row Storage and Compressed Column Storage.

7. Explain the concept of Compressed Diagonal Storage.
8. Describe the simplified form of JDS with example.
9. Discuss the importance of profile matrices.
10. CDS storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid. Comment.

Answers: Self Assessment

1. True
2. False
3. sparse matrices
4. triangular matrix
5. tridiagonal
6. compressed row storage (CRS)
7. Harwell-Boeing
8. Block
9. jagged diagonal storage (JDS)
10. ITPACK
11. nonzeros
12. Profile
13. columns
14. Vector
15. 3-tuple

6.6 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.c-program-example.com/2012/01/c-program-to-generate-sparse-matrix.html

http://www.microchip.com/forums/m563069-print.aspx

http://www.c4learn.com/c-programs/c-program-to-implement-stack-operations-using-array.html

http://www.gnu.org/software/octave/doc/interpreter/Creating-Sparse-Matrices.html

Notes

Unit 7: Linked Lists

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of linked lists
- Explain the representation of linked list in memory
- Discuss different types of Linked Lists

Introduction

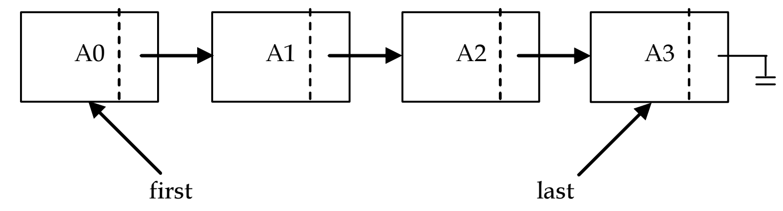
Static arrays are structures whose size is fixed at compile time and therefore cannot be extended or reduced to fit the data set. A dynamic array can be extended by doubling the size but there is overhead associated with the operation of copying old data and freeing the memory associated with the old data structure. One potential problem of using arrays for storing data is that arrays require a contiguous block of memory which may not be available, if the requested contiguous block is too large. However the advantages of using arrays are that each element in the array can be accessed very efficiently using an index. However, for applications that can be better managed without using contiguous memory we define a concept called “linked lists”.

Notes

7.1 Concept of Linked Lists

A linked list is a collection of objects linked together by references from one object to another object. By convention these objects are named as nodes. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node. The last node points to a NULL reference to indicate the end of the list.

Figure 7.1: Linked list



Source: http://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf

The entry point into a linked list is always the first or head of the list.

Did u know? Head is NOT a separate node, but a reference to the first Node in the list.

If the list is empty, then the head has the value NULL. Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be contiguous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

Linked lists have advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time.

Caution To create an array we must allocate memory for a certain number of elements.

To add more elements to the array then we must create a new array and copy the old array into the new array. This can take lot of time.

We can prevent this by allocating lots of space initially but then you might allocate more than you need and wasting memory. With a linked list we can start with space for just one element allocated and *add* on new elements easily without the need to do any copying and reallocating.

There are a number of issues in linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes 0(1) to access any element in the array. Linked lists takes 0(n) for access to an element in the list in the worst case. Another advantage of arrays in access time is special *locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods. Although the dynamic allocation of storage is a great advantage, the overhead with storing and retrieving data can make a big difference. Sometimes linked lists are hard to manipulate. If the last item is deleted, the last but one must now have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference Finally, linked lists wastes memory in terms of extra reference points.

7.1.1 Designing the Node of a Linked List

Linked list is a collection of linked nodes. A node is a struct with at least a data field and a reference to a node of the same type. A node is called a self-referential object, since it contains a pointer to a variable that refers to a variable of the same type.

Example: a struct Node that contains an int data field and a pointer to another node can be defined as follows.

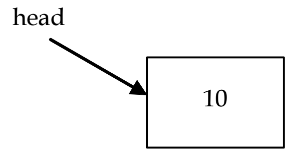
```
typedef struct node {
  int data;
  struct node* next;
} node;
node* head = NULL;
```

7.1.2 Creating the First Node

Memory must be allocated for one node and assigned to head as follows.

```
head = (node*) malloc(sizeof(node));
head→data = 10;
head→next = NULL;
```

Figure 7.2: Creating the First Node



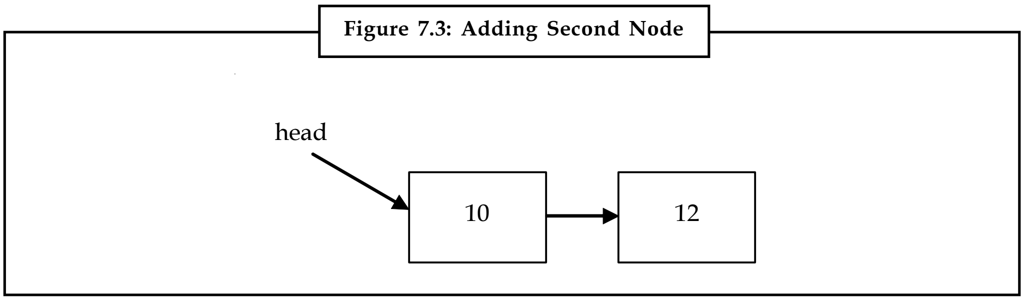
Source: http://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf

7.1.3 Adding the Second Node and Linking

The second node is added in the following way:

```
node* nextnode = malloc(sizeof(node));
nextnodeàdata = 12;
nextnodeànext = NULL;
headànext = nextnode;
```

Figure 7.3: Adding Second Node



Source: http://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf

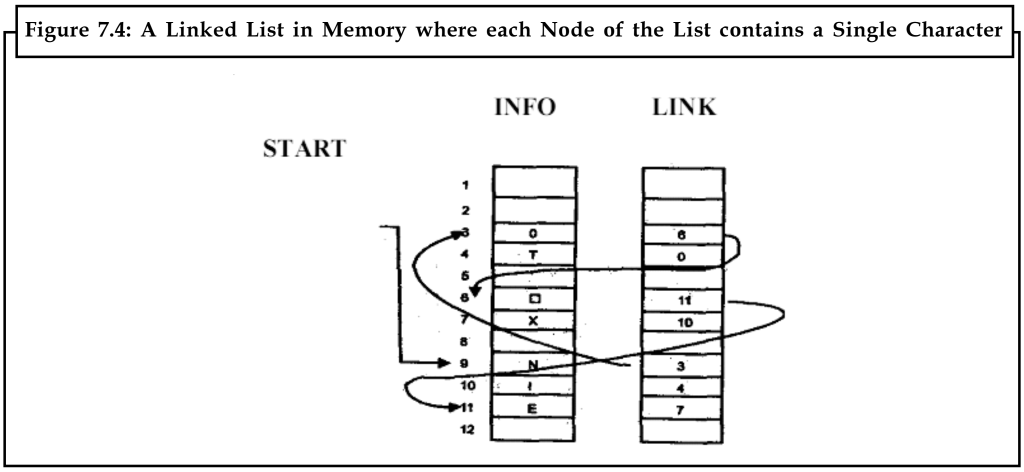
7.1.4 Representation of Linked Lists in Memory

Let LIST be a linked list. Then LIST will be maintained in memory as follows. First of all, LIST requires two linear arrays-we will call them here INFO and LINK - such that INFO[K] and LINK[K] contain the information part and the next pointer field of a node of LIST respectively. START contains the location of the beginning of the list, and a next pointer sentinel-denoted by NULL-which indicates the end of the list.

The following examples of linked lists indicate that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.

Example: Figure 7.4 pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

Figure 7.4: A Linked List in Memory where each Node of the List contains a Single Character



Source: http://www.csbdu.in/econtent/DataStructures/Unit1-DS.pdf

```
START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second character.
LINK[3] = 6, so 1INFO[6] = (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7] = X is the fifth character.
LINK[7] = 10, so INFO[10] = I is the sixth character.
```

Notes

```
LINK[10] = 4, so INFO[4] = T is the seventh character.
LINK[4] = 0, so the NULL value, so the list has ended.
```

Self Assessment Questions

State True or False:

1. A linked list is a collection of objects linked together by references from one object to another object.
2. The first node points to a NULL reference to indicate the end of the list.
3. If the list is empty, then the head has the value NULL.
4. If the last item is deleted, the last but one must now have its pointer changed to hold a NULL reference.
5. A node is a struct with at least a data field and a reference to a node of different types.
6. Linked list requires two linear arrays.

7.2 Types of Linked Lists

Linked lists are widely used in many applications because of the flexibility it provides. Unlike arrays that are dynamically assigned, linked lists do not require memory from a contiguous block. This makes it very appealing to store data in a linked list, when the data set is large or device (eg: PDA) has limited memory. One of the disadvantages of linked lists is that they are not random accessed like arrays. To find information in a linked list one must start from the head of the list and traverse the list sequentially until it finds (or not find) the node. Another advantage of linked lists over arrays is that when a node is inserted or deleted, there is no need to "adjust" the array.

There are few different types of linked lists.

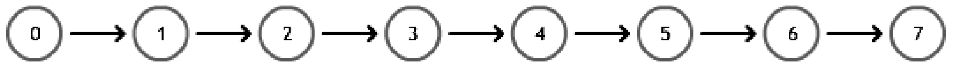
7.2.1 Singly Linked List

Generally "linked list" means a singly linked list. This list consists of a number of nodes in which each node has a next pointer to the following element.

Did u know? The link of the last node in the list is NULL which indicates end of the list.

A singly linked list as described above provides access to the list from the head node. Traversal is allowed only one way and there is no going back.

Figure 7.5: Singly linked list



Let us write a C program that create a singly linked list.

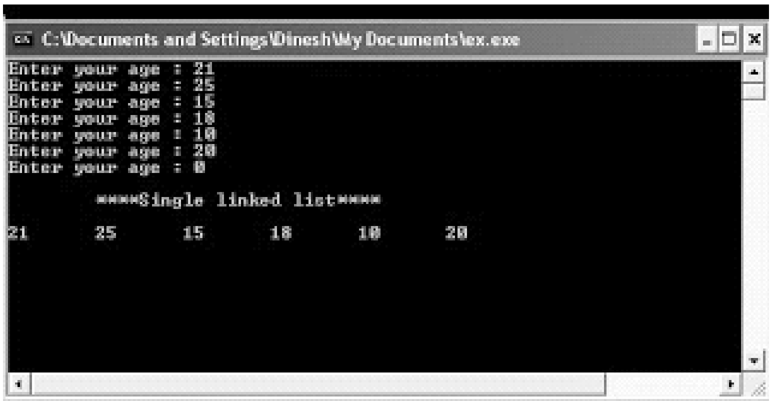
```
/*c program for creating singly linked list*/
#include<stdio.h>
```

```
#include<conio.h>
struct single_link_list
{
   int age;
   struct single_link_list *next;
};
typedef struct single_link_list node;

node *makenode(int );
int main()
{
   int ag;
   node *start,*last,*nn;    //nn=new node
   start=NULL;
   while(1)
   {
       printf(“Enter your age : “);
       scanf(“%d”,&ag);
       if(ag==0)
          break;
       nn=makenode(ag);
       if(start==NULL)
       {
          start = nn;
          last = nn;
       }
       else
       {
          last->next = nn;
          last = nn;
       }
   }
   printf(“\n\t****Single linked list****\n\n”);
   for(; start!=NULL; start=start->next)
        printf(“%d\t”,start->age);
   getch();
   return 0;
}
/*creation of node*/

node *makenode(int tmp)
{
```

Notes

```
    node *nn;
    nn = (node *)malloc(sizeof(node));
    nn->age = tmp;
    nn->next = NULL;
    return nn;
}
```

It shows the following output:



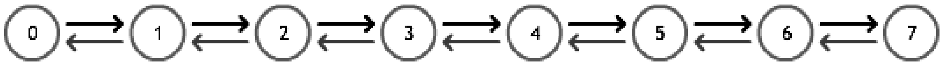
C:\Documents and Settings\Dinesh\My Documents\ex.exe

```
Enter your age : 21
Enter your age : 25
Enter your age : 15
Enter your age : 18
Enter your age : 10
Enter your age : 20
Enter your age : 0

         ****Single linked list****

21      25      15      18      10      20
```

7.2.2 Doubly Linked List

If a node also stores a reference to the previous node it can move in both directions, forth and back; this is obviously called a doubly linked list, which is shown below. A doubly linked list is a list that has two references, one to the next node and another to previous node. Doubly linked list also starts from head node, but provide access both ways. That is one can traverse forward or backward from any node.

Figure 7.6:Doubly linked list



Notes In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

The *advantage* of a doubly linked list (also called *two — way linked list)* is given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in doubly linked list we can delete a node even if we don't have previous nodes address (since, each node has left pointer pointing to previous node and can move backward).

The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations)

A generic doubly linked list node can be designed as:

```
typedef struct node {
 void* data;
 struct node* next;
 struct node* prev;
} node;
node* head = (node*) malloc(sizeof(node));
```

The design of the node allows flexibility of storing any data type as the linked list data.

Example:

```
Head→ data = malloc(sizeof(int)); head→ data = 12;
```

or

```
head→ data = malloc(strlen("guna")+1);strcpy(head→ data, "guna");
```

One operation that can be performed on doubly linked list is to delete a given node pointed by 'p'. Function for this operation is given below:

```
       delete(struct node *p)
       {
              if (p==Null) print f("Node Not Found")
              else
              {
                     p→llink →rlink=p→llink;
                     p→rlink→llink= p→llink;
                     free(p);
              }
       }
```

Task Analyze the applications of doubly linked lists.

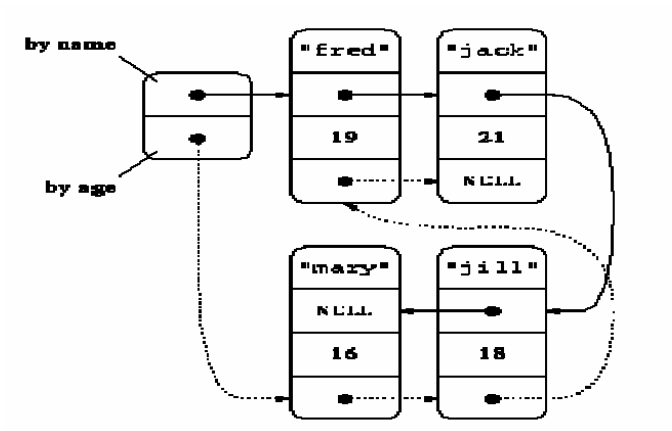
7.2.3 Multilinked List

A multilinked list is a more general linked list with multiple links from nodes.

For examples, we can define a Node that has two references, age pointer and a name pointer. With this structure it is possible to maintain a single list, where if we follow the name pointer we can traverse the list in alphabetical order of names and if we traverse the age pointer, we can traverse the list sorted by ages. This type of node organization may be useful for maintaining a customer list in a bank where same list can be traversed in any order (name, age, or any other criteria) based on the need.

Notes

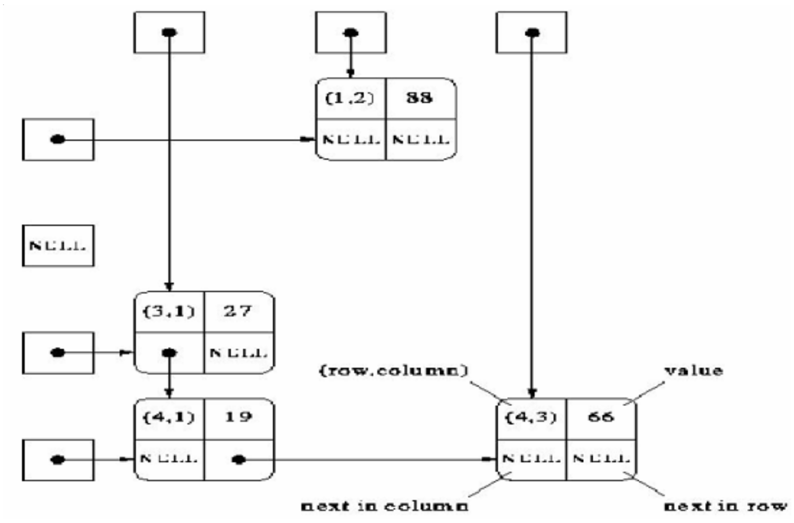
Figure 7.7: Linked List with two pointers



Source: http://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf

Example: Another example of multilinked list is a structure that represents a sparse matrix as shown below. Sparse matrices are widely used in applications. A sparse matrix is a large table of data where most are undefined. For example, we may have a 10^6 x 10^6 table of integers. If we just allocate an array to hold this table, we would require 4 x 10^6 x 10^6 bytes of memory. This is a very large chunk of contiguous memory that most computers don't have. Using a table where most entries are undefined to store this matrix is a bad idea. Therefore we come up with a data structure where only store the defined values. A typical record in the structure below consists of row and column index of the entry, value of the entry, a pointer to next row and a pointer to next column.

Figure 7.8: A Sparse Matrix Representation



Source: http://www.andrew.cmu.edu/user/rmemon/121-n11/assignment3/writeup.html

Notes

Task Compare and contrast doubly linked lists and multilinked lists.

7.2.4 Circular Linked List

Another important type of a linked list is called a circular linked list where last node of the list points back to the first node (or the head) of the list. Circular linked list is a more complicated linked data structure. In singly linked lists and doubly linked lists the end of lists are indicated with NULL value. But circular linked lists do not have ends.

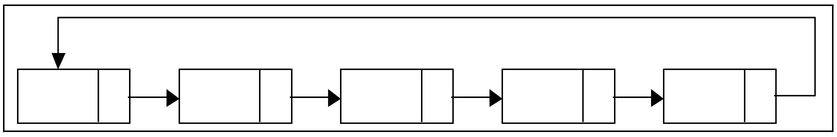
Caution While traversing the circular linked lists we should be careful otherwise we will be traversing the list infinitely.

In circular linked lists each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful.

Example: when several processes are using the same computer resource (CPU) for the same amount of time, and we have to assure that no process accesses the resource before all other processes did (round robin algorithm).

The elements can be placed anywhere in the heap memory unlike array which uses contiguous locations. Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node points back to the first node unlike singly linked list. It has a dynamic size, which can be determined only at run time.

Figure 7.9: Circular lists



The advantage is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time.

Notes For implementing queues we will only need one pointer namely tail, to locate both head and tail.

The disadvantage is that the algorithms have become more complicated.

Following is a type declaration for a circular linked list of integers:

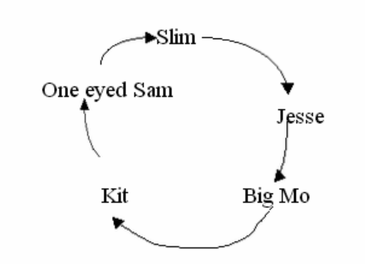
```
Typedef struct CLLNode {
 int data;
 struct ListNode *next;
}
```

Notes

Application of Circular Linked List

A famous problem that uses a circularly linked list is the Josephus problem. This problem has to do with selective deletion from a circularly linked list. There is a group of bandits in the Wild West who find themselves in a desperate predicament. The group is surrounded by the sheriff's posse and has no hope for mass escape. There is just one horse left, In order to select who shall take the horse and ride off with the booty, then do the following. Numbers are written on slips of paper and shuffled in a hat. The bandits stand in a circle. One is designed to be in the starting position.

Figure 7.10: Josephus Problem



A number, say n, is drawn from the hat and count begins around the circle (clockwise); the nth person is eliminated. He steps out of the circle; the circle's boundary is tightened. The count begins again with the man to his left being number1. Again the nth person is eliminated. The process continues until only one bandit remains. He grabs the goods, jumps on the horse and rides off into the sunset.

Assume the initial circle of bandits is as shown in figure and Jesse is selected number1 (he is the biggest). The number 4 is drawn , and One-eyed-Sam is the first bandit to be eliminated. The count begins again with Slim and Kit leaves the circle. Again the count begins with Slim, and the number 4 falls on Slim. Jesse becomes first, and Big Mo loses as number 4. Jesse grabs the horse.

You should write a program that simulates this procedure. Inputs should be the list of names of bandits, ordered by their positions in the circle, and the drawn number n. Output the names of the bandits as they are eliminated and the name of the winner. The obvious choice of data structure for this problem is a circularly linked list.

Self Assessment Questions

Fill in the blanks:

7. A .... linked list consists of a number of nodes in which each node has a next pointer to the following element.

8. To find information in a linked list one must start from the .... of the list and traverse the list sequentially until it finds (or not find) the node.

9. A .... linked list is a list that has two references, one to the next node and another to previous node.

10. The design of the node allows flexibility of storing any .... as the linked list data.

11. A .... list is a more general linked list with multiple links from nodes.

Notes

12. A .... is a large table of data where most are undefined.
13. In .... linked list, last node of the list points back to the first node (or the head) of the list.
14. In singly linked lists and doubly linked lists the end of lists are indicated with .... value.
15. A typical .... consists of row and column index of the entry, value of the entry, a pointer to next row and a pointer to next column.

Case Study

Case:

Write a C Program To Implement Single Linked List

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
void create();
void insert();
void delete();
void display();
struct  node
{
int data;
struct node *link;
};
struct node *first=NULL,*last=NULL,*next,*prev,*cur;
void create()
{
   cur=(struct node*)malloc(sizeof(struct node));
   printf("\nENTER THE DATA: ");
   scanf("%d",&cur->data);
   cur->link=NULL;
   first=cur;
   last=cur;
}
void insert()
{
   int pos,c=1;
   cur=(struct node*)malloc(sizeof(struct node));
   printf("\nENTER THE DATA: ");
   scanf("%d",&cur->data);
   printf("\nENTER THE POSITION: ");
```

Contd...

Notes

```
   scanf("%d",&pos);
   if((pos==1)  &&(first!=NULL))
  {
    cur->link = first;
    first=cur;
  }
  else
  {
     next=first;
     while(c<pos)
    {
     prev=next;
      next=prev->link;
    c++;
    }
     if(prev==NULL)
    {
      printf("\nINVALID POSITION\n");
    }
    else
    {
      cur->link=prev->link;
     prev->link=cur;
    }
 }
}
void delet()
{
int pos,c=1;
printf("\nENTER THE POSITION : ");
scanf("%d",&pos);
if(first==NULL)
{
printf("\nLIST IS EMPTY\n");
}
else if(pos==1 && first->link==NULL)
{
printf("\n DELETED ELEMENT IS %d\n",first->data);
free(first);
first=NULL;
}
```

Contd...

```
else if(pos==1 && first->link!=NULL)
{
cur=first;
first=first->link;
cur->link=NULL;
printf("\n DELETED ELEMENT IS %d\n",cur->data);
free(cur);
}
else
{
next=first;
while(c<pos)
{
cur=next;
next=next->link;
c++;
}
cur->link=next->link;
next->link=NULL;
if(next==NULL)
{
printf("\nINVALID POSITION\n");
}
else
{
printf("\n DELETED ELEMENT IS %d\n",next->data);
free(next);
}
}
}
void display()
{
   cur=first;
   while(cur!=NULL)
  {
      printf("\n %d",cur->data);
      cur=cur->link;
  }
}
void main()
{
```

Contd...

Notes

```
int ch;
clrscr();
printf("\n\nSINGLY LINKED LIST");
do
{
printf("\n\n1.CREATE\n2.INSERT\n3.DELETE\n4.EXIT");
printf("\n\nENTER YOUR CHOICE : ");
scanf("%d",&ch);
switch(ch)
{
case 1:
     create();
     display();
     break;
case 2:
     insert();
     display();
     break;
case 3:
     delet();
     display();
     break;
case 4:
     exit(0);
default:
    printf("Invalid choice...");
}
}while(1);
}
```

Sample Input and Output:

```
Singly Linked List
1.Create
2.Insert
3.Delete
4.Exit
Enter Your Choice : 1
Enter the Data: 10
 10
1.Create
```

Contd...

Notes

```
2.Insert
3.Delete
4.Exit
Enter Your Choice : 2
Enter the Data: 30
Enter the Position: 1
 30
 10
1.Create
2.Insert
3.Delete
4.Exit
Enter Your Choice : 3
Enter the Position : 2
List is Empty
```

Question

Write a program to create a linked list & display the elements in the list.

Source: http://enggedu.com/lab_exercise/data_structure_lab/C_program_to_implement_single_linked_list.php

7.3 Summary

- A linked list is a collection of objects linked together by references from one object to another object.
- We must begin from the head of the list and traverse the list sequentially to access the nodes in the list.
- A node is a struct with at least a data field and a reference to a node of the same type.
- A singly linked list consists of a number of nodes in which each node has a next pointer to the following element.
- A doubly linked list is a list that has two references, one to the next node and another to previous node.
- A multilinked list is a more general linked list with multiple links from nodes.
- A circular linked list is a list where last node of the list points back to the first node (or the head) of the list.
- A famous problem that uses a circularly linked list is the Josephus problem. This problem has to do with selective deletion from a circularly linked list.

7.4 Keywords

Linked List: A linked list is a collection of objects linked together by references from one object to another object.

Node: A node is a struct with at least a data field and a reference to a node of the same type.

Singly linked list: A singly linked list consists of a number of nodes in which each node has a next pointer to the following element.

Doubly linked list: A doubly linked list is a list that has two references, one to the next node and another to previous node.

Multilinked list: A multilinked list is a more general linked list with multiple links from nodes.

Circular linked list: A circular linked list is a list where last node of the list points back to the first node (or the head) of the list.

Sparse Matrix: A sparse matrix is a large table of data where most are undefined.

Malloc: The function malloc is used to allocate a certain amount of memory during the execution of a program.

7.5 Review Questions

1. Explain the concept of linked list with example.
2. Discuss the advantages and disadvantages of linked lists.
3. Discuss the various issues associated with linked lists.
4. Illustrate how to design the node of a linked list.
5. Describe the representation of linked lists in memory with example.
6. What is a singly linked list? Illustrate how to create a singly linked list.
7. Explain doubly linked list with example.
8. Describe the concept of Linked List with two pointers.
9. In circular linked lists each node has a successor. Comment.
10. Discuss the application of Circular Linked List.

Answers: Self Assessment

1. True
2. False
3. True
4. True
5. False
6. True
7. singly
8. Head
9. doubly
10. data type
11. multilinked
12. sparse matrix
13. circular
14. NULL
15. record

Notes

7.6 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://programmingexamples.wikidot.com/c-linked-lists

http://123techguide.blogspot.in/2012/02/create-linked-list-in-c.html#axzz2SUfoEZsc

http://www.martinbroadhurst.com/articles/circular-linked-list.html

http://www.c.happycodings.com/Data_Structures/code7.html

Notes

Unit 8: Operations on Linked List

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss basic Operations on a Singly Linked List
- Discuss basic Operations on Doubly Linked Lists
- Discuss basic Operations on Circular Linked Lists

Introduction

Dynamic allocation is specially appropriate for building lists, trees and graphs. We used an array for storing a collection of data items. Suppose the collection itself grows and shrinks then using a linked list is appropriate. It allows both insertion, deletion, search. But the random access capabilities of array is lost. Linked list is a collection of data item, where each item is stored in a structure (node). Linked list is accessed by accessing its first node. Subsequent nodes can be accessed by using the pointer next So after declaring, a (empty) list is initialized to NULL. Creating a list is done by creating nodes one after another. In this unit, we will discuss various operations used on linked lists.

Notes

8.1 Basic Operations on a Singly Linked List

The basic operations on a singly linked list include:

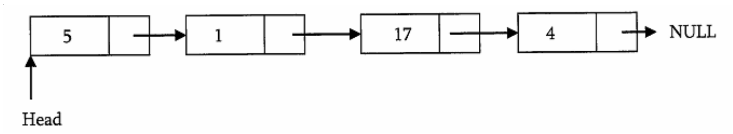
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

8.1.1 Traversing the Linked List

Let us assume that the head points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.

Figure 8.1: Traversing the linked lists



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring: dsame_chap3.pdf

The ListLength() function takes a linked list as input and counts the number of nodes in the list.

Example: below function can be used for printing the list data with extra print function.

```
int ListLength(stnict ListNode *head) f struct ListNode *current = head;
int count = 0;
```

8.1.2 Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

Notes To insert an element in the linked list at some position p, assume that after inserting the element the position of this new node is p.

Inserting a Node in Singly Linked List at the Beginning

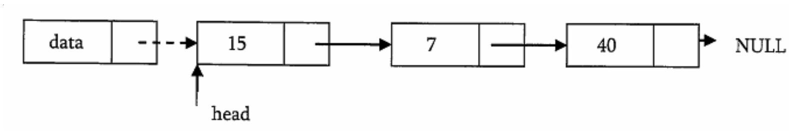
In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps.

Notes

Example: The example given below shows the steps for inserting a Node in Singly Linked List at the Beginning.

- Update the next pointer of new node, to point to the current head. New node

Figure 8.2: Update the next pointer of new node

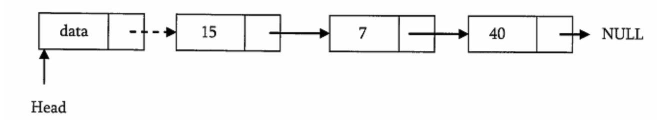


Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update head pointer to point to the new node.

New node

Figure 8.3: Update head pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

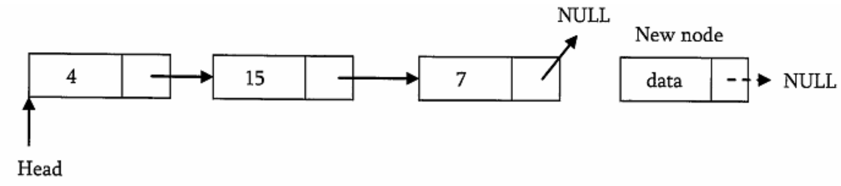
Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

Example: The example given below shows the steps for Inserting a Node in Singly Linked List at the Ending.

- New nodes next pointer points to NULL.

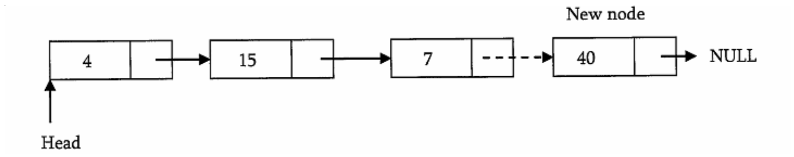
Figure 8.4: New nodes next pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Last nodes next pointer points to the new node.

Figure 8.5: Last Nodes Next Pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

8.1.3 Singly Linked List Deletion

Here also we have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

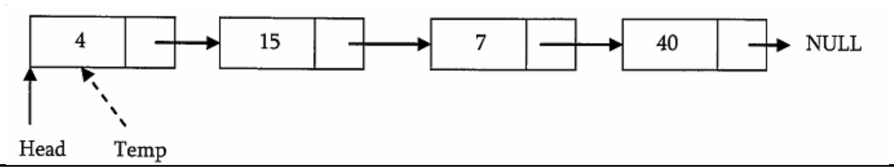
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps.

Example: The example given below shows the steps for deleting the first Node in Singly Linked List.

- Create a temporary node which will point to same node as that of head.

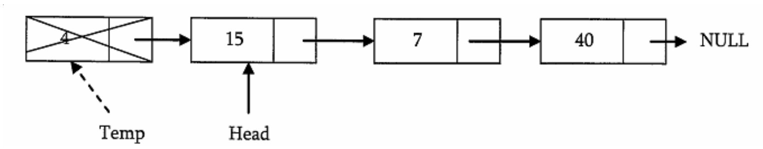
Figure 8.6: Create a Temporary Node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Now, move the head nodes pointer to the next node and dispose the temporary node.

Figure 8.7: move the head nodes pointer to the next node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

Notes

Deleting the last node in Singly Linked List

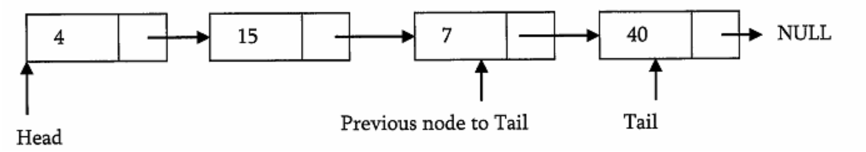
In this case, last node is removed from the list. This operation is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps.

Example: The example given below shows the steps for deleting the last node in Singly Linked List:

- Traverse the list and while traversing maintain the previous node address also.

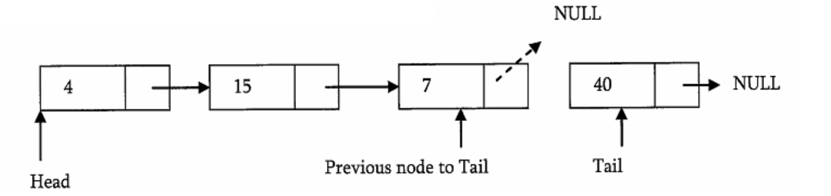
Did u know? By the time we reach the end of list, we will have two pointers one pointing to the tail node and other pointing to the node before tail node.

Figure 8.8: Traverse the list



- Update previous nodes next pointer with NULL.

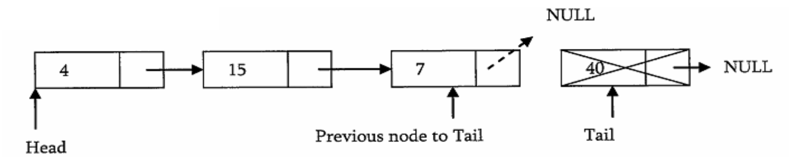
Figure 9.9: Update previous nodes



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Dispose the tail node.

Figure 8.10: Disposing the tail node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

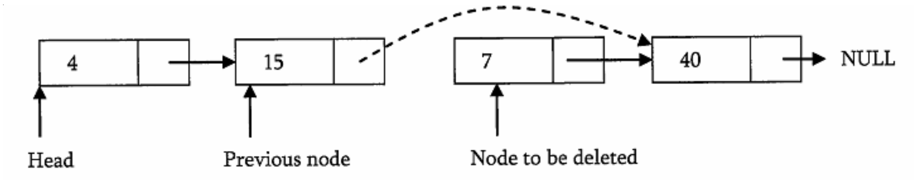
Deleting an Intermediate Node in Singly Linked List

In this case, node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps.

Example: The example given below shows the steps for deleting an Intermediate Node in Singly Linked List.

- As similar to previous case, maintain previous node while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to next pointer of the node to be deleted.

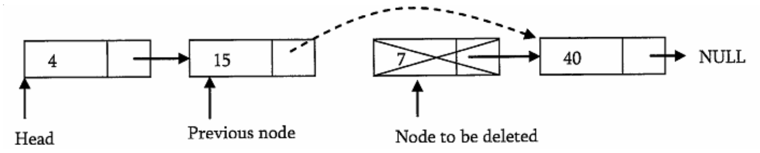
Figure 8.11: change the previous nodes next pointer to next pointer of the node to be deleted



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Dispose the current node to be deleted.

Figure 8.12: Dispose the current node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

The implementation is shown as below.

```
void DeleteNodeFromLinkedList (struct ListNode **head, int position) (
int k = 1;

struct ListNode *p, *q;

if(*head == NULL) {

printf (“List Empty”); return;

}

p = *head;

if(position == 1) {

p = *head;

*head = *head—>nlext;

free (p); return;

}

else [ //Traverse the list until the position from which we want to
delete while ((p != NULL) && (k < position - 1)) [

k++; q = p;

p = p—n.lext;

}
```

Notes

```
if(p — NULL)  /* At the end */
printf ("Position does not exist");
else { /* From the middle */
q→next = p→next;
free(p);
}
}
}
```

Deleting Singly Linked List

This works by storing the current node in some temporary variable and freeing the current node. After freeing the current node go to next node with temporary variable and repeat this process for all nodes.

```
Void DeleteLinkedList(struct ListNode **head) { struct ListNode
*auxilaryNode, *iterator; iterator = *head;
while (iterator) {
auxilaryNode = iterator>next;
free(iterator);
iterator = auxilaryNode;
}
*head = NULL; // to affect the real head back in the caller.

}
```

Self Assessment Questions

Fill in the blanks:

1. The .... function takes a linked list as input and counts the number of nodes in the list.
2. When Inserting a Node in Singly Linked List at the ...., a new node is inserted before the current head node.
3. Deleting the .... in Singly Linked List is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first.
4. When Deleting an .... Node in Singly Linked List, node to be removed is always located between two nodes.
5. Deleting Singly Linked List works by storing the .... node in some temporary variable and freeing the current node.

8.2 Basic Operations on Doubly Linked Lists

The Basic Operations on Doubly Linked Lists are discussed below:

8.2.1 Doubly Linked List Insertion

Insertion into a doubly-linked list has three cases (same as singly linked list):

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

Inserting a Node in Doubly Linked List at the Beginning

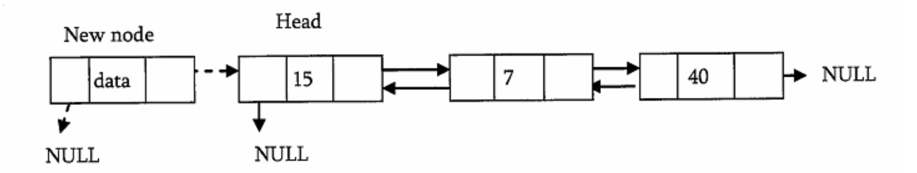
In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps.

Example: The example given below shows the steps for inserting a Node in Doubly Linked List at the Beginning.

- Update the right pointer of new node to point to the current head node and also make left pointer of new node as NULL.

Caution The current head node is shown by the dotted link in the figure given below:

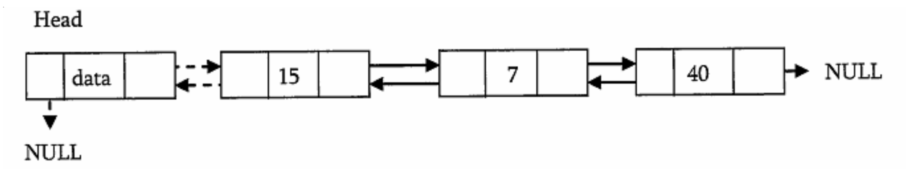
Figure 8.13: Update the right pointer of new node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update head nodes left pointer to point to the new node and make new node as head.

Figure 8.14: Update head nodes left pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

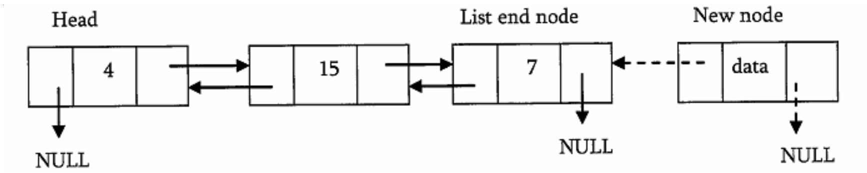
Notes

Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.

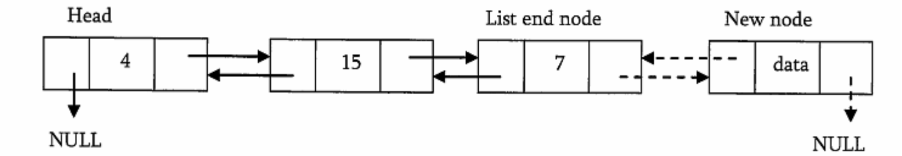
Figure 8.15: New node right pointer points to NULL



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update right of pointer of last node to point to new node.

Figure 8.16: Update right of pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

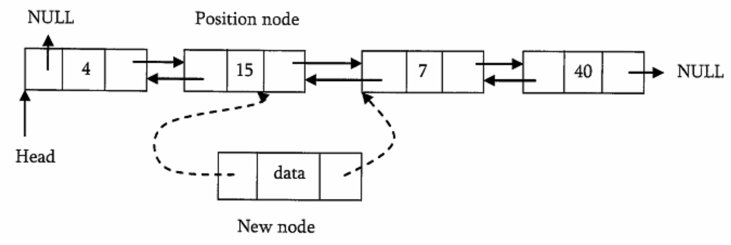
Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list till the position node and insert the new node.

- New node right pointer points to the next node of the position node where we want to insert the new node.

Did u know? New node left pointer points to the position node.

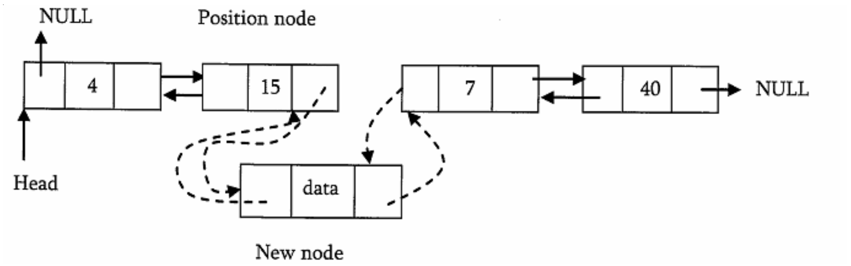
Figure 8.17: New node left pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Position node right pointer points to the new node and the next node of position nodes left pointer points to new node.

Figure 8.18: Position node right pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring: dsame_chap3.pdf

Now, let us write the code for all these three cases.

⚠ *Caution* We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send double pointer.

The following code inserts a node in the doubly linked list.

```
void DLLInsert(struct DLLNode **head, int data, int position) {int k =
1;
struct DLLNode *temp, *newNode;
newNode = (struct DLLNode *) malloc(sizeof ( struct DLLNode ));
 if(!newNode) {     //Always check for memory errors
printf (“Memory Error”);
return;
}
newNode—>data = data;
if(position == 1) { //Inserting a node at the beginning
newNode—>next = *head;
newNode—>Prev = NULL;
*head—> Prev = newNode;
*head = newNode;
return;
}
temp = *head;
while ( (k < position — 1) && temp—>next!=NULL){
temp = temp—>next;
k++;
 }
if( temp—>next== NULL)  //Insertion at the end
newNode —>next = temp—>next;
newNode—>prev = temp;
temp—>next = newNode;
}
```

Notes

```
else {        //Insertion in the middle
newNode—>next = temp—>next;
newNode—>prev = temp;
temp—>next—>prev = newNode;
temp—>next = newNode;
}
return;
}
```

8.2.2 Doubly Linked List Deletion

As similar to singly linked list deletion, here also we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

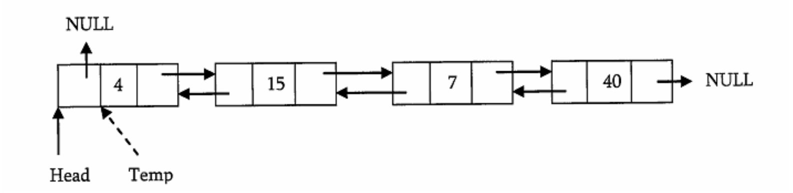
Deleting the First Node in Doubly Linked List

In this case, first node (current head node) is removed from the list. It can be done in two steps.

Example: The example given below shows the steps for deleting the First Node in Doubly Linked List.

- Create a temporary node which will point to same node as that of head.

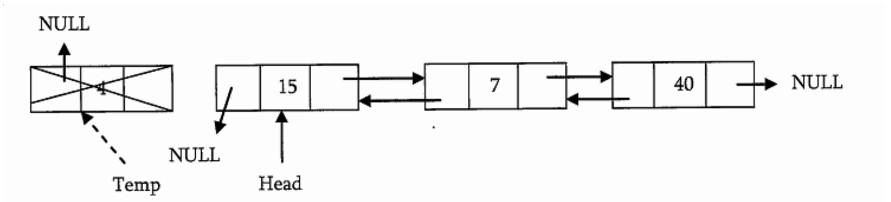
Figure 8.19: Create a temporary node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose the temporary node.

Figure 8.20: move the head nodes pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

Deleting the Last Node in Doubly Linked List

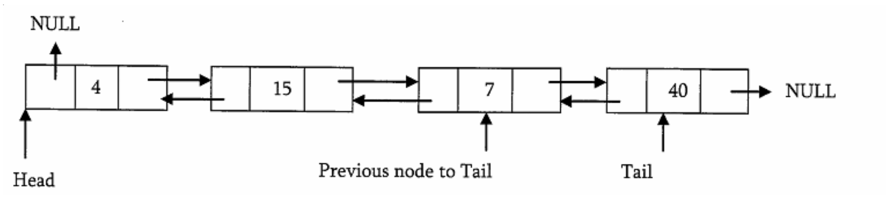
Notes

This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps.

Example: The example given below shows the steps for deleting the Last Node in Doubly Linked List.

- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the NULL (tail) and other pointing to the node before tail node.

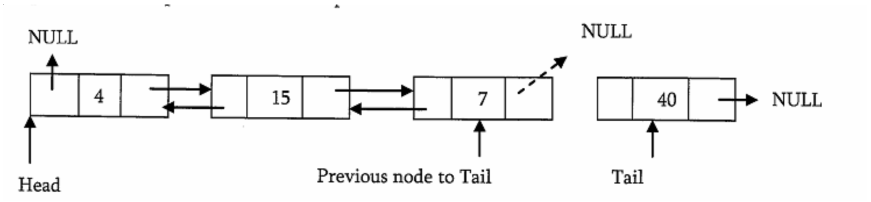
Figure 8.21: Traverse the list



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update tail nodes previous nodes next pointer with NULL.

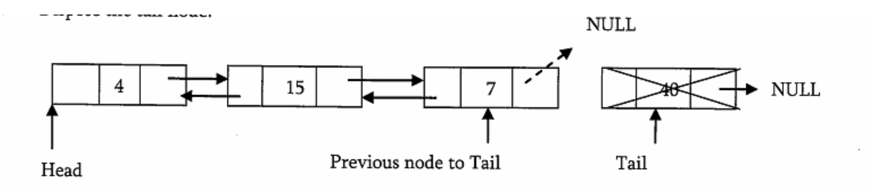
Figure 8.22: Update tail nodes



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Dispose the tail

Figure 8.23: Disposing the tail



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

Task Why is deleting the Last Node in Doubly Linked List is a bit trickier than removing the first node? Discuss.

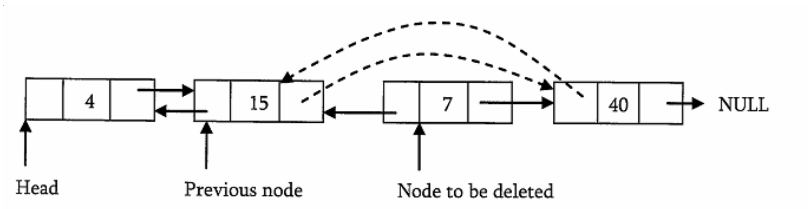
Deleting an Intermediate Node in Doubly Linked List

In this case, node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps.

Example: The example given below shows the steps for deleting an Intermediate Node in Doubly Linked List.

- As similar to previous case, maintain previous node also while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to the next node of the node to be deleted.

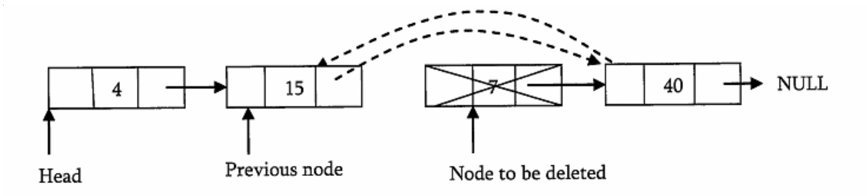
Figure 8.24: change the previous nodes



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Dispose the current node to be deleted.

Figure 8.25: Dispose the current node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

The implementation is shown as below:

```
void DLLDelete(struct DLLNode **head, int position) { stnict DLLNode
*temp, *temp2, temp = *head;

int k = 1;

if(*head == NULL) {

printf(“List is empty”);

return;

}

if(position   1){
```

```
*head = *head—>next;
if(*head != NULL)
*head—>prev = NULL;
free(temp);
return;
}

while((k < position - 1) && temp—>next1=NULL) {
temp = temp—>next;
k++;
}
if( temp—>next== NULL) {   //Deletion from end
temp2 = temp—>prev;
temp2—>next = NULL;
free(temp);
}
else  {
temp2 = temp-> Prev;
temp2—>next = temp—>next;
temp—>next—>prev = temp2;
free (temp);
}
return;
}
```

Self Assessment Questions

Fill in the blanks:

6. Insertion into a doubly-linked list has three cases, that is, Inserting a new node before the head, Inserting a new node after the tail (at the end of the list), and Inserting a new node at the .... of the list.

7. When Inserting a Node in Doubly Linked List at the Ending, ... the list till the end and insert the new node.

8. When Deleting an Intermediate Node in Doubly Linked List, ... and tail links are not updated.

9. Once we found the node to be deleted in .... linked list, change the previous nodes next pointer to the next node of the node to be deleted.

10. When Deleting the Last Node in Doubly Linked List, Update tail nodes previous nodes next pointer with ....

Notes

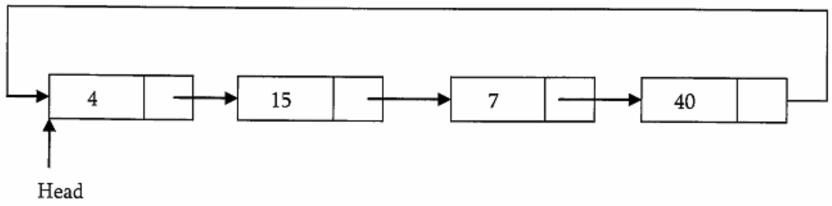
8.3 Basic Operations on Circular Linked Lists

The operations on Circular linked lists include counting, printing, inserting, deleting.

8.3.1 Counting Nodes in a Circular List

The circular list is accessible through the node marked head. To count the nodes, the list has to be traversed from node marked head, with the help of a dummy node current and stop the counting when current reaches the starting node head. If the list is empty, head will be NULL, and in that case set count = O. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

Figure 8.26: Counting Nodes



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

```
int CircularListLength(struct CLLNode *head) {
struct CLLNode *current = head;
int count = 0;
if(head== NULL) return 0;
do { current = current—> next;
count++;
} while (current != head);
return count;
}
```

8.3.2 Printing the Contents of a Circular List

We assume here that the list is being accessed by its head node. Since all the nodes are arranged in a circular fashion, the tail node of the list will be the node next to the head node. Let us assume we want to print the contents of the nodes starting with the head node. Print its contents, move to the next node and continue printing till we reach the head node again.

```
void PrintCircularListData(struct CLLNode *head) {
struct CLLNode *current = head;
if(head == NULL) return;
do { printf (“%d”, current—> data);
current = current—>next;
}
 while (current != head);
}
```

Task Write a program for printing the contents of a circular list.

8.3.3 Circular Linked List Insertion

Inserting a Node at the End of a Circular Linked List

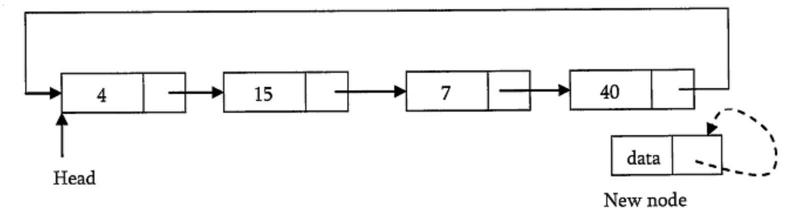
Let us add a node containing data, at the end of a list (circular list) headed by head.

Notes The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

Example: The example given below shows the steps for inserting a Node at the End of a Circular Linked List.

- Create a new node and initially keep its next pointer points to itself.

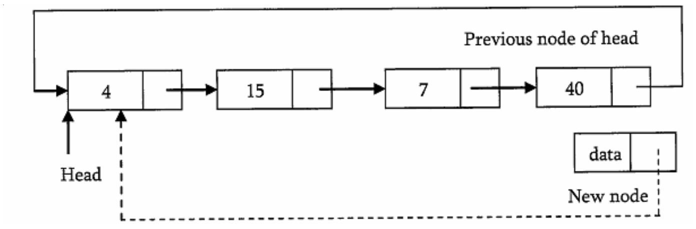
Figure 8.27: Create a new node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at a node whose next node is head.

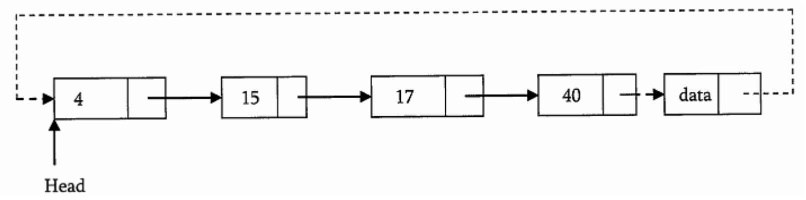
Figure 8.28: Update the next pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update the next pointer of previous node to point to new node and we get the list as shown below.

Figure 8.29: Update the next pointer of previous node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
struct CLLNode current = *head;
struct CLLNode *newNode = (struct node*) (malloc(sizeof(struct
CLLNode))); if(!newNode) {
printf("Memory Error"); return;
}
newNode-> data = data;
while (current->next != *head)
current = current->next;
newNode->next = newNode;
if(*head == NULL)
*head = newNode;
else
newNode->next = *head;
current->next = newNode;
}
}
```

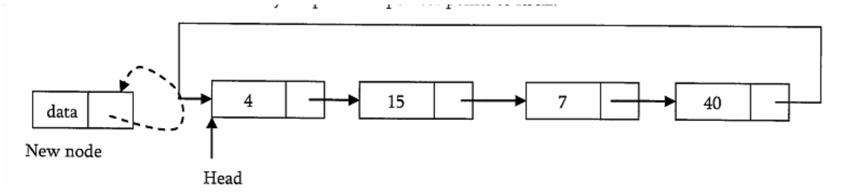
Inserting a Node at Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the ending is that, after inserting the new node we just need to update the pointer. Below are the steps for doing the same.

Example: The example given below shows the steps for Inserting a Node at Front of a Circular Linked List.

- Create a new node and initially keep its next pointer points to itself.

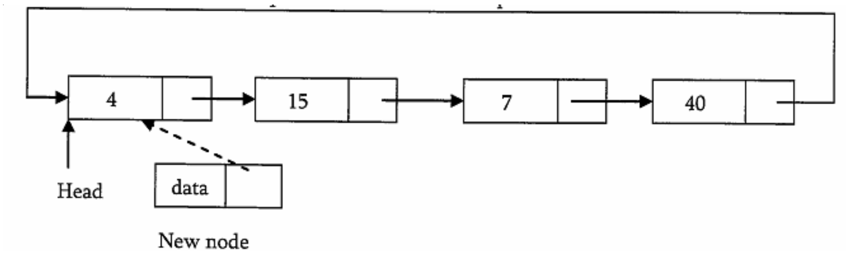
Figure 8.30: Create a new node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at the node which is its previous node in the list.

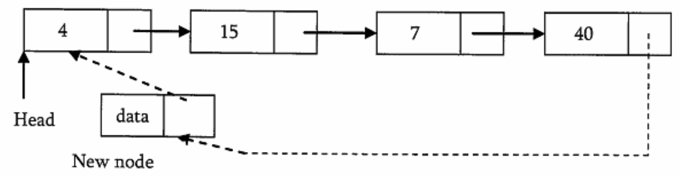
Figure 8.31: Update the next pointer



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update the previous node of head in the list to point to new node.

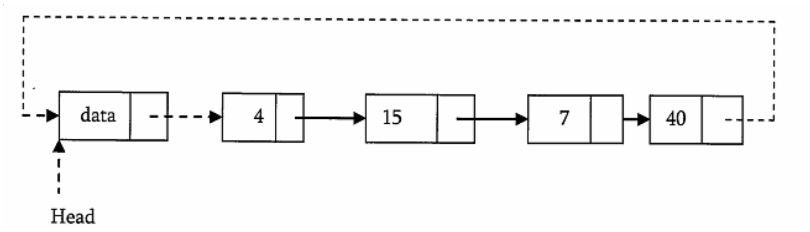
Figure 8.32: Update the previous node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Make new node as head.

Figure 8.33: Making new node as head



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

```
void InsertAtBeginInCLL (struct CLLNode **head, int data ) {
struct CLLNode *current = *head;
struct CLLNode * newNode = (struct node*) (malloc(sizeof(struct
CLLNode))); if(!newNode) {
printf(“Memory Error”); return;
}
newNode—>data = data;
while (current—>next != *head)
current = current—>next;
```

Notes

```
newNode—>next = newNode;
if(*head ==NULL)
*head = newNode;
else { newNode—>next = *head;
current—>next = newNode;
*head = newNode;
}
Return;
}
```

8.3.4 Circular Linked List Deletion

Deleting the Last Node in a Circular List

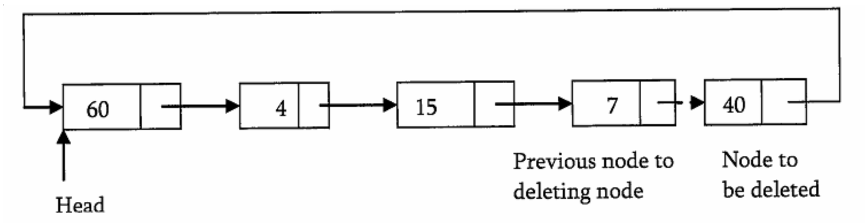
The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node.

Example: The example given below shows the steps for deleting the Last Node in a Circular List.

Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed pTail.

- Traverse the list and find the tail node and its previous node.

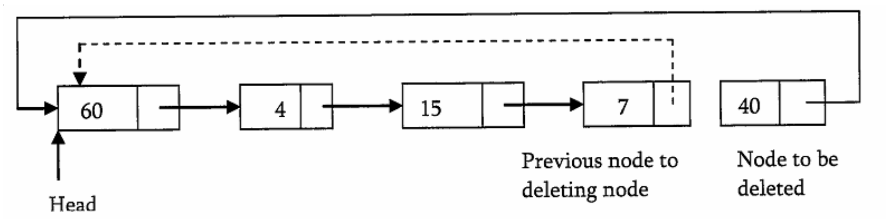
Figure 8.34: Traverse the list



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Update the tail nodes previous node next pointer to point to head.

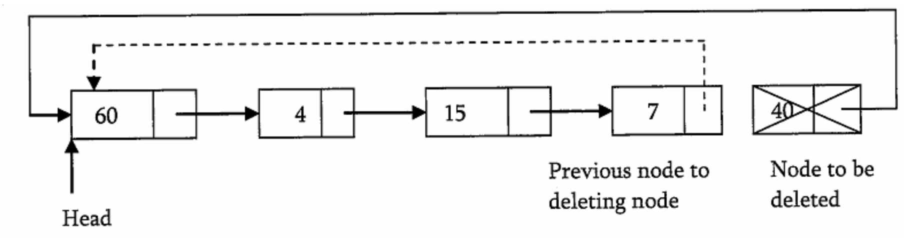
Figure 8.35: Update the tail nodes previous node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

- Dispose the tail node.

Figure 8.36: Dispose the tail node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

The code is given below.

```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
struct CLLNode *temp = *head;
struct CLLNode *current = *head;
if(*head == NULL) {
printf( “List Empty”);
return;
}
while (current—>next != *head) {
temp = current;
current = current—>next;
}
free(current);
return;
}
```

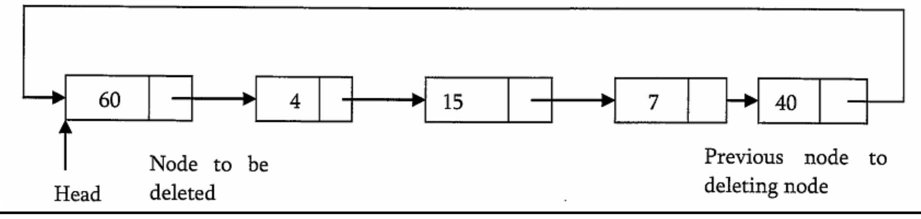
Deleting the First Node in a Circular List

The first node can be deleted by simply replacing the next field of tail node with the next field of the first node.

Example: The example given below shows the steps for deleting the First Node in a Circular List.

- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.

Figure 8.37: Find the tail node

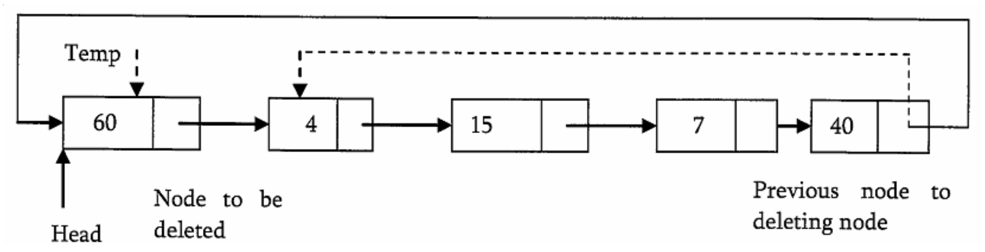


Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring:dsame_chap3.pdf

Notes

- Create a temporary which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).

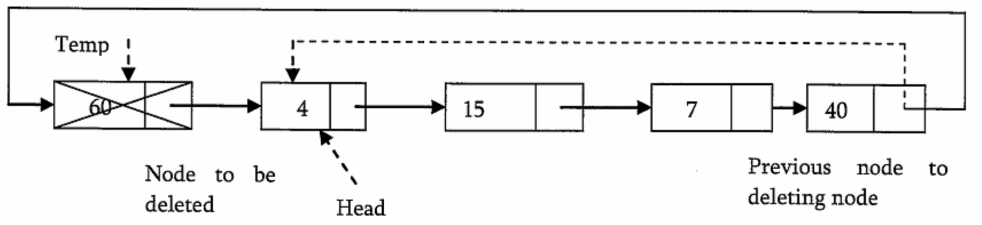
Figure 8.38: Creating a temporary pointing to head



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring: dsame_chap3.pdf

- Now, move the head pointer to next node. Create a temporary which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).

Figure 8.39: move the head pointer to next node



Source: http://www.csie.ntu.edu.tw/~hsinmu/courses/lib/exe/fetch.php?media=dsa_12spring: dsame_chap3.pdf

The code is given below.

```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
struct CLLNode *temp = *head;
struct CLLNode *current = *head;
if(*head == NULL) {
printf(“List Empty”);
return;
}
while (current—>next != *head)
current = current—>next;
current—> next = *head—>next;
*head = *head—)next;
free(temp);
return;
}
```

Self Assessment Questions

Notes

Fill in the blanks:

11. The .... list is accessible through the node marked head.
12. If the list is empty (In case of circular list), head will be NULL, and in that case set count =......................... .
13. When all the nodes are arranged in a circular fashion, the .... node of the list will be the node next to the head node.
14. The only difference between inserting a node at the beginning and at the ending is that, after inserting the new node we just need to update the ....
15. The first node of a circular list can be .... by simply replacing the next field of tail node with the next field of the first node.

8.4 Summary

- Linked list is a collection of data item, where each item is stored in a structure (node).
- The ListLength() function takes a linked list as input and counts the number of nodes in the list.
- When Inserting a Node in Singly Linked List at the Beginning, a new node is inserted before the current head node.
- Deleting the last node in Singly Linked List is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first.
- When Deleting an Intermediate Node in Singly Linked List, node to be removed is always located between two nodes.
- Deleting Singly Linked List works by storing the current node in some temporary variable and freeing the current node.
- The circular list is accessible through the node marked head.
- The only difference between inserting a node at the beginning and at the ending of a circular linked list is that, after inserting the new node we just need to update the pointer.

8.5 Keywords

Linked list: Linked list is a collection of data item, where each item is stored in a structure (node).

ListLength(): The ListLength() function takes a linked list as input and counts the number of nodes in the list.

Doubly linked list: A doubly linked list, in computer science, is a linked data structure that consists of a set of sequentially linked records called nodes.

Circular linked list: A circular linked list is a linked list in which the head element's previous pointer points to the tail element and the tail element's next pointer points to the head element.

8.6 Review Questions

1. Illustrate the steps used in Traversing the Linked List.
2. Discuss the concept of inserting a new node before the head and inserting a new node after the tail.

Notes

3. Illustrate how to delete Singly Linked List.
4. Describe the different cases included in doubly linked list insertion.
5. Explain the steps for deleting the first node in doubly linked list.
6. Deleting the Last Node in Doubly Linked List is trickier than removing the first node. Comment.
7. Write a program illustrating the Doubly Linked List deletion.
8. Describe the process of counting nodes in a circular list.
9. Illustrate how to print the contents of a circular list.
10. Explain the steps used for deleting the first node in a circular list.

Answers: Self Assessment

1. ListLength()
2. Beginning
3. last node
4. Intermediate
5. current
6. Middle
7. traverse
8. Head
9. doubly
10. NULL
11. circular
12. 0
13. circular
14. Tail
15. deleted

8.7 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.cs.wcupa.edu/~rkline/ds/linked-lists.html

http://www.cs.sunysb.edu/~cse130/slides/15-linked-lists.pdf

http://www.cs.sunysb.edu/~cse130/slides/15-linked-lists.pdf

http://www.indiastudychannel.com/

Unit 9: Stacks

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of stacks
- Explain sequential and linked list representation of stacks
- Describe operations on stack
- Discuss multi stacks
- Discuss applications of stack

Introduction

A stack is a data structure that is used to store elements but all the elements can be inserted or deleted from the TOP of the stack. The top of the stack is the top most part of the stack, something like a rectangular box which has only opening at the top. A stack is a linear data structure. It is very useful in many applications of computer science. A stack is generally implemented with two basic operations – push and pop. Push means to insert an item on to stack. The pop operation removes the topmost item from the stack. In this unit, we will discuss the concept of stacks. Also we will discuss the applications of stacks.

Notes

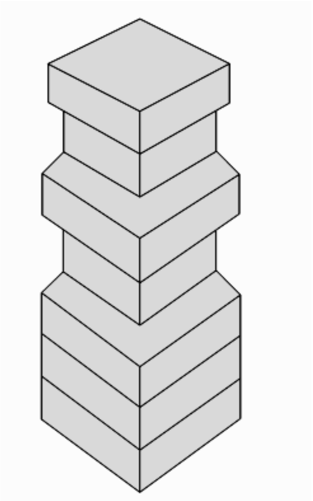
9.1 Concept of Stacks

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack.

Example: Some of the examples used to visualize this data structure are:

(a) ***Stack of Trays (or) Plates Placed on a Table:*** Here, one plate is placed on top of another, thus creating a stack of plates. Suppose, a person takes a plate off the top of the stack of plates. The plate most recently placed on the stack is the first one to be taken off. The bottom plate is the first one placed on the stack and the last one to be removed.

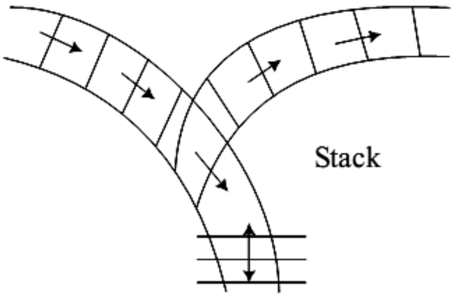
Figure 9.1: Stack of plates



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

(b) Another Familiar Example of a Stack is a Railway Station for Shunting Cars In this example, the last railway car placed on the stack is the first to leave.

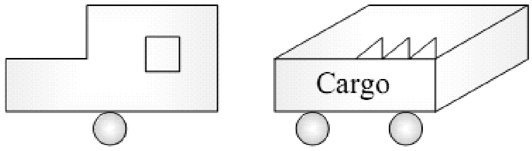
Figure 9.2: A railway shunting system—representation of a stack



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

(c) ***Shipment in a Cargo:*** For the shipment of goods, they have to be loaded into a cargo. During unloading, they are unloaded exactly in the opposite order as they are loaded, that is, the goods which are loaded last should be unloaded first.

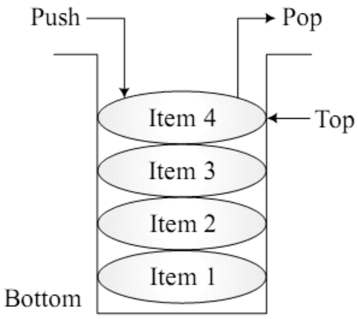
Figure 9.3: Shipment in a Cargo



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

Figure 9.4 shows the schematic diagram of a stack.

Figure 9.4: Schematic Diagram of a Stack



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

Other names for a stack are pushdown list, and LIFO (or) Last In First Out list. The stack allows access to only one item, i.e., the last item inserted. In the schematic diagram of a stack, after inserting Item 4 into the stack, it acts as the topmost element. It will be removed first. The very first item that was inserted into the stack is Item 1. It will be removed as the last item.

Self Assessment Questions

Fill in the blanks:

1. A .... is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack.
2. The stack allows access to only one item, i.e., the last item inserted. This is also known as .... system.

9.2 Operations on Stacks

The primitive operations that can be performed on a stack are given below:

1. Inserting an element into the stack (PUSH operation)
2. Removing an element from the stack (POP operation)
3. Determining the top item of a stack without removing it from the stack (PEEP operation)

PUSH operation is used to insert the data elements in a stack. The push operation inserts an element on to a stack and the element inserted settles down on the stack. Practically saying, a stack can be an array where elements are inserted in LIFO fashion.

The syntax used for the PUSH operation is:

```
PUSH (stack, item)
```

where stack is the name of the stack into which the item specified as the second argument is placed at the top position of the stack.

Algorithm to push an item onto the stack is given below:

Step 1: [Check for stack overflow]

```
  if tos >=MAXSTACK
```

print "Stack overflow" and exit

Step 2: [Increment the pointer value by one]

```
  tos=tos+1
```

Step 3: [Insert the item]

```
  arr[tos]=value
```

Step 4: Exit

Here, tos is a pointer which denotes the position of top most item in the stack. Stack is represented by the array arr and MAXSTACK represents the maximum possible number of elements in the stack

POP function is used for deleting the elements from a stack. As we know that stack follows a mechanism of LIFO, hence the last element to be inserted is the first element to be deleted. The syntax used for the POP operation is:

```
POP(stack)
```

where stack is the name of the stack from which the item has to be removed, i.e., the element at the topmost position of the stack is removed.

Algorithm to pop an element from the stack is given below:

Step 1: [Check whether the stack is empty]

```
  if tos = 0
```

print "Stack underflow" and exit

Step 2: [Remove the top most item]

```
  value=arr[tos]
  tos=tos-1
```

Step 3: [Return the item of the stack]

```
  return(value)
```

Consider a stack structure as given in Figure 9.5. After inserting (PUSH) an element (viz, 55) into this stack, the contents of the stack after the PUSH operation is given in Figure 9.6.

Figure 9.5: Stack Structure

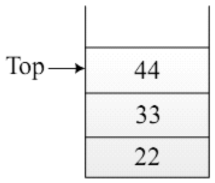
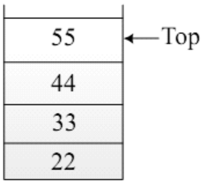


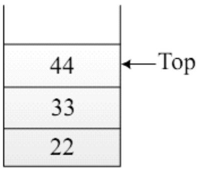
Figure 9.6: contents of the stack after the PUSH operation in figure 9.5



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

Suppose if the POP operation is specified as POP(st). Then the content of the stack can be visualized as in Figure 9.7.

Figure 9.7: Contents of the stack after applying pop operation in figure 9.6



Source: http://www.tup.tsinghua.edu.cn/Resource/tsyz/034111-01.pdf

That is, the element pointed out by Top will be removed and then Top will be decremented by 1. So it points the item placed below the removed item. It is also possible to verify the item placed at the top of the stack without removing it. This operation is called PEEP. The syntax used for this operation is:

```
PEEP(st)
```

This operation is the combination of POP and PUSH. It is equivalent to the following statements:

```
i = POP(st)
PUSH(st,i)
```

POP removes the item from the position pointed out by Top. It is in variable i. Then using the PUSH instruction we are inserting the same item i into the stack. Now the variable i has the value in the topmost position of the stack.

Task Compare and contrast push and pop operation.

9.2.1 Implementation of Push Operation

The push operation can be implemented as a function and this can be shown below:-

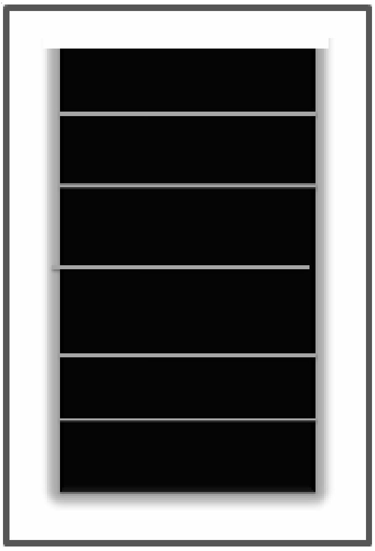
```
push(int element)
{
   ++top; //Incrementing the variable
   if(top==SIZE-1)
  {
```

Notes

```
      printf(‟ STACK FULL, cannot insert more elements”);
     exit(0);
  }
stack[top]=element; /*Inserting an element to the stack which is an
array*/
printf(“THE TOP OF STACK IS %d”, stack[top]);
}
```

In the above code we can notice that line starts the beginning of the push() function which accepts one argument that is the element to be inserted. The first operation we perform is to increment the variable top which is a variable that holds the index value of the array and practically holds the value of the TOP of stack. Initially the value of top is initialized to -1 which means that the stack is empty initially. Hence it is incremented first so that the value of top will be zero and it will be the first index of the array.

Figure 9.8: TOP = -1(no elements)



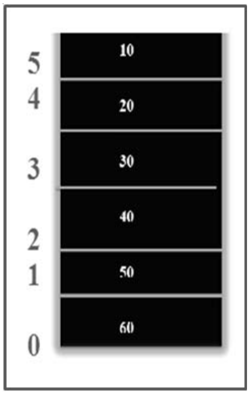
Source: http://newtonapples.com/operations-on-stack-c-language/

The variable size holds the size of the stack, i.e. the value of it suggests the maximum elements that can be accepted by the stack. Hence in line 4 the condition top == stack-1 suggests practically if the stack is full or not. The stack full condition is even known as stack overflow.

Notes The top variable increments on every insertion of element and when the value of top is equal to the value of size-1 (-1 since stack is an array and it starts with 0), it means that the stack cannot accept more elements.

Hence in line 5 we print a message that the stack is full and we cannot insert more elements. Line 6 executes the exit() function because it is not logical to get further in the function since we cannot insert any more elements.

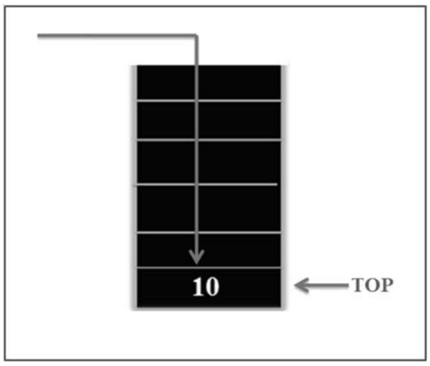
Figure 9.9: Top= 5 and size=6, STACK FULL



Source: http://newtonapples.com/operations-on-stack-c-language/

If the stack is not full, line 9 actually inserts the elements passed as argument in the array stack[] and it will be inserted at the same location where the top variable will be having value. Line 10 displays the element that is present in the top of the stack.

Figure 9.10: stack[top] = element



Source: http://newtonapples.com/operations-on-stack-c-language/

9.2.2 Implementation of Pop Operation

The pop operation can be shown as a function pop() which actually helps to remove an element from the stack.

```
pop()
{
   if(top== -1) // Checking if stack is empty
  {
      printf("STACK EMPTY, no elements to delete");
     exit(0);
  }
   int ele;
   ele=stack[top]; //ele holds the top most element that is to deleted
   printf("The element deleted is %d", stack[top]);
```

Notes

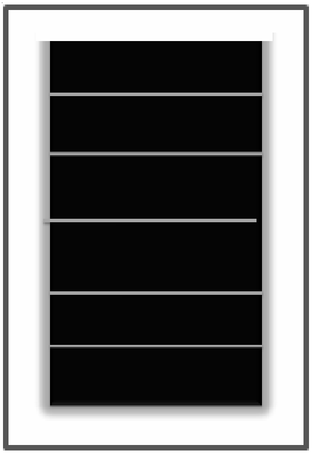
```
  top—;       // decrementing top to hold the index of next ele
}
```

Line 1 marks the beginning of the pop function and it does not accept any argument. Line 3 checks a condition if the stack is empty or not, it is essential to check it because if there are no elements in the stack it does not make any sense to perform deletion operation.

Did u know? The stack empty condition is even known as stack underflow.

Hence the condition top==-1 states if the value of top is -1 than logically the stack does not has any element. Immediately line 3 executes the exit() function which does not allow to go further in the function.

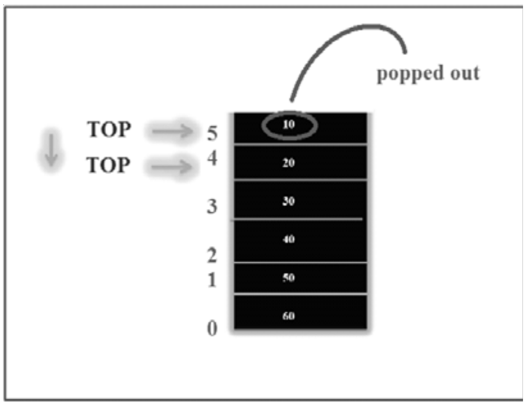
Figure 9.11: TOP = -1, Stack is empty



Source: http://newtonapples.com/operations-on-stack-c-language/

Line 8 states the declaration of a variable ele which is responsible to hold the value of the top most element. Line 9 states that the top most element in the stack is assigned to the variable ele and then line 10 prints the elements that will be logically deleted. Line 11 decrements the top variable so that it hold the index value of the next element in the stack.

Figure 9.12: Line 10 prints the element



Source: http://newtonapples.com/operations-on-stack-c-language/

Self Assessment Questions

Fill in the blanks:

3. .... operation is used to insert the data elements in a stack.

4. .... function is used for deleting the elements from a stack.

5. Verifying the item placed at the top of the stack without removing it is called .... operation.

6. When the value of top is initialized to -1 initially, it means that the stack is .... initially.

7. The stack empty condition is even known as ....

9.3 Representation of Stacks

We can represent stacks in two ways:

- Sequential Representation of Stacks
- Linked list Representation of Stacks

These are discussed below.

9.3.1 Sequential Representation of Stacks

A Stack contains an ordered list of elements and an array is also used to store ordered list of elements. Hence, it would be very easy to manage a stack using an array. However, the problem with an array is that we are required to declare the size of the array before using it in a program. Therefore, the size of stack would be fixed. Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack.

⚠ *Caution* We can declare the array with a maximum size large enough to manage a stack.

Program given below implements a stack using an array.

```
#include<stdio.h>
int choice, stack[10], top, element;
void menu();
void push();
void pop();
void showelements();
void main()
{ choice=element=1;
  top=0;
  menu();
}
void menu()
{
```

Notes

```
 printf(“Enter one of the following options:\n”);
 printf(“PUSH 1\n POP 2\n SHOW ELEMENTS 3\n EXIT 4\n”);
 scanf(“%d”, &choice);
 if (choice==1)
 {
 push(); menu();
 }
 if (choice==2)
 {
 pop();menu();
}
 if (choice==3)
 {
 showelements(); menu();
 }
 }
void push()
{
 if (top<=9)
 {
 printf(“Enter the element to be pushed to stack:\n”);
 scanf(“%d”, &element);
 stack[top]=element;
 ++top;
 }
 else
 {
 printf(“Stack is full\n”);
 }
 return;
}
void pop()
{
 if (top>0)
 {
 —top;
 element = stack[top];
 printf(“Popped element:%d\n”, element);
 }
 else
 {
```

```
 printf("Stack is empty\n");
 }
 return;
}
void showelements()
{
 if (top<=0)
 printf("Stack is empty\n");
 else
 for(int i=0; i<top; ++i)
 printf("%d\n", stack[i]);
}
```

In this program, the size of the stack was declared as 10. So, stack cannot hold more than 10 elements. The main operations that can be performed on a stack are push and pop. However, in a program, we need to provide two more options , namely, showelements and exit. showelements will display the elements of the stack. In case, the user is not interested to perform any operation on the stack and would like to get out of the program, then s/he will select exit option. It will log the user out of the program. choice is a variable which will enable the user to select the option from the push, pop, showelements and exit operations. top points to the index of the free location in the stack to where the next element can be pushed. element is the variable which accepts the integer that has to be pushed to the stack or will hold the top element of the stack that has to be popped from the stack. The array stack can hold at most 10 elements. push and pop will perform the operations of pushing the element to the stack and popping the element from the stack respectively.

9.3.2 Linked List Representation of Stacks

When a stack is implemented using arrays, it suffers from the basic limitation of an array – that is, its size cannot be increased or decreased once it is declared. As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In the case of a linked stack, we shall push and pop nodes from one end of a linked list. The stack, as linked list is represented as a singly connected list. Each node in the linked list contains the data and a pointer that gives location of the next node in the list.

Program given below implements a stack using linked lists.

```
#include<stdio.h>
 #include<conio.h>
 #include<stdlib.h>
 /* Definition of the structure node */
typedef struct node
{
 int data;
 struct node *next;
 } ;
 /* Definition of push function */
```

Notes

```
void push(node **tos,int item)
{
 node *temp;
 temp=(node*)malloc(sizeof(node)); /* create a new node dynamically */
 if(temp==NULL)      /* If sufficient amount of memory is */
 {     /* not available, the function malloc will */
 printf(“\nError: Insufficient Memory Space”);  /* return NULL to temp
*/
 getch();
 return;
 }
 else
 {
 temp->data=item;    /* put the item in the data portion of node*/
 temp->next=*tos;    /*insert this node at the front of the stack */
*tos=temp; /* managed by linked list*/
}
 }       /*end of function push*/
/* Definition of pop function */
int pop(node **tos)
{
 node *temp;
 temp=*tos;
 int item;
 if(*tos==NULL)
 return(NULL);
 else
{
 *tos=(*tos)->next; /* To pop an element from stack*/
 item=temp->data;    /* remove the front node of the */
 free(temp);         /* stack managed by L.L*/
 return (item);
 }
}      /*end of function pop*/
/* Definition of display function */
void display(node *tos)
{
 node *temp=tos;
 if(temp==NULL)      /* Check whether the stack is empty*/
 {
 printf(“\nStack is empty”);
 return;
 }
 else
 {
 while(temp!=NULL)
 {
 printf(“\n%d”,temp->data); /* display all the values of the stack*/
 temp=temp->next;    /* from the front node to the last node*/
 }
 }
 }     /*end of function display*/
/* Definition of main function */
void main()
{
 int item, ch;
 char choice=‘y’;
 node *p=NULL;
 do
 {
 clrscr();
 printf(“\t\t\t\t*****MENU*****”);
printf(“\n\t\t\t1. To PUSH an element”);
 printf(“\n\t\t\t2. To POP an element”);
 printf(“\n\t\t\t3. To DISPLAY the elements of stack”);
 printf(“\n\t\t\t4. Exit”);
 printf(“\n\n\n\t\t\tEnter your choice:-”);
 scanf(“%d”,&ch);
 switch(ch)
 {
 case 1:
 printf(“\n Enter an element which you want to push “);
 scanf(“%d”,&item);
 push(&p,item);
 break;
 case 2:
 item=pop(&p);
 if(item!=NULL);
 printf(“\n Detected item is%d”,item);
 break;
 case 3:
 printf(“\nThe elements of stack are”);
display(p);
 break;
 case 4:
 exit(0);
 }     /*switch closed */
 printf(“\n\n\t\t Do you want to run it again y/n”);
 scanf(“%c”,&choice);
 } while(choice==’y’);
 }      /*end of function main*/
```

Similarly, as we did in the implementation of stack using arrays, to know the working of this program, we executed it thrice and pushed 3 elements (10, 20, 30). Then we call the function display in the next run to see the elements in the stack.

In this program, initially, we defined a structure called node. Each node contains two portions, data and a pointer that keeps the address of the next node in the list. The Push function will insert a node at the front of the linked list, whereas pop function will delete the node from the front of the linked list. There is no need to declare the size of the stack in advance as we have done in the program where in we implemented the stack using arrays since we create nodes dynamically as well as delete them dynamically. The function display will print the elements of the stack.

Task Analyze the problems that takes place in the sequential implementation of stacks.

Self Assessment Questions

Fill in the blanks:

8. .... will display the elements of the stack.

9. In the case of a .... stack, we shall push and pop nodes from one end of a linked list.

9.4 Multi Stacks

Until now, we have discussed the representation of a single stack. Now we will discuss the concept of data representation for several stacks. Let us see an array X whose dimension is m. For convenience, we shall assume that the indexes of the array commence from 1 and end at m. If we have only 2 stacks to implement in the same array X, then the solution is simple.

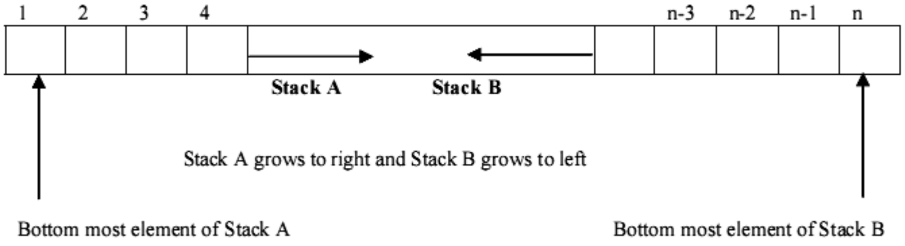
Suppose A and B are two stacks. We can define an array stack A with n_1 elements and an array stack B with n_2 elements.

Caution Overflow may occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose, instead of that, we define a single array stack with $n = n_1 + n_2$ elements for stack A and B together. See the Figure 9.13 below. Let the stack A “grow” to the right, and stack B “grow” to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. It does not matter how many elements individually are there in each stack.

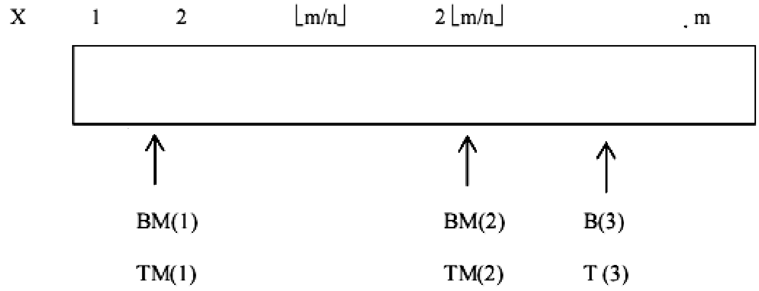
Notes

Figure 9.13: Implementation of multiple stacks using arrays


1
2
3
4
n-3
n-2
n-1
n
Stack A
Stack B
Stack A grows to right and Stack B grows to left
Bottom most element of Stack A
Bottom most element of Stack B

But, in the case of more than 2 stacks, we cannot represent these in the same way because a one-dimensional array has only two fixed points X(1) and X(m) and each stack requires a fixed point for its bottom most element. When more than two stacks, say n, are to be represented sequentially, we can initially divide the available memoryX(1:m) into n segments. If the sizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks. If the sizes of the stacks are not known, then, X(1:m) may be divided into equal segments. For each stack i, we shall use BM (i) to represent a position one less than the position in X for the bottom most element of that stack. TM(i), $1 \le i \le n$ will point to the topmost element of stack i. We shall use the boundary condition BM (i) = TM (i) iff the ithstack is empty. If we grow the ith stack in lower memory indexes than the i+1st stack, then, with roughly equal initial segments we have BM (i) = TM (i) = $m/n \ (i-1), 1 \le i \le n$, as the initial values of BM (i) and TM (i).

Figure 9.14: Initial configuration for n stacks in X(1:m)



All stacks are empty and memory is divided into roughly equal segments.

Self Assessment Questions

State True or False:

10. When more than two stacks, say n, are to be represented sequentially, we can initially divide the available memoryX(1:m) into n segments.

11. If the sizes of the stacks are known, then, we can allocate the segments to them in proportion to the expected sizes of the various stacks.

9.5 Application of Stacks

Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators. Polish notations are evaluated by stacks. Conversions of different notations (Prefix, Postfix, Infix) into one another are performed using stacks. Stacks are widely used inside computer when recursive functions are called. The computer evaluates an arithmetic expression written in infix notation in two steps. First, it converts the infix expression to postfix expression and then it evaluates the postfix expression. In each step, stack is used to accomplish the task.

Some applications of stack are listed below:

9.5.1 Parenthesis Checker

It is an algorithm that confirms that the number of closing parenthesis equals opening parenthesis by using stack. If number of closing parenthesis is not equal to the number of opening parenthesis, then it results in an error. Input a string from the user. The user must enter a set of opening and closing parenthesis as follows:

(()(()))

Scan the above input string from first character until NULL is found. If it is an opening parenthesis, then push it in the stack, otherwise, if a closing parenthesis is found, pop the topmost element of the stack. If the string ends and the stack becomes empty, then there is a perfect match, otherwise, you may report error by returning appropriate value to the invoking function.

Program for Parenthesis checker is as follows:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAX 20
struct stack
{
int top;
char arr[MAX];
};
void push(struct stack *s,char ch)
{
if(s->top==MAX-1)
{printf(“Stack Overflow.”);
exit(1);
}
s->arr[++(s->top)]=ch;
}
char pop(struct stack *s)
{
if(s->top==-1)
{printf(“Stack Underflow.”);
exit(1);
}
return(s->arr[s->top—]);
}
```

```
int parenthesischecker(char str[])
{
struct stack s;
char ch;
int i,flag=0;/*0 for no error, 1 for error*/
s.top=-1;
for(i=0;((str[i]!=’’) && (flag==0));i++)
{if(str[i]==’(‘)
push(&s,str[i]);
else
{ch=pop(&s);
if(ch!=’(‘)
flag=1;
}
}
if(s.top!=-1)
flag=1;
return flag;
}
void main()
{
char str[MAX],flag;
clrscr();
printf(“nEnter a string of parenthesis?”);
gets(str);
flag=parenthesischecker(str);
if(flag==0)
{printf(“nParenthesis Match O.K.”);
}
else
{printf(“nOpening/Closing Parenthesis do not match.”);
}
getch();
}
```

9.5.2 Evaluation of arithmetic expressions

Stacks are useful in evaluation of arithmetic expressions. Consider the expression

5 * 3 +2 + 6 * 4

The expression can be evaluated by first multiplying 5 and 3, storing the result in A, adding 2 and A, saving the result in A. We then multiply 6 and 4 and save the answer in B. We finish off by adding A and B and leaving the final answer in A.

A = 15 2 +

= 17

B = 6 4 *

= 24

A = 17 24 +

= 41

We can write this sequence of operations as follows:

5 3 * 2 + 6 4 * +

This notation is known as postfix notation and is evaluated as described above. We shall shortly show how this form can be generated using a stack.

Basically there are 3 types of notations for expressions. The standard form is known as the infix form. The other two are postfix and prefix forms.

- Infix: operator is between operands, that is, A + B
- Postfix : operator follows operands., that is, A B +
- Prefix: operator precedes operands, that is, + A B

Notes All infix expressions cannot be evaluated by using the left to right order of the operators inside the expression. However, the operators in a postfix expression are ALWAYS in the correct evaluation order.

Thus evaluation of an infix expression is done in two steps. The first step is to convert it into its equivalent postfix expression. The second step involves evaluation of the postfix expression. We shall see in this section, how stacks are useful in carrying out both the steps. Let us first examine the basic process of infix to postfix conversion.

The steps for converting infix to postfix form are shown in the examples given below.

Example: Consider the Infix form as a + b * c. This is to note that precedence of * is higher than of +.

1. a + (b * c) : convert the multiplication
2. a + (b c *) : convert the addition
3. a (b c *) + : Remove parentheses
4. a b c * + : Postfix form

Did u know? There is no need of parentheses in postfix forms.

Example:

1. (A + B) * C : Infix form
2. (A B +) * C : Convert the addition

3. (A B +) C * : Convert multiplication
4. A B + C * : Postfix form

Order of precedence for operators:

multiplication (*) and division (/)

addition (+) and subtraction (-)

The association is assumed to be left to right.

i.e. a + b + c = (a + b) + c = ab + c +

Evaluating a Postfix Expression

We can evaluate a postfix expression using a stack. Each operator in a postfix string corresponds to the previous two operands . Each time we read an operand we push it onto a stack. When we reach an operator its associated operands (the top two elements on the stack) are popped out from the stack.

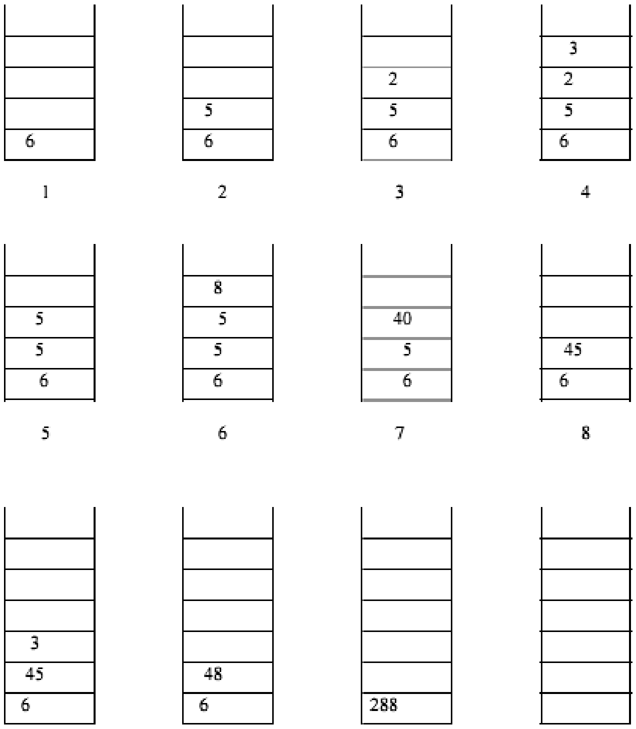
We then perform the indicated operation on them and push the result on top of the stack so that it will be available for use as one of the operands for the next operator .

The following example shows how a postfix expression can be evaluated using a stack.

Example:

6 5 2 3 + 8 * + 3 + *

Figure 9.15: Evaluation of postfix expression using stack



Source: http://www.eecs.ucf.edu/courses/cop3502h/spr2007/stacks.pdf

The process stops when there are no more operator left in the string. The result of evaluating the expression is obtained just by popping off the single element.

Converting an Infix Expression to Postfix

A stack can also be used to convert an infix expression in standard form into postfix form. We shall assume that the expression is a legal one (i.e. it is possible to evaluate it). When an operand is read, it will be placed on output list (printed out straight away). The operators are pushed on a stack. However, if the priority of the top operator in the stack is higher than the operator being read, then it will be put on output list, and the new operator pushed on to the stack. The priority is assigned as follows.

1. (Left parenthesis in the expression
2. * /
3. + –
4. (Left parenthesis inside the stack

The left parenthesis has the highest priority when it is read from the expression, but once it is on the stack, it assumes the lowest priority.

To start with , the stack is empty. The infix expression is read from left to right. If the character is an OPERAND, it is not put on the stack. It is simply printed out as part of the post fix expression.

The stack stores only the OPERATORS. The first operator is pushed on the stack. For all subsequent operators, priority of the incoming operator will be compared with the priority of the operator at the top of the stack.

If the priority of the incoming-operator is higher than the priority of topmost operator-on the stack , it will be pushed on the stack.

If the priority of the incoming-operator is same or lower than the priority of the operator at the top of the stack, then the operator at top of the stack will be popped and printed on the output expression.

The process is repeated if the priority of the incoming-operator is still same or lower than the next operator-in-the stack. When a left parenthesis is encountered in the expression it is immediately pushed on the stack, as it has the highest priority. However, once it is inside the stack, all other operators are pushed on top of it, as its inside-stack priority is lowest.

When a right parenthesis is encountered, all operators up to the left parenthesis are popped from the stack and printed out. The left and right parentheses will be discarded. When all characters from the input infix expression have been read, the operators remaining inside the stack, are printed out in the order in which they are popped.

Here is an algorithm for converting an infix expression into its postfix form.

Algorithm:

```
while there are more characters in the input
{
Read next symbol ch in the given infix expression.
If ch is an operand put it on the output.
If ch is an operator i.e.* , /, +, -, or (
 {
 If stack is empty push ch onto stack;
Else check the item op at the top of the stack;
 while (more items in the stack &&
 priority(ch) <= priority (op) )
 {
pop op and append it to the output, provided it
is not an open parenthesis
 op = top element of stack
}
push ch onto stack
 }
 If ch is right parenthesis ')'
 Pop items from stack until left parenthesis
 reached
Pop left parenthesis and discard both left
 and right parenthesis
}/* now no more characters in the infix expression*/
Pop remaining items in the stack to the output.
```

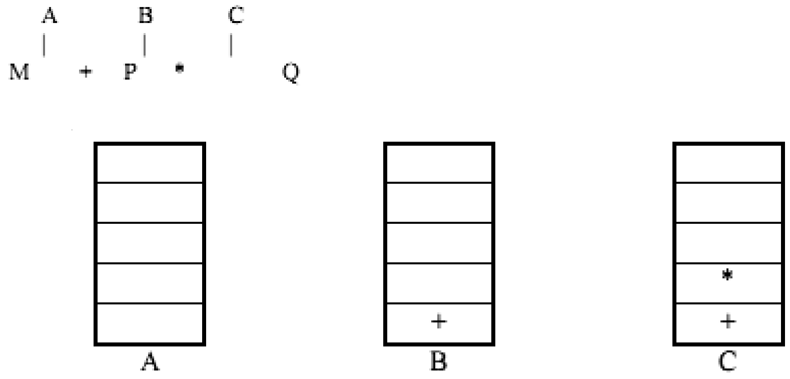
Example:

Using a stack to convert an Infix expression into its corresponding postfix form

Consider the following expressions with the infix notation. We want to transform these into postfix expressions using a stack. We also trace the state of the operator stack as each character of the infix expression is processed.

The contents of the operator stack at the indicated points in the infix expressions (points A, B and C) are shown below for each case.

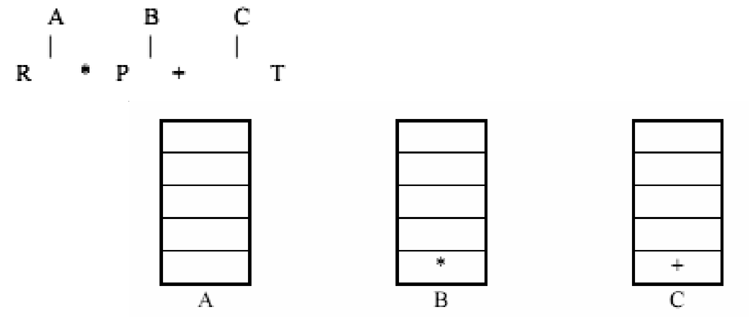
1.



Notes

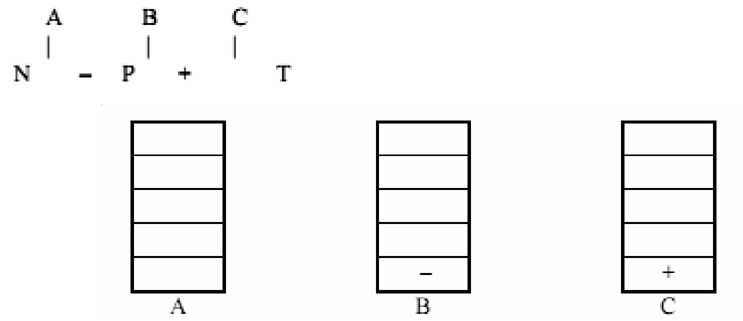
Resulting Postfix Expression : M P Q * +

2.



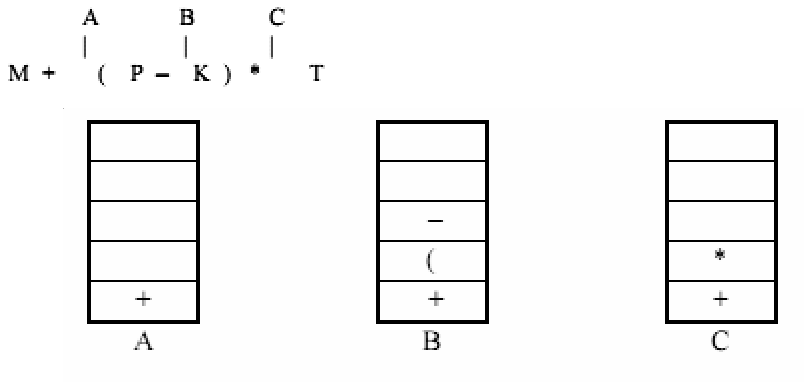
Resulting Postfix Expression : R P * T +

3.



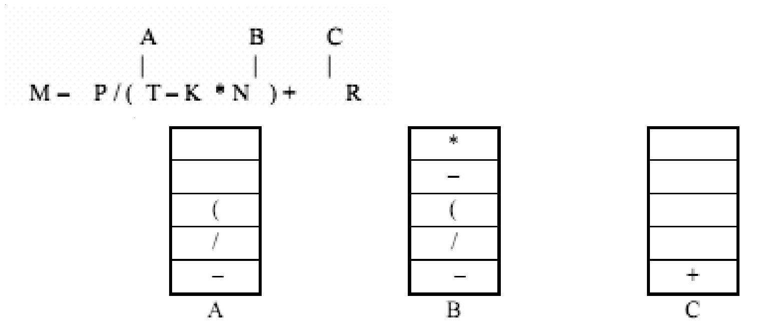
Resulting Postfix Expression : N P – T +

4.



Resulting Postfix Expression : M P K – T * +

5.



Source: http://www.eecs.ucf.edu/courses/cop3502h/spr2007/stacks.pdf

Resulting Postfix Expression : M P T K N * – / – R +

Self Assessment Questions

Fill in the blanks:

12. .... is an algorithm that confirms that the number of closing parenthesis equals opening parenthesis by using stack.

13. All .... expressions cannot be evaluated by using the left to right order of the operators inside the expression.

14. Each operator in a .... string corresponds to the previous two operands .

15. The left parenthesis has the .... priority when it is read from the expression

Case Study

Case:

Program to implement stack with its operations using pointers

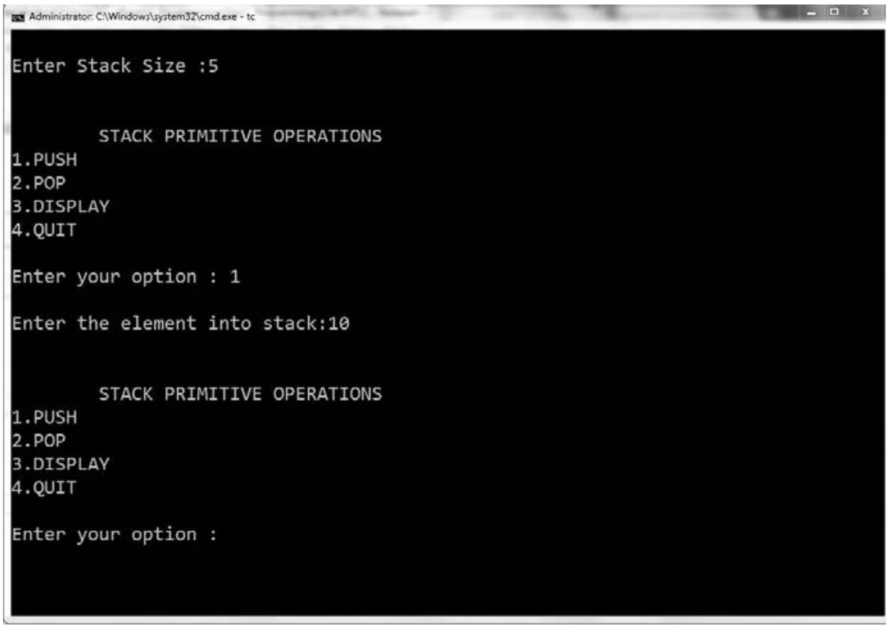
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 50
int size;
/* Defining the stack structure */
struct stack
{
     int arr[MAX];
     int top;
};
/* Initializing the stack(i.e., top=-1) */
void init_stk(struct stack *st)
{
     st->top = -1;
}
/* Entering the elements into stack */
void push (struct stack *st,int num)
{
     if(st->top == size-1)
        {
          printf("nStack overflow(i.e., stack full).");
         return;
        }
     st->top++;
     st->arr[st->top] = num;
}
```

Contd...

```
//Deleting an element from the stack.
int pop(struct stack *st)
{
        int num;
        if(st->top == -1)
              {
                  printf("nStack underflow(i.e., stack empty).");
               return NULL;
              }
         num = st->arr[st->top];
        st->top--;
        return num;
}
void display(struct stack *st)
{
        int i;
         for(i=st->top;i>=0;i--)
                printf("n%d",st->arr[i]);
}
int main()
{
         int element,opt,val;
        struct stack ptr;
         init_stk(&ptr);
         printf("nEnter Stack Size :");
         scanf("%d",&size);
        while(1)
       {
                printf("nntSTACK PRIMITIVE OPERATIONS");
              printf("n1.PUSH");
              printf("n2.POP");
               printf("n3.DISPLAY");
              printf("n4.QUIT");
             printf("n");
              printf("nEnter your option : ");
              scanf("%d",&opt);
             switch(opt)
            {
                   case 1:
                            printf("nEnter the element into stack:");
                          scanf("%d",&val);
```

```
                 push(&ptr,val);
                break;
            case 2:
                 element = pop(&ptr);
                  printf("nThe element popped from stack is :
%d",element);
                break;
            case 3:
                  printf("nThe current stack elements are:");
                 display(&ptr);
                break;
            case 4:
                exit(0);
            default:
                  printf("nEnter correct option!Try again.");
        }
    }
return(0);
}
```

Output:



```
Enter Stack Size :5


       STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option : 1

Enter the element into stack:10


       STACK PRIMITIVE OPERATIONS
1.PUSH
2.POP
3.DISPLAY
4.QUIT

Enter your option :
```

Question:

Write a C program to implement evaluation of Postfix expression using Stack.

Source: http://www.c4learn.com/c-programs/c-program-to-perform-stack-operations-using-pointer.html

Notes

9.6 Summary

- A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack.
- PUSH operation is used to insert the data elements in a stack. POP function is used for deleting the elements from a stack.
- A Stack contains an ordered list of elements and an array is also used to store ordered list of elements.
- In the case of a linked stack, we shall push and pop nodes from one end of a linked list.
- We can define an array stack A with n1 elements and an array stack B with n2 elements. Overflow may occur when either stack A contains more than n1 elements or stack B contains more than n2 elements.
- Parenthesis checker is an algorithm that confirms that the number of closing parenthesis equals opening parenthesis by using stack.
- Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operands and operators.
- We can evaluate a postfix expression using a stack. Each operator in a postfix string corresponds to the previous two operands .
- A stack can also be used to convert an infix expression in standard form into postfix form.

9.7 Keywords

Stack: A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack.

PUSH: PUSH operation is used to insert the data elements in a stack.

POP: POP function is used for deleting the elements from a stack.

Parenthesis checker: Parenthesis checker is an algorithm that confirms that the number of closing parenthesis equals opening parenthesis by using stack.

Arithmetic expression: An arithmetic expression consists of operands and operators.

PEEP: The process of verifying the item placed at the top of the stack without removing it is called PEEP.

Showelements Operation: Showelements will display the elements of the stack.

Multi stack: Multi stack is a program with more than one stack.

9.8 Review Questions

1. Explain the concept of stack with example.
2. Discuss the operations performed on a stack. Give example.
3. Illustrate the use of PEEP operation with example.
4. Describe the implementation of push and pop operation.
5. Explain the Sequential Representation of Stacks. Illustrate with a program.

6. What are multi stacks? Discuss the implementation of multiple stacks using arrays.
7. What is Parenthesis checker? Explain the use of stacks in Parenthesis checker.
8. Describe the Evaluation of arithmetic expressions with examples.
9. What are the 3 types of notations for expressions? Explain with example.
10. Discuss how stack is used to convert an Infix Expression to Postfix.

Answers: Self Assessment

1. stack
2. LIFO
3. PUSH
4. POP
5. PEEP
6. Empty
7. stack underflow
8. Showelements
9. linked
10. True
11. True
12. Parenthesis checker
13. infix
14. Postfix
15. highest

9.9 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.wiziq.com/tutorial/13556-STACKS-IN-DATA-STRUCTURE

http://www.zentut.com/c-tutorial/c-stack-using-array/

http://www.cmpe.boun.edu.tr/~akin/cmpe223/chap2.htm

http://groups.csail.mit.edu/graphics/classes/6.837/F04/cpp_notes/stack1.html

Notes

Unit 10: Queues

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the concept of linear queues and its representation.
- Explain the concept of circular queues and its representation.
- Explain the concept of priority queues and its representation.
- Explain the concept of dequeue and its representation.

Introduction

A queue is a data collection in which the items are kept in the order in which they were inserted, and the primary operations are enqueue (insert an item at the end) and dequeue (remove the item at the front). Because the first item in the queue will be the first one out, queues are known as a First-In-First-Out (FIFO) data structure. Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service. The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is the first to be served.

10.1 Representation of Linear Queue

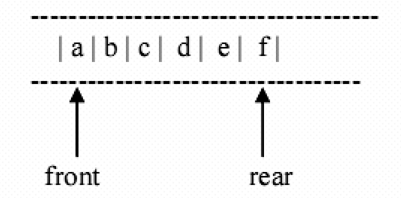
A physical analogy for a queue is a line at a booking counter. At a booking counter, customers go to the rear (end) of the line and customers are attended to various services from the front of the line. Unlike stack, customers are added at the rear end and deleted from the front end in a queue (FIFO).

Example: An example of the queue in computer science is print jobs scheduled for printers. These jobs are maintained in a queue. The job fired for the printer first gets printed first. Same is the scenario for job scheduling in the CPU of computer.

Did u know? Like a stack, a queue also (usually) holds data elements of the same type.

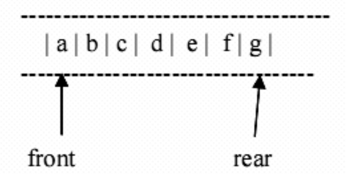
We usually graphically display a queue horizontally. Figure 10.1 depicts a queue of 5 characters.

Figure 10.1: A Queue of Characters



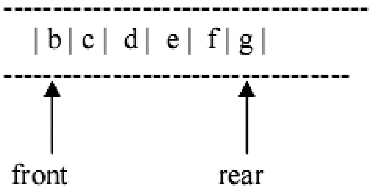
The rule followed in a queue is that elements are added at the rear and come off of the front of the queue. After the addition of an element to the above queue, the position of rear pointer changes as shown below. Now the rear is pointing to the new element 'g' added at the rear of the queue.

Figure 10.2: Queue of figure 10.1 after addition of new element



After the removal of element 'a' from the front, the queue changes to the following with the front pointer pointing to 'b'

Figure 10.3: Queue of figure 10.2 after deletion of an element



Queue has two ends. One is front from where the elements can be deleted and the other if rear to where the elements can be added. A queue can be implemented using Arrays or Linked lists. Each representation is having it's own advantages and disadvantages. The problems with arrays are that they are limited in space. Hence, the queue is having a limited capacity. If queues are implemented using linked lists, then this problem is solved. Now, there is no limit on the capacity of the queue. The only overhead is the memory occupied by the pointers.

Algorithm for addition of an element to the queue:

Step 1: Create a new element to be added

Step 2: If the queue is empty, then go to step 3, else perform step 4

Step 3: Make the front and rear point this element

Step 4: Add the element at the end of the queue and shift the rear pointer to the newly added element.

Algorithm for deletion of an element from the queue:

Step 1: Check for Queue empty condition. If empty, then go to step 2, else go to step 3

Step 2: Message "Queue Empty"

Step 3: Delete the element from the front of the queue. If it is the last element in the queue, then perform step a else step b

(a) make front and rear point to null

(b) shift the front pointer ahead to point to the next element in the queue

10.1.1 Sequential Representation of a Queue

As the stack is a list of elements, the queue is also a list of elements. The stack and the queue differ only in the position where the elements can be added or deleted. Like other liner data structures, queues can also be implemented using arrays.

Program given below lists the implementation of a queue using arrays.

```
#include "stdio.h"
#define QUEUE_LENGTH 50
struct queue
{ int element[QUEUE_LENGTH];
  int front, rear, choice,x,y;
}
struct queue q;
main()
{
int choice,x;
printf ("enter 1 for add and 2 to remove element front the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)
  {
```

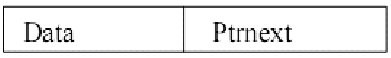
Notes

```
case 1 :
 printf ("Enter element to be added :");
scanf("%d",&x); Queues
add(&q,x);
break;
case 2 :
delete();
break;
 }
}
add(y)
{
++q.rear;
if (q.rear < QUEUE_LENGTH)
 q.element[q.rear] = y;
else
 printf("Queue overflow")
}
delete()
{
if q.front > q.rear printf("Queue empty");
else{
 x = q.element[q.front];
 q.front++;
 }
return x;
}
```

10.1.2 Linked List Implementation of a queue

The basic element of a linked list is a "record" structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node.

Figure 10.4: Structure of a node

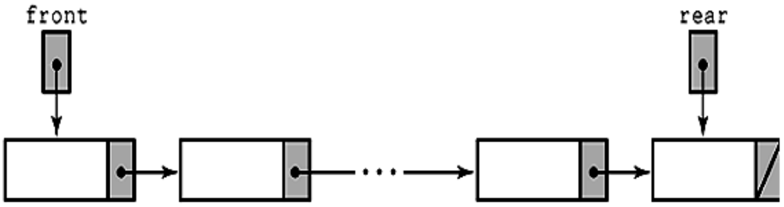


Notes The data component may contain data of any type. Ptrnext is a reference to the next element in the queue structure.

Notes

Figure 10.5 depicts the linked list representation of a queue.

Figure 10.5: A linked list representation of a Queue



Self Assessment Questions

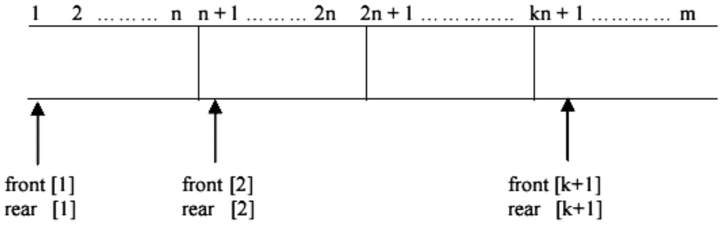
Fill in the blanks:

1. The rule followed in a queue is that elements are added at the .... and come off of the front of the queue.
2. The stack and the queue differ only in the position where the .... can be added or deleted.
3. The object that holds the data and refers to the next element in the list is called a ....

10.2 Representation of Multiple Queues

So far, we have seen the representation of a single queue, but many practical applications in computer science require several queues. Multiqueue is a data structure where multiple queues are maintained. This type of data structures are used for process scheduling. We may use one dimensional array or multidimensional array to represent a multiple queue.

Figure 10.6: Multiple queues in an array



A multiqueue implementation using a single dimensional array with m elements is depicted in Figure 10.6. Each queue has n elements which are mapped to a liner array of m elements.

Self Assessment Questions

State True or False:

4. Linear queue is a data structure where multiple queues are maintained.
5. Multiqueues are used for process scheduling.

10.3 Representation of a Circular Queue

One of the major problems with the linear queue is the lack of proper utilization of space. Suppose that the queue can store 100 elements and the entire queue is full. So, it means that the queue is holding 100 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full. In this way, space utilization in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

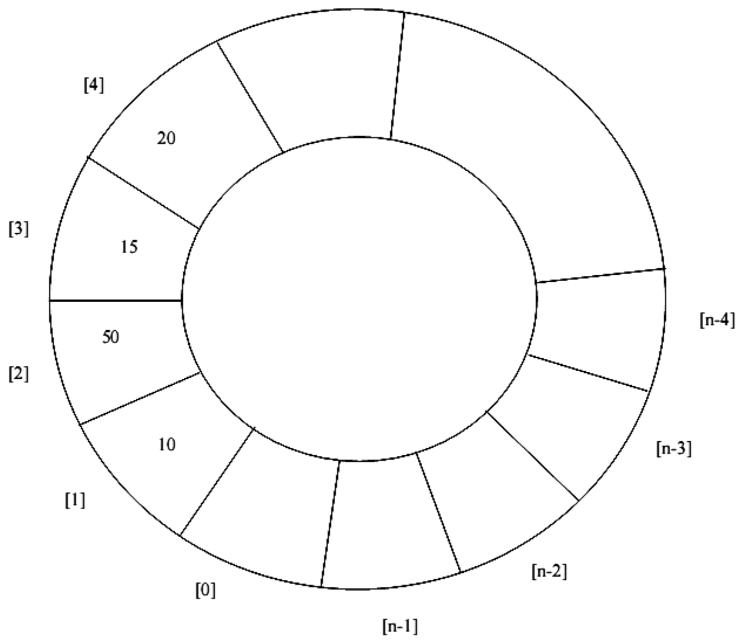
The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with n elements starts from index 0and ends at n-1.So, clearly , the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1(both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing ton-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilization of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element anti-clock wise. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1.

Caution The circular queue is empty in case both front and rear point to the same index.

Figure 10.7 depicts a circular queue.

Figure 10.7: A Circular Queue (Front = 0, Rear = 4)



Algorithm for Addition of an element to the circular queue:

Step-1: If "rear" of the queue is pointing to the last position then go to step-2 or else Step-3

Step-2: make the "rear" value as 0

Step-3: increment the "rear" value by one

Step-4: a. if the "front" points where "rear" is pointing and the queue holds a not

NULL value for it, then its a “queue overflow” state, so quit; else go to step-b

b. add the new value for the queue position pointed by the “rear”

Algorithm for deletion of an element from the circular queue:

Step-1: If the queue is empty then say “queue is empty” and quit; else continue

Step-2: Delete the “front” element

Step-3: If the “front” is pointing to the last position of the queue then go to step-4 else go to step-5

Step-4: Make the “front” point to the first position in the queue and quit

Step-5: Increment the “front” position by one.

10.3.1 Array Implementation of a Circular Queue

A circular queue can be implemented using arrays or linked lists. Program given below gives the array implementation of a circular queue.

```
#include “stdio.h”
void add(int);
void deleteelement(void);
int max=10; /*the maximum limit for queue has been set*/
static int queue[10];
int front=0, rear=-1; /*queue is initially empty*/
void main()
{
int choice,x;
printf (“enter 1 for addition and 2 to remove element front the queue
and 3 for exit”);
printf(“Enter your choice”);
scanf(“%d”,&choice);
switch (choice)
{
case 1 :
printf (“Enter the element to be added :”);
scanf(“%d”,&x);
add(x);
break;
case 2 :
deleteelement();
break;
 }
}
void add(int y)
{
 if(rear == max-1)
```

Notes

```
 rear = 0;
 else
 rear = rear + 1;
 if( front == rear && queue[front] != NULL)
 printf(“Queue Overflow”);
 else
queue[rear] = y;
}
void deleteelement()
{
int deleted_front = 0;
 if (front == NULL)
 printf(“Error - Queue empty”);
 else
 {
deleted_front = queue[front];
queue[front] = NULL;
 if (front == max-1)
 front = 0;
 else
 front = front + 1;
 }
}
```

Task Compare and contrast linear queue and circular queue.

10.3.2 Linked List Implementation of a Circular Queue

Linked list representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers. Program given below gives the linked list representation of a circular queue.

```
#include “stdio.h”
 struct cq
{ int value;
 int *next;
 };
typedef struct cq *cqptr
cqptr p, *front, *rear;
main()
{
int choice,x;
/* Initialise the circular queue */
```

Notes

```
cqptr = front = rear = NULL;
printf ("Enter 1 for addition and 2 to delete element from the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)
{
case 1 :
 printf ("Enter the element to be added :");
 scanf("%d",&x);
 add(&q,x);
 break;
case 2 :
 delete();
break;
 }
}
/************ Add element *****************/
add(int value)
{
 struct cq *new;
 new = (struct cq*)malloc(sizeof(queue));
 new->value = value
 new->next = NULL;
 if (front == NULL)
 {
 cqptr = new;
 front = rear = queueptr;
 }
 else
 {
 rear->next = new;
 rear=new;
 }
}
/* *************** delete element ***********/
delete()
{
int delvalue = 0;
if (front == NULL)
{ printf("Queue is empty");
 delvalue = front->value;
 if (front->next==NULL)
 {
 free(front);
 queueptr = front = rear = NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr = front;
}
}
}
```

Self Assessment Questions

Fill in the blanks:

6. In a .... queue, front will point to one position less to the first element anti-clock wise.

7. .... representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers.

10.4 Priority Queue

A priority queue is a collection of elements where each element is assigned a priority and the order in which elements are deleted and processed is determined from the following rules: An element of higher priority is processed before any element of lower priority.

Two elements with the same priority are processed according to the order in which they are added to the queue.

Example: An example of a priority queue can be time sharing system : programs of high priority are processed first, and programs with the same priority form a standard queue.

10.4.1 Array Implementation of Priority Queue

One way to represent priority queue is through arrays.

If an array is used to store the elements of priority queue then insertion is easy but deletion of elements would be difficult. This is because while inserting elements in the priority queue they are not inserted in an order. As a result, deleting an element with the highest priority would require examining the entire array to search for such an element.

Did u know? An element in a queue can be deleted from the front end only.

The array element of priority queue can have the following structure:

```
struct data
{
 int item;
 int priority;
 int order;
}
```

The structure holds the type of data item, priority of the element and the order in which the element has been added.

Algorithm on inserting element in priority queue.

```
PQInsert (M, Item)
Step 1 Find the Row M
Step2 [Reset the Rear Pointer]
 If Rear[M] = N-1 then
 Rear[M] = 0
 Else
 Rear[M] = Rear[M]+1
Step 3 [Overflow]
 If Front[M] = Rear[M] then
 Write ("This Priority Queue is full")
 Return
Step 4 [Insert Element]
 Q[M] [Rear[M]] = then
Step 5 [Is Front Pointer Properly Set]
 If Front[M] = -1 then
 Front[m] = 0
 Return
Step 6 Exit
Algorithm deleting element from Priority Queue

PQDelete (K, Item)
Step 1 Initialize K = 0
Step 2 while (Front[K] = -1)
 K = K+1
[To find the first non empty queue]
Step 3 [Delete Element]
   Item = Q[K] [Front[K]
Step 4 [Queue Empty]
 If Front[K] = N-1 then
 Front[K] = 0
 Else
 Front[K] = Front[K]+1
```

```
  Return Item
Step 6 Exit
```

10.4.2 Linked List Implementation of Priority Queue

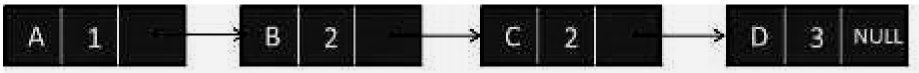
Another way to represent a priority queue in memory is by means of one-way list:

- Each node in the list contains three items of information - an information field INFO, a priority number PRNO and the link NEXT.
- Node A will precede Node B in the list when A has higher priority than B or when both the nodes have same priority but A was added to the list before B.

Caution The order in one-way list corresponds to the order of priority queue.

Figure 10.8 shows priority queue with 4 elements where B and C have same priority numbers.

Figure 10.8: Priority Queue with 4 Elements



Source: http://www.eshikshak.co.in/index.php/data-structure/queue/priority-queue

Self Assessment Questions

Fill in the blanks:

8. A .... is a collection of elements where each element is assigned a priority.

9. An element of higher priority is processed before any element of .... priority.

10. Two elements with the same priority are processed according to the .... in which they are added to the queue.

11. Deleting an element with the .... priority would require examining the entire array to search for such an element.

12. The .... of priority queue holds the type of data item, priority of the element and the order in which the element has been added.

10.5 Representation of Dequeue

Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends. Like a linear queue and a circular queue, a dequeue can also be implemented using arrays or linked lists.

10.5.1 Array Implementation of a Dequeue

If a Dequeue is implemented using arrays, then it will suffer with the same problems that a linear queue had suffered. Program given below gives the array implementation of a Dequeue.

```
#include "stdio.h"
#define QUEUE_LENGTH 10;
```

Notes

```
int dq[QUEUE_LENGTH];
int front, rear, choice,x,y;
main()
{
 int choice,x;
 front = rear = -1; /* initialize the front and rear to null i.e empty queue */
 printf (“enter 1 for addition and 2 to remove element from the front of the queue”);
 printf (“enter 3 for addition and 4 to remove element from the rear of the queue”);
 printf(“Enter your choice”);
 scanf(“%d”,&choice);
 switch (choice)
 {
 case 1:
 printf (“Enter element to be added :”);
 scanf(“%d”,&x);
 add_front(x);
 break;
 case 2:
 delete_front();
 break;
 case 3:
 printf (“Enter the element to be added :”);
 scanf(“%d “,&x);
 add_rear(x);
 break;
 case 4:
 delete_rear();
 break;
 }
}
/**************** Add at the front ***************/
add_front(int y)
{
if (front == 0)
{
 printf(“Element can not be added at the front”);
 return;
else
 {
 front = front - 1;
```

Notes

```
dq[front] = y;
 if (front == -1 ) front = 0;
 }
 }
/**************** Delete from the front ***************/
delete_front()
{
 if front == -1
 printf("Queue empty");

else
 return dq[front];
 if (front = = rear)
 front = rear = -1
 else
 front = front + 1;
}
/**************** Add at the rear ***************/
add_rear(int y)
if (front == QUEUE_LENGTH -1 )
{
 printf("Element can not be added at the rear ")
 return;
else
{
 rear = rear + 1;
 dq[rear] = y;
 if (rear = = -1 )
 rear = 0;
}
}
/**************** Delete at the rear ***************/
delete_rear()
{
 if rear == -1
 printf("deletion is not possible from rear");
 else
 {
 if (front = = rear)
 front = rear = -1
 else
 { rear = rear - 1;
```

```
  return dq[rear];
 }
}
}
```

10.5.2 Linked List Implementation of a Dequeue

Double ended queues are implemented with doubly linked lists.

> *Notes* A doubly link list can traverse in both the directions as it has two pointers namely left and right. The right pointer points to the next node on the right where as the left pointer points to the previous node on the left.

Program given below gives the linked list implementation of a Dequeue.

```
# include “stdio.h”
#define NULL 0
struct dq {
 int info;
 int *left;
 int *right;
 };
typedef struct dq *dqptr;
dqptr p, tp;
dqptr head;
dqptr tail;
main()
{
 int choice, I, x;
dqptr n;
dqptr getnode();
 printf(“\n Enter 1: Start 2 : Add at Front 3 : Add at Rear 4: Delete at
Front 5:
Delete at Back”);
while (1)
{
 printf(“\n 1: Start 2 : Add at Front 3 : Add at Back 4: Delete at Front
5: Delete
at Back 6 : exit”);
 scanf(“%d”, &choice);
 switch (choice)
 {
 case 1:
 create_list();
 break;
 case 2:
 eq_front();
 break;
 case 3:
eq_back();
break;
 case 4:
dq_front();
break;
 case 5:
dq_back();
break;
 case 6 :
exit(6);
 }
 }
}
create_list()
{
 int I, x;
 dqptr t;
 p = getnode();
 tp = p;
 p->left = getnode();
 p->info = 10;
 p_right = getnode();
 return;
}
dqptr getnode()
{
 p = (dqptr) malloc(sizeof(struct dq));
 return p;
}
dq_empty(dq q)
{
 return q->head = = NULL;
}
eq_front(dq q, void *info)
{
 if (dq_empty(q))
 q->head = q->tail = dcons(info, NULL, NULL);
```

Notes

```
else
{
 q-> head -> left =dcons(info, NULL, NULL);
 q->head -> left ->right = q->head;
 q ->head = q->head ->left;
}
}
eq_back(dq q, void *info)
{
 if (dq_empty(q))
 q->head = q->tail = dcons(info, NULL, NULL)
 else
{
 q-> tail -> right =dcons(info, NULL, NULL);
 q->tail -> right -> left = q->tail;
 q ->tail = q->tail ->right;
}
}
 dq_front(dq q)
{
 if dq is not empty
{
dq tp = q-> head;
void *info = tp -> info;
q ->head = q->head-> right;
free(tp);
if (q->head = = NULL)
 q -> tail = NULL;
else
 q -> head -> left = NULL;
return info;
}
}
dq_back(dq q)
{
 if (q!=NULL)
 {
 dq tp = q-> tail;
 *info = tp -> info;
 q ->tail = q->tail-> left;
 free(tp);
 if (q->tail = = NULL)
q -> head = NULL;
 else
 q -> tail -> right = NULL;
 return info;
 }
}
```

Notes

Task Analyze the use of dequeue.

Self Assessment Questions

Fill in the blanks:

13. .... is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends.

14. Double ended queues are implemented with ....

15. A doubly link list can traverse in both the directions as it has two ....

10.6 Summary

- The rule followed in a queue is that elements are added at the rear and come off of the front of the queue.
- Queue has two ends. One is front from where the elements can be deleted and the other if rear to where the elements can be added.
- The stack and the queue differ only in the position where the elements can be added or deleted.
- The basic element of a linked list is a "record" structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node.
- Multiqueue is a data structure where multiple queues are maintained. This type of data structures are used for process scheduling.
- In a circular queue, front will point to one position less to the first element anti-clock wise.
- Linked list representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers.
- A priority queue is a collection of elements where each element is assigned a priority.
- Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends.

10.7 Keywords

Queue: A queue is a data collection in which the items are kept in the order in which they were inserted.

Node: The object that holds the data and refers to the next element in the list is called a node.

Multiqueue: Multiqueue is a data structure where multiple queues are maintained.

Notes

Circular queue: A circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.

Priority queue: A priority queue is a collection of elements where each element is assigned a priority.

Dequeue: Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends.

Record: A record is a special type of data structure that collects different data types that define a particular structure.

One-dimensional array: The array which is used to represent and store data in a linear form is called as 'single or one dimensional array.'

10.8 Review Questions

1. Explain the representation of Linear Queue with example.
2. Describe the implementation of a queue using arrays.
3. What is a multiple queue? Discuss the representation of Multiple Queues
4. Explain the concept of representing circular queues?
5. What are the different methods used for the implementation of circular queue? Explain.
6. Explain the concept of priority queues with example.
7. Illustrate the process of inserting element in priority queue.
8. Illustarte the Linked List Implementation of Priority Queue with example.
9. Describe the array and linked list implementation of a dequeue.
10. Make distinction between priority queue and dequeue.

Answers: Self Assessment

1. rear
2. Elements
3. node
4. False
5. True
6. Circular
7. Linked list
8. priority queue
9. lower
10. Order
11. highest
12. Structure
13. Dequeue
14. doubly linked lists
15. pointers

10.9 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.thelearningpoint.net/computer-science/data-structures-queues—with-c-program-source-code

http://www.cs.ucf.edu/courses/cop3502/spr07/730/implementation.pdf

http://jpkc.seiee.sjtu.edu.cn/ds/ds2/Course%20lecture/chapter%203.pdf

http://www.cs.cmu.edu/~wlovas/15122-r11/lectures/09-queues.pdf

Notes

Unit 11: Operations and Applications of Queues

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss various operations on queues
- Explain Enqueue and Dequeue operation
- Discuss applications of queues

Introduction

The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is the first to be served. As pointed out earlier, Abstract Data Types describe the properties of a structure without specifying an implementation in any way. Thus, an algorithm which works with a "queue" data structure will work wherever it is implemented. Different implementations are usually of different efficiencies. In this unit, we will discuss operations on queues. Also, we will discuss various applications of queues.

11.1 Operations on Queues

Queue is a data structure that maintain "First In First Out" (FIFO) order. And can be viewed as people queueing up to buy a ticket. In programming, queue is usually used as a data structure for BFS (Breadth First Search).

In multiuser system, there will be request from different users for CPU time. The operating system puts them in queue and they are disposed on FIFO(First In, First Out) basic. We will discuss the queue data structure and the operations that it allows in this section. Similar to stack operations, operations that can be carried out In a queue are:

- Create a queue.
- Check whether a queue is empty (Is empty): Checks whether the queue is empty. (If Is empty is true, then the addition of an item is possible. If this is false, then acknowledge with the message "Queue is empty").

Notes

- Check whether a queue is full (Is full): Checks whether the queue is full. (If Is full is true, then the removal of an item is possible. If this is false, then acknowledge with the message "Queue is full").
- Add item at the rear of the queue(enqueue).
- Remove item from front of queue(dequeue).
- Read the front of the queue.
- Print the entire queue.

Notes The primitive isEmpty(Q) is required to know whether the queue is empty or not, because calling next on an empty queue should cause an error. Like stack, the situation may be such when the queue is "full" in the case of a finite queue. But we avoid defining this as it would depend on the actual length of the Queue defined in a specific problem.

Task Make distinction between Is empty and Is full operation.

11.1.1 Enqueue Operation

This operation is used to add an item to the queue at the rear end. So, the head of the queue will be now occupied with an item currently added in the queue. Head count will be incremented by one after addition of each item until the queue reaches the tail point.

Caution This operation will be performed at the rear end of the queue.

The following code below will state the Enqueue operation through a function:-

```
enqueue()
{
int item;
if (rear==MAX-1) /* QUEUE FULL condition */
printf(“Queue Overflow, Cannot insert more elements\n”);
else
{
if (front==-1) /*If queue is initially empty */
front=0;
printf(“Enter the data element that is to be inserted in queue: “);
scanf(“%d”, &item);
rear=rear+1; /* incrementing rear for holding the index of added ele*/
queue[rear] = item ;
}
}
```

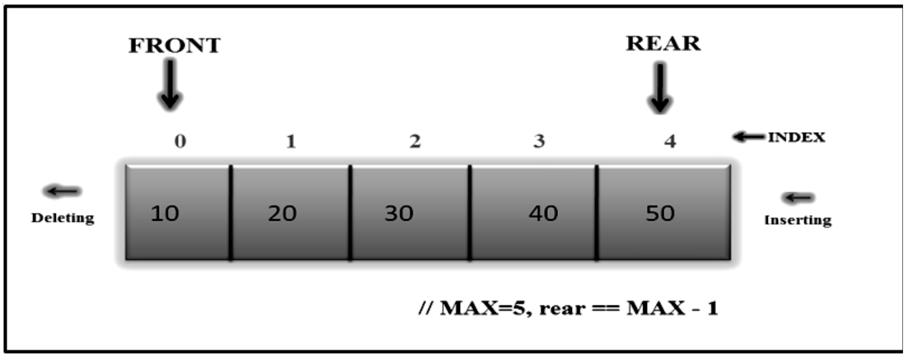
Notes

The code written above shows the enqueue operation on a queue, i.e. inserting elements to it. Line marks the beginning of the enqueue function. Line 3 declares a variable item that will supposedly hold the value that is to be inserted.

> *Notes* Although we can even approach in a different way, we could have even passed the element to be inserted as an argument to the function, well let's proceed with our current approach.

Line 4 is the condition for a full queue or we can say queue overflow. Logically speaking if the value of rear is equal to MAX-1 (stating the maximum number of elements a queue can accept), than it obviously states that the queue is full. Line 5 displays the appropriate message if the condition is true.

Figure 11.1: Enqueue Operation on a Queue



Line 8 is the part if the queue overflow condition is false, another if condition is encountered and it checks if the queue is empty or queue underflow. The variable queue is initially initialized as -1 which states that the queue is empty. When the first element is inserted the queue is initialized to zero. Hence since its time for inserting the first element, we initialize the queue as zero if the queue is empty. Line 11 accepts the data that is to be inserted and saves it in variable item. Line 12 increments the rear variable since it has to be incremented every time we will insert an element. We insert the element to be added on to the array queue[].

11.1.2 Dequeue Operation

This operation is used to remove an item from the queue at the front end. Now the tail count will be decremented by one each time when an item is removed from the queue until the queue reaches the head point.

> *Caution* This operation will be performed at the front end of the queue.

The dequeue operation is shown by the dequeue function which helps to delete an element:-

```
1.   dequeue()
2.   {
3.   if (front == -1 || front > rear) /* queue empty condition */
4.   {
```

```
5.   printf("Queue Underflow\n"); return ;
6.   }
7.   else
8.   {
9.   printf("Element deleted from queue is : %d\n", queue[front]); front++;
10.  }
11.  }
```

Line 1 marks the beginning of the queue function. Line 3 states the condition of queue underflow.

Did u know? Whenever we are deleting elements the queue is incremented.

Gradually after deleting all elements if the value of queue has exceeded rear, than it means that the queue is empty. Line 5 prints the appropriate message. The other condition displays the element that is deleted and we know that the first element to be inserted is deleted. The front variable holds the index of the first element and it's deleted. Line 11 increments the front variable eventually.

Example: The source code below implements the enqueue and dequeue operations on a queue.

```
# include<stdio.h>
# define MAX 5
int queue[MAX];
int rear = -1,front = -1;
void main()
{
  int ch;
  while(1)
  {
     printf("1.Enqueu\n");
     printf("2.Dequeue\n");
     printf("3.Display\n");
     printf("4.EXIT\n");
     printf("Enter your choice : ");
     scanf("%d",&ch);
     switch(ch)
     {
      case 1 :
       enqueue();
      break;
      case 2 :
       dequeue();
```

Notes

```
            break;
            case 3:
             display();
            break;
            case 4:
             exit(1);
             default:
              printf("Wrong choice\n");
          }
        }
      }
      enqueue()
      {
        int item;
        if (rear==MAX-1)            /* QUEUE FULL condition */
           printf("Queue Overflow, Cannot insert more elements\n");
        else
        {
          if (front==-1)            /*If queue is initially empty */
          front=0;
           printf("Enter the data element that is to be inserted in queue: ");
           scanf("%d", &item);
          rear=rear+1;                      /* incrementing rear for holding the
      index of added ele*/
          queue[rear] = item ;
        }
      }
      dequeue()
      {
        if (front == -1 || front > rear) /* queue empty condition */
        {
           printf("Queue Underflow\n");
          return ;
        }
        else
        {
           printf("Element deleted from queue is : %d\n", queue[front]);
          front++;
        }
      }
      display()
      {
```

```
  int i;
  if (front == -1)
     printf("Queue is empty\n");
  else
  {
     printf("Queue is :\n");
      for(i=front;i<= rear;i++)
      printf("%d ",queue[i]);
     printf("\n");
  }
}
```

Let us now consider the following example of using the operation Enqueue and Dequeue.

Example: A queue of an array A[3] is initialized and now the size of the queue is 3 which represents the queue can hold a maximum of 3 items. The enqueue operation is used to add an item to the queue.

Enqueue (3) – This will add an item called 3 to the queue in the front end. (Front count will be incremented by 1).

Again we are adding another item to the queue.

Enqueue (5) – This will add an item called 5 to the queue (Front count will be incremented by 1)

Again we are adding the last item to the queue.

Enqueue (8) – This will add an item called 8 to the queue (Now the queue has reached its maximum count and hence no more addition of an item is possible in the queue, as the queue has reached the rear end).

Now dequeue operation is performed to remove an item from the queue.

Dequeue () – This will remove the item that has been added first in the queue, i.e., the item called 3 by following the concept of FIFO. Now the queue consists of the remaining items 5 and 8 in which the object 5 will be in the front end. The dequeue operation continues until the queue is reaching its last element.

Task Illustrate the working of dequeue operation with example.

Self Assessment Questions

Fill in the blanks:

1. Queue is a data structure that maintain .... order.
2. Queue is usually used as a data structure for ....
3. .... operation checks whether the queue is empty.
4. .... operation checks whether the queue is full.
5. The primitive isEmpty(Q) is required to know whether the queue is empty or not, because calling next on an empty queue should cause an ....

6. .... Operation is used to add an item to the queue at the rear end.
7. Head count will be incremented by one after addition of each item until the queue reaches the .... point.
8. .... operation is used to remove an item from the queue at the front end.
9. The dequeue operation continues until the queue is reaching its .... element.

11.2 Applications of Queues

The application areas where the queue concept is implemented are discussed as below:

- Print queue – Jobs sent to the printer
- Operating system maintain queue in allocating the process to each unit by storing them in buffer.

Did u know? An operating system always makes use of a queue (ready queue, waiting queue) for scheduling processes or jobs.

- The interrupts that occurs during the system operation are also maintained in queue
- The allocation of memory for a particular resources (Printer, Scanner, any hand held devices) are also maintained in queue.
- Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.
- When a resource is shared among multiple consumers.

  Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.

  Examples include IO Buffers, pipes, file IO, etc.
- All the input output calls made by the disk to and from the memory are handled by a queue.
- We have many processors like 80386 and queue plays an active part in there architecture.
- Queues can be commercially used in online business applications for processing customer requests in first in first serve manner.
- When a resource like CPU is shared between among multiple consumers, a queue solves the hatchet.
- In all the basic algorithms for searching or sorting, everyone of them make use of queue.
- Queues are very efficient when we want to take out elements in the same order we have put in inside.

Self Assessment Questions

State True or False:

10. Operating system maintain queue in allocating the process to each unit by storing them in buffer.

11. The allocation of memory for a particular resources are maintained in queue.
12. An operating system does not makes use of a queue for scheduling processes or jobs.
13. Queue is used when data is transferred synchronously between two processes.
14. When a resource like CPU is shared between among multiple consumers, a queue solves the hatchet.
15. All the input output calls made by the disk to and from the memory are handled by a stack.

Case Study

Case:

C Program to Enqueue and Dequeue in a Circular Queue using Singly Linked List

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
typedef struct nodes
{
       int x;
        struct nodes *next;
}node;
//Function Declarations
void enqueue(node **,node **, int);//Use call by reference node **
is pointer to pointer
int dequeue(node **,node **);
void display(node *);
void main()
{
              node *front=NULL, *rear=NULL;
             int choice,element;
            clrscr();
           do
          {
                    printf("\n1. Enqueue\n2. Dequeue\n3.Print Linked
List\n4.Exit\nChoice : ");
                   scanf("%d",&choice);
                  switch(choice)
                 {
                      case 1:
                                    printf("Enter the element to insert : ");
                                   scanf("%d",&element);

                                                    enqueue(&front,&rear,element);/
```

Contd...

Notes

```
/Function call
                        break;
               case 2:
                          element = dequeue(&front,&rear);
                          printf("The deleted element is :
%d",element);
                       break;
               case 3:
                        display(front);
                       break;
                          default:
                        printf("Enter the right choice : ");
                  }//switch
           }while(choice !=4);
}//main

void enqueue(node **front, node **rear, int element)
{
     node * temp,*current;
      current = (node*)malloc(sizeof(node*));
     current->x = element;
     current->next = NULL;
     if(*front == NULL)
    {
            *front = *rear = current;
     }
      else
    {
            temp = *rear; //use pointer to structure to access
structure members
            temp->next=current;
            temp=temp->next;
           *rear=temp;
     }
} //enqueue

int dequeue(node **front,node **rear)
{
      node *temp;
      int element;
       if(*front == *rear) //Deleting last node
    {
```

Contd...

```
            temp = *front;
            element = temp->x;
            free(*front);
            *front = *rear = NULL;
    }
    else
    {
           temp = *front;
           element = temp->x;
           temp = temp->next;
           free(*front);
          *front = temp;
}
return(element);
}//dequeue

void display(node *front)
{
         node *current;
         current = front;
          printf("Front =%u",front);
         while(current != NULL)
       {
              printf(" Data = %d, Next = %u",current->x,current->next);
             current = current->next;
         }//while
}//display
```

Question:

Write a C program to insert and delete elements from a Queue.

Source: http://datastructuresinterview.blogspot.in/2013/02/c-program-to-enqueue-and-dequeue-using.html

11.3 Summary

- Is empty operation checks whether the queue is empty. (If Is empty is true, then the addition of an item is possible.
- Is full operation checks whether the queue is full.
- Enqueue operation is used to add an item to the queue at the rear end. So, the head of the queue will be now occupied with an item currently added in the queue.
- When the first element is inserted the queue is initialized to zero.
- Dequeue operation is used to remove an item from the queue at the front end.

- The dequeue operation continues until the queue is reaching its last element.
- Operating system maintain queue in allocating the process to each unit by storing them in buffer.
- Queues are very efficient when we want to take out elements in the same order we have put in inside.

11.4 Keywords

Queue: Queue is a data structure that maintain "First In First Out" (FIFO) order.

Is empty: Is empty operation checks whether the queue is empty. (If Is empty is true, then the addition of an item is possible.

Is full: Is full operation checks whether the queue is full.

Enqueue: Enqueue operation is used to add an item to the queue at the rear end.

Dequeue: Dequeue operation is used to remove an item from the queue at the front end.

FIFO: FIFO (first-in, first-out) is an approach to handling program work requests from queues or stacks so that the oldest request is handled next.

BFS: Breadth-first search (BFS) is a general technique for traversing a graph.

11.5 Review Questions

1. Discuss various operations carried out in a queue.
2. Explain the use of Is empty operation and Is full operation.
3. Explain the concept of Enqueue operation with example.
4. Write a program illustrating enqueue operation on a queue.
5. Describe dequeue operation with example.
6. Make distinction between enqueue and dequeue operation.
7. The dequeue operation continues until the queue is reaching its last element. Comment.
8. Discuss various applications of queues.
9. Disxuss the use of queues in business applications.
10. Write a program illustrating enqueue and dequeue operation on a queue.

Answers: Self Assessment

1. First In First Out (FIFO)
2. BFS (Breadth First Search).
3. Is empty
4. Is full
5. error
6. Enqueue
7. tail
8. Dequeue
9. last
10. True
11. True
12. False
13. False
14. True
15. False

Notes

11.6 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.thefluentadmin.com/operations-on-queues-in-c/

http://www.cprograms.in/Queue/insert-delete-queue.html

http://sourcecode4u.com/categories/data-structures/183-write-c-programs-that-implement-queue-its-operations-using-pointers

http://world-of-c-programming.blogspot.in/2012/11/queue-operations.html

Unit 12: Introduction to Trees

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of trees
- Explain Binary Tree and its representation
- Discuss traversal of Binary tree

Introduction

All data structures provide a way to organize data. Different structures serve different purposes. Everyone is familiar with the concept of a family tree. In fact, a nice exercise would be to print the family tree for an individual whose family history is stored in the data base of that case study. The family tree is actually embedded within the lists containing the family history. Trees may be viewed as a special case of lists in which no intertwining or sharing takes place. They are the basis for structures and operations used in all aspects of programming, from the structure of programs for compilers, to work in data processing, information retrieval, and artificial intelligence. Trees are ubiquitous; they seem to sprout everywhere—even in the field of computer science! In this unit, we will discuss the concept of trees and binary trees. We will also discuss the traversal of binary trees.

12.1 Concept of Trees

Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion. Assume a Tree representing your family structure. Let say we start with your Grand parent; then come to your parent and finally, you and your siblings.

Let us discuss the terminology related to trees.

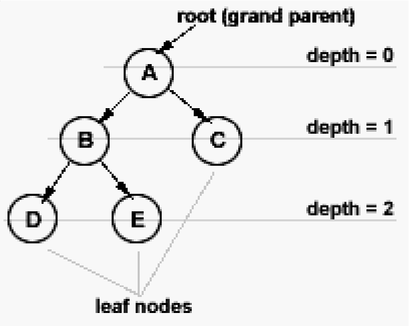
- A node is a structure which may contain a value, a condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.
- Nodes that do not have any children are called leaf nodes. They are also referred to as terminal nodes.
- The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path). This is commonly needed in the manipulation of the various self balancing trees, AVL Trees in particular. Conventionally, the value -1 corresponds to a subtree with no nodes, whereas zero corresponds to a subtree with one node.
- The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. (In the formal definition, each such path is also unique). In diagrams, it is typically drawn at the top.

Notes In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

- An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.
- A subtree of a tree T is a tree consisting of a node in T and all of its descendants in T. (This is different from the formal definition of subtree used in graph theory.) The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Example: Consider the tree given below.

Figure 12.1:Tree



Source: http://datastructures.itgo.com/trees/concept.htm

As you can see, the top most element is called the "root". The nodes with no children are called "Leaf" nodes. Here, if we say "D" is representing you then "E" becomes your sibling; "B"

Notes

becomes your parent; "A" becomes your ancestor (grand parent) and "C" is sibling of your parent. If we call A as the root then B onwards is one sub tree and C onwards is another.

Did u know? A sub-tree can have zero or more sub-trees in it.

Every tree has a depth and height factor associated. These factors are associated with every node in the tree. The depth of a particular node tells you how many generations down, that node is from the Root. Similarly, height of a particular node tells you its distance from the deepest leaf (or leaves) in that branch (or sub tree).

Example: In figure 12.1, D and E are at height = 0; C is at height = 0; and A is at height = 2.

The height of a Tree is nothing but the height of its root. Both of these factors have their own importance at times.

Task Compare and contrast root node and internal node.

Self Assessment Questions

Fill in the blanks:

1. .... is a data structure which allows you to associate a parent-child relationship between various pieces of data.
2. A .... is a structure which may contain a value, a condition, or represent a separate data structure.
3. Nodes that do not have any children are called ....
4. .... is the node at which operations on the tree commonly begin.
5. A .... of a tree T is a tree consisting of a node in T and all of its descendants in T.
6. The .... of a particular node tells you how many generations down, that node is from the Root.

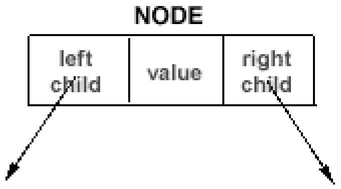
12.2 Binary Tree

Any Tree whose nodes can have at the most two children is called a Binary tree or a tree with order 2. Every node in a Binary tree has a structure like this :

```
Struct NODE
 {
  struct NODE *leftchild;
  Int nodevalue;              /*this can be of any type*/
  struct NODE *rightchild;
 }
```

The binary tree appears as shown in the figure given below.

Figure 12.2: Representation of Binary Tree



Source: http://datastructures.itgo.com/trees/concept.htm

As you can see, 'leftchild' and 'rightchild' are pointers to another tree-node. The "leafnode" will have NULL values for these pointers.

12.2.1 Binary Tree Creation

The binary tree creation follows a very simple principle — for the new element to be added, compare it with the current element in the tree. If its value is less than the current element in the tree then move towards the left side of that element or else to its right. If there is no sub tree on the left, make your new element as the left child of that current element or else compare it with the existing left child and follow the same rule. Exactly same has to done for the case when your new element is greater than the current element in the tree but this time with the right child.

The algorithm is given as below:

1. ***Step-1:*** If Value of New element < current element then go to step-2 or else step-3
2. ***Step-2:*** If the Current element does not have a left sub-tree then make your new element the left child of the current element; else make the existing left child as your current element and go to step-1
3. ***Step-3:*** If the current element does not have a right sub-tree then make your new element the right child of the current element; else make the existing right child as your current element and go to step-1

C implementation is given as below.

```
struct NODE
{
  struct NODE *left;
  int value;
  struct NODE *right;
}
create_tree( struct NODE *curr, struct NODE *new )
{
  if(new->value <= curr->value)
   {
     if(curr->left != NULL)
         create_tree(curr->left, new);
    else
       curr->left = new;
```

Notes

```
   }
  else
   {
     if(curr->right != NULL)
        create_tree(curr->right, new);
    else
       curr->right = new;
   }
}
```

12.2.2 Binary Tree Representation

There are two ways by which we can represent a binary tree.

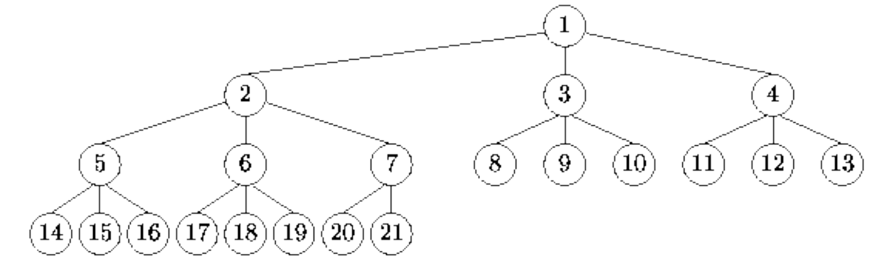
- Linked representation of a binary tree
- Array representation of a binary tree

Array representation of binary trees

When a binary tree is represented by arrays three separate arrays are required. One array stores the data fields of the trees. The other two arrays represents the left child and right child of the nodes.

Consider the binary tree given below:

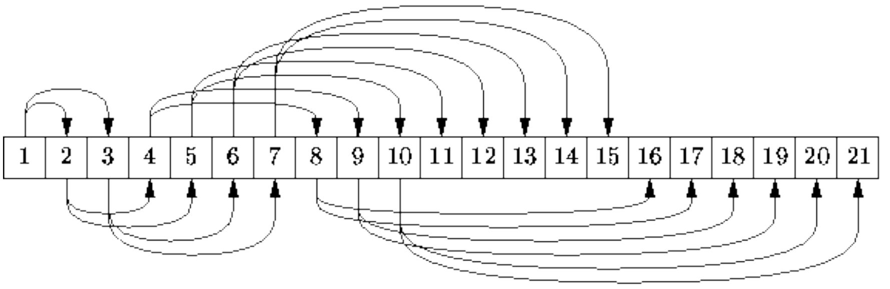
Figure 12.3:Binary Tree



Source: http://www.ustudy.in/node/7015

Now the Array representation of this binary tree is given below:

Figure 12.4: Array Representation of the Binary Tree given in Figure 12.3



Source: http://www.ustudy.in/node/7015

Following program show how a binary tree can be represented using arrays:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
struct node *left;
char data;
struct node *right;
};
struct node *buildtree(int);
void inorder(struct node);
char arr[]={‘a’,’b’,’c’,’d’,’e’,’f’,’g’,\0’,’\0’,’\h’};
int ic[]={1,3,5,-1,9,-1,-1,-1-,-1,-1};
int rc[]={2,4,6,-1,-1,-1-,-1,-1,-1,-1};
void main()
{
struct node *root;
clrscr();
root=buildtree(0);
printf(“in-order traversal:\n”);
inorder(root);
getch();
}
struct node *buildtree(it index)
{
struct node *temp=null;
if(index!=-1)
{
temp=(struc node *)malloc (sizeof(struct node));
temp->left=buildtree(lc[index]);
temp->data=arr[index];
temp->right=buildtree(rc[index]);
}
return temp;
}
void inorder(struct node *root)
{
if (root!=null)
{
inorder(root->left);
printf(“%c]t”,root->data);
```

Notes

```
inorder(root->right);
}
}
```

Linked representation of binary trees

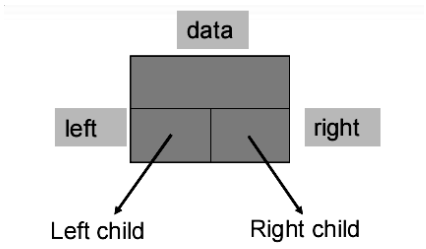
Binary trees can be represented by links, where each node contains the address of the left child and the right child. These addresses are nothing but kinds to the left and right child respectively.

Caution A node that does not have a left or a right child contains a NULL value in its link fields.

Linked structures uses space more efficiently and provides more flexibility. In order to implement a binary tree using a linked structure, each node should have:

- A data component
- Two link components:
- left, pointing to the left child
- right, pointing to the right child

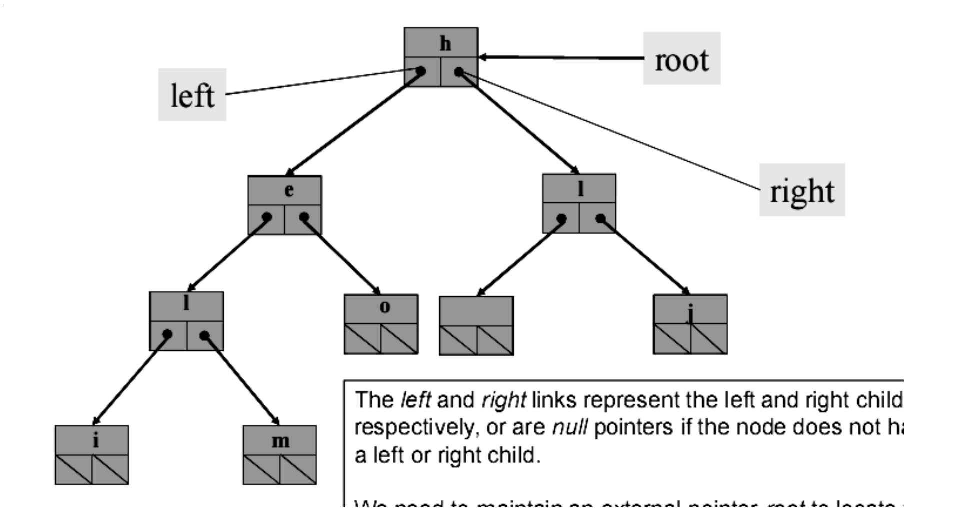
Figure 12.5: Linked Structure of a Binary Tree



Source: http://www.cs.tut.fi/~prog2/material/Lec8.pdf

The linked representation is shown as below:

Figure 12.6: Linked Representation


h
root
left
e
l
right
l
o
j
i
m
The *left* and *right* links represent the left and right child respectively, or are *null* pointers if the node does not h
a left or right child.
We need to maintain an external pointer root to locate

Source: http://www.cs.tut.fi/~prog2/material/Lec8.pdf

The left and right links represent the left and right children respectively, or are null pointers if the node does not have a left or right child.

We need to maintain an external pointer root to locate the root of the tree.

Self Assessment Questions

Fill in the blanks:

7. Any tree whose nodes can have at the most two children is called a ....

8. Binary trees can be represented by links, where each node contains the ....of the left child and the right child.

9. A node that does not have a left or a right child contains a ....value in its link fields.

10. The left and right links represent the ....if the node does not have a left or right child.

12.3 Traversal of a Binary Tree

The process of systematically visiting all the nodes in a tree and performing some computation at each node in the tree is called a tree traversal. When we want to visit each and every element of a tree, there are three different ways to do the traversal, that is, preorder, inorder and postorder.

Caution These traversal methods are limited to Binary trees only and not for any other tree.

The methods differ primarily in the order in which they visit the root node the nodes in the left sub-tree and the nodes in the right sub-tree.

Notes A binary tree is of recursive nature, i.e. each sub-tree is a binary tree itself. Hence the functions used to traverse a tree using these methods can use recursion.

To traverse a non-empty binary tree in pre-order, we need to perform the following three operations:

- Visit the root
- Traverse the left sub-tree in pre-order
- Traverse the right sub-tree in pre-order

To traverse a non-empty binary tree in in-order, we need to perform the following three operations:

- Traverse the left sub-tree in in-order.
- Visit the root
- Traverse the right sub-tree in in-order

Notes

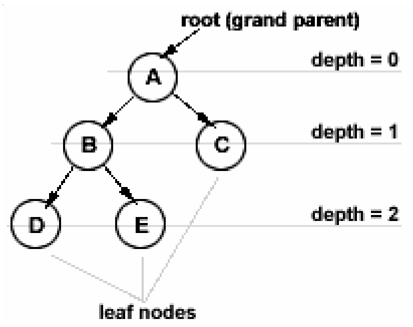
To traverse a non-empty binary tree in post-order, we need to perform the following three operations:

- Traverse the left sub-tree in post-order.
- Traverse the right sub-tree in post-order
- Visit the root

In all the three methods nothing needs be done to traverse an empty binary tree.

Example:

Figure 12.7: Tree Traversal



Source: http://datastructures.itgo.com/trees/traversal.htm

In case of INORDER traversal, we start from the root i.e A. We are supposed to visit its left sub-tree then visit the node itself and its right sub-tree. Here, A has a left sub-tree rooted at B. So, we move to B and check for its left sub-tree. Again, B has a left sub-tree rooted at D. So, we check for D's left sub-tree now, but D doesn't have any left sub-tree and thus we will visit node D first and check for its right sub-tree. As D doesn't have any right sub-tree, we'll track back and visit node B; and check for its right sub-tree. B has a right sub-tree rooted at E and so we move to E. Well, E doesn't have any left or right sub-trees so we just visit E and track back to B. We already have visited B, so we track back to A. We are yet to visit the node itself and so we visit A first; then we go for checking the right sub-tree of A, which is rooted at C. As C doesn't have any left or right tree we visit C. So, the INORDER becomes - D B E A C.

Similarly, in PREORDER, we visit the current node first, then we visit its left sub-tree and then its right sub-tree. In POSTORDER, for every node, we visit the left sub-tree first and then the right sub-tree and finally the node itself.

An algorithm is given as below:

1. ***Step-1:*** For the current node check whether it has a left child. If it has then go to step-2 or else step-3
2. ***Step-2:*** Repeat step-1 for this left child
3. ***Step-3:*** Visit (i.e printing in our case) the current node
4. ***Step-4:*** For the current node check whether it has a right child. If it has then go to step-5
5. ***Step-5:*** Repeat step-1 for this right child

This is the recursive algorithm for INORDER traversal.

C implementation is given as below.

```
struct NODE
{
 struct NODE *left;
 int value;
 struct NODE *right;
}
inorder(struct NODE *curr)
{
 if(curr->left != NULL) inorder(curr->left);   /*step-1 & step-2*/
 printf(“%d”, curr->value);    /*step-3*/
 if(curr->right != NULL) inorder(curr->right);   /*step-4*/
}
```

Task Make distinction between preorder and postorder.

12.3.1 Threaded Binary Tree

`Trees->Threaded Binary Tree->Concepts`

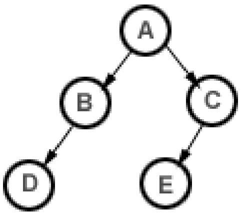
A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

The node structure for a threaded binary tree varies a bit and its like this :

```
struct NODE
{
struct NODE *leftchild;
int node_value;
struct NODE *rightchild;
struct NODE *thread;
}
```

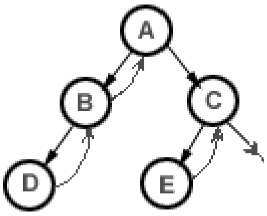
Let's make the Threaded Binary tree out of a normal binary tree.

Figure 12.8: Normal Binary Tree



The INORDER traversal for the above tree is — D B A E C. So, the respective Threaded Binary tree will be:

Figure 12.9: Threaded Binary Tree created from Normal Binary Tree



B has no right child and its inorder successor is A and so a thread has been made in between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread.

12.3.2 Non-recursive traversal using a Threaded Binary Tree

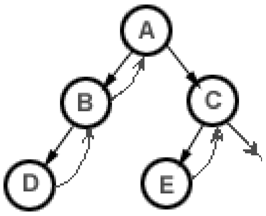
As this is a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree.

We will consider the INORDER traversal again. Here, for every node, we'll visit the left sub-tree (if it exists) first (if and only if we haven't visited it earlier); then we visit (i.e print its value, in our case) the node itself and then the right sub-tree (if it exists).

Did u know? If the right sub-tree is not there, we check for the threaded link and make the threaded node the current node in consideration.

Example:

Figure 12.10: right sub-tree missing



	List of visited nodes:	INORDER:
step-1: 'A' has a left child i.e B, which has not been visited. So, we put B in our "list of visited nodes" and B becomes our current node in consideration. and finally..... D B E A C	B	
step-2: 'B' also has a left child, 'D', which is not there in our list of visited nodes. So, we put 'D' in that list and make it our current node in consideration.	B D	
step-3: 'D' has no left child, so we print 'D'. Then we check for its right child. 'D' has no right child and thus we check for its thread-link. It has a thread going till node 'B'. So, we make 'B' as our current node in consideration.	BD	D
step-4: 'B' certainly has a left child but its already in our list of visited nodes. So, we print 'B'. Then we check for its right child but it doesn't exist. So, we make its threaded node (i.e 'A') as our current node in consideration.	BD	DB
step-5: 'A' has a left child, 'B', but its already there in the list of visited nodes. So, we print 'A'. Then we check for its right child. 'A' has a right child, 'C' and its not there in our list of visited nodes. So, we add it to that list and we make it our current node in consideration.	BDC	DBA
step-6: 'C' has 'E' as the left child and its not there in our list of visited nodes even. So, we add it to that list and make it our current node in consideration.	B D C E	D B A
		D B E A C

Algorithm is given as below:

1. ***Step-1:*** For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.
2. ***Step-2:*** Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.
3. ***Step-3:*** For the current node check whether it has a right child. If it has then go to step-4 else go to step-5
4. ***Step-4:*** Make that right child as your current node in consideration. Go to step-6.
5. ***Step-5:*** Check for the threaded node and if its there make it your current node.
6. ***Step-6:*** Go to step-1 if all the nodes are not over otherwise quit.

C implementation is given below:

```
struct NODE
{
  struct NODE *left;
  int value;
```

Notes

```
 struct NODE *right;
 struct NODE *thread;
}
inorder(struct NODE *curr)
{
 while(all the nodes are not over )
   {
      if(curr->left != NULL &&  ! visited(curr->left))
       {
         visit(curr->left);
         curr = curr->left;
       }
     else
       {
        printf("%d", curr->value);
        if(curr->right != NULL)
           curr = curr->right;
       else
            if(curr->thread != NULL)
                curr = curr->thread;
       }
   }
}
```

The looping condition has to be implemented. You can use your own logic for that. The functions - visited( ) and visit( ) have to be written. visit( ) maintains a linked list of already visited nodes. visited( ) returns a TRUE value if it finds a particular node, in the list maintained by visit( ), otherwise FALSE.

Self Assessment Questions

Fill in the blanks:

11. The process of systematically visiting all the nodes in a tree and performing some computation at each node in the tree is called a ....
12. A binary tree is of..........................nature.
13. In ...., we visit the current node first, then we visit its left sub-tree and then its right sub-tree.
14. A .... Binary Tree is a binary tree in which every node that does not have a right child has a THREAD to its INORDER successor.
15. If the right sub-tree is not there, we check for the threaded link and make the threaded node the .... node in consideration.

Case Study

Binary Tree Plants its Roots in Cloud Computing to Accelerate Partner Training

Binary Tree is a leading provider of software for migrating from Exchange 2003/ 2007 and Lotus Notes to on-premises and online versions of Exchange and SharePoint. Binary Tree selected Skytap to accelerate and improve the delivery of its partner certi?cation courses, and provide instant access to demos and proof-of-concept projects.

Situation

Binary Tree is represented by business partners worldwide who provide specialized services and a proven methodology for guiding customers through complex IT transitions. The company has been an avid user of cloud computing services for many years, and decided to evaluate the cloud to improve its partner enablement and certi?cation process. Prior to its adoption of the cloud for training, Binary Tree leveraged a combination of physical and virtual environments to provide partner certi?cation training. This process was expensive, time consuming, and put the onus of ? exibility on the partner because training classes were only offered on a limited basis. In Q1 2010, Traci Blowers, Director of Alliances and Business Development, started investigating cloud solutions that could automate the provisioning of classroom environments and enable her team to:

- Accelerate training delivery by quickly spinning up classroom environments to numerous students at one time
- Provide self-service access to virtual classrooms and training sandboxes - no matter where the partner was located
- Improve the partner enablement and certi?cation experience

Challenges

Prior to Skytap, Binary Tree's certi?cation classes were instructor led and calendared against the availability of Binary Tree's subject matter experts (SMEs). As a result, classes were limited to one class per quarter. Adding to this challenge, the system could only accommodate eight students at a time. If a minimum of ?ve students did not register, classes were cancelled. Binary Tree also struggled to provide partners with hands-on skills training prior to certi?cation. To become certi?ed, partners were required to schedule non-billable shadowing engagements under the supervision of Binary Tree's SMEs. This often resulted in up to six weeks of billable time lost due to delays in scheduling the SMEs. In addition to the dif?culty scheduling these "shadow sessions," partners were required to pull billable consultants off engagements to sit through multi-day instruction courses before becoming certi?ed. The situation was equally challenging for Binary Tree because SME instructors could not be billed out during class time.

Binary Tree also struggled to provide partners with access to virtual images for classroom training. Binary Tree SMEs had to create and update materials regularly, which involved crafting virtual images on drives and uploading the images to FTP servers. As a result of these challenges, Binary Tree experienced signi? cant delays in enablement and certi?cation, impacting the company's ability to build a worldwide channel of delivery-certi?ed consultants. Traci Blowers quickly recognized that Binary Tree needed to identify a better solution.

Contd...

Notes

Solution

After a thorough evaluation of the marketplace, Traci Blowers selected Skytap based on its ability to quickly and cost-effectively support its partner enablement and certification courses. "We saw the Skytap Cloud as a means of providing our partners with 24x7 worldwide access to standardized training resources," said Traci Blowers. "We can now enable any student to become certified without having to leverage a Binary Tree instructor or execute a nonbillable shadowing exercise."

Skytap provides Binary Tree SMEs with immediate access to virtual machines for knowledge development, enablement and certification. The company is currently utilizing Skytap across its worldwide client base, and initiatives for Microsoft and the Microsoft Partner Channel. These groups leverage Skytap for virtual, interactive classrooms that help partners gain certifications and/or execute messaging coexistence and migration proof-of-concept projects.

Using Skytap, Binary Tree created a centralized library of existing classroom images based on its current training machines. As a result, Binary Tree has eliminated the need to constantly upload images to its FTP server. "Our SMEs can use Skytap's self-service web UIto provision new classrooms and run multiple classes instantly." Because Skytap is a cloud based service, Binary Tree can scale the number of virtual classrooms to match our business needs." Students can securely access the classroom for which they are authorized from anywhere in the world. Classes can be scheduled to run at specific times enabling Binary Tree to provide a new level of flexibility to its partners. Once a class is completed, Binary Tree can remove or clone student environments for later use.

Benefits

Since adopting Skytap Cloud for its partner training, demo and proof of concept deployments, Binary Tree has received a tremendous amount of positive feedback. Binary Tree partner, Microsoft, views the cloud as a new means of scaling the Binary Tree process globally and offering effortless coexistence demonstrations and migration proof-of-concept projects. "We are experiencing significant demand from our worldwide partners for training material to help them meet the demand they are seeing from customers looking to migrate to the Microsoft Platform," said Terry Birdsell, Director for Productivity & Enterprise Sales at Microsoft, a Binary Tree partner. "The portal Binary Tree has created will go a long way towards sharing the invaluable knowledge they have gathered after years of successful migration and coexistence projects."

Additionally, Binary Tree has seen a tremendous increase in the number of global partners leveraging its new CMT University offering and CMT Sandbox, which enables hands-on skill set building to become certified for delivery. "Instead of asking partners to participate in a single calendared training event, we now enable them to become certified at their convenience and on their timeline." Furthermore, only one administrator is needed to monitor requests, saving significant time and budget.

Question:

Discuss how does the representation of binary tree accelerates partner training.

Source: http://www.skytap.com/downloads/case-studies/skytap_casestudy_binarytree.pdf

12.4 Summary

- Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, data and files in a hierarchical fashion.
- A node is a structure which may contain a value, a condition, or represent a separate data structure.
- The top most element is called the "root". The nodes with no children are called "Leaf" nodes.
- Any Tree whose nodes can have at the most two children is called a Binary tree or a tree with order 2.
- The binary tree creation follows a very simple principle — for the new element to be added, compare it with the current element in the tree.
- When a binary tree is represented by arrays three separate arrays are required. One array stores the data fields of the trees.
- Binary trees can be represented by links, where each node contains the address of the left child and the right child.
- The process of systematically visiting all the nodes in a tree and performing some computation at each node in the tree is called a tree traversal.
- A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor.

12.5 Keywords

Tree: Tree is a data structure which allows you to associate a parent-child relationship between various pieces of data.

Node: A node is a structure which may contain a value, a condition, or represent a separate data structure.

Leaf nodes: Nodes that do not have any children are called leaf nodes.

Root node: The topmost node in a tree is called the root node.

Internal node: An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

Binary tree: Any Tree whose nodes can have at the most two children is called a Binary tree or a tree with order 2.

Tree traversal: The process of systematically visiting all the nodes in a tree and performing some computation at each node in the tree is called a tree traversal.

Threaded Binary Tree: A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD to its INORDER successor.

12.6 Review Questions

1. Explain the concept of trees with example.
2. Discuss the terminologies related to trees.

3. What is a binary tree? Discuss the structure of a binary tree with example.
4. Discuss the process of creating binary tree with example.
5. Illustrate the representation of binary trees using arrays.
6. Describe Linked representation of binary trees with example.
7. What are the different ways of tree traversal? Discuss.
8. In PREORDER, we visit the current node first, then we visit its left sub-tree and then its right sub-tree. Comment.
9. Illustrate the steps used for traversing a non-empty binary tree in post-order.
10. Describe the concept of non-recursive traversal using a Threaded Binary Tree.

Answers: Self Assessment

1. Tree
2. Node
3. leaf nodes
4. Root node
5. subtree
6. Depth
7. Binary tree
8. Address
9. NULL
10. null pointers
11. tree traversal
12. Recursive
13. PREORDER
14. Threaded
15. current

12.7 Further Readings

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://ideainfo.8m.com/

http://claymore.engineer.gvsu.edu/~jackh/books/analysis/pdf/trees.pdf

http://www.cs.auckland.ac.nz/~jmor159/PLDS210/trees.html

http://www.i-programmer.info/babbages-bag/477-trees.html

Unit 13: Sorting

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of Insertion Sort
- Discuss the concept of Selection Sort
- Discuss the concept of Merge Sort
- Discuss the concept of Radix Sort
- Discuss the concept of Hashing

Notes

Introduction

Sorting in general refers to various methods of arranging or ordering things based on criterias (numerical, chronological, alphabetical, heirarchial etc.). Sorting the elements is an operation that is encountered very often in the solving of problems. For this reason, it is important for a programmer to learn how sorting algorithms work. A sorting algorithm refers to putting the elements of a data set in a certain order, this order can be from greater to lower or just the opposite, and the programmer determines this. In practice, the most common order types are those of numbers and strings (chars) or lexicographical order. Having an array of elements, a1, a2, ... , an, it is required to sort the elements that the following condition is assured:a1 < = a2 < = a3 < = ... < = ai <= aj < = ... < = an. Due to obvious reasons, Sorting (of data) is of immense importance and is one of the most extensively researched subjects. It is one of the most fundamental algorithmic problems. So much so that it is also fundmental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc. It is estimated that around 25% of all CPU cycles are used to sort data. There are many approaches to sorting data and each has its own merits and demerits.

13.1 Insertion Sort

Insertion sort is an elementary sorting algorithm. The insertion sort is commonly compared to organizing a handful of playing cards. You pick up the random cards one at a time. As you pick up each card, you insert it into its correct position in your hand of organized cards. The insertion sort maintains the two sub-arrays within the same array. At the beginning of the sort, the first element of the first sub-array is considered the "sorted array". With each pass through the loop, the next element in the unsorted second sub-array is placed into its proper position in the first sorted sub-array.

Did u know? The insertion sort can be very fast and efficient when used with smaller arrays. Unfortunately, it loses this efficiency when dealing with large amounts of data.

In an insertion sort, all the keys in a given list are assumed to be in random order before sorting. That is, all possible orderings of the keys within the list are equally likely.

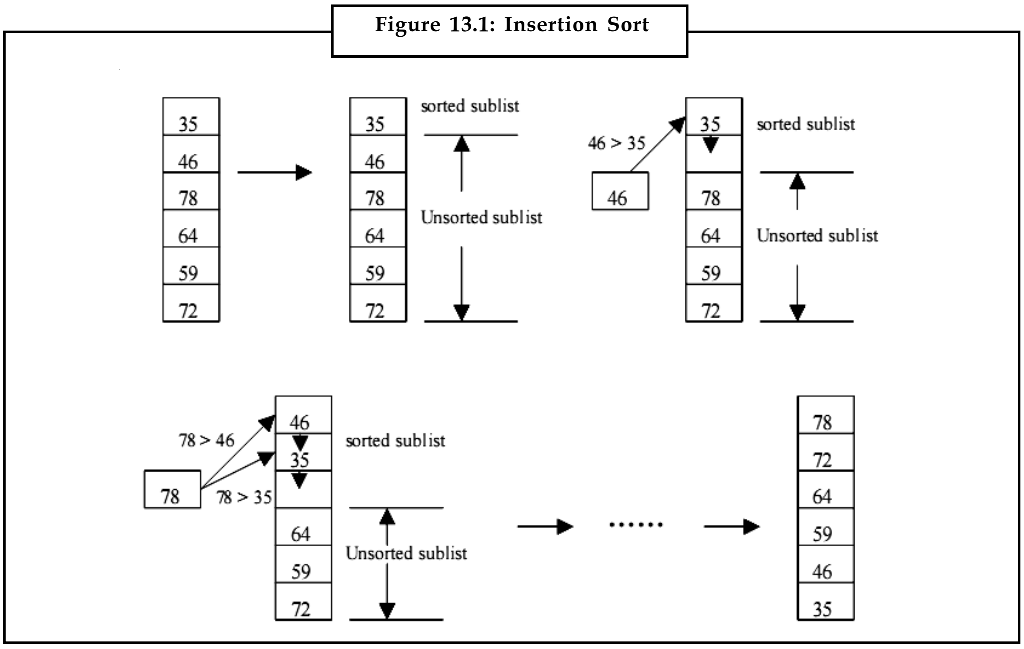
1. First, imagine an item is to be inserted to an "empty" list, which is actually the original list.

2. Then, the first item in the original list is defaulted into the first position in the list. This item is now said to be in a sorted list. The sublist from the second item to the end is regarded as the unsorted list. This means that the original list is divided into sorted and unsorted sublists.

3. Then, the first (next) item of the unsorted list (i.e. the second item of the original list) is inserted into the sorted sublist. At this time, the sorted sublist is expanding while the unsorted sublist is shrinking. And so it goes — step 3 repeats itself until the original list is exhausted.

The following figure shows a list of integers being sorted in descending order to explain this algorithm.

Figure 13.1: Insertion Sort



Source: http://ee.sjtu.edu.cn/po/Class-web/data-stucture/csc120ch9.pdf

On average, there are $n^2/4 + O(n) = O(n^2)$ for both the number of comparisons of keys and the number of assisgnments of entries. For the best case, there are (n-1) comparisons which is of O(n) because they have already been sorted.

Now let us see the step by step example:

Example: Having the following list, let's try to use insertion sort to arrange the numbers from lowest to greatest:

Unsorted list: 6 **4** 5 10 3 1 8 9 7 2

Step 1:

4 6 **5** 10 3 1 8 9 7 2 - As 6 was the first element, at step 1, the element 4 is being inserted, and 4 is smaller than 6, so it must place before 6, so the order would be 4 6.

Step 2:

4 5 6 **10** 3 1 8 9 7 2 - At step 2, 5 was the element to be inserted, and being smaller than 6, it goes before it, but then it is compared to 4 which is smaller than 5, now the order would be 4 5 6.

Step 3:

4 5 6 10 **3** 1 8 9 7 2 - At step 3, 10 was the new element, and being greater than 6, the element was in the right place and no moving had to take place, now the order would be 4 5 6 .

Step 4:

3 4 5 6 10 **1** 8 9 7 2 - At step 4, 3 was the new element, and compared to 10 it is smaller, so it has to move to the left, also, it is compared to 6 then to 5 and then to 4, and is smaller than any of these, so the position of the element 3 is at the left of all these numbers, now the order would be 3 4 5 6.

Notes

Step 5:

1 3 4 5 6 10 **8** 9 7 2 - Step 5, the new element is 1, which is again smaller than 10, then like in the previous steps, it is compared to the next numbers 6,5,4,3 and is smaller than an of these, so it ends at the left of the array that has been sorted out so far, with the order being 1 3 4 5 6

Step 6:

1 3 4 5 6 8 10 **9** 7 2 - At step 6, the element 8 was compared to 10 and moved to the left, but compared to 6 it is greater, so now the list would look like 1 3 4 5 6 8

Step 7:

1 3 4 5 6 8 9 10 **7** 2 - Step 7, the element 9 was compared to 10 and moved to the left having it being smaller, but it is greater than 8 so the list is : 1 3 4 5 6 8 9

Step 8:

1 3 4 5 6 7 8 9 10 **2** - Step 8, the element of 7 was compared to 10, so it had to move to the left of 10, also 7 is smaller than 8 which it moves another position to the left, but it is greater than 6, so it found its position, with the array looking like : 1 3 4 5 6 7 8 9

Step 9:

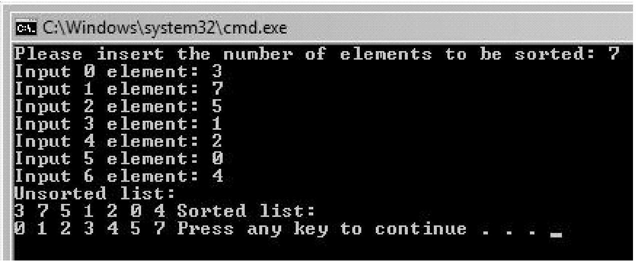
1 2 3 4 5 6 7 8 9 10 - At the final step 9, the element 2 was compared to 10, being smaller it moves to the left, then again it is compared to 9, then 8, 7, 6, 5, 4, 3 and is smaller than any of these numbers, so the right position is to the left, but being greater than 1, it's at one position to the right of the element 1, now the final order is the right one : 1 2 3 4 5 6 7 8 9 10

Having no more elements to compare 10 to, the algorithm now knows that the list is sorted.

The following is an implementation in C.

```
#include < iostream >
using namespace std;
int main(void)
{
         int i;
          long int A[100];
          long int n=100;
           cout << “Please insert the number of elements to be sorted: “;
          cin >> n;         // The total number of elements
          for(int i=0;i < n;i++)
         {
                   cout << “Input “ << i << “ element: “;
                    cin >> A[i]; // Adding the elements to the array
         }
          cout << “Unsorted list:” << endl;         // Displaying the
unsorted array
          for(int i=0;i < n;i++)
         {
                   cout << A[i] << “ “;
         }
           for (int k=1;k < n;k++)
         {
                 int temp=A[k];  // the k element to be inserted is
retained in a temporal variable,
                i = k-1;                //at first it is the A[1] value,
which is the second value of the array
                while (i >=0 && A[i] > temp)
               {
                       A[i+1] = A[i];
                      i = i-1;
               }
                A[i+1] = temp;
        }
          cout << "Sorted list:" << endl; // Displaying the sorted array
         for(int i=0;i < n;i++)
        {
                cout << A[i] << " ";
        }
        return 0;
}
```

The output is shown as below:



```
Please insert the number of elements to be sorted: 7
Input 0 element: 3
Input 1 element: 7
Input 2 element: 5
Input 3 element: 1
Input 4 element: 2
Input 5 element: 0
Input 6 element: 4
Unsorted list:
3 7 5 1 2 0 4 Sorted list:
0 1 2 3 4 5 7 Press any key to continue . . .
```

The temp variable retains the value to be inserted, which at the first run of the loop would be A[1], being the second value as A[0] is the first value. Then the while loop checks to see if A[0] is greater than A[1], and if it is true, a swap between the values takes place. Also, i is decremented by 1. At the next run, A[2] is compared to A[1], and if it is greater a swap takes place, and after that it is compared to A[0], and if its greater there is a swap between the values, if not, the proper order has been found out.

How many compares are done? 1+2+...+(n-1) which results in O(n^2) worst case and (n-1)* 1 which is O(n) on best case. Also, how many element shifts are done? 1+2+...+(n-1) on O(n^2) worst case and 0 , O(1) best case.

13.1.1 Advantages and Disadvantages of Insertion Sort

The advantages of Insertion Sort are:

- it is easy to learn;
- few lines of code;
- efficient for small data sets;
- stable, does not change the relative order of elements with equal keys;

Notes

Disadvantages of Insertion Sort are:

- not effective for large numbers of sorting elements;
- needs a large number of element shifts;
- as the number of elements increases the performance of the program would be slow.

Self Assessment Questions

State true or false:

1. The insertion sort maintains the two sub-arrays within the same array.
2. Insertion sort is not efficient for small data sets.

13.2 Selection Sort

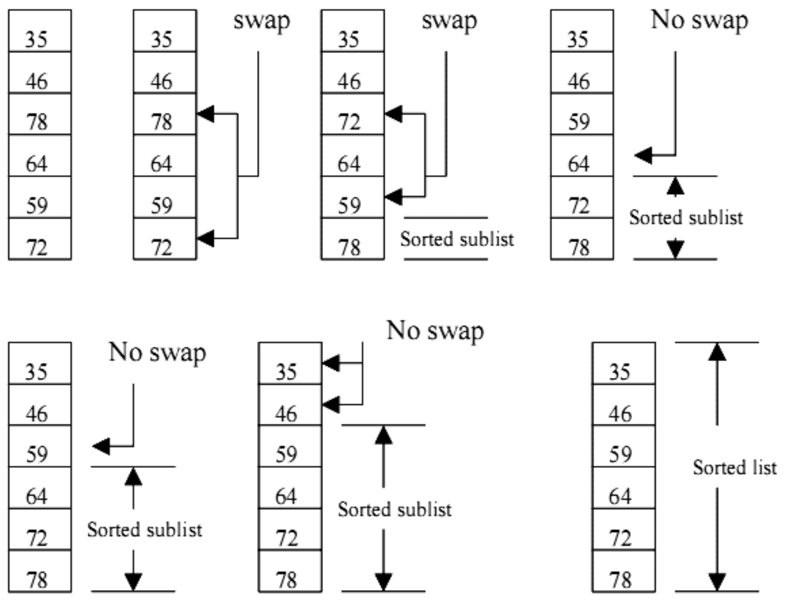
Selection sort, also called naive (selection) sort, is an in-place comparison sort. It may look pretty similar to insertion sort, but it performs worse. It is a quite simple sorting algorithm, and may perform better than more complicated algorithms in particular cases, for example in the ones where the auxiliary memory is limited.

Like insertion sort, all the keys in the list for a selection sort are assumed to be in random order before sorting, and there are two sublists of sorted and unsorted keys during the sorting process. This is different from the insertion sort in that no extra variable is required to aid the comparisons, and the movement of data is limited to one-to-one exchange.

Selection sort first finds the smallest number in the array and exchanges it with the element from the first position, then it finds the second smallest number and exchanges it with the element from the second position, and so on until the entire list is sorted.

The selection sort is demonstrated by sorting a list of integers in ascending order as in the following figure.

Figure 13.2: Selection Sort



Source: http://ee.sjtu.edu.cn/po/Class-web/data-stucture/csc120ch9.pdf

Now let us see the step by step example:

Example: Having the following list, let's try to use selection sort to arrange the numbers from lowest to greatest:

6, 3, 5, 4, 9, 2, 7.

First run:

2, 3, 5, 4, 9, 6, 7 (2 was the smallest number and 6 which was the first element were swapped)

Second run:

2, 3, 4, 5, 9, 6, 7 (4 and 5 were swapped, as 3 was already on the good position, with no other element being smaller)

Third run:

2, 3, 4, 5, 6, 9, 7 (6 and 9 were swapped, 5 was already in the good position)

Forth run:

2, 3, 4, 5, 6, 7, 9 (7 and 9 were swapped)

Fifth run:

2, 3, 4, 5, 6, 7, 9 (no swap)

Sixth run:

2, 3, 4, 5, 6, 7, 9 (no swap)

Even though in the last two runs there were no swaps, the algorithm still makes passes, of a total of n-1 passes, where n is the number of elements.

The big-O notation for this algorithm is of O(n^2), which is of the same order as the insertion sort.

Caution The condition of applying these sorting algorithms is dependent on speeds of comparison associated with data moving .

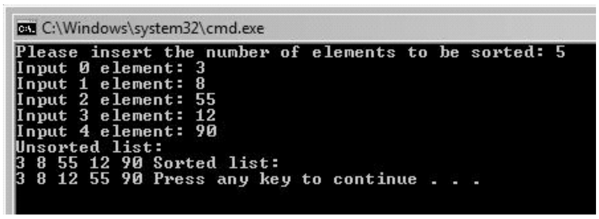
The below is an implementation of the algorithm in C.

```
#include < iostream >
using namespace std;
int main()
{
          long int n=100;
          long int a[100];
           cout << "Please insert the number of elements to be sorted: ";
          cin >> n;         // The total number of elements
          for(int i=0;i < n;i++)
         {
                   cout << "Input " << i << " element: ";
                    cin >> a[i]; // Adding the elements to the array
         }
```

Notes

```
          cout << “Unsorted list:” << endl;              // Displaying the
unsorted array
          for(int i=0;i < n;i++)
         {
                 cout << a[i] << “ “;
         }
          for (int i=n-1;i>1;–i)
         {
                 int locmax=0;
                 int maxtemp=a[0];
                 for (int j=1;j < =i;++j)
                {
                        if(a[j]>maxtemp)
                       {
                              locmax=j;
                              maxtemp=a[j];
                       }
                }
                 a[locmax]=a[i];
                 a[i]=maxtemp;
         }
          cout << “Sorted list:” << endl; // Displaying the sorted array
          for(int i=0;i < n;i++)
         {
                 cout << a[i] << “ “;
         }
          return 0;
}
```

The output is shown as below:



```
Please insert the number of elements to be sorted: 5
Input 0 element: 3
Input 1 element: 8
Input 2 element: 55
Input 3 element: 12
Input 4 element: 90
Unsorted list:
3 8 55 12 90 Sorted list:
3 8 12 55 90 Press any key to continue . . .
```

The algorithm executes n-1 iterations always, no matter if the correct order was achieved earlier or the array was already sorted.

The first element is retained in the maxtemp variable, then the next element is checked with maxtemp to see which is greater, and if this is true, maxtemp retains the next big value and so on. After this has been completed, on the last position, there will be the greatest element from the list, so the greatest element ends on the right position after the first run.

This code example of the algorithm sorts the array from greater to lower, as opposing to the step by step example that I gave earlier.

Clearly, the outer loop runs n-1 times. The only complexity in this analysis is the inner loop. If we think about a single time the inner loop runs, we can get a simple bound by noting that it can never loop more than n times. Since the outer loop will make the inner loop complete n times, the comparison can't happen more than O(n2) times. Also, the same complexity applies to worst case or even best case scenario.

13.2.1 Advantages and Disadvantages of Selection Sort

The advantages of Selection Sort are:

- easy to implement;
- requires no additional storage space.
- it performs well on a small list.

Disadvantages of Selection Sort are

- poor efficiency when dealing with a huge list of items
- its performance is easily influenced by the initial ordering of the items before the sorting process.

Selection sort is sorting algorithm known by its simplicity. Unfortunately, it lacks efficiency on huge lists of items, and also, it does not stop unless the number of iterations has been achieved (n-1, n is the number of elements) even though the list is already sorted.

Self Assessment Questions

Fill in the blanks:

3. ...., also called naive (selection) sort, is an in-place comparison sort.

4. The .... for selection algorithm is of O(n^2), which is of the same order as the insertion sort.

13.3 Merge Sort

Merging means combining elements of two arrays to form a new array. The simplest way of merging two arrays is to first copy all the elements of one array into a new array and then append all the elements of the second array to the new array. If you want the resultant array to be sorted, you can sort it by any of the sorting techniques.

If the arrays are originally in sorted order, they can be merged in such a way as to ensure that the combined array is also sorted. This technique is known as merge sort.

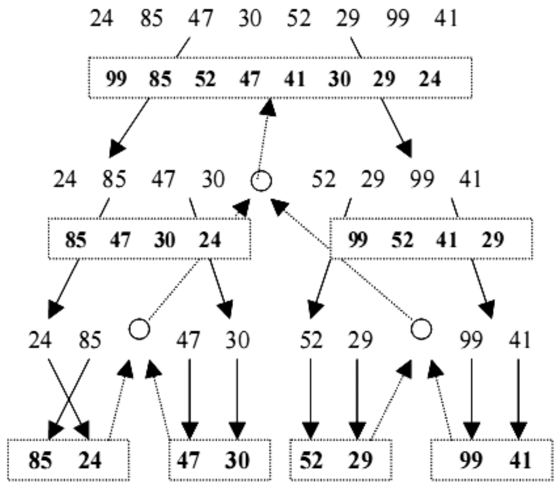
Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

A list of keys for mergesort is divided into two sublists of about the same length. Then, these two sublists are further divided into half each. We keep doing this until each sublist consists of only one key. Then, the way we conquer the sorting is to merge the sublists two pieces by two pieces. According to our order requirement, when we merge two sublists, we put the smaller one in the front for ascending order, or the larger one in the front for descending order. Now, each sublist consists of two keys in order. Then, we merge the sublists two pieces by two pieces again in the same way of ordering. By repeating this process, we finally obtain a sorted list.

The following figure illustrates this algorithm by sorting a list of keys associated with a binary tree in descending order. Each node of the tree represents a recursive call of mergesort, and the list is divided there.

Did u know? When we traverse back to the node, the sublists are merged in the required order.

Figure 13.3: Merge Sort



Source: http://ee.sjtu.edu.cn/po/Class-web/data-stucture/csc120ch9.pdf

Note that ....▶ : It will group 2 sorted sublists and form 1 sorted list

⬚ : Sorted sublist

Let us see a step by step example of merge sort:

Example: Having the following list, let's try to use merge sort to arrange the numbers from lowest to greatest:

Unsorted list: 50, 81, 56, 32, 44, 17, 99

Divide the list in two: the first list is 50, 81, 56, 32 and the second is 44, 17, 99 .

Divide again the first list in two which results: 50, 81 and 56, 32.

Divide one last time, and will result in the elements of 50 and 81. The element of 50 is just one, so you could say it is already sorted. 81 is the same one element so it's already sorted.

Now, It is time to merge the elements together, 50 with 81, and it is the proper order.

The other small list, 56, 32 is divided in two, each with only one element. Then, the elements are merged together, but the proper order is 32, 56 so these two elements are ordered.

Next, all these 4 elements are brought together to be merged: 50, 81 and 32, 56. At first, 50 is compare to 32 and is greater so in the next list 32 is the first element: 32 * * *. Then 50 is again compared to 56 and is smaller, so the next element is 50: 32 50 * *. The next element is 81, which is compared to 56, and being greater, 56 comes before, 32 50 56 *, and the last element is 81, so the list sorted out is 32 50 56 81.

We do the same thing or the other list, 44, 17, 99, and after merging the sorted list will be: 17, 44, 99.

The final two sub-lists are merged in the same way: 32 is compared to 17, so the latter comes first: 17 * * * * * *. Next, 32 is compared to 44, and is smaller so it ends up looking like this : 17 32 * * * * *. This continues, and in the end the list will be sorted.

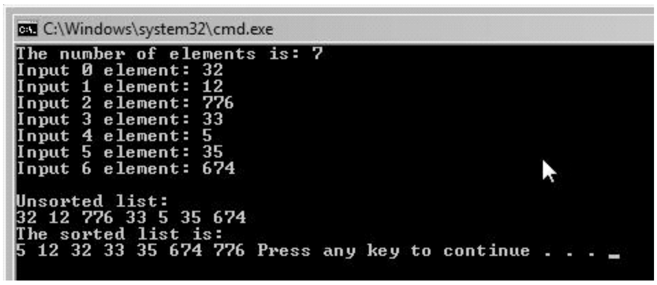
Now let us see the implementation of merge sort in c:

```
#include < iostream >
using namespace std;
void Merge(int *a, int low, int mid, int high)
{
          int i = low, j = mid + 1, k = low;
          int b[100];
          while ((i <= mid) && (j <= high))
          {
                    if (a[i] < a[j])
                    {
                              b[k] = a[i];
                              i++;
                    }
                    else
                    {
                              b[k] = a[j];
                              j++;
                    }
                    k++;
          }
          while (i <= mid)
          {
                    b[k] = a[i];
                    i++;
                    k++;
          }
          while (j <= high)
          {
                    b[k] = a[j];
                    j++;
                    k++;
          }
          for (k = low; k <= high; k++)
                    a[k] = b[k];
}
void MergeSort(int *a, int low, int high)
{
          int mid;
```

Notes

```
         if(low >= high)
                 return;
         else
         {
                 mid =(low + high) / 2;
                  MergeSort(a, low, mid);
                  MergeSort(a, mid+1, high);
                  Merge(a, low, mid, high);
         }
}
int  main()
{
         int *a;
         int n;
          cout << “The number of elements is: “;
         cin>>n;
          a = (int*) calloc (n,sizeof(int));
          for(int i=0;i < n;i++)
         {
                  cout << “Input “ << i << “ element: “;
                   cin >> a[i]; // Adding the elements to the array
         }
          cout << “nUnsorted list:” << endl;        // Displaying the
unsorted array
          for(int i=0;i < n;i++)
         {
                 cout << a[i] << “ “;
         }
          MergeSort(a, 0, n-1);
          cout<<“nThe sorted list is: “ << endl;
          for (int i=0;i < n;i++)
                 cout << a[i] << “ “;
         return 0;
}
```

The output is shown as below:



```
The number of elements is: 7
Input 0 element: 32
Input 1 element: 12
Input 2 element: 776
Input 3 element: 33
Input 4 element: 5
Input 5 element: 35
Input 6 element: 674

Unsorted list:
32 12 776 33 5 35 674
The sorted list is:
5 12 32 33 35 674 776 Press any key to continue . . .
```

Notes

The MergeSort procedure splits recursively the lists into two smaller sub-lists. The merge procedure is putting back together the sub-lists, and at the same time it sorts the lists in proper order, just like in the example from above.

Merge sort guarantees O(n*log(n)) complexity because it always splits the work in half. In order to understand how we derive this time complexity for merge sort, consider the two factors involved: the number of recursive calls, and the time taken to merge each list together.

13.3.1 Advantages and Disadvantages of Merge Sort

Advantages of Merge sort are:

- is stable;
- It can be applied to files of any size.
- Reading through each run during merging and writing the sorted record is also sequential. The only seeking necessary is as we switch from run to run.

Disadvantages of Merge Sort are:

- it requires an extra array;
- is recursive;

Thus we can say that merge sort is a stable algorithm that performs even faster than heap sort on large data sets. Also, due to use of divide-and-conquer method merge sort parallelizes well. The only drawback could be the use of recursion, which could give some restrictions to limited memory machines.

Self Assessment Questions

Fill in the blanks:

5. Merge sort is based on .... method.

6. The .... procedure splits recursively the lists into two smaller sub-lists.

13.4 Radix Sort

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.

Radix sort is a sorting algorithm that sorts the data, integers for example, by splitting the numbers into individual digits and comparing the individual digits sharing the same significant position.

Also, a positional notation is required, because instead of integers there could be strings of characters or floating point numbers.

The radix sort can be classified into 2 types: LSD (least significant digit) or MSD (most significant digit). The LSD sorting type begins from the least significant digit to the most significant digit and the MSD works the other way around.

⚠

Caution Having the number 150, the number 0 is the least significant digit and 1 is the most significant digit.

Notes

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process. In other words:

1. Take the least significant digit of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives:

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.

Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075,090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:

2 (bucket size for digits of 0: 170, 090)

2 (bucket size for digits of 2: 002, 802)

1 (bucket size for digits of 4: 024)

2 (bucket size for digits of 5: 045, 075)

1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:

2 (bucket size for digits of 0: 002, 802)

1 (bucket size for digits of 2: 024)

1 (bucket size for digits of 4: 045)

1 (bucket size for digits of 6: 066)

2 (bucket size for digits of 7: 170, 075)

1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:

6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)

1 (bucket size for digits of 1: 170)

1 (bucket size for digits of 8: 802)

Let us see another example.

Example:

Having the following list, let's try to use radix sort to arrange the numbers from lowest to greatest:

Unsorted list: 12, 8, 92, 7, 100, 500.

Because there are numbers with 3 digits, we can write the initial list like this: 012, 008, 092, 007, 100, 500.

Now by using the LSD, the sorting may begin with step 1:

Write all the numbers that have one of the following digit as the LSD of the numbers from the list:

0:100, 500

1:-

2:012, 092

3:-

4:-

5:-

6:-

7:007

8:008

9:-

Also, if you have two numbers with the same digit like 012 and 092, you write them in the order from the list.

Now, the order of the numbers according to step 1 is: 100, 500, 012, 092, 007, 008, which differs a bit from the initial list.

We repeat step 1, but now we use the second digit :

0:100, 500, 007, 008

1:012

2:-

3:-

4:-

Notes

5:-

6:-

7:-

8:-

9:092

After step 2, the order is: 100, 500, 007, 008, 012, 092.

Step 3, using the MSB:

0:007, 008, 012, 092

1: 100

2:-

3:-

4:-

5: 500

6:-

7:-

8:-

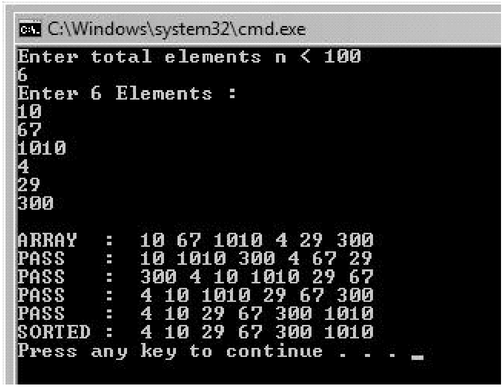
9:-

The final list is the sorted one: 007, 008, 012, 092, 100, 500.

Now let use the implementation of radix sort in c:

```
#include < iostream >
using namespace std;
#define MAX 100
void print(int *a, int n)          //Prints the numbers of an array
{
         int i;
         for (i = 0; i < n; i++)
                  cout << “ “ << a[i];
}
void radixsort(int *a, int n)
{
          int i, b[MAX], m = 0, exp = 1;
         for (i = 0; i < n; i++)
         {
                  if (a[i] > m)
                          m = a[i];
         }
           while (m / exp > 0)
         {
                int bucket[10] ={0};
                for (i = 0; i < n; i++)
                        bucket[a[i] / exp % 10]++;
                for (i = 1; i < 10; i++)
                        bucket[i] += bucket[i - 1];
                for (i = n - 1; i >= 0; i—)
                        b[—bucket[a[i] / exp % 10]] = a[i];
                for (i = 0; i < n; i++)
                       a[i] = b[i];
               exp *= 10;
                cout << "nPASS   : ";
                 print(a, n);     //Prints the results from the first pass
        }
}
  int main()
{
        int arr[MAX];
        int i, n;
         cout << "Enter total elements n < " << MAX << endl;
        cin>>n;
         cout << "Enter " << n << " Elements : " << endl;
        for (i = 0; i < n; i++)
                cin>>arr[i];
         cout<<"nARRAY  : ";
         print(&arr[0], n);
         radixsort(&arr[0], n);
        cout << "nSORTED : ";
         print(&arr[0], n);
        printf("n");
        return 0;
}
```

The output is shown as below:



C:\Windows\system32\cmd.exe

```
Enter total elements n < 100
6
Enter 6 Elements :
10
67
1010
4
29
300

ARRAY  :  10 67 1010 4 29 300
PASS   :  10 1010 300 4 67 29
PASS   :  300 4 10 1010 29 67
PASS   :  4 10 1010 29 67 300
PASS   :  4 10 29 67 300 1010
SORTED :  4 10 29 67 300 1010
Press any key to continue . . .
```

Notes

At the first for loop, the variable m will store the highest number from the array.

At the while loop, the condition is to check if the number still has digits, and while this is true, a bucket (an array) with maxim 10 storage places (the values from 0 to 9) will store the proper numbers, just like in the example from above. At the end of the while loop the exp must increase by a factor of 10 compared to the previous one, and at the next check of the while loop, the condition will be tested to see of the number of digits is greater than 0.

The time complexity of the algorithm is as follows: Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes O(k*n) time. If the numbers are of finite size, the algorithm runs in O(n) asymptotic time.

13.4.1 Advantages and Disadvantages of Radix Sort

The advantages of Radix Sort are:

- if it's implemented in Java, it would be faster than QuickSort or HeapSort;
- is stable, meaning it preserves existing order of equal keys.
- is quite good on small keys.

Disadvantages of Radix Sort are:

- does not work well when you have very long keys, because the total sorting time is proportional to key length and to the number of items to sort.
- you have to write an unconventional compare routine.
- It requires fixed size keys, and some standard way of breaking the keys into pieces.

Task Make distinction between Selection sort and Radix sort.

Self Assessment Questions

Fill in the blanks:

7. .... is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value

8. The .... sorting type begins from the least significant digit to the most significant digit.

13.5 Hashing

Searching an element from a large collection of data is always an exhausting job. There are many searching algorithms and techniques to perform searching operation but to search an element from a large collection of data is a challenging task to complete in efficient time. Hashing is a process of storing a large amount of data and retrieving elements from it in optimal time. Hashing is the solution when we want to search an element from a large collection of data.

For example suppose we store the information of 10,000 students of a university using a structure in C. Every student has a unique id. If we want to search the attributes of a particular student then there are various approaches that can strike our mind for doing this searching operation. A linked list or a binary search tree can be used. An array can be used which can be efficient than the previous two but storing 10,000 students information will need a lot of memory for an array. The best technique to search in efficient time in this type of situation is hashing with minimum wastage of memory.

We need to know about two important features that will be used during hashing:-

- Hash Table
- Hash Function

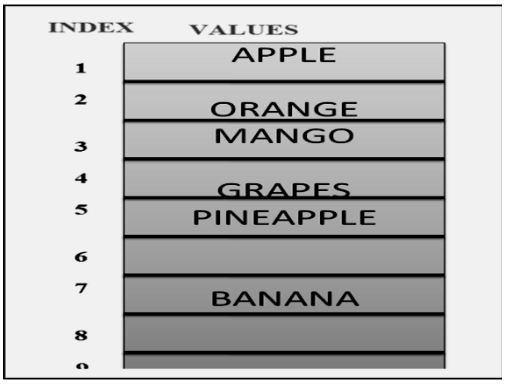
The idea is simple in hashing is to use hash function to map keys(data to be searched) in the hash table. Let's first know what exactly a hash table & hash function is.

13.5.1 Hash Table

All the large collection of data are stored in a hash table. The size of the hash table is usually fixed and it is always bigger than the number of elements we are going to store. The load factor defines the ration of the number of data to be stored to the size of the hash table. Hashing is the procedure of inserting and searching a data from the hash table in optimal time.

For storing a element in the hash table, we assign a key to each element that is inserted. This key is the important criteria in a hash table which allows searching to be performed much faster. Hash tables are generally array of cells of fixed size containing data. This key is helpful is mapping the element to be searched in the hash table.

Figure 13.4: Hash Table



Source: http://newtonapples.com/understanding-hashing-in-c-language/

The above figure shows a sample hash table whose size is 9 and there are 7 elements stored currently. The first column ranging from 1 to 9 are the INDEX assigned to all the individual elements and the second column are the specific values stored in a hash table.

13.5.2 Hash Function

The element that is to be retrieved from the hash table is known as a key. A hash function helps in connecting a key to the index of the hash table. The key can be a number or string. A hash function should be perfect in storing the elements in the exact index in the hash table and tell us where to retrieve the searched data from the table. A hash function should be easy and quick to compute.

It is possible that when a hash function is mapping the keys into a hash table, two elements might hash to the same location in the hash table. This is known as collision. But this is for sure that the first element that is already present in the hash table cannot be modified. So the new element has to be brought up with a solution. Inorder to avoid this collision we can prefer one of these solutions:-

- Search from that position for an empty location.
- Use a second hash function.
- Use that array location as the header of a linked list of values that hash to this location.

13.5.3 Working of Hashing

Suppose we need to store all the information of 5 employees in a hash table and unique security number of the employees is used as a key. Let's assume that the size of the hash table is 6. Hence the hash function should be:-

Value modulo 6

The value is the ssn number of the employee and modulo (%) is the arithmetic modulo operation. 6 is the size of the hash table. The hash function is usually modulated like this in case of integers. Hence we can write it in simpler words as:

Hash function= (sum of all the digits in ssn) % 6

For employee1 ssn = 2342158231IN

2+3+4+2+1+5+8+2+3+1=31

Hash code= 31 % 6 =1.

So the hash function maps the ssn of employee 1 to index 1 in the hash table.

Similarly all the employees are mapped onto the hash table by the hash function and there is a chance of collision. If there is a collision then the above solutions can be followed to solve the collision problem.

13.5.4 Types of hashing

The different types of hashing are:

- Static Hashing
- Dynamic Hashing

Static Hashing

In static hashing the process is carried out without the usage of an index structure. Data is normally accessed on a disk by computing a function on key instead. A bucket in a hash file is a storage unit that can hole records. This bucket can be a block in the disk. For inserting a data we need to find the hash code subsequently and place the record in the bucket. For searching the data compute the hash value and match with each record in the bucket. The hash function can assign equal or random number of records to all the buckets. Bucket overflow can occur if:-

- There are not enough buckets to store data.
- A few buckets have more records then others. (Multiple records have same hash value)

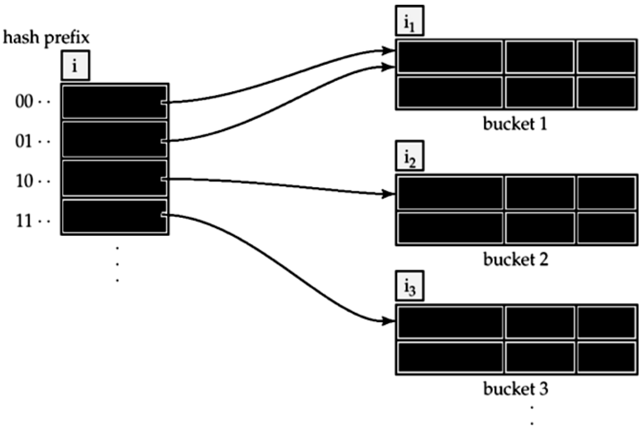
Inorder to provide a solution for bucket overflow issue, we need to provide more buckets then are needed. If a bucket is full, then link another bucket to it and repeat this process.

Dynamic Hashing

Since static hashing had a problem of bucket overflow and slow processing was an added disadvantage dynamic hashing was introduced. It is more effective then static hashing where the database can grow and sink.

Extendable Hashing allows dynamic allocation of buckets, i.e. according to the demand of database the buckets can be allocated making this approach more efficient. The hash function produces a large number of values but only a part of the value is used to map. Hash indices are used from an indexed table which is used as prefix of an entire hash value. More than one consecutive index can point to a specific bucket.

Figure 13.5:Bucket Hashing with Dynamic hashing(i1, i2, i3 indices)



Source: http://newtonapples.com/understanding-hashing-in-c-language/

The table in the left is bucket address table which consists of the bucket addresses and an index to point to each bucket. For searching any record from the bucket, we take the first i bits of the hash value and match the corresponding entry in the bucket address table with the index to receive the address of the corresponding bucket. After we get the address of the bucket, we can go to that specific bucket and search the record. Follow the same procedure for insertion but we can only add the record if the bucket is not full.

Notes The use of buckets in static and dynamic hashing is known as bucket hashing.

Task Compare and contrast static hashing and dynamic hashing.

Self Assessment Questions

Fill in the blanks:

9. .... is the solution when we want to search an element from a large collection of data.

10. The .... defines the ration of the number of data to be stored to the size of the hash table.

11. A .... helps in connecting a key to the index of the hash table.

12. In .... hashing the process is carried out without the usage of an index structure.

13.6 Some other Sorting Techniques

Let us now discuss some other sorting techniques.

13.6.1 Bubble Sort

Bubble sort is a simple sorting algorithm, which compares repeatedly each pair of adjacent items and swaps them if they are in the incorrect order. At the first run, the highest element of the list ends on the last place of the sorted list, and then, at the following run, the next highest element ends on one position lower than the previous element and so on. The initial list is finally sorted, when at a run, no swaps occur.

Even though it's one of the simplest sorting algorithms, it's almost never used in real life because of the bad performance on large sorting lists. It's easy to learn, so it's one of the first algorithms that people learn.

Example: Having the following list, let's try to use bubble sort to arrange the numbers from lowest to greatest:

Unsorted list: 9 2 0 1 4 6

First run:

(**9 2** 0 1 4 6) -> (**2 9** 0 1 4 6) : 9>2, swap occurs

(2 **9 0** 1 4 6) -> (2 **0 9** 1 4 6) : 9>0, swap occurs

(2 0 **9 1** 4 6) -> (2 0 **1 9** 4 6) : 9>1, swap occurs

(2 0 1 **9 4** 6)-> (2 0 1 **4 9** 6) : 9>4, swap occurs

(2 0 1 4 **9 6**) -> (2 0 1 4 **6 9**) : 9>6, swap occurs. The last two elements are in the right order, no swaps occur, it's the end of the first run.

Second run:

(**2 0** 1 4 6 9) -> (**0 2** 1 4 6 9) : 2>0, swap occurs

(0 **2 1** 4 6 9) -> (0 **1 2** 4 6 9): 2>1, swap occurs

(0 1 **2 4** 6 9) -> (0 1 **2 4** 6 9): 2<4, no swap occurs

(0 1 2 **4 6** 9) -> (0 1 2 **4 6** 9): 4<6, no swap occurs

(0 1 2 4 **6 9**) -> (0 1 2 4 **6 9**): 6<9, no swap occurs, it is the end of the list -> end of the second run.

The array is already sorted, but the algorithm doesn't know that, so it requires another pass with no swaps in order to know the elements are in the right place.

Third run:

(**0 1** 2 4 6 9) -> (**0 1** 2 4 6 9): 0<1, no swap occurs

(0 **1 2** 4 6 9) -> (0 **1 2** 4 6 9): 1<2, no swap occurs

(0 1 **2 4** 6 9) -> (0 1 **2 4** 6 9): 2<4, no swap occurs

(0 1 2 **4 6** 9) -> (0 1 2 **4 6** 9): 4<6, no swap occurs

(0 1 2 4 **6 9**) -> (0 1 2 4 **6 9**): 6<9, no swap occurs

The algorithm now knows that the list is sorted.

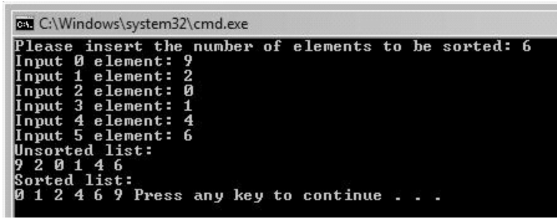
The implementation of bubble sort in c is shown as below:

```
#include < iostream >
using namespace std;
int main()
{
          int a[100];  // The sorting array
         int n;           // The number of elements
          cout << “Please insert the number of elements to be sorted: “;
         cin >> n;        // The total number of elements
          for(int i=0;i< n;i++)
         {
                   cout << “Input “ << i << “ element: “;
                    cin >>a[i]; // Adding the elements to the array
         }
          cout << “Unsorted list:” << endl;         // Displaying the
unsorted array
          for(int i=0;i< n;i++)
         {
                  cout << a[i] << “ “;
         }
          int flag=-1;     // The flag is necessary to verify is the array
is sorted, 0 represents sorted array
         int k=n;                   // Another variable is required to hold
the total number of elements
          while(flag!=0 && k>=0)            // Conditions to check if the
array is already sorted
         {
                  k=k-1;
                  flag=0;
                   for (int i=0;i < k;i++)
                  {
                            if(a[i]>a[i+1])          // Check if the two
adjacent values are in the proper order, if not, swap them
                         {
                                    int aux=a[i];   // The swap procedure
                                   a[i]=a[i+1];
                                   a[i+1]=aux;
                                   flag=1;          // A swap has taken
place, this means a new run of the algorithm must take place
                            }         // to determine if the array is sorted.
```

Notes

```
                    }
            }
             cout << “nSorted list:” << endl;          // Display the sorted
array
             for(int i=0;i < n;i++)
            {
                    cout << a[i] << “ “;
            }
             return 0;
}
```

The output is shown as below:



C:\Windows\system32\cmd.exe

```
Please insert the number of elements to be sorted: 6
Input 0 element: 9
Input 1 element: 2
Input 2 element: 0
Input 3 element: 1
Input 4 element: 4
Input 5 element: 6
Unsorted list:
9 2 0 1 4 6
Sorted list:
0 1 2 4 6 9 Press any key to continue . . .
```

Source: http://www.exforsys.com/tutorials/c-algorithms/bubble-sort.html

A flag is required to check if any swaps have occurred at a run of the algorithm. If a swap has taken place, the flag becomes 1, and the while loop gets at least one more pass. When the flag is 0, it means that no swap has taken place so the arrays has been sorted and the while loop gets ignored.

Also, the k must be greater or equal to 0, this represents the number of elements of the list, and at each pass it decrements its value by 1. If this condition and the previous one are both true at the same time, then the array isn't sorted, and the program enters the while loop. If one of the conditions is false, then the array has been sorted out.

There is a for loop embedded inside a while loop (n-1)+(n-2)+(n-3)+ ... +1 which results in O(n^2), Swaps - Best Case 0, or O(1) and on Worst Case (n-1)+(n-2)+(n-3) + ... +1 , or O(n^2).

Some of the advantages of bubble sort are:

- it is easy to learn;
- few lines of code;
- works very well on already sorted lists, or lists with just a few permutations.

Some of the disadvantages of bubble sort are:

- not effective for large numbers of sorting elements;
- complexity is O(n^2).

13.6.2 Quick Sort

Quick sort is a comparison sort developed by Tony Hoare. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set.

A pivot is chosen from the elements of the data set.

> *Notes* The list must be reordered in such a way that the elements with the value less than the pivot come before it and the ones that are greater come after it. This is called the partition operation, and after this was completed, the pivot is in its final position, then, recursively, sort the rest of the elements.

Example: Having the following list, let’s try to use quick sort to arrange the numbers from lowest to greatest:

Unsorted list:

6	1	4	9	0	3	5	2	7	8

We go from START to END, with the pivot being the first element 6, and let us hide it. :

8	1	4	9	0	3	5	2	7	6
I								j	

Now we scan with i and j. We have to find with i the first value greater than the pivot and with j the first value lower than the pivot. We found that the first value greater than the pivot is 8, and we are still looking for the first value lower than the pivot :

8	1	4	9	0	3	5	2	7	6
I								j	

8	1	4	9	0	3	5	2	7	6
i							j		

We found with j the number 2 which is lower than the pivot, so now we swap the elements pointed by i with j:

2	1	4	9	0	3	5	8	7	6
i							j		

Now, we look again for greater values than the pivot with i:

2	1	4	9	0	3	5	8	7	6
			i				j		

We found the element of 9 which is greater. Now we seek an element lower with j:

2	1	4	9	0	3	5	8	7	6
			i			j			

We found 5. Now we swap them:

2	1	4	5	0	3	9	8	7	6
			i			j			

We begin the search again, but this time, the index i passes over the index j, so this means at the position where i is the index is the proper place for the pivot, and we swap the value of the pivot with the value that is currently there:

2	1	4	5	0	3	9	8	7	6
					j	i			

2	1	4	5	0	3	6	8	7	9
					j	i			

If you look closely, all the elements before 6 are smaller and all the elements after the former pivot are greater. Now, we split the initial list into two smaller lists according to the former pivot: 2, 1, 4, 5, 0, 3 and 8, 7, 9. We repeat the same procedure for each of the sub-lists. The pivot will be the first element in each case, and when the proper place for the pivot was discovered and moved in accordance to this, again we split the list into two smaller ones using the same rule: one list will be composed by the elements smaller than the pivot and the next list composed by elements greater than the pivot. By the end, the initial list will be put together and it will be sorted.

The implementation of Quick Sort in C is shown as below:

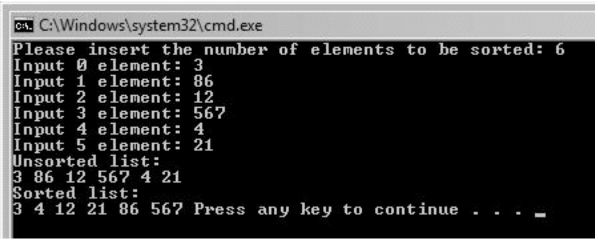
```
#include < iostream >
using namespace std;
void swap(int &a,int &b)
{
```

```
          int aux;
          aux = a;
          a = b;
          b = aux;
}
int partition (int *a, int low,int high)
{
          int l = low;
          int h = high;
          int x = a[l];
          while (l < h)
          {
                     while ((a[l] <= x) && (l <= high))
                             l++;
                     while ((a[h] > x) && (h >= low))
                             h—;
                    if (l < h)
                              swap(a[l], a[h]);
          }
           swap(a[low], a[h]);
          return h;
}

void QuickSort (int *a, int low, int high)
{
          int k;
          if (high > low)
          {
                     k = partition(a, low, high);
                     QuickSort(a, low, k-1);
                     QuickSort(a, k+1, high);
          }
}
int main()
{
          int n;
          int *a;
            cout << “Please insert the number of elements to be sorted: “;
           cin >> n;         // The total number of elements
            a = (int *)calloc(n, sizeof(int));
           for(int i=0;i< n;i++)
          {
```

Notes

```
                        cout << "Input " << i << " element: ";
                         cin >>a[i]; // Adding the elements to the array
           }
            cout << "Unsorted list:" << endl;        // Displaying the
unsorted  array
            for(int i=0;i< n;i++)
           {
                      cout << a[i] << " ";
           }
            QuickSort(a, 0, n-1);
            cout << "nSorted list:" << endl;         // Display the sorted
array
            for(int i=0;i < n;i++)
           {
                      cout << a[i] << " ";
           }
           return 0;
}
```

The output is shown below:



```
Please insert the number of elements to be sorted: 6
Input 0 element: 3
Input 1 element: 86
Input 2 element: 12
Input 3 element: 567
Input 4 element: 4
Input 5 element: 21
Unsorted list:
3 86 12 567 4 21
Sorted list:
3 4 12 21 86 567 Press any key to continue . . .
```

Source: http://www.exforsys.com/tutorials/c-algorithms/quick-sort.html

The partition function does what it is called, it chooses the pivot, then, like in the example, while l is smaller than high (or i smaller than j like in the example) it checks the list for the elements that must be greater or smaller, and when it has found them, a swap function is executed. Then, inside the QuickSort function, after the pivot was found, the list is divided into two smaller lists, according to the pivot, then recursively the partition function is called for each sub-list and in the end the initial list is sorted.

Some of the advantages of quick sort are:

- One of the fastest algorithms on average;
- Does not need additional memory (the sorting takes place in the array - this is called in-place processing).

Some of the advantages of quick sort are:

- it The worst-case complexity is O(N2);
- is recursive;

13.6.3 Heap Sort

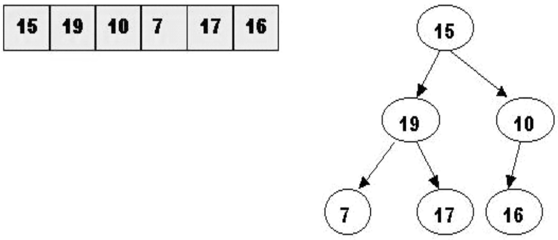
Heap Sort is a comparison based algorithm. It bases on building a heap tree from the data set, and then it removes the greatest element from the tree and adds it to the end of the sorted list. There are two ways to do this, either to add the highest value to the root of the tree, or as one of the left/right child with the greatest depth.

The heap tree is built according to the data set. Then the greatest value is move to the root of the tree, and then it is removed. This represents the highest value from the data set, so it is added to the end of the sorted list. Next, the heap is reconstructed, and the second highest value is added to the top and then it is removed. This continues until the tree is empty and the list is full.

Example: Having the following list, let's try to use heapsort to arrange the numbers from lowest to greatest:

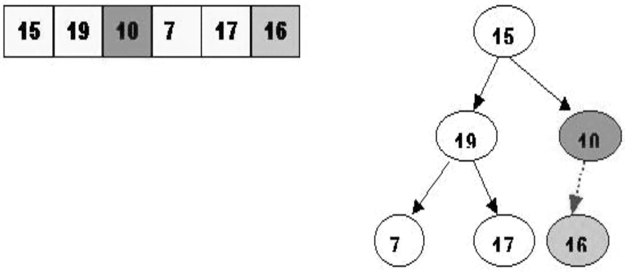
Unsorted list: 15, 19, 10, 7, 17, 6.

Building the heap tree:



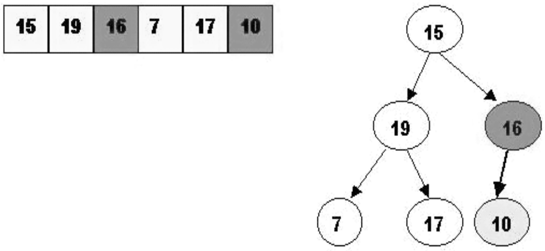
15	19	10	7	17	16

Start with the rightmost node at height 1 - the node at position 3 = Size/2. It has one greater child and has to be percolated down:



15	19	10	7	17	16

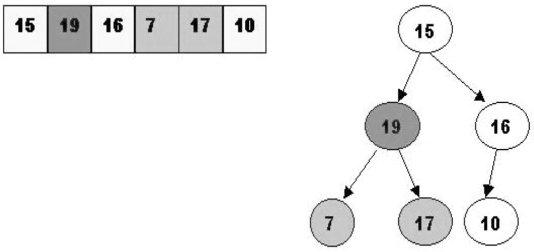
And it will look like this:



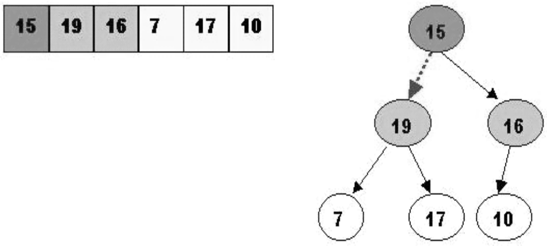
15	19	16	7	17	10

Notes

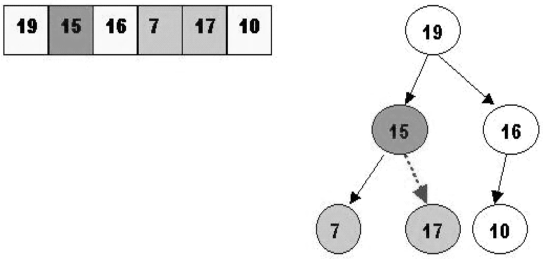
Then, we check the children of 19, but both are smaller so no swap is needed:



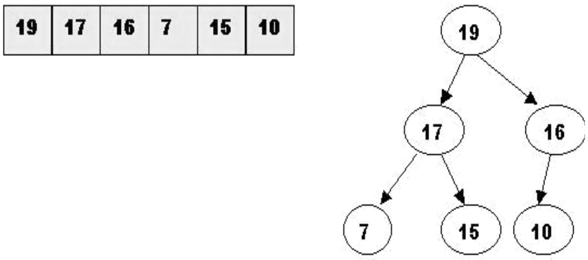
The children of 15 are 19 and 16, and is smaller, so a swap is required:



Also, 15 is smaller than 17 one of the children, so it is again time to swap :

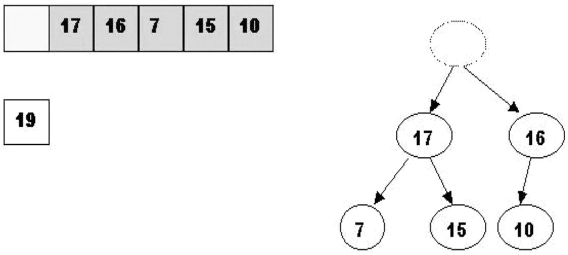


Now the heap was built.

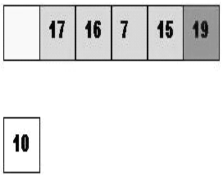


Next, it is time to sort the heap:

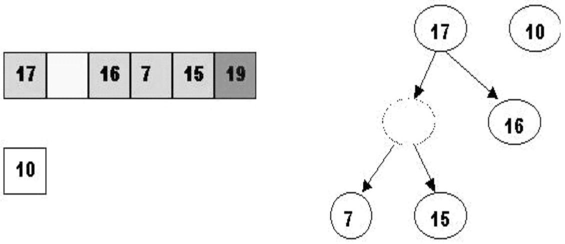
19, being the highest value, was removed from the top:



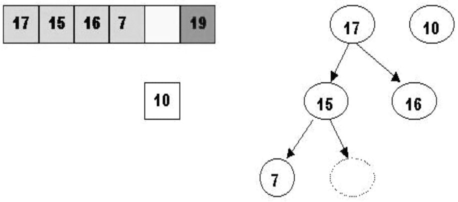
Swap 19 with the last element of the heap (As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array):



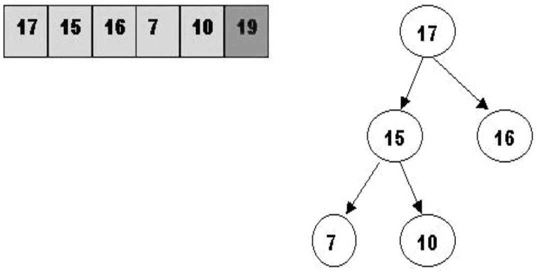
The element 10 will have to go down in the tree:



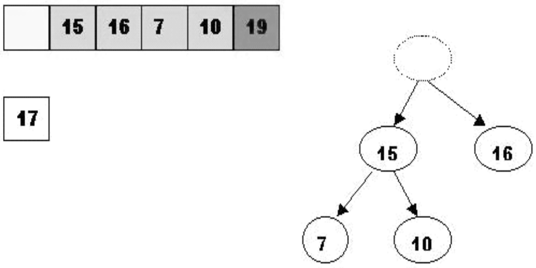
And again down one more level, 10 is less that 15, so it cannot be inserted in the previous hole:



Now 10 has found the right spot:

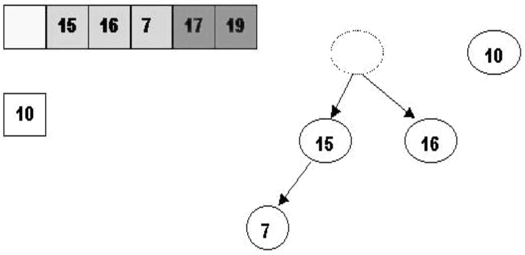


The next greatest element is 17, which has to be removed :

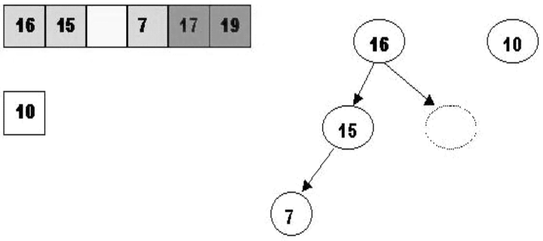


Also, swap 17 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap, instead it becomes a cell from the sorted array:

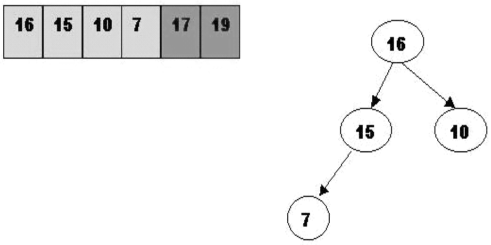
Notes



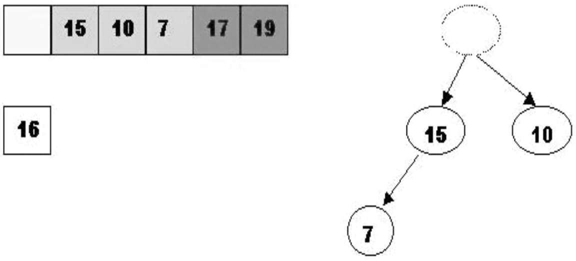
The element 10 is less than the children of the hole, and we percolate the hole down:



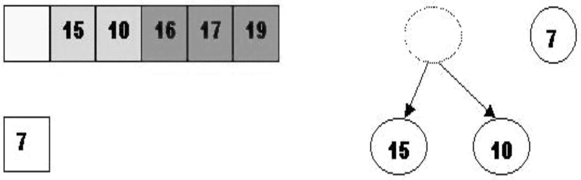
Insert 10 in the hole



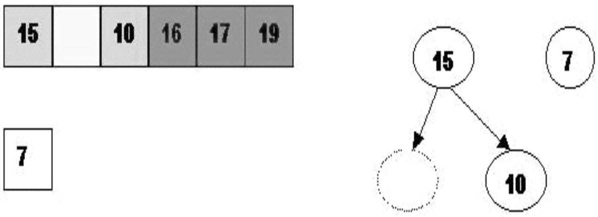
Delete the next biggest element which is 16:



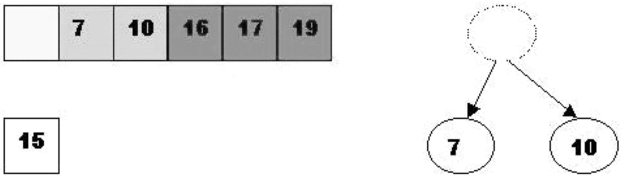
Swap 16 with the last element of the heap. As 7 will be adjusted in the heap, its cell will no longer be a part of the heap, instead it becomes a cell from the sorted array:



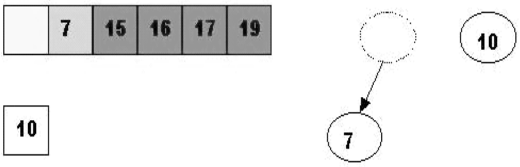
Go down one level to find the right position for 7:



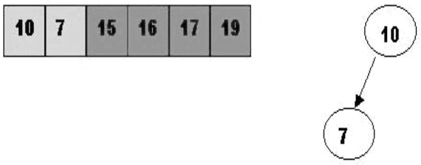
Remove 15:



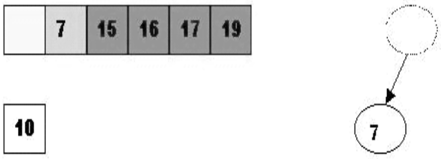
Swap 15 with the last element of the heap. As 10 will be adjusted in the heap, its cell will no longer be a part of the heap, instead it becomes a position from the sorted array:



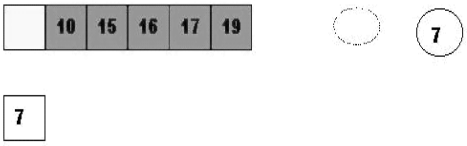
Store 10 in the hole (10 is greater than the children of the hole)



Remove 10:

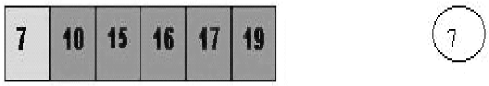


Swap 10 with the last element of the heap. As 7 will be adjusted in the heap, its cell will no longer be a part of the heap, instead it becomes a cell from the sorted array:

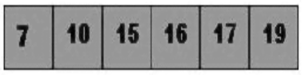


Notes

Store 7 in the hole (as the only remaining element in the heap):



7 is the last element from the heap, so now the array is sorted :



The implementation of heap sort in C is shown below:

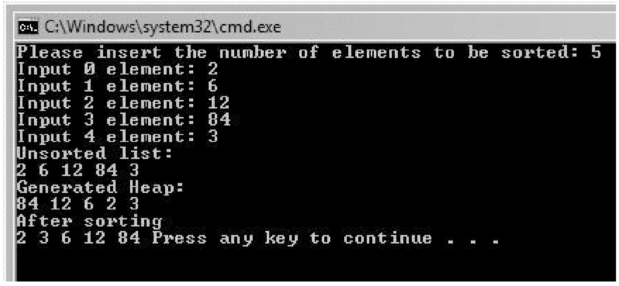
```
#include< iostream >
using namespace std;

void insert (int v[],int &N, int a)
{
        v[N]=a;
        N++;
         int child=N-1;
          int parent=abs((child-1)/2);
         while(parent >=0 )
        {
                if(v[child]>v[parent])
                {
                        v[child]=v[parent]+v[child];
                        v[parent]=v[child]-v[parent];
                        v[child]=v[child]-v[parent];
                       child=parent;
                        parent=abs((child-1)/2);
                }
                else
                        parent=-1;
        }
}int remove(int v[],int N)
{
        int     a=v[0];
         v[0]=v[N-1];
        N—;
         int parent=0;
         int child=1;
          while(child<=N-1)
        {
```

```
                    if(child+1< =N-1 && (v[child+1]>v[child]))
                          child++;
                   if (v[parent]< v[child])
                 {
                          int      aux=v[parent];
                          v[parent]=v[child];
                          v[child]=aux;
                          parent=child;
                          child=2*parent+1;}
                 else
                          break;
         }
         return a;
}
void heap_sort(int a[],int N,int v[])
{
          for(int i=N-1;i>=0;i–)
         {
                 N=i+1;
                  a[i]=remove(v,N);
         }
}
void   heap_gen1(int a[],int v[],int n)
{
         int N=1;
         v[0]=a[0];
          for(int i=1;i< n;i++)
         {
                  insert(v,N,a[i]);
         }
}
void display(int a[],int n)
{
          for(int i=0;i < n;i++)
         {
                 cout << a[i] << " ";
         }
}
int main()
{
          int *a=new int[100];
          int *v=new int[100];
```

Notes

```
          int n;
           cout << "Please insert the number of elements to be sorted: ";
          cin >> n;       // The total number of elements
          for(int i=0;i < n;i++)
          {
                    cout << "Input " << i << " element: ";
                     cin >> a[i]; // Adding the elements to the array
          }
           cout << "Unsorted list:" << endl;        // Displaying the
unsorted array
          for(int i=0;i < n;i++)
          {
                    cout << a[i] << " ";
          }
           heap_gen1(a,v,n);
           cout << "nGenerated Heap:n";
          display(v,n);
           cout << "nAfter sorting";
           heap_sort(a,n,v);
          cout << "n";
          display(a,n);
          return 0;
}
```

The output is shown as below:



```
Please insert the number of elements to be sorted: 5
Input 0 element: 2
Input 1 element: 6
Input 2 element: 12
Input 3 element: 84
Input 4 element: 3
Unsorted list:
2 6 12 84 3
Generated Heap:
84 12 6 2 3
After sorting
2 3 6 12 84 Press any key to continue . . .
```

Source: http://www.exforsys.com/tutorials/c-algorithms/heap-sort.html

The function heap_sort is the one that is removing the top of the heap tree and making the sorting list. The implementation of the algorithm simply follows the same steps as those from the above example.

Complexity

HeapSort's space-complexity is O(1), just a few scalar variables. It uses a data structure known as a max-heap which is a complete binary tree where the element at any node is the maximum of itself and all its children. A tree can be stored either with pointers as per the pictorial representation we are normally used to visualize, or by mapping it to a vector. Here if a node in the tree is mapped to the ith element of an array, then its left child is at 2i, its right child is at (2i+1) and its parent is at floor(i/2). We can construct a heap by starting with a an existing heap, adding an element at the bottom of the heap, and allow it to migrate up the father chain till it finds its proper position. Complexity T(n) = O(n*log (n)).

Some of the advantages of heap sort are:

- it does not use recursion;
- works just as fast or any data order.
- there is no worst case scenario.

Some of the disadvantages of heap sort are:

- slower that quick and merge sort;
- memory requirement, it requires both an array and a heap of size n;
- not stable.

Self Assessment Questions

Fill in the blanks:

13. .... is a simple sorting algorithm, which compares repeatedly each pair of adjacent items and swaps them if they are in the incorrect order.
14. Quick Sort uses a .... chosen by the programmer, and passes through the sorting list and on a certain condition.
15. .... bases on building a heap tree from the data set, and then it removes the greatest element from the tree and adds it to the end of the sorted list.

13.7 Summary

- A sorting algorithm refers to putting the elements of a data set in a certain order, this order can be from greater to lower or just the opposite, and the programmer determines this.
- In an insertion sort, all the keys in a given list are assumed to be in random order before sorting.
- Selection sort, also called naive (selection) sort, is an in-place comparison sort.
- Selection sort first finds the smallest number in the array and exchanges it with the element from the first position, then it finds the second smallest number and exchanges it with the element from the second position, and so on until the entire list is sorted.
- Merging means combining elements of two arrays to form a new array. The simplest way of merging two arrays is to first copy all the elements of one array into a new array and then append all the elements of the second array to the new array.
- Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists.
- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- Hashing is the solution when we want to search an element from a large collection of data.
- A hash function helps in connecting a key to the index of the hash table. The key can be a number or string.

13.8 Keywords

Insertion Sort: The insertion sort maintains the two sub-arrays within the same array.

Selection sort: Selection sort, also called naive (selection) sort, is an in-place comparison sort.

Merge Sort: Merge sort is based on Divide and conquer method which takes the list to be sorted and divide it in half to create two unsorted lists.

Radix Sort: Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

Hashing: Hashing is the solution when we want to search an element from a large collection of data.

Key: The element that is to be retrieved from the hash table is known as a key.

Hash function: A hash function helps in connecting a key to the index of the hash table.

Bubble Sort: Bubble sort is a simple sorting algorithm, which compares repeatedly each pair of adjacent items and swaps them if they are in the incorrect order.

13.9 Review Questions

1. Explain the concept of insertion sort with example.
2. Discuss the advantages and disadvantages of selection sort.
3. Describe merge sort with example. Also illustrate its implementation in C.
4. Explain the working of radix sort with example.
5. Discuss the features of Hashing.
6. For storing a element in the hash table, we assign a key to each element that is inserted. Comment.
7. Discuss the use of hash function.
8. Illustrate the working of hashing with example.
9. Make distinction between bubble sort and quick sort.
10. Describe the concept of heap sort with example.

Answers: Self Assessment

1. True
2. False
3. Selection sort
4. big-O notation
5. Divide and conquer
6. Merge Sort
7. Radix sort
8. LSD
9. Hashing
10. load factor
11. hash function
12. Static
13. Bubble sort
14. Pivot
15. Heap sort

13.10 Further Readings

Notes

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://www.codebeach.com/2008/09/sorting-algorithms-in-c.html

http://www.cquestions.com/2009/09/insertion-sort-algorithm.html

http://cs.nyu.edu/courses/spring01/V22.0102-002/heap.txt

http://www.cprogramming.com/tutorial/computersciencetheory/hashtable.html

Notes

Unit 14: Searching

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the concept of Linear Search
- Discuss Binary Search
- Discuss the characteristics of Linear Search and Binary Search

Introduction

Searching is the process of determining whether or not a given value exists in a data structure or a storage media. Searching is an important function in computer science. Many advanced algorithms and data structures have been devised for the sole purpose of making searches more efficient. And as the data sets become larger and larger, good search algorithms will become more important. At one point in the history of computing, sequential search was sufficient. But that quickly changed as the value of computers became apparent. Linear search has many interesting properties in its own right, but is also a basis for all other search algorithms. Learning how it works is critical. Binary search is the next logical step in searching. By dividing the working data set in half with each comparison, logarithmic performance, O(log n), is achieved. That performance can be improved significantly when the data set is very large by using interpolation search, and improved yet again by using binary search when the data set gets smaller.

14.1 Linear Search

Linear search is a basic form of search. Linear search is the basic searching algorithm, also called as sequential search. Algorithm searches the element by comparing the each element in the list, until the desired element found.

This method of searching for data in an array is straightforward and easy to understand. To find a given item, begin your search at the start of the data collection and continue to look until you have either found the target or exhausted the search space. Clearly to employ this method you must first know where the data collection begins and the size of the area to search. Alternatively, a unique value could be used to signify the end of the search space.

Did u know? This method of searching is most often used on an array data structure whose upper and lower bounds are known.

The complexity of this type of search is O(N) because, in the worst case all items in the search space will be examined. This type of search is $\Theta(n/2)$ as, in the average case, one-half of the items in the search space will be examined before a match is found. There are many algorithms for improving search time that can be used in place of a linear search. For instance, the binary search algorithm operates much more efficiently than a linear search but requires that the data being searched be in sorted order. Because there are faster ways of searching a memory space, the linear search is sometimes referred to as a brute force search.

14.1.1 Characteristics

The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item isn't found. In other words, if you have N items in your collection, the worst case scenario to find an item is N iterations. This is known as O(N) using the Big O Notation.

Caution The speed of search grows linearly with the number of items within your collection.

Linear searches don't require the collection to be sorted.

In some cases, you'll know ahead of time that some items will be disproportionally searched for. In such situations, frequently requested items can be kept at the start of the collection. This can result in exceptional performance, regardless of size, for these frequently requested items.

Despite its less than stellar performance, linear searching is extremely common. Many of the built-in methods that you are familiar with, like ruby's find_index, or much of jQuery, rely on linear searches. When you are dealing with a relatively small set of data, it's often good enough (and for really small unordered data is can even be faster than alternatives).

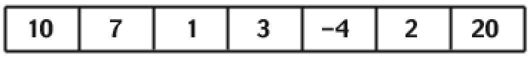
Now let us understand linear search with real world example.

Example: As a real world example, pickup the nearest phonebook and open it to the first page of names. We're looking to find the first "Smith". Look at the first name. Is it "Smith"? Probably not (it's probably a name that begins with 'A'). Now look at the next name. Is it "Smith"? Probably not. Keep looking at the next name until you find "Smith".

The above is an example of a sequential search. You started at the beginning of a sequence and went through each item one by one, in the order they existed in the list, until you found the item you were looking for.

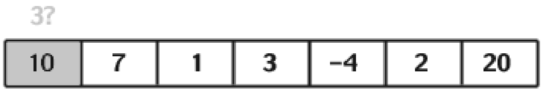
Now we'll look at this as related to computer science. Instead of a phonebook, we have an array. Although the array can hold data elements of any type, for the simplicity of an example we'll just use an array of integers, like the following:

Figure 14.1: The array we're searching



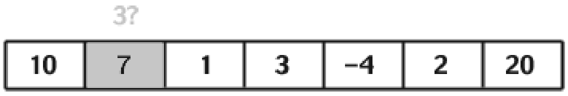
Let's search for the number 3. We start at the beginning and check the first element in the array.

Figure 14.2: Is the first value 3?



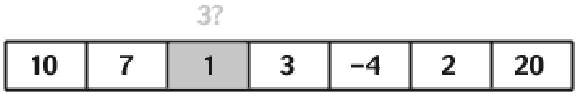
As we can see, it is not 3, move to the next element.

Figure 14.3: Is the second value 3?



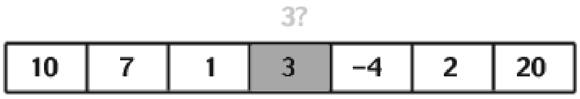
Again move to the next element, as it is not 3.

Figure 14.4: Is the third value 3?



As we can see in the figure below, the next value is number 3..

Figure 14.5: Is the fourth value 3? Yes!



Source: http://www.sparknotes.com/cs/searching/linearsearch/section1.rhtml

Now you understand the idea of linear searching; we go through each element, in order, until we find the correct value.

Sequential search has some advantages over other searches. Most importantly, it doesn't require the array to be sorted, since every array element is examined. In addition, linear search is quite easy to implement, as evidenced by the relative simplicity of the code above. The disadvantage of sequential search is efficiency. Since this approach examines every element in the list, it does work for every element. Therefore, linear search is O(n) , relatively inefficient, as sorting algorithms go.

Task Analyze the real world examples of linear search.

Notes

14.1.2 A Function to Implement Linear Search

Let's apply a linear search algorithm and write a function to carry it out. Our function will take three arguments: the array to search, the number of elements in the array, and a value to search for. The function will return the index into the array that the value was found at, or -1 if the value wasn't found.

> *Notes* Remember that in programming languages like C, arrays of length N have indices numbered 0 through N-1; therefore a return value of -1 cannot be valid place in the array and the calling function will know that the value wasn't found.

We declare our function as follows:

```
int sequential_search(int arr[], int n, int value);
```

Step 1: We need to search through every element in the array. This can be easily accomplished using a loop.

```
for(i=0; i<n; i++) {
        ...
}
```

Step 2: At every place in the array, we need to compare the array element to the value we're searching for. If this index stores the value, then immediately return the correct answer. Otherwise, keep going.

```
for(i=0; i<n; i++) {
        if (arr[i] == value) return i;
}
```

Step 3: What happens if the value is never found? The loop will end and the function will continue. So after the loop we need to return the value -1.

```
for(i=0; i<n; i++) {
      if (arr[i] == value) return i;
}
return -1;
```

Step 4: Putting this all together we end up with a function to do a linear search of an array:

```
int sequential_search(int arr[], int n, int value) {
         int i;
        /* loop through entire array */
        for(i=0; i<n; i++) {
                /* if this element contains the value being searched for,
                 * return the location in the array
                 */
                if (arr[i] == value) return i;
        }
                /* if we went through the entire array and couldn't find
         * the element, return -1.  as 0 is the smallest index in
         * the array, -1 represents an error and tells the calling
```

Notes

```
          * function that a value wasn't found
          */
        return -1;
}
```

14.1.3 Sequential Search on Linked Lists

Sequential search is the most common search used on linked list structures. Let's take a look at how this search can be applied to linked lists.

We define our linked list structure as:

```
typedef struct _list_t_ {
     int data;
      struct _list_t_ *next;
} list_t;
```

Our sequential search function for linked lists will take two arguments: a pointer to the first element in the list and the value for which we are searching. The function will return a pointer to the list structure containing the correct data, or will return NULL if the value wasn't found. Why do we need to pass in the number of elements in the array for the array version and not the number of elements in the list for the linked list version? Because we can easily tell when we're at the end of our list by checking to see if the element is NULL, whereas with our array the computer has no way of knowing how big it is without us telling it. Our function declaration is as follows:

```
list_t *sequential_search(list_t *list, int value);
```

Step 1: Just like the array version, we need to look at every element in the list, so we need a loop. We start at the first element in list, and while the current element is not NULL (meaning we haven't reached the end), we go on to the next element:

```
for( ; list!=NULL; list=list->next) {
     ...
}
```

Step 2: If the current element contains the data we're looking for, then return a pointer to it.

```
for( ; list!=NULL; list=list->next) {
      if (list->data == value) return list;
}
```

Step 3: If after looking at every element in the list, we haven't found the value for which we are searching, then the value doesn't exist in the list. Return an error flag appropriately:

```
for( ; list!=NULL; list=list->next) {
      if (list->data == value) return list;
}
return NULL;
```

Step 4: Our final search function is:

```
list_t *sequential_search(list_t *list, int value)
{
          /* look at every node in the list */
```

Notes

```
          for( ; list!=NULL; list=list->next) {
              /* if this node contains the given value, return the node */
              if (list->data == value) return list;
        }

          /* if we went through the entire list and didn't find the
            * value, then return NULL signifying that the value wasn't
found
          */
         return NULL;
}
```

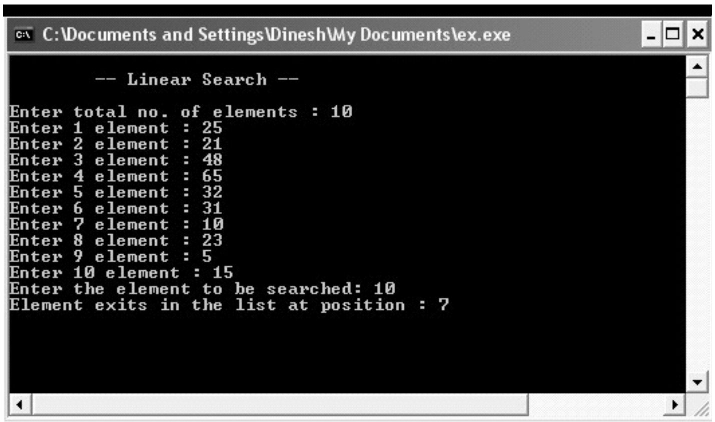
Example:

Write a C program to search an element in a given list of elements using Liner Search.

```
/*c program for linear search*/
#include<stdio.h>
#include<conio.h>
int main()
{
   int arr[20];
   int i,size,sech;
   printf("\n\t— Linear Search —\n\n");
   printf("Enter total no. of elements : ");
   scanf("%d",&size);
   for(i=0; i<size; i++)
  {
      printf("Enter %d element : ",i+1);
      scanf("%d",&arr[i]);
  }
   printf("Enter the element to be searched: ");
   scanf("%d",&sech);
   for(i=0; i<size; i++)
  {
      if(sech==arr[i])
     {
        printf("Element exits in the list at position : %d",i+1);
       break;
     }
  }
   getch();
   return 0;
}
```

Notes

The output is given below:



```
        -- Linear Search --

Enter total no. of elements : 10
Enter 1 element : 25
Enter 2 element : 21
Enter 3 element : 48
Enter 4 element : 65
Enter 5 element : 32
Enter 6 element : 31
Enter 7 element : 10
Enter 8 element : 23
Enter 9 element : 5
Enter 10 element : 15
Enter the element to be searched: 10
Element exits in the list at position : 7
```

Self Assessment Questions

Fill in the blanks:

1. Linear search is the basic searching algorithm, also called as .... search.
2. The complexity of this type of search is .... because, in the worst case all items in the search space will be examined.
3. The .... case performance scenario for a linear search is that it needs to loop through the entire collection.
4. The function will return the index into the array that the value was found at, or .... if the value wasn't found.
5. In C, arrays of length N have indices numbered 0 through ....
6. Our sequential search function for linked lists will take two arguments: a .... to the first element in the list and the value for which we are searching.
7. To employ linear search, you must first know where the data collection begins and the .... of the area to search.
8. Because there are faster ways of searching a memory space, the linear search is sometimes referred to as a ....

14.2 Binary Search

When it is known that a data set is in sorted order it is possible to drastically increase the speed of a search operation in most cases. An array-based binary search selects the median (middle) element in the array and compares its value to that of the target value. Because the array is known to be sorted, if the target value is less than the middle value then the target must be in the first half of the array. Likewise if the value of the target item is greater than that of the middle value in the array, it is known that the target lies in the second half of the array. In either case we can, in effect, "throw out" one half of the search space with only one comparison.

Now, knowing that the target must be in one half of the array or the other, the binary search examines the median value of the half in which the target must reside. The algorithm thus narrows the search area by half at each step until it has either found the target data or the search fails.

The algorithm is easy to remember if you think about a child's guessing game. Imagine a number between 1 and 1000 and guess the number. Each time you guess, you will get to know "higher" or "lower." Of course you would begin by guessing 500, then either 250 or 750 depending on the response.

You would continue to refine your choice until you got the number correct.

14.2.1 Characteristics

Every iteration eliminates half of the remaining possibilities. This makes binary searches very efficient - even for large collections. Our implementation relies on recursion, though it is equally as common to see an iterative approach.

Binary search requires a sorted collection. This means the collection must either be sorted before searching, or inserts/updates must be smart. Also, binary searching can only be applied to a collection that allows random access (indexing).

In the real world, binary searching is frequently used thanks to its performance characteristics over large collections. The only time binary searching doesn't make sense is when the collection is being frequently updated (relative to searches), since re-sorting will be required.

Hash tables can often provide better (though somewhat unreliable) performance. Typically, it's relatively clear when data belongs in a hash table (for frequent lookups) versus when a search is needed.

14.2.2 Analysis

A binary search on an array is O(log2 n) because at each test you can "throw out" one half of the search space. If we assume n, the number of items to search, is a power of two (i.e. n = 2x) then, given that n is cut in half at each comparison, the most comparisons needed to find a single item in n is x.

Notes It is noteworthy that for very small arrays a linear search can prove faster than a binary search. However as the size of the array to be searched increases the binary search is the clear victor in terms of number of comparisons and therefore overall speed.

Still, the binary search has some drawbacks. First of all, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched it can throw off the entire process.

Caution When presented with a set of unsorted data the efficient programmer must decide whether to sort the data and apply a binary search or simply apply the less-efficient linear search.

Even the best sorting algorithm is a complicated process. Is the cost of sorting the data is worth the increase in search speed gained with the binary search? If you are searching only once, it is probably better to do a linear search in most cases.

Once the data is sorted it can also prove very expensive to add or delete items. In a sorted array, for instance, such operations require a ripple-shift of array elements to open or close a "hole" in the array.

Notes

Did u know? This is an expensive operation as it requires, in worst case, log2 n comparisons and n item moves.

The binary search assumes easy random-access to the data space it is searching. An array is the data structure that is most often used because it is easy to jump from one index to another in an array. It is difficult, on the other hand, to efficiently compute the midpoint of a linked list and then traverse there inexpensively. The binary search tree data structure and algorithm attempt to solve these array-based binary search weaknesses.

Example: Below is an iterative implementation of the binary search in C. The binary search algorithm can be implemented as a recursive algorithm also.

In this program, we search for the target value in an array. Return the index on success and NOTFOUND, or -1, on failure.

```
#include “global.h”
 #include “debug.h”
  typedef int KEY;
 #define NOTFOUND -1
 extern int NumElements(KEY *x);   // to return the # of elements in an
array


DWORD bsearch(KEY kTarget, KEY *pkSearchspace)
 {


      BOOL fFound = NO;             // have we found it yet?
      DWORD dwFirst = 0;             // search space starts at index zero
      DWORD dwLast = NumElements(searchspace); // # of elements in
search space
      DWORD dwMidpoint;             // current midpoint
      // Check return value of NumElements and make precondition
assertions
      ASSERT(dwLast >= dwFirst);
      ASSERT(pkSearchspace);
      // iterative implementation – could also be done recursively
      while ((dwFirst <= dwLast) && (!fFound))
      {
            // calculate the dwMidpoint of the current [sub]range
            dwMidpoint = (dwFirst + dwLast) / 2;
            // compare the target with the value of dwMidpoint element
            if (pkSearchspace[dwMidpoint] == kTarget)
            {
                  fFound = YES;
            }
            else
```

```
               {
                      // if the dwMidpoint is not the target adjust the
range accordingly
                      if (target < searchspace[dwMidpoint])
                      {
                             dwLast = dwMidpoint - 1;
                      }
                      else
                      {
                             dwFirst = dwMidpoint + 1;
                      }
               }

               // Return value

               if (fFound)
               {
                      return(dwMidpoint);
               }
               else
               {
                      return(NOTFOUND);
               }
        }
 }
```

Task Compare and contrast Linear search and Binary search.

Self Assessment Questions

State True or False:

9. A linear search selects the median (middle) element in the array and compares its value to that of the target value.
10. Every iteration eliminates half of the remaining possibilities.
11. Binary search requires a sorted collection.
12. A binary search on an array is O(log n).
13. For very small arrays a binary search can prove faster than a linear search.
14. The binary search assumes easy random-access to the data space it is searching.
15. Once the data is sorted it can also prove very expensive to add or delete items.

Notes

Case Study

Case:

Write C programs that use both recursive and non recursive functions to perform the following searching operations for a Key value in a given list of integers : i) Linear search ii) Binary search

```
/* C program Linear Search Using Non-Recursive Functions*/
#include<stdio.h>
#include<conio.h>
main()
{
      int a[100],i,n,ele,loc=-1;
     clrscr();
      printf("Enter Size");
      scanf("%d",&n);
      printf("Enter %d elements",n);
      for(i=0;i<n;i++)
      scanf("%d",&a[i]);
      printf("The array elements");
      for(i=0;i<n;i++)
      printf("%4d",a[i]);
      printf("\nEnter element to search");
      scanf("%d",&ele);
      for(i=0;i<=n-1;i++)
     {
        if(ele==a[i])
       {
         loc=i;
         break;
       }
     }
      if(loc>=0)
      printf("\nThe element %d found at %d location",ele,loc+1);
     else
      printf("\nThe element %d is not found",ele);
     getch();
}
int lin_search(int x[100],int n,int ele)
{
     if(n<=0)
     return -1;
```

Contd...

```
      if(x[n]==ele)
     return n;
     else
      return  lin_search(x,n-1,ele);
}
```

Output:

```
Enter Size:
5
Enter 5 elements;
12
30
25
4
85
The array elements:
12 30 25  4 85
Enter element to search:
25
The element 25 found at 3 location
```

Question:

Write a program to search for a particular string from given set of strings using linear search and binary search techniques.

Source: http://stylewap.comoj.com/wordpress/write-c-programs-that-use-both-recursive-and-non-recursive-functions-to-perform-the-following-searching-operations-for-a-key-value-in-a-given-list-of-integers-i-linear-search-ii-binary-search/

14.3 Summary

- Linear search is the basic searching algorithm, also called as sequential search.
- The complexity of linear search is O(N) because, in the worst case all items in the search space will be examined.
- The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item isn't found.
- Sequential search is the most common search used on linked list structures.
- A binary search selects the median (middle) element in the array and compares its value to that of the target value.
- Binary search requires a sorted collection. This means the collection must either be sorted before searching, or inserts/updates must be smart.

- A binary search on an array is O(log2 n) because at each test you can “throw out” one half of the search space.
- The binary search assumes easy random-access to the data space it is searching.

14.4 Keywords

Searching: Searching is the process of determining whether or not a given value exists in a data structure or a storage media.

Linear Search: In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned.

Binary Search: In binary search, we first compare the key with the item in the middle position of the array.

14.5 Review Questions

1. Explain the concept of searching.
2. What is linear search? Illustrate with example.
3. Discuss the complexity of linear search.
4. Discuss the characteristics of linear search.
5. Discuss the advantages of linear search over other searches.
6. Describe the steps used in implementing linear search.
7. Illustrate the use of sequential search on linked lists.
8. Explain the steps included in performing binary search.
9. Describe the analysis of binary search with example.
10. Binary searching can only be applied to a collection that allows random access. Comment.

Answers: Self Assessment

1. sequential
2. O(N)
3. worst
4. -1
5. N-1
6. Pointer
7. size
8. brute force search
9. False
10. True
11. True
12. False
13. False
14. True
15. True

14.6 Further Readings

Notes

Books

Sartaj Sahni, 1976, Fundamentals of data structures, Computer Science Press

Samir Kumar Bandyopadhyay, 2009, Data Structures Using C, Pearson Education India

Karthikeyan, Fundamentals Data Structures And Problem Solving, PHI Learning Pvt. Ltd

Davidson, 2004, Data Structures (Principles And Fundamentals), Dreamtech Press

Online links

http://bcahub.com/sy-bca/dfs/sorting-and-searching/binary-search-in-data-structure/

http://www.ustudy.in/node/2749

http://datastructures2010.blogspot.in/2010/07/linear-search-and-binary-search.html

http://www.programsformca.com/2012/02/data-structure-searching-algo-c-program.html