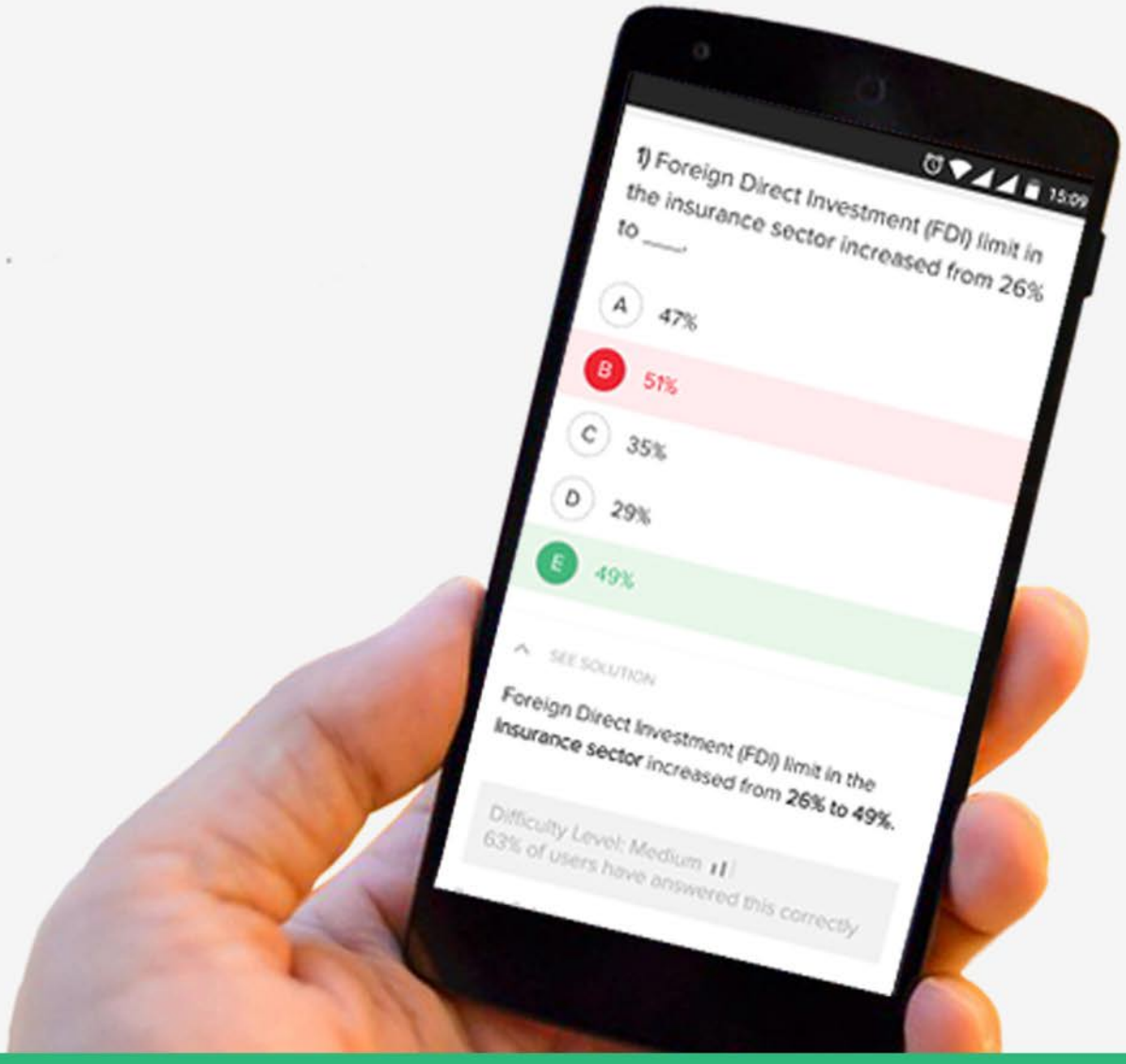
gradeup

Formulas on PROGRAMMING & DATA STRUCTURES

for GATE CS Exam



www.gradeup.co

Basic Data Types

char	1 byte (8 bits) with range -128 to 127
int	16-bit OS : 2 bytes with range -32768 to 32767 32-bit OS : 4 bytes with range -2,147,483,648 to 2,147,483,647
float	4 bytes with range 10^{-38} to 10^{38} with 7 digits of precision
double	8 bytes with range 10^{-308} to 10^{308} with 15 digits of precision
void	generic pointer, used to indicate no function parameters etc.

Note: Except for type void *the meaning of the above basic types may be altered when combined with the following keywords. signed, unsigned, long, short*

Variables: A variable is a named piece of memory which is used to hold a value which may be modified by the program. A variable thus has three attributes that are of interest to us : its type, its value and its address.

type variable-list ;

Variables are declared in three general areas in a C program: 1) Local variables, 2) Global variables, and 3) Formal parameters of the function.

Storage Classes: There are four storage class modifiers used in C which determine an identifier's storage duration and scope. ***auto, static , register, extern***

Storage class	Auto	Static	Register	Extern
Default Initial Value	Garbage value	0 (Zero)	Garbage	0(Zero)
Location	RAM	RAM	CPU registers	RAM
Scope	Local to thevariable where the variable is defined	Local to the variable where the variable is defined	Local to the variable where the variable is defined	Entire Program
Life	As long as the control is within the block where the variable is defined	As long as the program is under execution	As long as the control is within the block where the variable is defined	As long as the program is under execution

Constants: Constants are fixed values that cannot be altered by the program and can be numbers, characters or strings.

Examples :-

char : 'a', '$', '7'
int : 10, 100, -100
unsigned : 0, 255
float : 12.23456, -1.573765e10, 1.347654E-13
double : 1433.34534545454, 1.35456456456456E-200
long : 65536, 2222222
string : “Hello World\n”

Special character constants: Also called *Escape Sequences*, which are preceded by the backslash character '\', and have special meanings in C.

\n	newline
\t	tab
\b	backspace
\'	single quote
\"	double quote
\0	null character
\xdd	represent as hexadecimal constant

printf() : *The printf() function is used for formatted output and uses a control string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required. The more common format specifiers are given below*

%c character	%f floating point
%d signed integer	%lf double floating point
%i signed integer	%e exponential notation
%u unsigned integer	%s string
%ld signed long	%x unsigned hexadecimal
%lu unsigned long	%o unsigned octal
	%% prints a % sign

Field Width Specifiers: Field width specifiers are used in the control string to format the numbers or characters output appropriately .

Syntax :- **%[total width printed][.decimal places printed]format specifier**

where square braces indicate optional arguments.

For Example :-
int i = 15 ;

```
float f = 13.3576 ;
printf( "%3d", i ) ;   /* prints "_15 " where _ indicates a space
  character */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width
  of 6 and displays 2 decimal places  */
printf( “%*.*f”, 6,2,f ) ;   /* prints  "_13.36" as above. Here * is used as replacement character
for field widths     */
```

Assignment Operator

lvalue -- left hand side of an assignment operation
rvalue -- right hand side of an assignment operation

```
int x ;
x = 20 ;
```

Arithmetic Operators

+ - * / --- same rules as mathematics with * and / being evaluated before + and -.
% -- modulus / remainder operator

Increment and Decrement Operators

There are two special unary operators in C, Increment ++, and Decrement -- , which cause the variable they act on to be incremented or decremented by 1 respectively.

Special Assignment Operators

Many C operators can be combined with the assignment operator as shorthand notation.
+=, -=, *=, /=, %=, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

Relational Operators

The full set of relational operators are provided in shorthand notation

> >= < <= == !=

Logical Operators

&&	--	Logical AND
||	--	Logical OR
!	--	Logical NOT

Bitwise Operators

These are special operators that act on **char or int arguments only**. They allow the programmer to get closer to the machine level by operating at bit-level in their arguments.

&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR	~	Ones Complement
>>	Shift Right	<<	Shift left

Implicit & Explicit Type Conversions

Normally in mixed type expressions all operands are converted **temporarily** up to the type of the largest operand in the expression.

Normally this automatic or implicit casting of operands follows the following guidelines in ascending order.

long double
double
float
unsigned long
long
unsigned int
signed int

Explicit casting coerces the expression to be of specific type and is carried out by means of the **cast operator** which has the following syntax.

Syntax : **(type) expression**

Sizeof Operator

The sizeof operator gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types as we will see later on).

Syntax : ***sizeof (expression)***

Precedence of Operators

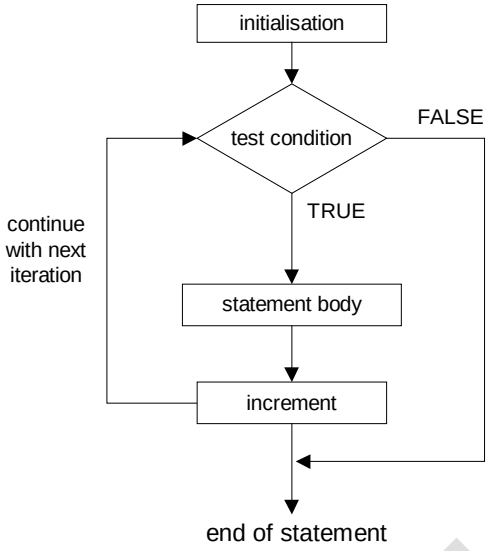
Precedence	Operator	Associativity
Highest	( ) [ ] -> .	left to right
	! ~ ++ -- +(unary) -(unary) (type) * & sizeof	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= &= ^= |= <<= >>=	right to left
Lowest	,	left to right

for statement

Syntax : **for ([initialisation] ; [condition] ; [increment])**
[statement body] ;

initialisation :- this is usually an assignment to set a loop counter variable for example.
condition :- determines when loop will terminate.
increment :- defines how the loop control variable will change each time the loop is executed.
statement body :- can be a single statement, no statement or a block of statements.

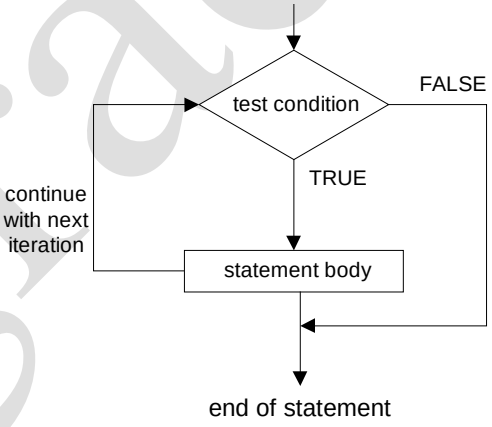
The for statement executes as follows :-



while statement

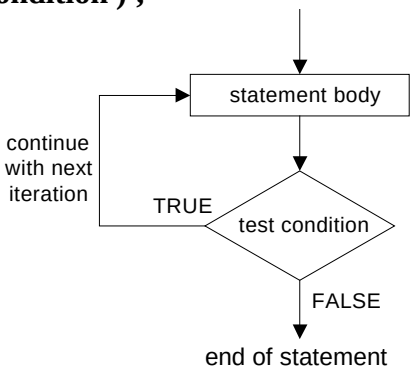
The ***while*** statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax : **while (condition)**
statement body ;



do while

Syntax : **do**
{
statement body ;
} while (condition) ;



break statement

When a ***break*** statement is encountered inside a while, for, do/while or switch statement the statement is immediately terminated and execution resumes at the next statement following the statement.

continue statement

The ***continue*** statement terminates the current iteration of a while, for or do/while statement and resumes execution back at the beginning of the loop body with the next iteration.

if statement

The ***if*** statement is the most general method for allowing conditional execution in C.

Syntax : **if (condition)**
statement body ;
else
statement body ;
or just :
if (condition)
statement body ;

Conditional Operator :- ?:

This is a special shorthand operator in C and replaces the following segment

```
if ( condition )
        expr_1 ;
else
        expr_2 ;
```

with the more elegant

condition ? expr_1 : expr_2 ;

The switch Statement

This is a multi-branch statement similar to the if - else ladder (with limitations) but clearer and easier to code.

Syntax : **switch (expression)**

```
{
case constant1 :   statement1 ;
                   break ;

case constant2 :   statement2 ;
                   break ;

...

default :   statement ;
}
```

Functions

Syntax : **return_type function_name (parameter_list)**

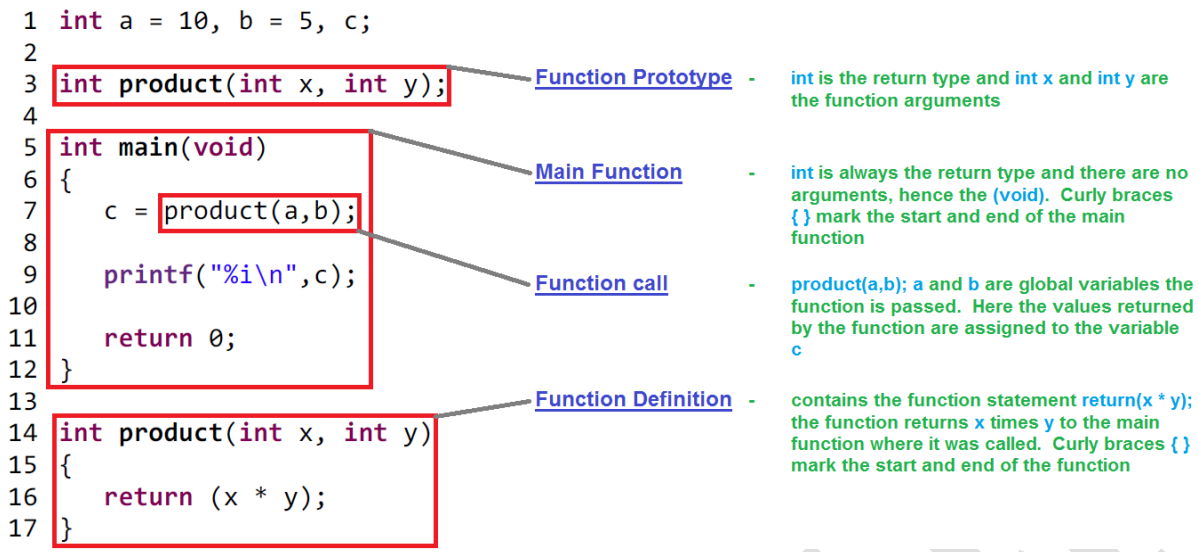
```
{
body of function ;
}
```

Function Prototype (declaration)

Syntax : **type_spec function_name(type_par1, type_par2, etc.);**

This declaration simply informs the compiler what type the function returns and what type and how many parameters it takes. Names may or may not be given to the parameters at this time.

Example:



Recursion: A recursive function is a function that calls itself either directly or indirectly through another function.

Arrays: An array is a collection of variables of the same type that are referenced by a common name. Specific elements or variables in the array are accessed by means of an index into the array.

- In C all arrays consist of contiguous memory locations. The lowest address corresponds to the first element in the array while the largest address corresponds to the last element in the array.
- C supports both single and multi-dimensional arrays.

Single Dimension Arrays

Syntax : **type var_name[size] ;**

where type is the type of each element in the array, var_name is any valid C identifier, and size is the number of elements in the array which has to be a constant value.

Note : In C all arrays use zero as the index to the first element in the array.

Strings: In C a string is defined as a character array which is terminated by a special character, the null character '\0', as there is no string type as such in C. Thus the string or character array must always be defined to be one character longer than is needed in order to cater for the '\0'.

A string constant is simply a list of characters within double quotes e.g. "Hello" with the '\0' character being automatically appended at the end by the compiler.

Multidimensional Arrays: Multidimensional arrays of any dimension are possible in C but in practice only two or three dimensional arrays are workable. The most common multidimensional array is a two dimensional array for example the computer display, board games, a mathematical matrix etc.

Syntax : **type name [rows] [columns] ;**

For Example :- 2D array of dimension 2 X 3.

int d[2] [3] ;

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

A two dimensional array is actually an array of arrays, in the above case an array of two integer arrays (the rows) each with three elements, and is stored row-wise in memory.

Pointers: A pointer is a variable that is used to store a memory address. Most commonly the address is the location of another variable in memory.

Pointer Variables:

Syntax : **type *ptr ;**

which indicates that *ptr* is a pointer to a variable of type *type*.
The type of the pointer variable ptr is ***int ****. The declaration of a pointer variable normally sets aside just two or four bytes of storage for the pointer whatever it is defined to point to.

Pointer Operators * and & : & is a unary operator that returns the address of its operand which must be a variable. The * operator is the complement of the address operator & and is normally termed the indirection operator. Like the & operator it is a unary operator and it returns the value of the variable located at the address its operand stores.

Structures: A structure is a customised user-defined data type in C. It is by definition a collection of variables of any type that are referenced under one name, providing a convenient means of keeping related information together.

structure definition :- the template used to create structure variables.
structure elements :- the member variables of the structure type

Defining Structures

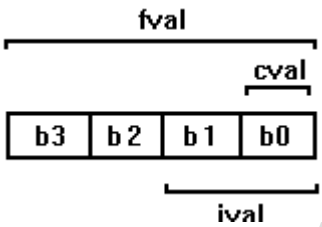
Syntax :

```
struct  tag   {
        type  var_1 ;
        type var_2 ;
        ...
type var_n ;
        }  ;
```

The keyword **struct** tells the compiler we are dealing with a structure and must be present whenever we refer to the new type, **tag** is an identifier which is the name given to the customised "type".

Unions: A union is data type where the data area is shared by two or more members generally of different type at different times.

For Example :-




```
union u_tag {
    short ival ;
    float fval ;
    char cval ;
     } uval ;
```

The size of uval will be the size required to store the largest single member, 4 bytes in this case to accommodate the floating point member.

ASCII Character Set:

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.

Arrays & Linked lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

	ArrayList	LinkedList
Structure	ArrayList is an index based data structure where each element is associated with an index.	Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.
Insertion And Removal	Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted.	Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed.
	Insertion and removal operations in ArrayList are of order O(n).	Insertion and removal in LinkedList are of order O(1).
Retrieval (Searching or getting an element)	Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.	Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element.
	Retrieval operation in ArrayList is of order of O(1).	Retrieval operation in LinkedList is of order of O(n).
Random Access	ArrayList is of type Random Access. i.e elements can be accessed randomly.	LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element.
Usage	ArrayList can not be used as a Stack or Queue.	LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List.
Memory Occupation	ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it's index.	LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements.
When To Use	If your application does more retrieval than the insertions and deletions, then use ArravList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out acces	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)*	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)* *(Similar trees good for disk storage)*	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex