```
alpha = malloc(sizeof(node));    /* allocate space for a node from the heap */
⋮
free(alpha);        /* return node to the heap */
```

In our notation, the equivalent statements would be

call GETNODE(α)

⋮

call RETNODE(α)

1.5 Putting it all together: formulating a solution to the ESP

Consider again the exam-scheduling problem: We are given n courses and a list of students enrolled in each course. We want to come up with a schedule of examinations 'without conflicts' using a minimal number of time slots. Let us now apply the ideas we discussed above in formulating a solution to this problem.

To be more specific, consider the courses, and corresponding class lists, shown in Figure 1.7. For brevity, only initials are shown; we assume that each initial identifies a unique student. We have here the problem data in its 'raw' form. We can deduce from this raw data two kinds of information, viz., values of certain quantities and relationships among certain data elements. For instance, we can easily verify that there are 11 courses, that there are 12 students enrolled in C01, 14 in C02, etc.; that the maximum class size is 15, that the average class size is 14, and so forth. In terms of relationships, we can see that RDA and ALB are classmates, that RDA and MCA are not, that C01 and C02 have two students in common, that C01 and C03 have none, and so on. Clearly, not all of this information is useful in solving the ESP. Our first task, then, is to extract from the given raw data those quantities and those relationships which are relevant to the problem we are supposed to solve, and to set aside those which are not.

Course	Students
C01	RDA ALB NAC EDF BMG VLJ IVL LGM EGN KSO EST VIV
C02	MCA EDF SLF ADG BCG AAI RRK LGM RLM JGP LRQ WAR KLS DDY
C03	ABC BBD GCF ADG AKL BCL MIN JGP RSQ DBU IEY RAW ESZ
C04	ANA JCA CAB NAC GCF GLH VLJ LLM MAN PEP PQQ ERR SET MAV REW
C05	BBC EDD HSE ELG ISH JEI EMJ RRK TPL RER EPS AVU CDW ELY
C06	ALA MCA ABB BCF GLH AKL HGN RON JGP ALQ EPR ABT KEV YEZ
C07	CDC ISH ABI DHJ ESM FBM RMN PEP VIR JLS LOT MAV TEX
C08	AAA HLA BBD WRE ECG HLH DHJ RON TSO PQQ MBT REW BAX TRY BDZ
C09	MCA JCA BCF EGG AAI XTK WIL CSM HLO RSP APR RER JET DBU
C10	RDA BBB CLC ECG MNH EMJ JOK ARM NFM EGN RCN RSP LEQ YIR AVU
C11	ADB WDB BKC CLC SDE UKF BMG HRH BTK LGM QJP EPS KLS BST YNZ

Figure 1.7 Class lists for 11 courses

A careful analysis of the exam-scheduling problem should convince you that the fact that RDA and ALB are both taking C01 is not relevant to the ESP, but the fact that RDA is taking both C01 and C10 is. More to the point, the *really* relevant information is that there is a student who is taking both C01 and C10, thereby establishing a relationship between C01 and C10; the identity of the student, in this case RDA, is in itself immaterial. Thus, in solving the ESP, *it suffices to simply indicate which courses have common students*.

The process by which we set aside extraneous information from the given raw data and cull only that which is relevant to the problem we are solving is called *data abstraction*. Our next task is to devise a *data representation* for those aspects of the problem data that we have abstracted out. This is where the notion of an ADT plays an important role.

The CS literature is replete with ADT's which have been found to be extremely versatile tools in solving problems on a computer. Many of these derive from familiar mathematical structures such as sets, trees, graphs, tables, and the like. It is not very often that we need to invent an entirely new ADT; usually, all it takes is to see an old one in a new light. Take, for instance, the graph. A graph may be defined, for now, as *a set of vertices connected by edges*. Figure 1.8 depicts a graph.

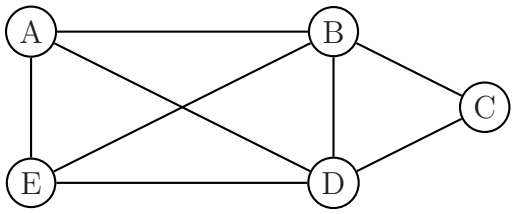


Figure 1.8 A graph on 5 vertices and 8 edges

Earlier, we found that to solve the ESP, it suffices to have a data representation which indicates which courses have common students. Can a graph serve this purpose? Indeed, it can, and in a most natural way. Specifically, we adopt the following conventions:

1. A vertex of the graph represents a course.
2. Two vertices (courses) are connected by an edge if one or more students are taking both courses. Such vertices are said to be *adjacent*.

Figure 1.9 shows the graph which represents the relevant relationships abstracted out from the data given in Figure 1.7 following the conventions stated above. For instance, vertices C01 and C02 are connected by an edge because two students, namely EDF and LGM, are taking both courses. In fact, LGM is also taking C11; hence, an edge joins vertices C01 and C11, and another edge joins vertices C02 and C11. Clearly, examinations for these three courses must be scheduled at different times. In general, the data representation shown in Figure 1.9 immediately identifies which courses *may not have* examinations scheduled at the same time; the crucial task, then, is to find a way to identify which courses *may have* examinations scheduled at the same time such that we minimize the number of time slots needed. You may want, at this point, to verify that the graph of Figure 1.9 does in fact capture all the relevant relationships implicit in the class lists of Figure 1.7.

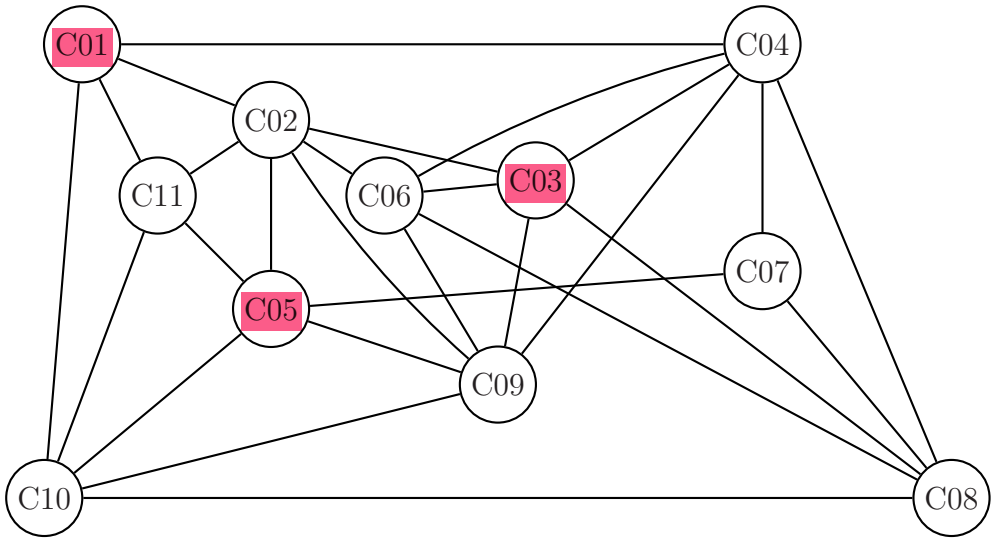


Figure 1.9 Graph representing the class lists of Figure 1.7

Now we get to the heart of the problem: using the graph of Figure 1.9, how do we determine which courses may have examinations scheduled at the same time? To this end, we apply a technique called *graph coloring*: we color the vertices of the graph such that no two adjacent vertices are assigned the same color, using the least number of colors. We then assign to the same time slot the examinations for all vertices (courses) which have the same color. In graph theory, the least number of colors needed to color a graph is called its *chromatic number*. In the context of the exam-scheduling problem, this number represents the minimum number of time slots needed for a conflict-free schedule. [Suggestion: Try coloring the graph of Figure 1.9; remember, no adjacent vertices may have the same color and use as few colors as possible.]

If you have colored the graph as suggested, you will have noticed that to obtain a *legal* coloring (no adjacent vertices are assigned the same color) is easy enough, but to obtain a coloring that is also *optimal* (uses as few colors as possible) is not quite that easy anymore. In fact, this latter requirement makes the graph coloring problem (and the ESP, as well) extremely difficult to solve; such a problem is said to be *intractable*. This problem has its beginnings in the middle of the nineteenth century (c. 1850); to date, the brightest minds in mathematics and in CS have yet to find an algorithm that is guaranteed to give an optimal solution, apart from the brute force approach in which we try all possibilities. Unfortunately, trying all possibilities takes exponential time in the size of the input, making this approach computationally very expensive even for graphs of 'practical' size. (More on this in Session 3.)

Actually, the graph coloring problem is just one in a large class of intractable problems called *NP-complete problems*, all of which take at least exponential time to solve. In practice, such problems are solved using algorithms which yield an approximate solution (for instance, one that is suboptimal) in a reasonable amount of time. Such an algorithm is called a *heuristic*.

For the graph coloring problem, the following heuristic produces a legal, though not necessarily optimal, coloring (AHO[1983], p. 5).

Algorithm G: Approximate graph coloring

1. Select an uncolored vertex and color it with a new color.
2. For each remaining uncolored vertex, say v, determine if v is connected by an edge to some vertex already colored with the new color; if it is not, color vertex v with the new color.
3. If there are still uncolored vertices, go to 1; else, stop.

Let us apply this algorithm to the graph of Figure 1.9. We choose C01 and color it red. C02, C04, C10 and C11 cannot be colored red since they are adjacent to C01; however, anyone among C03, C04 through C09 can be colored red. We choose C03 and color it red. Now, C06, C08 and C09 can no longer be colored red, but either C05 or C07 can be. We choose C05 and color it red; no more remaining uncolored vertex can be colored red. Next, we choose C02 and color it blue. C06, C09 and C11 cannot be colored blue, but any one among C04, C07, C08 and C10 can be. We choose C04 and color it blue. Now, C07 and C08 can no longer be colored blue, but C10 can be; we color C10 blue. Next, we choose C06 and color it green. C08 and C09 cannot be colored green, but C07 and C11 can be. We color C07 and C11 green. Finally, we color C08 and C09 yellow. Figure 1.10 summarizes the results.

Vertices	Color
C01, C03, C05	Red
C02, C04, C10	Blue
C06, C07, C11	Green
C08, C09	Yellow

Figure 1.10 A coloring of the graph of Figure 1.9

It turns out that the coloring of Figure 1.10 is optimal. To see this, note that vertices C02, C03, C06 and C09 constitute a subgraph in which there is an edge connecting every pair of vertices. Such a subgraph is called a *4-clique*. (Can you find another 4-clique in Figure 1.9?) Clearly, four colors are needed to color a 4-clique, so the coloring of Figure 1.10 is optimal. In terms of the original exam-scheduling problem, the colors are the time slots, and all vertices (courses) with the same color may have examinations assigned to the same time slot, for instance:

Courses	Schedule of exams
C01, C03, C05	Monday 8:30 – 11:30 AM
C02, C04, C10	Monday 1:00 – 4:00 PM
C06, C07, C11	Tuesday 8:30 – 11:30 AM
C08, C09	Tuesday 1:00 – 4:00 PM

Figure 1.11 An optimal solution to the ESP of Figure 1.7

We have solved a particular instance of the ESP. But more important than the solution that we obtained (Figure 1.11) is the conceptual framework that we developed to arrive at the solution. This is more important because now we have the means to solve *any other* instance of the ESP. At the heart of this conceptual framework is a data representation (a graph) which captures the essential relationships among the given data elements, and an algorithm (to color a graph, albeit suboptimally) which produces the desired result (or something that approximates this). These two, together, constitute an abstract data type. While our ultimate goal is to implement on a computer the method of solution that we have formulated, we have deliberately left out, until now, any consideration of implementation issues. We have formulated a solution to the ESP without having to think in terms of any specific programming language.

Finally, with our method of solution firmly established, we get down to details of implementation. This task involves:

1. choosing a *data structure* to represent a graph in the memory of a computer
2. writing the *code* to convert Algorithm G into a running program

One obvious way to represent a graph on n vertices is to define a matrix, say M, of size n by n, such that $M(i, j) = 1$ (or *true*) if an edge connects vertices i and j, and 0 (or *false*), otherwise. This is called the *adjacency matrix* representation of a graph (more on this in Session 9). Figure 1.12 depicts the adjacency matrix M for the graph of Figure 1.9.

$$M = \begin{array}{c} \\ C01 \\ C02 \\ C03 \\ C04 \\ C05 \\ C06 \\ C07 \\ C08 \\ C09 \\ C10 \\ C11 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} C01 & C02 & C03 & C04 & C05 & C06 & C07 & C08 & C09 & C10 & C11 \end{array} \\ \left[\begin{array}{ccccccccccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \end{array}$$

Figure 1.12 Adjacency matrix representation of the graph of Figure 1.9

The adjacency matrix representation of a graph is readily implemented as an *array* in many programming languages. For instance, we could specify the matrix M as follows:

Language	Array declaration
C	**int** M[11][11];
Pascal	**var** M:**array**[1..11,1..11] **of** 0..1;
FORTRAN	**integer**∗1 M(1:11,1:11)

Having chosen a particular data structure, in this case a two-dimensional array, to represent a graph, we finally set out to write the code to implement Algorithm G as a computer program. This is a non-trivial but straightforward task which is left as an exercise for you to do. With this, our solution to the exam-scheduling problem is complete.

Summary

- Solving a problem on a computer involves two intimately related tasks: the structuring of data and the synthesis of algorithms. *Data structures* and *algorithms* are the building blocks of computer programs.
- A set of data elements plus a set of operations defined on the data elements constitute an *abstract data type* (ADT). The word 'abstract' means an ADT is implementation independent.
- The implementation, or realization, of an ADT in terms of the constructs of a specific programming language is a *data structure*. The usual implementations are *sequential* (using arrays) and *linked* (using dynamic variables), or a mix of the two.
- Thinking in terms of ADT's when conceptualizing a solution to a problem helps one cope with the complexities of the problem-solving process on a computer. Only after the solution has been completely formulated should one begin to transcribe it into a running program.
- An *algorithm* is a finite set of well-defined instructions for solving a problem in a finite amount of time. An algorithm can be carried out 'by hand' or executed by a computer.
- An algorithm must terminate after a finite amount of time. An algorithm which, though finite, takes years or centuries or millenia to execute is not finite enough to be of any practical use. An example of such an algorithm is Cramer's rule for solving systems of linear equations. Much faster algorithms are available to solve such linear systems.
- The *only* algorithm that is guaranteed to yield an optimal solution to the exam-scheduling problem (ESP) would require an inordinately large amount of time even for inputs of moderate size. A problem such as the ESP is said to be *intractable*. The ESP can be solved suboptimally in a reasonable amount of time by using an approximate algorithm, or a *heuristic*.

Exercises

1. Consult a book on Automata Theory and make a list of five unsolvable problems. Consult a book on Algorithms and make a list of five tractable and five intractable problems.
2. What happens if initially $m < n$ in Euclid's algorithm?

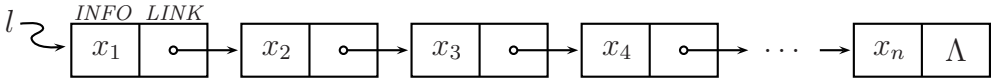
3. Apply Euclid's algorithm to find the GCD of the following pairs of integers.

   (a) 2057 and 1331 (b) 469 and 3982 (c) 610 and 377 (d) 611953 and 541

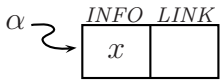
4. Give a recursive version of Euclid's algorithm.

5. Find the time required to solve n linear equations in n unknowns using Cramer's rule on a computer that can perform 1 *trillion* multiplications per second for $n =$ 19, 21, 22, 23 and 24.

6. Given a linked list pointed to by l



   and a new node pointed to by α:



   (a) Write an algorithm to insert the new node at the head of the list, i.e., node α becomes the first node of list l.

   (b) Write an algorithm to insert the new node at the tail of the list, i.e., node α becomes the last node of list l.

   (c) Assume that the nodes in list l are sorted in increasing order of the contents of the *INFO* field, i.e., $x_1 < x_2 < x_3 < \ldots < x_n$. Write an algorithm to insert the new node into the list such that the list remains sorted.

7. (a) Draw the graph which represents the relevant information to solve the exam-scheduling problem for the class lists given below.
   (b) Apply Algorithm G to color the graph in (a). Is the coloring optimal?

C01	ADB	WDB	BKC	OBC	SDE	UKF	BMG	NOH	BTK	LGM	EPS	KLS	ODT	YNZ
C02	BEB	XEB	CLC	DMC	TEE	VLF	CNG	ISH	CUK	MHM	RKP	FQS	LMS	CTT	YNZ
C03	SEA	OBC	DMC	FDG	NOH	FNJ	KPK	BSM	OGM	FHN	SDN	STP	MFQ	ZJR	BWU
C04	NDA	KDA	CDF	FHG	BBI	YUK	XJL	DTM	LMO	STP	BQR	SFR	KFT	ECU	
C05	BBB	IMA	CCD	XSE	FDG	IMH	EIJ	SPN	UYO	ERQ	ODT	SFW	CBX	USY	CEZ
C06	DEC	JTH	BCI	EIJ	FTM	GCM	SNN	QFP	WJR	KMS	MPT	NBV	UFX		
C07	BMA	NDA	BCB	CDF	HMH	BLL	IHN	SPN	KHP	BMQ	FQR	BCT	LFV	ZFZ	
C08	CCC	FED	ITE	FMG	JTH	KFI	FNJ	SSK	UQL	PEP	SFR	FQS	BWU	DEW	FMY
C09	BOA	KDA	DBB	OBC	HDF	HMH	WMJ	MMM	NBN	QFP	ERQ	FSR	TFT	NBV	SFW
C10	BCC	CCD	HDF	BEG	BLL	CDL	NJN	KHP	STQ	ECU	JFY	SBW	FTZ		
C11	NDA	FEF	TMF	BEG	CDG	BBI	SSK	MHM	SMM	KHP	MSQ	XBR	LMS	EEY	
C12	SEA	BMB	OBC	FEF	CNG	WMJ	JWL	MHM	FHN	LTO	PEP	FTT	WJV		

8. Algorithm G is a *greedy algorithm* in that it tries to color as many uncolored vertices as it legally can with the current color before turning to another color. The idea is that this will result in as few colors as possible. Show that this technique does *not* necessarily yield an optimal coloring for the graph in Item 7(a).

9. Write a program, using a language of your choice, to implement the solution to the ESP as formulated in this session. Program input will consist of a set of class lists

in the manner of Figure 1.7. Program output should consist of: (a) the adjacency matrix for the graph which indicates which courses have common students, in the manner of Figure 1.12, and (b) the generated schedule of exams, in the manner of Figure 1.11. Test your program using: (a) the class lists in Item 7, and (b) the class lists given below.

C01	RDA	ALB	NAC	EDF	BMG	VLJ	IVL	LGM	EGN	KSO	EST	VIV			
C02	MCA	EDF	SLF	ADG	BCG	AAI	RRK	LGM	RLM	JGP	LRQ	WAR	KLS	DDY	
C03	ABC	BBD	GCF	ADG	AKL	BCL	MIN	JGP	RSQ	DBU	IEY	RAW	ESZ		
C04	ANA	JCA	CAB	NAC	GCF	GLH	VLJ	LLM	MAN	PEP	PQQ	ERR	SET	MAV	REW
C05	BBC	EDD	HSE	ELG	ISH	JEI	EMJ	RRK	TPL	RER	EPS	AVU	CDW	ELY	
C06	ALA	MCA	ABB	BCF	GLH	AKL	HGN	RON	JGP	ALQ	EPR	ABT	KEV	YEZ	
C07	CDC	ISH	ABI	DHJ	ESM	FBM	RMN	PEP	VIR	JLS	LOT	MAV	TEX		
C08	AAA	HLA	BBD	WRE	ECG	HLH	DHJ	RON	TSO	PQQ	MBT	REW	BAX	TRY	BDZ
C09	MCA	JCA	BCF	EGG	AAI	XTK	WIL	CSM	HLO	RSP	APR	RER	JET	DBU	
C10	RDA	BBB	CLC	ECG	MNH	EMJ	JOK	ARM	NFM	EGN	RCN	RSP	LEQ	YIR	AVU
C11	ADB	WDB	BKC	CLC	SDE	UKF	BMG	HRH	BTK	LGM	QJP	EPS	KLS	BST	YNZ

Bibliographic Notes

The basic concepts discussed in this introductory session can be found in most books on Data Structures. The references listed in the READINGS for this session are particularly recommended. Section 1.2 (*Blending Mathematics, Science and Engineering*) and section 1.3 (*The Search for Enduring Principles in Computer Science*) of STANDISH[1994] provide an excellent take-off point for our study of data structures and algorithms. Section 1.1 (*Algorithms*) of KNUTH1[1997] is a nifty exposition on the concept of an algorithm. Section 1.1 (*From Problems to Programs*) of AHO[1983] provides us with a template to follow as we solve problems on a computer. Our approach in solving the exam-scheduling problem (ESP) parallels that given in this reference in solving a traffic light controller problem.