Unit 14 Advanced Data Representation

Structure:

14.1 Introduction

Learning a computer language is like learning music, carpentry, or engineering. At first, you work with the tools of the trade, playing scales, learning which end of the hammer to hold and which end to avoid, solving countless problems involving falling, sliding, and balanced objects. Acquiring and practicing skills is what you've been doing so far in this book, learning to create variables, structures, functions, and the like. Eventually, however, you move to a higher level in which using the tools is second nature and the real challenge is designing and creating a project. You develop an ability to see the project as a coherent whole. This unit concentrates on that higher level. You may find the material covered here a little more challenging than the preceding units, but you may also find it more rewarding as it helps you move from the role of apprentice to the role of craftsperson.

We'll start by examining a vital aspect of program design: the way a program represents data. Often the most important aspect of program development is finding a good representation of the data manipulated by that program. Getting data representation right can make writing the rest of the program simple. By now you've seen C's built-in data types: simple variables, arrays, pointers, structures, and unions.

Finding the right data representation, however, often goes beyond simply selecting a type. You should also think about what operations will be necessary. That is, you should decide how to store the data, and you should define what operations are valid for the data type. For example, C implementations typically store both the C int type and the C pointer type as integers, but the two types have different sets of valid operations. You can multiply one integer by another, for example, but you can't multiply a pointer by a pointer. You can use the * operator to dereference a pointer, but that operation is meaningless for an integer. The C language defines the valid operations for its fundamental types. However, when you design a scheme to represent data, you might need to define the valid operations yourself. In C, you can do so by designing C functions to represent the desired operations. In short, then, designing a data type consists of deciding on how to store the data and of designing a set of functions to manage the data.

You will also look at some algorithms, recipes for manipulating data. As a programmer, you will acquire a repertoire of such recipes that you apply over and over again to similar problems.

This unit looks into the process of designing data types, a process that matches algorithms to data representations. In it, you'll meet some common data forms, such as the stacks and queues.

You'll also be introduced to the concept of the abstract data type (ADT). An ADT packages methods and data representations in a way that is problem-oriented rather than language-oriented. After you've designed an ADT, you can easily reuse it in different circumstances. Understanding ADTs prepares you conceptually for entering the world of object-oriented programming (OOP) and the C++ language.

Objectives:

After studying this unit, you should be able to:

- use C to represent a variety of data types
- implement the new algorithms and increasing your ability to develop programs conceptually
- implement abstract data types (ADTs)
- implement two Abstract Data Types-stacks and queues
- discuss the applications of stacks and queues

14.2 Exploring Data Representation

Let's begin by thinking about data. Suppose you had to create an address book program. What data form would you use to store information? Because there's a variety of information associated with each entry, it makes sense to represent each entry with a structure. How do you represent several entries? With a standard array of structures? With a dynamic array? With some other form? Should the entries be alphabetized? Should you be able to search through the entries by zip code? by area code? The actions you want to perform might affect how you decide to store the information. In short, you have a lot of design decisions to make before plunging into coding.

How would you represent a bitmapped graphics image that you want to store in memory? A bitmapped image is one in which each pixel on the screen is set individually. In the days of black-and-white screens, you could use one computer bit (1 or 0) to represent one pixel (on or off), hence the name bitmapped. With color monitors, it takes more than one bit to describe a single pixel. For example, you can get 256 colors if you dedicate 8 bits to each pixel. Now the industry has moved to 65,536 colors (16 bits per pixel), 16,777,216 colors (24 bits per pixel), 2,147,483,648 colors (32 bits per pixel), and even beyond. If you have 16 million colors and if your monitor has a resolution of 1024x768, you'll need 18.9 million bits (2.25 MB) to represent a single screen of bitmapped graphics. Is this the way to go, or can you develop a way of compressing the information? Should this compression be lossless (no data lost) or lossy (relatively unimportant data lost)? Again, you have a lot of design decisions to make before diving into coding.

Let's tackle a particular case of representing data. Suppose you want to write a program that enables you to enter a list of all the movies (including videotapes) you've seen in a year. For each movie, you'd like to record a variety of information, such as the title, the year it was released, the director, the lead actors, the length, the kind of film (comedy, science fiction, romance, drivel, and so forth), your evaluation, and so on. That suggests using a structure for each film and an array of structures for the list. To simplify matters, let's limit the structure to two members: the film title and your evaluation, a ranking on a 0 to 10 scale. Program 6.1 shows a bare-bones implementation using this approach.

Program 14.1: The *films1.c* Program

```
/* films1.c -- using an array of structures */
#include <stdio.h>
#define TSIZE       45      /* size of array to hold title   */
#define FMAX         5      /* maximum number of film titles */
struct film {
   char title[TSIZE];
   int rating;
} ;
int main(void)
{
   struct film movies[FMAX];
   int i = 0;
   int j;

   puts(“Enter first movie title:”);
   while (i < FMAX && gets(movies[i].title) != NULL &&
      movies[i].title[0] != '\0')
   {
      puts("Enter your rating <0-10>:");
      scanf("%d", &movies[i++].rating);
      while(getchar() != '\n')
         continue;
      puts("Enter next movie title (empty line to stop):");
   }
   if (i == 0)
```

```
    printf("No data entered. ");
  else
    printf ("Here is the movie list:\n");
  for (j = 0; j < i; j++)
    printf("Movie: %s  Rating: %d\n", movies[j].title,
      movies[j].rating);
  printf("Bye!\n");
  return 0;
}
```

The program creates an array of structures and then fills the array with data entered by the user. Entry continues until the array is full (the FMAX test), until end of file (the NULL test) is reached, or until the user presses the Enter key at the beginning of a line (the '\0' test).

This formulation has some problems. First, the program will most likely waste a lot of space because most movies don't have titles 40 characters long, but some movies do have long titles. Second, many people will find the limit of five movies a year too restrictive. Of course, you can increase that limit, but what would be a good value? Some people see 500 movies a year, so you could increase FMAX to 500, but that still might be too small for some, yet it might waste enormous amounts of memory for others. Also, some compilers set a default limit for the amount of memory available for automatic storage class variables such as movies, and such a large array could exceed that value. You can fix that by making the array a static or external array or by instructing the compiler to use a larger stack, but that's not fixing the real problem.

The real problem here is that the data representation is too inflexible. You have to make decisions at compile time that are better made at runtime. This suggests switching to a data representation that uses dynamic memory allocation. You could try something like this:

```
#define TSIZE  45          /* size of array to hold title  */
struct film {
  char title[TSIZE];
  int rating;
} ;
...
```

```
int n, i;
struct film * movies;       /* pointer to a structure       */
...
printf("Enter the maximum number of movies you'll enter:\n");
scanf("%d", &n);
movies = (struct film *) malloc(n * sizeof(struct film));
```

Here, you can use the pointer movies just as though it were an array name:

```
while (i < FMAX && gets(movies[i].title) != NULL &&
        movies[i].title[0] != '\0')
```

By using malloc(), you can postpone determining the number of elements until the program runs, so the program need not allocate 500 elements if only 20 are needed. However, it puts the burden on the user to supply a correct value for the number of entries.

Self Assessment Questions

1. A _____ image is one in which each pixel on the screen is set individually.
2. The best solution to allocate memory locations for the objects when the actual number of locations is not known in advance is by using _____ memory allocations.

14.3 Abstract Data Types

An Abstract Data Type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and implementation of the operations.

Some programming languages provide explicit mechanisms to support the distinction between specification and implementation. Although C does not have an explicit mechanism for implementing ADTs, it is still possible and desirable to design your data types using the same notion.

How does the specification of the operations of an ADT differ from the implementation of the operations? The specification consists of the names of every function, the type of its arguments, and the type of its result. There should also be a description of what the function does, but without appealing to internal representation or implementation details. This requirement is

quite important, and it implies that an abstract data type is implementation independent. Furthermore, it is possible to classify the functions of a data type into several categories:

1) **Creator/constructor:** These functions create a new instance of the designated type.
2) **Transformers:** These functions also create an instance of the designated type, generally by using one or more other instances.
3) **Observers/reporters:** These functions provide information about an instance of the type, but they do not change the instance.

Typically, an ADT definition will include at least one function from each of these three categories.

14.4 Stack as an Abstract Data Type

A stack is simply a list of elements with insertions and deletions permitted at one end – called the stack top. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure exhibits the LIFO (last in first out) property. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively. Shown in Figure 14.1 are the effects of push and pop operations on the stack.

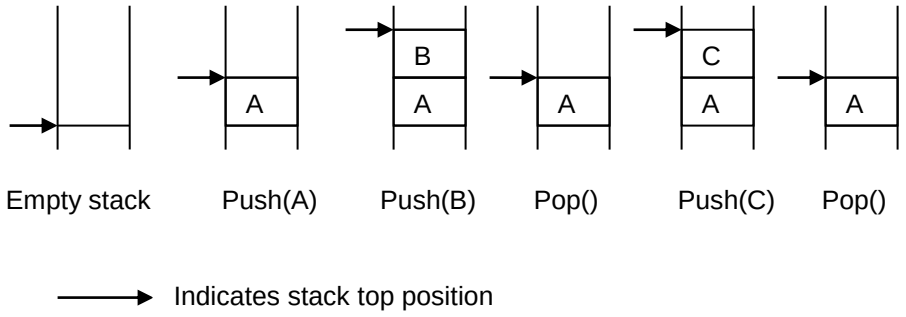

Figure 14.1: Stack operations

Since a stack is basically a list, it can be implemented by using an array or by using a linked representation.

14.4.1 Array Implementation of a Stack

When an array is used to implement a stack, the push and pop operations are realized by using the operations available on an array. The limitation of

an array implementation is that the stack cannot grow and shrink dynamically as per the requirement.

Program 14.2: A complete C program to implement a stack using an array appears here:

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the stack */
#include <stdlib.h>
void push(int stack[], int *top, int value)
{
  if(*top < MAX )
  {
     *top = *top + 1;
     stack[*top] = value;
  }
 else
 {
     printf("The stack is full can not push a value\n");
     exit(0);
  }
}
void pop(int stack[], int *top, int * value)
{
  if(*top >= 0 )
  {
      *value = stack[*top];
     *top = *top - 1;
  }
  else
  {
     printf("The stack is empty can not pop a value\n");
     exit(0);
  }
}
void main()
{
```

```
  int stack[MAX];
  int top = -1;
  int n,value;
  do
  {
     do
     {
          printf("Enter the element to be pushed\n");
        scanf("%d",&value);
        push(stack,&top,value);
          printf("Enter 1 to continue\n");
        scanf("%d",&n);
     } while(n == 1);

     printf("Enter 1 to pop an element\n");
     scanf("%d",&n);
     while( n == 1)
     {
          pop(stack,&top,&value);
        printf("The value popped is %d\n",value);
          printf("Enter 1 to pop an element\n");
          scanf("%d",&n);
     }
     printf("Enter 1 to continue\n");
     scanf("%d",&n);
  } while(n == 1);
}
```

14.4.2 Implementation of a Stack Using Linked Representation

Initially the list is empty, so the top pointer is NULL. The push function takes a pointer to an existing list as the first parameter and a data value to be pushed as the second parameter, creates a new node by using the data value, and adds it to the top of the existing list. A pop function takes a pointer to an existing list as the first parameter, and a pointer to a data object in which the popped value is to be returned as a second parameter. Thus it retrieves the value of the node pointed to by the top pointer, takes

the top point to the next node, and destroys the node that was pointed to by the top.

If this strategy is used for creating a stack with the four data values: 10, 20, 30, and 40, then the stack is created as shown in Figure 14.2. Initially top=NULL.

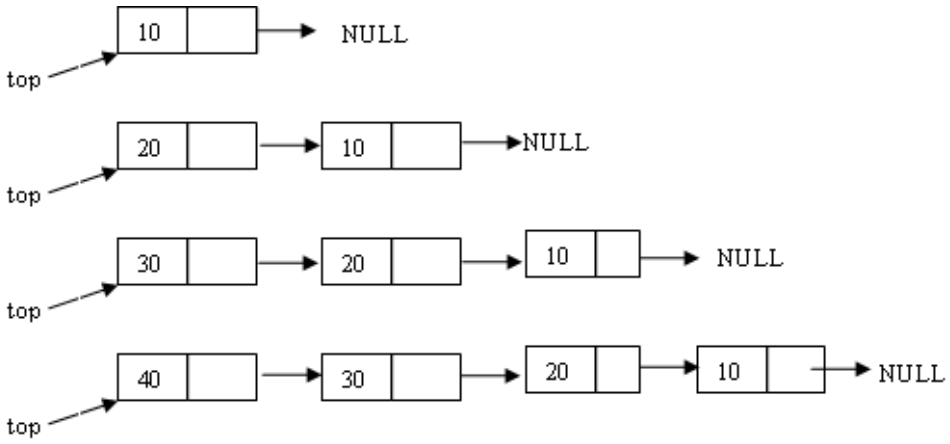

Figure 14.2: Linked stack

Program 14.3: A complete C program for implementation of a stack using the linked list is given here

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
  int data;
  struct node *link;
};
struct node *push(struct node *p, int value)
{
  struct node *temp;
  temp=(struct node *)malloc(sizeof(struct node));
     /* creates new node
     using data value
     passed as parameter */
  if(temp==NULL)
```

```
  {
    printf("No Memory available Error\n");
    exit(0);
  }
  temp->data = value;
  temp->link = p;
  p = temp;
  return(p);
}

struct node *pop(struct node *p, int *value)
{
  struct node *temp;
  if(p==NULL)
  {
    printf(" The stack is empty can not pop Error\n");
    exit(0);
  }
  *value = p->data;
  temp = p;
  p = p->link;
  free(temp);
  return(p);
}

void main()
{
  struct node *top = NULL;
  int n,value;
  do
  {
    do
    {
        printf("Enter the element to be pushed\n");
      scanf("%d",&value);
      top = push(top,value);
```

```
        printf("Enter 1 to continue\n");
        scanf("%d",&n);
      } while(n == 1);

      printf("Enter 1 to pop an element\n");
      scanf("%d",&n);
      while( n == 1)
      {
           top = pop(top,&value);
        printf("The value popped is %d\n",value);
           printf("Enter 1 to pop an element\n");
           scanf("%d",&n);
      }
      printf("Enter 1 to continue\n");
      scanf("%d",&n);
    } while(n == 1);
}
```

14.4.3 Applications of Stacks

One of the applications of the stack is in expression evaluation. A complex assignment statement such as a = b + c*d/e–f may be interpreted in many different ways. Therefore, to give a unique meaning, the precedence and associativity rules are used. But still it is difficult to evaluate an expression by computer in its present form, called the infix notation. In *infix notation*, the binary operator comes in between the operands. A unary operator comes before the operand. To get it evaluated, it is first converted to the postfix form, where the operator comes after the operands. For example, the postfix form for the expression a*(b–c)/d is abc – *d/. A good thing about postfix expressions is that they do not require any precedence rules or parentheses for unique definition. So, evaluation of a postfix expression is possible using a stack-based algorithm.

Program 14.4: Program to convert an infix expression to prefix form

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define N 80
typedef enum {FALSE, TRUE} bool;
#include "stack.h"
#include "queue.h"
#define NOPS 7

char operators [] = "()^/*+-";
int priorities[] = {4,4,3,2,2,1,1};
char associates[] = " RLLLL";

char t[N]; char *tptr = t; // this is where prefix will be saved.
int getIndex( char op ) {
   /*
    * returns index of op in operators.
    */
   int i;
   for( i=0; i<NOPS; ++i )
      if( operators[i] == op )
            return i;
   return -1;
}
int getPriority( char op ) {
   /*
    * returns priority of op.
    */
   return priorities[ getIndex(op) ];
}

char getAssociativity( char op ) {
   /*
    * returns associativity of op.
    */
   return associates[ getIndex(op) ];
```

```
}
void processOp( char op, queue *q, stack *s ) {
   /*
    * performs processing of op.
    */
   switch(op) {
     case ')':
          printf( "\t S pushing )...\n" );
          sPush( s, op );
          break;
     case '(':
          while( !qEmpty(q) ) {
               *tptr++ = qPop(q);
               printf( "\tQ popping %c...\n", *(tptr-1) );
          }
          while( !sEmpty(s) ) {
               char popop = sPop(s);
               printf( "\tS popping %c...\n", popop );
               if( popop == ')' )
                    break;
               *tptr++ = popop;
          }
          break;
     default: {
          int priop;    // priority of op.
          char topop;   // operator on stack top.
          int pritop;   // priority of topop.
          char asstop;  // associativity of topop.
          while( !sEmpty(s) ) {
               priop = getPriority(op);
               topop = sTop(s);
               pritop = getPriority(topop);
               asstop = getAssociativity(topop);
if( (pritop < priop) || (pritop == priop && asstop == 'L') || (topop == ')' ))// IMP.
                break;
            while( !qEmpty(q) ) {
```

```
                    *tptr++ = qPop(q);
                printf( "\tQ popping %c...\n", *(tptr-1) );
                }
                *tptr++ = sPop(s);
                printf( "\tS popping %c...\n", *(tptr-1) );
        }
        printf( "\tS pushing %c...\n", op );
        sPush( s, op );
        break;
      }
   }
}
bool isop( char op ) {
    /*  is op an operator?  */
     return (getIndex(op) != -1);
}
char *in2pre( char *str ) { /*   returns valid infix expr in str to prefix. */
    char *sptr;
    queue q = {NULL};
    stack s = NULL;
    char *res = (char *)malloc( N*sizeof(char) );
    char *resptr = res;
    tptr = t;
    for( sptr=str+strlen(str)-1; sptr!=str-1; -sptr ) {
       printf( "processing %c tptr-t=%d...\n", *sptr, tptr-t );
       if( isalpha(*sptr) ) // if operand.
             qPush( &q, *sptr );
       else if( isop(*sptr) )      // if valid operator.
             processOp( *sptr, &q, &s );
       else if( isspace(*sptr) )    // if whitespace.
             ;
       else {
             fprintf( stderr, "ERROR:invalid char %c.\n", *sptr );
             return "";
       }
   }
```

```
  while( !qEmpty(&q) ) {
     *tptr++ = qPop(&q);
     printf( "\tQ popping %c...\n", *(tptr-1) );
  }
  while( !sEmpty(&s) ) {
     *tptr++ = sPop(&s);
     printf( "\tS popping %c...\n", *(tptr-1) );
  }
  *tptr = 0;
  printf( "t=%s.\n", t );
  for( -tptr; tptr!=t-1; -tptr ) {
    *resptr++ = *tptr;
  }
  *resptr = 0;
  return res;
}
int main() {
   char s[N];

   puts( "enter infix freespaces max 80." );
   gets(s);
   while(*s) {
     puts( in2pre(s) );
     gets(s);
   }

  return 0;
}
```

Explanation

1. In an infix expression, a binary operator separates its operands (a unary operator precedes its operand). In a postfix expression, the operands of an operator precede the operator. In a prefix expression, the operator precedes its operands. Like postfix, a prefix expression is parenthesis-free, that is, any infix expression can be unambiguously written in its prefix equivalent without the need for parentheses.

2. To convert an infix expression to reverse-prefix, it is scanned from right to left. A queue of operands is maintained noting that the order of operands in infix and prefix remains the same. Thus, while scanning the infix expression, whenever an operand is encountered, it is pushed in a queue. If the scanned element is a right parenthesis (‘)’), it is pushed in a stack of operators. If the scanned element is a left parenthesis (‘(‘), the queue of operands is emptied to the prefix output, followed by the popping of all the operators up to, but excluding, a right parenthesis in the operator stack.
3. If the scanned element is an arbitrary operator o, then the stack of operators is checked for operators with a greater priority than o. Such operators are popped and written to the prefix output after emptying the operand queue. The operator o is finally pushed to the stack.
4. When the scanning of the infix expression is complete, first the operand queue, and then the operator stack, are emptied to the prefix output. Any whitespace in the infix input is ignored. Thus the prefix output can be reversed to get the required prefix expression of the infix input.

Example

If the infix expression is a*b + c/d, then different snapshots of the algorithm, while scanning the expression from right to left, are shown in Table 14.1.

Table 14.1: Scanning the infix expression a*b+c/d from right to left

Step	Remaining expression	Scanned element	Queue of operands	Stack of operators	Prefix output
0	a*b+c/d	nil	empty	empty	nil
1	a*b+c/	D	d	empty	nil
2	a*b+c	/	d	/	nil
3	a*b+	C	d c	/	nil
4	a*b	+	empty	+	dc/
5	a*	B	b	+	dc/
6	A	*	b	* +	dc/
7	Nil	A	b a	* +	dc/
8	Nil	nil	empty	empty	dc/ba*+

The final prefix output that we get is dc/ba*+ whose reverse is +*ab/cd, which is the prefix equivalent of the input infix expression a*b+c*d. Note that all the operands are simply pushed to the queue in steps 1, 3, 5, and 7. In step 2, the operator / is pushed to the empty stack of operators.

In step 4, the operator + is checked against the elements in the stack. Since / (division) has higher priority than + (addition), the queue is emptied to the prefix output (thus we get 'dc' as the output) and then the operator / is written (thus we get 'dc/' as the output). The operator + is then pushed to the stack. In step 6, the operator * is checked against the stack elements. Since * (multiplication) has a higher priority than + (addition), * is pushed to the stack. Step 8 signifies that all of the infix expression is scanned. Thus, the queue of operands is emptied to the prefix output (to get 'dc/ba'), followed by the emptying of the stack of operators (to get 'dc/ba*+').

Points to remember

1. A prefix expression is parenthesis-free.
2. To convert an infix expression to the prefix equivalent, it is scanned from right to left. The prefix expression we get is the reverse of the required prefix equivalent.
3. Conversion of infix to prefix requires a queue of operands and a stack, as in the conversion of an infix to postfix.
4. The order of operands in a prefix expression is the same as that in its infix equivalent.
5. If the scanned operator o1 and the operator o2 at the stack top have the same priority, then the associativity of o2 is checked. If o2 is right-associative, it is popped from the stack.

Self Assessment Questions:

3. Stack is also called _____ data structure.
4. The drawback of implementing a stack using an array is:
   (a) Stack cannot grow
   (b) Stack can grow
   (c) Array cannot grow even if stack is empty
   (d) Array can grow infinitely
5. Postfix expression is used for Evaluation of Expressions. (True/False)

14.5 Queue as an Abstract Data Type

A *queue* is also a list of elements with insertions permitted at one end – called the rear, and deletions permitted from the other end – called the front. This means that the removal of elements from a queue is possible in the same order in which the insertion of elements is made into the queue. Thus, a queue data structure exhibits the *FIFO* (*first in first out*) property. Insert and delete are the operations that are provided for insertion of elements into the queue and the removal of elements from the queue, respectively. Shown in Figure 14.3 are the effects of insert and delete operations on the queue. Initially both front and rear are initialized to -1.

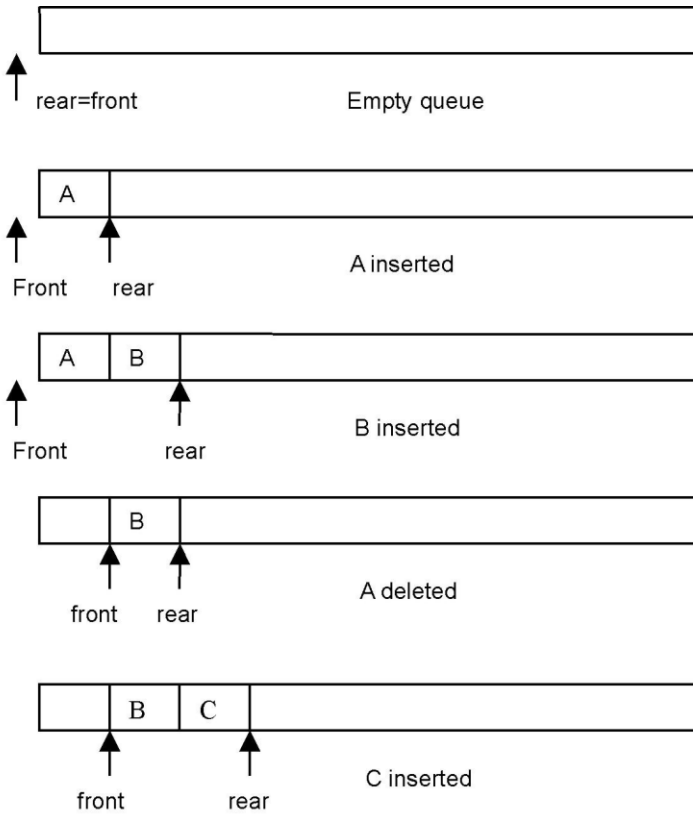

Figure 14.3: Queue operation

14.5.1 Array Implementation of a Queue

When an array is used to implement a queue, then the insert and delete operations are realized using the operations available on an array. The

limitation of an array implementation is that the queue cannot grow and shrink dynamically as per the requirement.

Program 14.5: A complete C program to implement a queue by using an array is shown here

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the queue */
#include <stdlib.h>

void insert(int queue[], int *rear, int value)
{
  if(*rear < MAX-1)
  {
    *rear= *rear +1;
    queue[*rear] = value;
  }
  else
  {
    printf("The queue is full can not insert a value\n");
    exit(0);
  }
}

void delete(int queue[], int *front, int rear, int * value)
{
  if(*front == rear)
  {
    printf("The queue is empty can not delete a value\n");
    exit(0);
  }
  *front = *front + 1;
  *value = queue[*front];
}

void main()
{
  int queue[MAX];
```

```
  int front,rear;
  int n,value;
  front=rear=(-1);
  do
  {
     do
     {
          printf("Enter the element to be inserted\n");
        scanf("%d",&value);
        insert(queue,&rear,value);
        printf("Enter 1 to continue\n");
           scanf("%d",&n);
     } while(n == 1);

     printf("Enter 1 to delete an element\n");
     scanf("%d",&n);
     while( n == 1)
     {
          delete(queue,&front,rear,&value);
        printf("The value deleted is %d\n",value);
           printf("Enter 1 to delete an element\n");
        scanf("%d",&n);
     }
     printf("Enter 1 to continue\n");
     scanf("%d",&n);
  } while(n == 1);
}
```

14.5.2 Implementation of a Queue using Linked List Representation

Initially, the list is empty, so both the front and rear pointers are NULL. The insert function creates a new node, puts the new data value in it, appends it to an existing list, and makes the rear pointer point to it. A delete function checks whether the queue is empty, and if not, retrieves the data value of the node pointed to by the front, advances the front, and frees the storage of the node whose data value has been retrieved.

If the above strategy is used for creating a queue with four data values – 10, 20, 30, and 40, the queue gets created as shown in Figure 14.4.

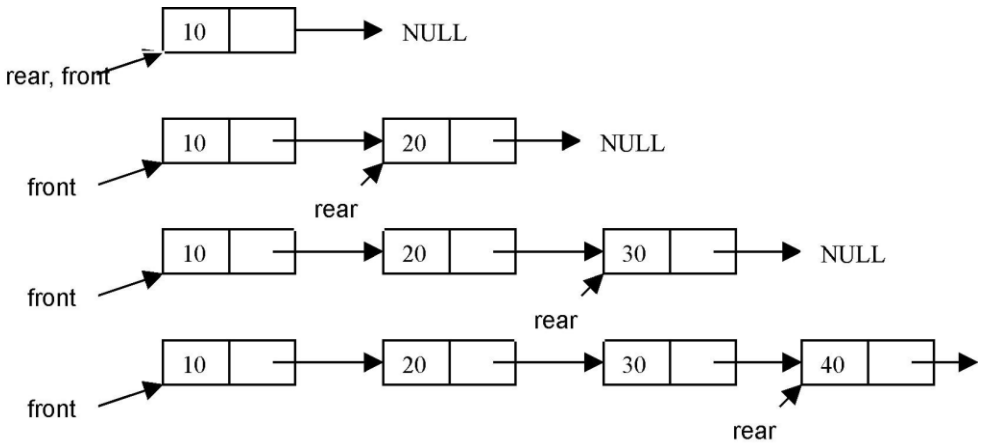

Figure 14.4: Linked queue

Program 14.6: A complete C program for implementation of a queue using the linked list is shown here

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
  int data;
  struct node *link;
};

void insert(struct node **front, struct node **rear, int value)
{
  struct node *temp;
  temp=(struct node *)malloc(sizeof(struct node));
     /* creates new node using data value passed as parameter */
  if(temp==NULL)
  {
    printf("No Memory available Error\n");
    exit(0);
  }
  temp->data = value;
  temp->link=NULL;
```

```
  if(*rear == NULL)
  {
    *rear = temp;
    *front = *rear;
  }
  else
  {
    (*rear)->link = temp;
    *rear = temp;
  }
}

void delete(struct node **front, struct node **rear, int *value)
{
  struct node *temp;
  if((*front == *rear) && (*rear == NULL))
  {
    printf(" The queue is empty cannot delete Error\n");
    exit(0);
  }
  *value = (*front)->data;
  temp = *front;
  *front = (*front)->link;
  if(*rear == temp)
  *rear = (*rear)->link;
  free(temp);
}

void main()
{
  struct node *front=NULL,*rear = NULL;
  int n,value;
  do
  {
    do
    {
```

```
        printf("Enter the element to be inserted\n");
      scanf("%d",&value);
      insert(&front,&rear,value);
      printf("Enter 1 to continue\n");
      scanf("%d",&n);
    } while(n == 1);

    printf("Enter 1 to delete an element\n");
    scanf("%d",&n);
    while( n == 1)
    {
        delete(&front,&rear,&value);
        printf("The value deleted is %d\n",value);
        printf("Enter 1 to delete an element\n");
        scanf("%d",&n);
    }
    printf("Enter 1 to continue\n");
    scanf("%d",&n);
  } while(n == 1);
}
```

14.5.3 Circular Queues

The problem with the previous implementation is that the insert function gives a queue-full signal even if a considerable portion is free. This happens because the queue has a tendency to move to the right unless the 'front' catches up with the 'rear' and both are reset to 0 again (in the delete procedure). To overcome this problem, the elements of the array are required to shift one position left whenever a deletion is made. But this will make the deletion process inefficient. Therefore, an efficient way of overcoming this problem is to consider the array to be circular, as shown in Figure 14.5. Initially both front and rear are intitialized to 0.

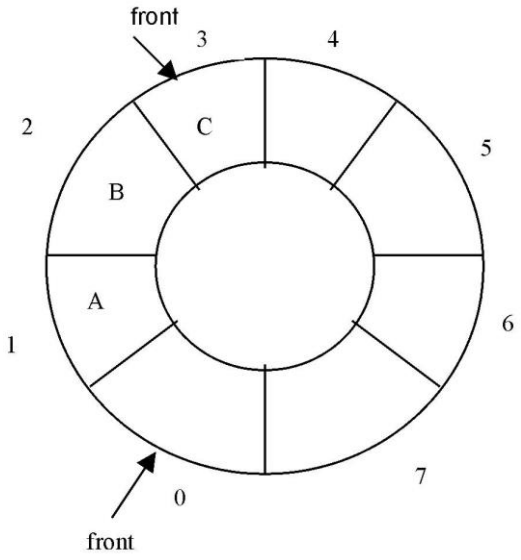

Figure 14.5: Circular queues

Program 14.7: A complete C program for implementation of a circular queue is shown here

```
#include <stdio.h>
#define MAX 10 /* The maximum size of the queue */
#include <stdlib.h>

void insert(int queue[], int *rear, int front, int value)
{
  *rear= (*rear +1) % MAX;
  if(*rear == front)
  {
    printf("The queue is full can not insert a value\n");
    exit(0);
  }
  queue[*rear] = value;
}

void delete(int queue[], int *front, int rear, int * value)
{
  if(*front == rear)
  {
    printf("The queue is empty can not delete a value\n");
```

```
    exit(0);
  }
  *front = (*front + 1) % MAX;
  *value = queue[*front];
}
void main()
{
  int queue[MAX];
  int front,rear;
  int n,value;
  front=0; rear=0;
  do
  {
    do
    {
        printf("Enter the element to be inserted\n");
      scanf("%d",&value);
      insert(queue,&rear,front,value);
      printf("Enter 1 to continue\n");
      scanf("%d",&n);
    } while(n == 1);
  printf("Enter 1 to delete an element\n");
  scanf("%d",&n);
  while( n == 1)
  {
     delete(queue,&front,rear,&value);
    printf("The value deleted is %d\n",value);
     printf("Enter 1 to delete an element\n");
    scanf("%d",&n);
  }
  printf("Enter 1 to continue\n");
  scanf("%d",&n);
  } while(n == 1);
}
```

14.5.4 Applications of Queues

One application of the queue data structure is in the implementation of priority queues required to be maintained by the scheduler of an operating system. It is a queue in which each element has a priority value and the elements are required to be inserted in the queue in decreasing order of priority. This requires a change in the function that is used for insertion of an element into the queue. No change is required in the delete function.

Program 14.8: A complete C program implementing a priority queue is shown here

```
# include <stdio.h>
# include <stdlib.h>
struct node
{
  int data;
  int priority;
  struct node *link;
};
void insert(struct node **front, struct node **rear, int value, int priority)
{
  struct node *temp,*temp1;
  temp=(struct node *)malloc(sizeof(struct node));
    /* creates new node using data valuepassed as parameter */
if(temp==NULL)
  {
    printf("No Memory available Error\n");
    exit(0);
  }
  temp->data = value;
  temp->priority = priority;
  temp->link=NULL;
  if(*rear == NULL) /* This is the first node */
  {
    *rear = temp;
    *front = *rear;
  }
  else
```

```
  {
    if((*front)->priority < priority)
         /* the element to be inserted has  highest priority hence should
be the first element*/
    {
        temp->link = *front;
        *front = temp;
    }
    else
        if( (*rear)->priority > priority)
             /* the element to be inserted has lowest priority hence should
be the last element*/
        {
            (*rear)->link = temp;
            *rear = temp;
        }

    else
    {
            temp1 = *front;
            while((temp1->link)->priority >= priority)
   /* find the position and insert the new element */
                    temp1=temp1->link;
         temp->link = temp1->link;
         temp1->link = temp;
         }
}
void delete(struct node **front, struct node **rear, int *value, int *priority)
{
  struct node *temp;
  if((*front == *rear) && (*rear == NULL))
  {
    printf(" The queue is empty cannot delete Error\n");
    exit(0);
  }
  *value = (*front)->data;
```

```
   *priority = (*front)->priority;
   temp = *front;
   *front = (*front)->link;
   if(*rear == temp)
      *rear = (*rear)->link;
   free(temp);
}

void main()
{
   struct node *front=NULL,*rear = NULL;
   int n,value, priority;
   do
   {
   do
   {
        printf("Enter the element to be inserted and its priority\n");
     scanf("%d %d",&value,&priority);
     insert(&front,&rear,value,priority);
     printf("Enter 1 to continue\n");
     scanf("%d",&n);
   } while(n == 1);
   printf("Enter 1 to delete an element\n");
   scanf("%d",&n);
   while( n == 1)
   {
        delete(&front,&rear,&value,&priority);
        printf("The value deleted is %d\ and its priority is %d \n",value,priority);
        printf("Enter 1 to delete an element\n");
      scanf("%d",&n);
    }
    printf("Enter 1 to delete an element\n");
    scanf("%d",&n);
  }  while( n == 1)
}
```

Self Assessment Questions

6. Queue is also called a _____ data structure.
7. In a linked implementation of a queue, the newly added element is the front element. (True/False)
8. An application of a _____ queue is in the implementation of priority queues required to be maintained by the scheduler of an operating system.

14.6 Summary

The most important aspect of program development is finding a good representation of the data manipulated by that program. The stack and the queue are examples of ADTs commonly used in computer programming. A stack is basically a list with insertions and deletions permitted from only one end, called the stack-top, so it is a data structure that exhibits the LIFO property. One of the important applications of a stack is in the implementation of recursion in the programming language. A queue is also a list with insertions permitted from one end, called rear, and deletions permitted from the other end, called front. So it is a data structure that exhibits the FIFO property. When the size of the stack/queue is known beforehand, the array implementation can be used and is more efficient. When the size of the stack/queue is not known beforehand, then the linked representation is used. It provides more flexibility. A circular queue is a queue in which the element next to the last element is the first element.

14.7 Terminal Questions

1. Convert the following infix expression into postfix:
   a/b-c+d*e-a*c
2. Convert the following infix expression into prefix
   a*(b+c)/d-g
3. Write an expression to point the front pointer in a circular queue to the next position?
4. State true or false:
   Stack is used in recursive programs.
5. The runtime memory allocation is also called _____.

14.8 Answers to Self Assessment Questions

1. Bitmapped
2. Dynamic
3. LIFO
4. (a) Stack cannot grow
5. True
6. FIFO
7. False
8. priority

14.9 Answers to Terminal Questions

1. ab/c-de*+ac*-
2. -/*a+bcdg
3. *front = (*front + 1) % MAX;
4. True
5. Dynamic Memory Allocation.

14.10 Exercises

1. Write a C program to implement a stack of characters.
2. Write a C program to implement a double-ended queue, which is a queue in which insertions and deletions may be performed at either end. Use a linked representation.
3. Write a program to reverse a linked list.
4. Write a program to delete a node from a linked list.
5. Write a program to insert a node into after the given node in a linked list.
6. What is meant by postfix expression? Write a program to convert an infix expression into a postfix expression.
7. Write a program to evaluate a postfix expression.
8. Write a function to check whether a stack is full.
9. Write a function to check whether a queue is empty.
10. Why stack and queues are called Abstract Data Types? Explain.

References:

1. E. Balagurusamy, “Programming with ANSI C”, Tata McGraw-Hill Publishers, New Delhi.
2. Byron S. Gottfried, Schaum’s Outline Series, “Theory and Problems of Programming wiith C”, ata McGraw-Hill Publishers, New Delhi.
3. Stephen C. Kochan, “Programming in C”, CBS Publishers, Revised Edition, New Delhi.
4. Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, Second Edition, Prentice-Hall of India, New Delhi.