DATA STRUCTURE

Tanjina Akter
10/10/2018

Contents

➢ INTRODUCTION TO DATA STRUCTURE

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Interrelationship among data elements that determine how data is recorded, manipulated, stored, and presented by a database.

➢ BASIC DATA STRUCTURES

Data structures may be classified into two types -

- Primitive Data Structures
- Abstract Data Structures

As we have discussed above, anything that can store data can be called as a data Structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures or Built-in Data Structures .

Also we have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are:

- STACK
- QUEUE
- LINKED LIST etc.

❖ Array

Array is collection of similar data type, you can insert and deleted element form array without follow any order.

❖ **There are 2 types of C arrays. They are,**

- One dimensional array
- Multi dimensional array
  - Two dimensional array
  - Three dimensional array
  - four dimensional array etc…

➢ ONE DIMENSIONAL ARRAY

Syntax : data-type arr_name[array_size]

Array declaration syntax:
data_type arr_name [arr_size];

Array initialization syntax:
data_type arr_name [arr_size]=(value1, value2, value3,….);

Array accessing syntax:
arr_name[index];

EXAMPLE PROGRAM FOR ONE DIMENSIONAL ARRAY

```
#include<stdio.h>
int main()
{
    int i;
    int arr[5] = {10,20,30,40,50};

      // declaring and Initializing array in C
      //To initialize all array elements to 0, use int arr[5]={0};
      /* Above array can be initialized as below also
      arr[0] = 10;
      arr[1] = 20;
      arr[2] = 30;
      arr[3] = 40;
      arr[4] = 50; */

    for (i=0;i<5;i++)
    {
       // Accessing each variable
       printf("value of arr[%d] is %d \n", i, arr[i]);
     }

}
```

- **TWO DIMENSIONAL ARRAY**

Two dimensional array is nothing but array of array.

syntax : data_type array_name[num_of_rows][num_of_column];

Array declaration syntax:

data_type arr_name [num_of_rows][num_of_column];**Array initialization syntax:**
data_type arr_name[2][2] = {{0,0},{0,1},{1,0},{1,1}};**Array accessing syntax:**
arr_name[index];

EXAMPLE PROGRAM FOR TWO DIMENSIONAL ARRAY

```
#include<stdio.h>
int main()
{
  int i,j;
  // declaring and Initializing array
  int arr[2][2] = {10,20,30,40};
  /* Above array can be initialized as below also
     arr[0][0] = 10; // Initializing array
     arr[0][1] = 20;
     arr[1][0] = 30;
     arr[1][1] = 40; */
  for (i=0;i<2;i++)
```

```
  {
    for (j=0;j<2;j++)
    {
      // Accessing variables
      printf("value of arr[%d] [%d] : %d\n",i,j,arr[i][j]);
    }
  }
}
```

❖ STACK

Stack is a linear data structure which follows a particular order in which the Operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out)

Stack Operation

In stack data structure mainly perform two operation; push and pop

- **pop:** In case of stack deletion of any item from stack is called pop.
- **push:** In case of stack Insertion of any item in stack is called push.

`Stack<item-type>` Operations

`push``(`*`new-item`*`:item-type)`
: Adds an item onto the stack.

`top``():item-type`
: Returns the last item pushed onto the stack.

`pop``()`
: Removes the most-recently-pushed item from the stack.

`is-empty``():Boolean`
: True if no more items can be popped and there is no top item.

`is-full``():Boolean`
: True if no more items can be pushed.

`get-size``():Integer`
: Returns the number of elements on the stack.

All operations except `get-size()` can be performed in time. `get-size()` runs in at worst

```
/*
 initialize stack pointer
*/
void init(int *top)
{
    *top = 0;
}

/*
 push an element into stack
 precondition: the stack is not full
*/
void push (int *s,int* top, int element)
{
    s[(*top)++] = element;
}
/*
 remove an element from stack
 precondition: stack is not empty
*/
int pop(int *s,int *top)
{
    return s[--(*top)];
}
/*
 return 1 if stack is full, otherwise return 0
*/
int full(int *top,const int size)
{
    return *top == size ? 1 : 0;
}
/*
 return 1 if the stack is empty, otherwise return 0
*/
int empty(int *top)
{
    return *top == 0 ? 1 : 0;
}

/*
    display stack content
*/
void display(int *s,int *top)
{
    printf("Stack: ");
    int i;
    for(i = 0; i < *top; i++)
    {
        printf("%d ",s[i]);
    }
    printf("\n");
}
```

❖ QUEUE

Queue is also an abstract data type or a linear data structure, in which the element is inserted from one end called REAR(also called tail), and the deletion of existing element takes place from the other end called as FRONT(also called head).
This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.
The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

```
/ * Enqueue and Dequeue operation  of   Queue * /

front = −1 , rear = −1;
Enqueue (A [ ] , size)    / / here ”A” is the name o f the queue and
                                / / ” size  ” is size  of  the queue
{
            if ( rear  = size − 1)
           print ” queue is full ” ;
       else
{
         print” enter element to insert” ;
         read the element x ( say ) ;

      rere= rear + 1 ;
          A[ rear ] = x ;
  }

}
```

```
/ *it is not practically possible to insert one element at a time,so while writing code,we
  shall use the same logic but modify a little bit so that desired number of elements
  may be inserted at a time until queue is full * /
```

```
Dequeue (A [ ] )
{
    if ( front = −1)
  print” queue i s empty ” ;
 e l s e
    {
        front= front + 1 ;
           print ” deleted element is A[ front] ” ;
     }
  }
```

/ ∗ here also one can delete multiple number of elements but deletion will take place from the top position by deleting one by one element until the stack is empty ∗ /

```
print (A [ ] )       / / print the elements o f  the stack .
{
 if (  rear == −1)
 print ” queue is empty ” ;
 else
   {
      printf ” queue i s ” ;
for ( i = front + 1 ; i <= rear ; i ++)
  print ( queue [ i ] ) ;
    }
}
```

- Implementation of circular queue

```
/ * circular queue algorithm * /

Is Full ( size )
 {
 if ( head == ( tail + 1) % size )
    return ( 10 ) ;
  }


Is Empty ( size )
{
 if ( head == tail ) return ( 10 ) ;


Enqueue ( queue [ ] , size )
 {
  if ( Is Full ( size ) == 10)
  print ( " the queue is full " ) ;
 else
 {
 print ( " enter the elements to insert " ) ;
 tail = ( tail + 1) % size                / / to remove the disadvantage of  linear array
  queue [ t a i l ] = x ;

    }
}


Dequeue ( queue [ ] , size )
 {
 if ( Is Empty ( size ) == 1)
 print ( " the queue is empty " ) ;
 e l s e
  {
    head = ( head + 1) % size ;
  print ( deleted element is " queue [ head ] " ) ;

     }
}



p r i n t ( queue [ ] , s i z e )
{
 if ( Is Empty ( size ) == 1)
```

```
print ( " the queue is empty " ) ;
else
   {
for ( i = ( head + 1) % size ; i ! = ( tai l + 1) % size ; i = ( i + 1) % size )
print ( queue [ i ] ) ;
    }
}
```

❖ LINKED LIST

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain(as shown in figure). Linked lists may be of different types but our job is restricted only to singly linked list.

Basic Operations:

- Insertion at the beginning: Insertion always takes place at the beginning i.e. "head" pointer always points to the newly inserted node.
- Insertion at the end: Insertion always takes place at the end in the sense that the pointer of the newly inserted node always points to Null.
- Deletion at the beginning: The node pointed by "head" pointer is deleted.
- Deletion at the end: The node at the end pointing to Null is deleted.

```
/ * Linked List operations ( algorithm ) * /
struct Node
 {
     int data ;
     struct Node * n ex t ;        / / pointer denoted by " nex t " .

   } ;
   Struct Node * head ;       / / head pointer is def ined .
 Insert begin ( )
    {
    print " enter the element " ;
    struct Node * temp = ( Node * ) malloc ( size of ( s t r u c t Node ) )
/ / memory a l location by malloc



temp–>d a t a = el em e n t ;
temp–>n ex t = head ;
 head = temp ;
}

/ / more t ha n one el em e n t s can be i n s e r t e d by a s m a l l m o d i f i c a t i o n , giv en
i n th e code .

I n s e r t l a s t ( )
{
 p r i n t f " e n t e r t h e el em e n t " ;
s t r u c t Node * temp = ( Node * ) m all o c ( s i z e o f ( s t r u c t Node ) ) ;
 temp–>d a t a = el em e n t ;
temp–>n ex t = NULL ;
```

```
 i f ( head == NULL) head = temp ;
 e l s e
{
 s t r u c t Node * p = head ;
 w hil e ( p->n ex t ! = NULL)
 p = p->n ex t ;
 p->n ex t = temp ;
     }
}

d e l e t e b e g i n ( )
 {
i f ( head == NULL)
p r i n t ” l i s t i s empty ” ;
 e l s e
   {
 s t r u c t Node * p = head ;
head = head->n ex t ; f r e e ( p ) ;
/ / d e a l l o c a t e s t h e p r e v i o u s l y a l l o c a t e d s p a c e .
     }
 }



d e l e t e l a s t ( )
{
  i f ( head == NULL)
 p r i n t ” l i s t i s empty ” ;
e l s e
 {
```

```
 s t r u c t Node * p = head ;
w hil e ( ( p–>n ex t)–>n ex t ! = NULL)
 {
p = p–>n ex t ;
  }
 f r e e ( p–>n ex t ) ; p–>n ex t = NULL ;
   }
 }


p r i n t ( )
{
 s t r u c t Node * temp = head ;
 i f ( temp == NULL)
 p r i n t f ” t h e l i n k e d l i s t i s empty ” ;
e l s e
 {
 p r i n t f ” l i s t i s ” ;
 w hil e ( temp ! = NULL)
 {
 p r i n t ( temp–>d a t a ) ;
 temp = temp–>n ex t ;
  }
 p r i n t new l i n e ;
    }
}
```

References

1. http://www.java2novice.com/data-structures-in-java/queue/array-implementation/
2. https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
3. https://www.sitesbay.com/data-structure/c-stack-example