FINAL REPORT

DATA STRUCTURE AND APPLICATIONS

by HRIDAY PAUL

24th March 2018
Contents

1  INTRODUCTION TO DATA STRUCTURE ........................................... 5

2  BASIC DATA STRUCTURES ......................................................... 5

2.1  STACK ............................................................................. 8

2.1.1  Basic Features: ......................................................... 8

2.1.2  Applications of Stack .................................................. 9

2.1.3  Implementation using Array in C ............................... 9

2.2  QUEUE ........................................................................... 12

2.2.1  Basic Features: ......................................................... 12

2.2.2  Applications of Queue ............................................... 13

2.2.3  Implementation using Array in C: ............................. 13

2.2.4  Demerits of Linear Queue And Introduction to Circular Queue: ......................................................... 15

2.2.5  Implementation of circular queue in C ...................... 17

2.3  LINKED LIST .................................................................. 20

2.3.1  Advantages of using Linked List ............................... 20

2.3.2  Basic Operations: ...................................................... 20

2.3.3  Implementation in C .................................................... 21
3 TIME COMPLEXITY:

3.1 Time Complexity of Stack ........................................ 25
3.2 Time Complexity of Queue ....................................... 25
3.3 Time Complexity of Linked list .................................. 25

4 IMPLEMENTATION OF STACK AND QUEUE USING LINKED LIST

4.1 Disadvantage of using array in implementation of Stack and Queue ........................................ 26
4.2 Stack implementation using Linked List ....................... 27
   4.2.1 Algorithm ...................................................... 27
4.3 Queue implementation using Linked List ..................... 29
   4.3.1 Algorithm ...................................................... 29

5 APPLICATIONS

5.1 TOWERS OF HANOI .............................................. 32
   5.1.1 A brief description of the Puzzle: ......................... 32
   5.1.2 Description of the Algorithms to solve the puzzle . 33
   5.1.3 Algorithms ................................................... 34
5.2 HASH TABLE ..................................................... 36
5.2.1 Hash Function: ................................................. 36
5.2.2 Hash Table ....................................................... 37
5.2.3 Hash Table implementation on a text file: ............ 37
1  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. Data Structures are structures programmed to store data, so that various operations can be performed on it easily. It represents the knowledge of data to be organized in memory. It should be designed and implemented in such a way that it reduces the complexity and increases the efficiency.

2  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.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in coming sections.

The data structures can also be classified on the basis of the following characteristics:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>In Linear data structures, the data items are arranged in a linear sequence. Example: <em>Array</em></td>
</tr>
<tr>
<td>Non-Linear</td>
<td>In Non-Linear data structures, the data items are not in sequence. Example: <em>Tree, Graph</em></td>
</tr>
<tr>
<td>Homogeneous</td>
<td>In homogeneous data structures, all the elements are of same type. Example: <em>Array</em></td>
</tr>
<tr>
<td>Non-Homogeneous</td>
<td>In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: <em>Structures</em></td>
</tr>
<tr>
<td>Static</td>
<td>Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: <em>Array</em></td>
</tr>
<tr>
<td>Dynamic</td>
<td>Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: <em>Linked List</em> created using pointers</td>
</tr>
</tbody>
</table>
NOTE:  Our study here does not include the discussion of Non-Linear Data structures.

Now, we start discussion about the basic data structures(stated above) including algorithms and codes.
2.1 STACK

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple 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, just like a pile of objects (shown in figure).

2.1.1 Basic Features:

1. Stack is an ordered list of similar data type

2. Stack is a LIFO structure. (Last in First out).

3. push() function is used to insert new elements into the Stack and pop() function is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.

4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.
2.1.2 Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
There are other uses also like: Parsing, Expression Conversion (Infix to Post-fix, Post-fix to Prefix etc) and many more.

2.1.3 Implementation using Array in C

Here is algorithms for Stack operations.

Algorithm

/* Stack operations – Push and Pop algorithms (using array) */

int top = -1;
Push(A[], size) // here "A" is the name of the stack and "size" is size of the stack
{
    if (top = size - 1)
        print "stack is full";
    else
        {
        print "enter element to push";
        read the element x (say);
        top = top + 1;
        }
A[top] = x;
}
}

/* it is not practically possible to push 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 stack overflow. */

pop(A[])
{
    if(top = -1)
        print " stack is empty";
    else
    {
        print "deleted element is A[top]";
        top = top - 1;
    }
}

/* 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 of the stack.
{
    if (top == -1)
        print"stack is empty";
    else
    {
        print"\nstack is: ";
        for (i = top; i >= 0; i--)
            print(stack[i]);
    }
}
2.2 QUEUE

Queue is also an abstract data type or a linear data structure, in which the first 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**.

2.2.1 Basic Features:

1. Like Stack, Queue is also an ordered list of elements of similar data types.

2. Queue is a **FIFO** (First in First Out) structure.

3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
2.2.2 **Applications of Queue**

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

2.2.3 **Implementation using Array in C:**

**Algorithms**

```c
/* Enqueue and Dequeue operations of linear Queue algorithm */

front = -1, rear = -1;
Enqueue(A[], size) // here "A" is the name of 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);
```

```c
```
rear = 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 is empty";
    else
    {
        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 of the stack.
{  
    if (rear == -1)  
        printf("queue is empty");  
    else  
    {  
        printf("queue is");  
        for (i = front + 1; i <= rear; i++)  
            printf(queue[i]);  
    }  
}

2.2.4 Demerits of Linear Queue And Introduction to Circular Queue:

In a Linear Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

By the definition of a queue, when we add an element in Queue, rear pointer is increased by 1 whereas, when we remove an element front pointer is increased by 1. But in array implementation of queue this may cause problem as follows:

Consider operations performed on a Queue (with SIZE = 5) as follows:

1. Initially empty Queue is there so, front = 0 and rear = -1

2. When we add 5 elements to queue, the state of the queue becomes as
follows with front = 0 and rear = 4

| 10 | 20 | 30 | 40 | 50 |

3. Now suppose we delete 2 elements from Queue then, the state of the Queue becomes as follows, with front = 2 and rear = 4

|   | 30 | 40 | 50 |

4. Now, actually we have deleted 2 elements from queue so, there should be space for another 2 elements in the queue, but as rear pointer is pointing at last position and Queue overflow condition (Rear == SIZE-1) is true, we can’t insert new element in the queue even if it has an empty space. To overcome this problem there is another variation of queue called CIRCULAR QUEUE.

**Circular Queue**

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called ‘Ring Buffer’.
2.2.5 Implementation of circular queue in C

Algorithms

/* circular queue algorithm */

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

IsEmpty(size)
{
    if(head == tail)
        return(10);
}
enqueue(queue[], size)
{
    if(IsFull(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[tail] = x;
    }
}

decqueue(queue[], size)
{
    if(IsEmpty(size) == 1)
        print("the queue is empty");
    else
    {
        head = (head + 1) % size;
        print(deleted element is "queue[head]");
    }
}

print(queue[], size)
if(IsEmpty(size) == 1) 
print("the queue is empty");
else
{
    for(i = (head + 1) % size; i != (tail + 1) % size; i = (i + 1) % size)
        print(queue[i]);
}
2.3 **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.

![Linked List Diagram](image)

2.3.1 **Advantages of using Linked List**

1. They are a dynamic in nature which allocates the memory when required.

2. Insertion and deletion operations can be easily implemented.

3. Stacks and queues can be easily executed.

4. Linked List reduces the access time.

2.3.2 **Basic Operations:**

1. **Insertion at the beginning:** Insertion always takes place at the beginning i.e. "head" pointer always points to the newly inserted node.

2. **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.
3. **Deletion at the beginning**: The node pointed by "head" pointer is deleted.

4. **Deletion at the end**: The node at the end pointing to Null is deleted.

### 2.3.3 Implementation in C

Implementation of Linked List data structure needs **dynamic** memory allocation which is achieved by **Malloc** in C programming. Malloc allocates requested size of bytes and returns a pointer first byte of allocated space.

Here is the algorithm and code for implementing a linked list using C program.

**Algorithms**

```c
/* Linked List operations (algorithm) */

struct Node
{
    int data;
    struct Node* next; // pointer denoted by "next".
};

struct Node* head; // head pointer is defined.

Insert_begin()
{
    print "enter the element";
    struct Node* temp = (Node*)malloc(sizeof(struct Node));
    // memory allocation by malloc
```
temp->data = element;

temp->next = head;

head = temp;

} // more than one elements can be inserted by a small modification, given in the code.

Insert_last() {

    printf("enter the element");
    struct Node* temp = (Node*)malloc(sizeof(struct Node));
    temp->data = element;
    temp->next = NULL;

    if(head == NULL)
        head = temp;

    else
    {
        struct Node* p = head;
        while(p->next != NULL)
            p = p->next;
        p->next = temp;
    }
}
;  

delete_begin()
    if(head == NULL)
        print "list is empty";
    else
        {
            struct Node* p = head;
            head = head->next;
            free(p); // deallocates the previously allocated space.
        }
}

delete_last()
    if(head == NULL)
        print "list is empty";
    else
        {
            struct Node* p = head;
            while((p->next)->next != NULL)
            {
                p = p->next;
            }
            free(p->next);
            p->next = NULL;
        }
}
print()
{
    struct Node* temp = head;
    if(temp == NULL)
        printf "the linked list is empty";
    else
    {
        printf "list is ";
        while(temp != NULL)
        {
            printf (temp->data);
            temp = temp->next;
        }
        printf new line;
    }
}
3 TIME COMPLEXITY:

3.1 Time Complexity of Stack

<table>
<thead>
<tr>
<th>Operations</th>
<th>Time Complexity in Array</th>
<th>Time complexity in Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>POP</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

3.2 Time Complexity of Queue

<table>
<thead>
<tr>
<th>Operations</th>
<th>Time Complexity in Array</th>
<th>Time complexity in Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQUEUE</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>DEQUEUE</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

3.3 Time Complexity of Linked list

<table>
<thead>
<tr>
<th>Operations</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion at the Beginning</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insertion at the end</td>
<td>O(n)</td>
</tr>
<tr>
<td>Deletion at the Beginning</td>
<td>O(1)</td>
</tr>
<tr>
<td>Deletion at the end</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
4 IMPLEMENTATION OF STACK AND QUEUE USING LINKED LIST

4.1 Disadvantage of using array in implementation of Stack and Queue

Using array is sometimes disadvantageous while implementing STACK and QUEUE data structures to store some data due to the following reasons.

1. We must know in advance that how many elements are to be stored in array.

2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array can not be increased or reduced.

3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.

4. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

So, we proceed for LINKED LIST to implement STACK and QUEUE.
4.2 Stack implementation using Linked List

Here is the algorithm and C code for implementation of STACK (without loss of generality, integer type data has been used here. One can also use character or float or string type data.) using LINKED LIST.

4.2.1 Algorithm

/* implementation of stack using linked list algorithm */

struct Node
{
    int data;
    struct Node* next;
};

struct Node* top = NULL;    // initializes top at null.

push()
{
    printf("enter number of elements to push\n");
    read the number n [say];
    printf("enter the elements\n");
    for(i = 1; i <= n; i++)
    {
        struct Node* temp = (Node*)malloc(sizeof(struct Node));
        temp->data = element;
        temp->next = top;
    }
top = temp;       // insertion takes place where the top
                  pointer points.
}
}
pop()
{
    if (top == NULL)
        printf("stack is empty");
    else
    {
        struct Node* p = top;
        top = top->next;
        free(p);
    }
}
print()
{
    struct Node* temp = top;
    if (temp == NULL)
        printf("\nthe stack is empty");
    else
    {
        printf("stack is");
        while (temp != NULL)
        {
            printf(temp->data);
        }
    }
}
temp = temp->next;
}
}
} /* implementation of stack using linked list */

4.3 **Queue implementation using Linked List**

Here is the algorithm and C code for implementation of QUEUE(without loss of
generality, integer type data has been used here. One can also use character or float
or string type data.) using LINKED LIST.

4.3.1 **Algorithm**

/* queue using linked list algorithm */

```c
struct Node
{
    int data;
    struct Node *next;
};
struct Node *front = NULL; // initializes rear and front at Null
struct Node *rear = NULL;
```
enqueue()
{
    printf("enter numbers of elements to insert ");
    read the number n[say];
    printf("enter the elements ");
    for (i = 1; i <= n; i++)
    {
        struct Node* temp = (Node*)malloc(sizeof(struct Node));
        temp->data = element;
        temp->next = NULL;
        if (front = NULL and rear = NULL)
            front = rear = temp;
        else
        {
            rear->next = temp;
            rear = temp;
        }
    }
}
void dequeue()
{
    struct Node* p = front;
    if (front == NULL)
        printf("\nqueue is empty");
    else if (front == rear)
front = rear = NULL;
else
front = front->next;
free(p);
}
void print()
{
struct Node* temp = front;
if(front == NULL)
print "the queue is empty";
else
{
print "queue is ";
while(temp != NULL)
{
print(temp->data);
temp = temp->next;
}
}
5 APPLICATIONS

5.1 TOWERS OF HANOI

5.1.1 A brief description of the Puzzle:

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with the following constraints

1. Only one disk can be moved at a time and

2. A larger disk can not be placed on the top of a smaller one.

The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

Moreover, the number of moves required to correctly move a tower of 64 disks from one pole to other is \(2^{64} - 1 = 18,446,744,073,709,551,615\). At a rate of one move per second, that is 584,942,417,355,584,942,417,355 years! Clearly there is more to this puzzle than meets the eye.
5.1.2 **Description of the Algorithms to solve the puzzle**

Suppose, we have 3 disks in pole A and our goal is to move the disks from pole A to pole C subject to the above mentioned constraints(as shown in figure). Now, we solve the problem by recursive algorithm and apply STACK operations(Push and Pop).

Suppose, we know how to move top two disks from pole A to pole B. Then our job is done because we can shift the remaining one(the largest one) from pole A to pole C and then the two disks(which are presently in B) can be moved from pole B to pole C(as we have assumed that we know how to move two disks(??)). Obviously there is big question mark of how we know the strategy to move two disks from one pole to another. Now, to clear this doubt what we do is - we first move the top one(i.e. the smaller one) disk from A and move it to C, next we shift the top disk from A to B, and finally the disk in pole C get shifted to pole B. Thus, we can shift two disks from pole A to pole B and following similar type of argument, we can move two disks from pole C to pole B.

The another important observation comes into mind when we think Towers of Hanoi as a data structure, where transfer of disk from one pole to another is similar to using POP operation in one pole(from which the disk get transferred) and PUSH operation in the other(in which the disk is placed on the top).
5.1.3 **Algorithms**

Here is a brief outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole

- Move a tower of height-1 to an intermediate pole, using the final pole.
- Move the remaining disk to the final pole.
- Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

As long as we always obey the rule that the larger disks remain on the bottom of the stack, we can use the three steps above recursively, treating any larger disks as though they were not even there. The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination. A tower of one disk will be our base case. In addition, the steps outlined above move us toward the base case by reducing the height of the tower. So, the algorithm required
for programming is as follows

/* towers of Hanoi algorithm */
Hanoi(from, to, other, n)
{
    if (n > 0)
    {
        Hanoi(from, other, to, n-1);
        print from -> to;
        if (from == 1 && to == 2)
            push2(pop1());
        else if (from == 1 && to == 3)
            push3(pop1());
        else if (from == 2 && to == 1)
            push1(pop2());
        else if (from == 2 && to == 3)
            push3(pop2());
        else if (from == 3 && to == 1)
            push1(pop3());
        else if (from == 3 && to == 2)
            push2(pop3());
        Hanoi(other, to, from, n-1);
    }
}
5.2 **HASH TABLE**

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

1. In universities, each student is assigned a unique roll number that can be used to retrieve information about them.

2. In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number. The usual practice is to assign keys to the objects to make searching easy. To store the key/value pair, a simple array may be used like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, one should look for hashing. In hashing, large keys are converted into small keys by **Hash Function**. Here we define what a hash function is.

5.2.1 **Hash Function:**

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.
To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

- **Easy to compute:** It should be easy to compute and must not become an algorithm in itself.

- **Uniform distribution:** It should provide a uniform distribution across the hash table and should not result in clustering.

- **Less collisions:** Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

**Note:** Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

### 5.2.2 Hash Table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well.

### 5.2.3 Hash Table implementation on a text file:

Hash Table may be implemented to store and display the words from a text file. Here is the algorithm.
Algorithm

/* hash table algo */

hash_func(word[], size) // it gives the hash value
{
    i = 0, sum = 0
    while(word[i] != '\0')
    {
        sum = sum + word[i]
        i++
    }
    return(sum % size)
}

Insert_word(hash_table[], size) // to insert words
{
    fpointer = fopen("/file name", "r")
    int h, r; char ch;
    while(ch = fgetc(fpointer) != EOF)
    {
        struct pair* temp
        struct pair* t
        temp = (struct pair*)malloc(sizeof(struct pair))
        fscanf(fpointer, "%s", temp->data)
        temp->next = NULL
        h = hash_func(temp->data, size)
r = Search_word(hash_table, size, temp->data)
if (r == 0)
{
    t = hash_table[h]
    if (t == NULL)
        hash_table[h] = temp
    else
        {
            while (t->next != NULL)
                t = t->next

            t->next = temp
        }
}

fclose(fpointer)
Search_word(hash_table[], size, word[]) // for searching any particular word
{
    int f = 0;
    h = hash_func(word, size)
    if(hash_table[h] == NULL)
    {
        f = 0
        return(f)
    }
    else
    {
        struct pair* temp = hash_table[h]
        while(temp != NULL)
        {
            if(strcmp(temp->data, word) == 0)
            {
                f = 1
                break
            }
            temp = temp->next;
        }
        return(f)
    }
}
Display(hash_table[], size)  // it displays the hash table
{
    struct pair* s;
    for (i = 0; i < size; i++)
    {
        print("index of the hash table array")
        s = hash_table[i]
        while (s != NULL)
        {
            print(s->data)
            s = s->next;
        }
        print(new line);
    }
}