ESO211 (Data Structures and Algorithms)
Lectures 4 to 7

Shashank K Mehta

1 Abstract Data Types (ADTs)

Data forms provided by most languages such as integers, floating point numbers, characters, booleans etc are the basic data-types. In most programs we may need to build more complex data-types from basic types and previously user-defined types.

2 Structures for Data Types

There are two types of data collections for which we need to design structures: a fixed collection of heterogeneous data-types, i.e., records; and a homogeneous (multi)set/po(multi)set (partially ordered (multi) set) of a given data-type. The structure for the former is provided by most languages in which one can access any field of a record in a fixed time once we have the address of the record. In Pascal it looks like

```
type
nameoftype = record
field1 : type1;
field2 : type2;
                ...
end
```

In the latter case, if the collection is a poset (partially ordered set) or po-multiset, then it might be ordered based on inherent value of the data-items or based on their arrival into the set.

Several non-trivial structures have been developed to store sets/multisets and partially ordered sets/multisets. No single structure is efficient in performing all operations on these sets. Besides the complexity in implementing these data-structures also vary.

3 Basic Structures

There are two basic topologies: (i) lists or sequences and (ii) trees. Others are mostly their derivatives.

3.1 Lists

There are lists/sequences of a given data type: $a_1, a_2, \cdots, a_n$ where each data item a_i is an instance of a given (fixed) type. Each item has a next and a previous item, except the extreme items.

3.2 Trees

There structures resemble the natural trees. It is a generalization of lists, which are 1-dimensional. In trees an item may have any number of neighboring items.

3.3 List Operations

Here *position* refers to the accessing address of an item.

$First(L)$ returns the position (pointer/index) of the first element of the list L,

$End(L)$ returns the position after the last element (it may be an index or pointer or nil depending on the implementation)

$Locate(x, L)$ returns the position of the first occurrence of x. But if x does not occur in L, then returns $End(L)$.

$Retrieve(p, L)$ returns the element at position p if $p < End(L)$.

$Delete(p, L)$ deletes the element at position immediately after position p.

$DeleteHead(L)$ deletes the first element of L.

$Insert(x, p, L)$ if $p < End(L)$, then inserts element x at position immediately after the position p.

$Insert_Head(x, L)$ Insert x as the first element of L.

$Next(p, L)$, $Previous(p, L)$ returns the position after/before the position p.

$MakeNull(L)$ makes L empty and returns $End(L)$.

Example Algorithm 1 shows how to delete entries at even positions from a list.

```
p := First(L);
if p ≠ End(L) then
    while Next(p, L) ≠ End(L) do
        Delete(p, L);
        p := Next(p, L);
    end
end
```

Algorithm 1: PurgeEven(L: List)

3.4 List Implementations

(a) *Array of Items*: Declare array of a fixed length in which each element is a list item. This way we can define a list of basic elements such as numbers or more complex user-defined structures.

```
type
List = record
elements = array[1..maxlength] of elementtype
head : int;
tail : int
end
```

(b) *Linked Lists Using Pointers*: Define a record with one or more pointer fields. This way we can define singly-linked, doubly linked lists, circular lists, etc. We will use "new(p)" to get a new record pointed by p.

```
type
celltype = record
element: elementtype;
next: ^ celltype
end

type
List = record
head: ^celltype;
tail: ^celltype
end
```

(c) *Simulating Linked List on an Array*: Use a base array (say space[]) of records. The records must have fields for data and pointer fields. In this case the pointers are the array indices. Here too we can define singly-linked, doubly-linked, circular lists etc. Here we keep all the cells not used by any list in a linked list called Free. Note: we will not discuss this implementation. *Examples*:

Algorithm 2 implements *Insert* and Algorithm 3 implements *InsertHead* for array based lists.

Examples: Algorithm 4 implements *Insert* and Algorithm 5 implements *InsertHead* for pointer based lists.
Exercise Implement Previous(p,L) operation on a singly linked list.

4 Data-structures Derived from Lists

4.1 Stack

Operations allowed on a stack are

if $p \geq End(L)$ OR $END(L) > maxIndex$ **then**
| **return** *error*
end
;
for $i := End(L)$ *down to* $p + 2$ **do**
| $L.elements[i] := L.elements[i - 1]$;
end
$L.elements[p + 1] := x$;

Algorithm 2: Insert(x, p, L)

if $END(L) > maxIndex$ **then**
| **return** *error*
end
;
for $i := End(L)$ *down to* 2 **do**
| $L.elements[i] := L.elements[i - 1]$;
end
$L.elements[1] := x$;

Algorithm 3: InsertHead(x, L)

$Push(x, S)$ insert at head

$Pop(S)$ delete head

$TopVal(S)$ retrieve head without deleting

$Empty(S)$ return *true* of S is empty else returns $false$

Implementation of Stack in an array:

```
type
stack = record
top: integer;
elements: array[1..maxlength] of elementtype
end
```

Implementation of Stack using a Singly Linked List

```
type
celltype = record
element: elementtype;
next: ^ celltype
end
```

```
type
stack = record
head: ^celltype;
end
```

if $p \neq nil$ **then**
  new-cell(q);
  $q \uparrow .element := x$;
  $q \uparrow .next := p \uparrow .next$;
  $p \uparrow .next := q$;
end

Algorithm 4: Insert(x, p, L)

new-cell(q);
$q \uparrow .element := x$;
$q \uparrow .next := L.head$;
$L.head := q$;

Algorithm 5: InsertHead(x, L)

Stack Operations on Linked List Implementation: Algorithms 6 to 9 implement *Empty*(S), $TopVal(S)$, *Pop*(S), and $Push(x, S)$ respectively.

if $S.head = null$ **then**
  return *true*;
end
else
  return *false;*
end

Algorithm 6: Empty(S)

Example Evaluate an arithmetic expression in prefix form.

We will need to store operator symbols as well as numbers in a stack we define an element-type in which is the union of integer type and the operator type.

```
type
elementtype = record
category: {"O","D"};
op      : {+,-,*,/};
dat     :integer
end
```

Recursive Solution:

Assume that the expression is already entered in stack S with the left-side at the top. Algorithm 10 evaluates the expression at the top of the stack recursively and pushes the final value back in the stack. See the code in Algorithm **??**.

Non-Recursive Solution:

if ($S.head = null$ **then**
  | **return** *error*;
end
else
  | **return** $S.head{\cdot}element$;
end

Algorithm 7: $TopVal(S)$

if ($S.head = null$) **then**
  | **return** *error*;
end
else
  | $S.head := S.head \uparrow .next$;
end

Algorithm 8: $Pop(S)$

Assume that the expression is already entered in stack S_1 with the left-side at the top. We also have another stack S_2 which is initially empty. Algorithm **??** gives the code for an iterative solution.

4.2 Queue

Operations Allowed on a Queue:

Enqueue(x,Q) insert at the rear of Q

Dequeue(Q) delete from the front of Q

Front(Q) retrieve the front element of Q

Empty(Q) return true of Q is empty else returns false

Implementation of a Queue in an Array

```
type
queue = record
elements: array[0..maxlength-1] of elementtype;
front: integer
end
```

Question: How to implement a circular queue in an array?

Implementation of a Queue Using a Linked List:

```
type
celltype = record
element: elementtype;
next : ^ celltype
        end
```

```
new-cell(p);
p↑.element := x;
p↑.next := S.head;
S.head := p;
```

Algorithm 9: $Push(x, S)$

```
if Empty(S) then
    return error;
    x := TopVal(S);
    Pop(S);
    if x.category = "D" then
        Push(x, S);
    end
    else
        REvaluate(S);
        u := TopVal(S);
        Pop(S);
        REvaluate(S);
        v := TopVal(S);
        Pop(S);
        u.dat := x.op(u.dat, v.dat);
        Push(u, S);
    end
end
```

Algorithm 11: REvaluate(S)

```
type
queue = record
front, rear: ^celltype
end
```

Queue Operations on Linked-List Implementation: Algorithms 12 to 15 implement $Empty(S)$, $Front - val(S)$, $Dequeue(Q)$, and $Enqueue(x, Q)$ respectively.

5 Application of Queue Data Structure

Consider a set of points $p_1, p_2, \ldots, p_n$ in a 2-D plane such that rays $qp_1, qp_2, \ldots, qp_n$ is sorted in clock-wise order. Here q is some fixed point. Following algorithm computes the convex-hull of the $p_1, \ldots, p_n$, which is the smallest convex polygon containing these points. See Algorithm **??**. Note that this algorithm is not efficient as it takes $O(n^2)$ time. Can you improve the complexity?

```
if Empty(S) then
    return error;
end
p := TopVal(S_1);
Pop(S_1);
while ¬Empty(S_2) OR p.category = "O" do
    if p.category = "O" then
        Push(p, S_2);
        p := TopVal(S_1);
        Pop(S_1);
    end
    else
        q := Top(S_2);
        Pop(S_2);
        if q.category = "D" then
            r := Top(S_2);
            Pop(S_2);
            p.dat := r.op(q.dat, p.dat);
        end
        else
            Push(q, S_2);
            Push(p, S_2);
            p := Top(S_1);
            Pop(S_1);
        end
    end
end
return p.dat;
```

Algorithm 13: NEvaluate(S)

```
if Q.rear = null then
    return true;
end
else
    return false;
end
```

Algorithm 14: $Empty(Q)$

```
if Q.rear = null then
|   return error;
end
else
|   return Q.front ↑ .element;
end
```

Algorithm 15: $Front - val(Q)$

```
if Q.front = null then
|   return error;
end
else
|   Q.front := Q.front ↑ .next;
|   if Q.front = null then
|   |   Q.rear := null;
|   end
end
```

Algorithm 16: $Dequeue(Q)$

```
new − cell(p);
p ↑ .element := x;
if Q.front = null then
|   Q.front := p;
|   Q.rear := p;
|   p ↑ .next := null;
end
else
|   Q.rear ↑ .next := p;
|   Q.rear := p;
|   p ↑ .next := null;
end
```

Algorithm 17: $Enqueue(x, Q)$

```
for i := 1 to n do
|  Enqueue(Q, p_i);
end
p := Dequeue(Q);
start := p;
firstPass := true;
while p ≠ start OR first_pass = true do
|  q := Dequeue(Q);
|  r := Head(Q)  /* this operation does not modify Q        */
|  ;
|  if p → q → r is a left turn then
|  |  start := p;
|  |  firstPass := true;
|  else
|  |  Enqueue(Q, p);
|  |  p := q;
|  |  first_pass := false;
|  end
end
return p and all the points in Q;
```

Algorithm 18: Computation of convex-hull of an ordered set of points