ESO211 (Data Structures and Algorithms)
Lectures 4 to 7

Shashank K Mehta

1 Abstract Data Types (ADTs)

Data types provided by most languages such as integers, floating point numbers, characters, booleans etc are called 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 memory address of the record. In Pascal-like languages 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 type. Each item has a next and a previous item, except the extreme items.

3.2 Trees

These structures resemble the natural trees. It is a generalization of lists, which are 1-dimensional. In trees an item may have any number of next (child) items.

3.3 List Operations

Here is a list of some of the opertions possible on a list of data items.

$Size(L)$ returns the number of elements in the list L,

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

$Last(L)$ returns the pointer/index after the last element of the list L,

$Locate(x, L)$ returns the pointer/index of the first occurrence of x in the list L. But if x does not occur in L, then returns nil,

$Retrieve(p, L)$ returns the element at index p,

$Delete(p, L)$ deletes the element at index p,

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

$Insert(x, p, L)$ if $p \leq Size(L)$, then inserts element x at position immediately after index 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 index p.

$MakeNull(L)$ makes L empty.

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
listnode = record
element: elementtype;
next: ^listnode
end

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

Examples: Algorithm 1 implements *Insert* and Algorithm 2 implements *Insert-Head* for array based lists.

```
if L.Head = 1 AND L.Tail = maxIndex then
    return error
end
;
if L.Head > 1 then
    for i := 0 to p − 1 do
        L.elements[L.Head + i − 1] := L.elements[L.Head + i];
    end
    L.elements[L.Head + p − 1] := x;
    L.Head := L.Head − 1;
else
    for i := L.Tail to p + 1 do
        L.elements[i + 1] := L.elements[i];
    end
    L.elements[p + 1] := x;
    L.tail := L.Tail + 1;
end
```

Algorithm 1: Insert(x, p, L)

Examples: Algorithm 3 implements *Insert* and Algorithm 4 implements *Insert-Head* for pointer based lists.
Exercise Implement Previous(p,L) operation on a singly linked list.

```
if L.Tail + 1 − L.Head = maxIndex then
    return error;
end
if L.Head > 1 then
    L.Head := L.Head − 1;
    L.elements[L.Head] := x;
else
    for i := L.Tail down to 1 do
        L.elements[i + 1] := L.elements[i];
    end
    L.elements[1] := x;
end
```

Algorithm 2: InsertHead(x, L)

```
if p ≠ nil then
    new-cell(q);
    q↑.element := x;
    q↑.next := p↑.next;
    p↑.next := q;
end
```

Algorithm 3: Insert(x, p, L)

4 Data-structures Derived from Lists

4.1 Stack

Operations allowed on a stack are

$Push(x, S)$ 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

```
new-cell(q);
q↑.element := x;
q↑.next := L.head;
L.head := q;
```

Algorithm 4: InsertHead(x, L)

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

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

Stack Operations on Linked List Implementation: Algorithms 5 to 8 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 5: Empty(S)

```
if (S.head = null then
 | return error;
end
else
 | return S.head·element;
end
```

Algorithm 6: $TopVal(S)$

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 which is the union of integer type and the operator type.

```
type
elementtype = record
category: {"O","D"};
```

```
if (S.head = null) then
 | return error;
end
else
 | x := S.head.element;
 | S.head := S.head ↑ .next;
 | return x;
end
```

Algorithm 7: $Pop(S)$

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

Algorithm 8: $Push(x, S)$

```
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 9.

Non-Recursive Solution:

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 10 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 without deleting

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;
```

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

Algorithm 10: REvaluate(S)

```
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

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 Applications of Queue Data Structure

Example 1: Merge of two sorted sequences.

Example 2: Design MergeSort algorithm.

Example 3: Given a tree T rooted at vertex r, a breadth-first-sequence of the nodes of T is a sequence of its nodes such that if x is an ancestor of y, then

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

Algorithm 12: NEvaluate(S)

x must be to the left of y in the sequence. Design an algorithm to compute a a breadth-first sequence.

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

Algorithm 13: $Empty(Q)$

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

Algorithm 14: $Front - val(Q)$

```
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 15: $Dequeue(Q)$

```
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 16: $Enqueue(x, Q)$

```
while (¬Empty(Q1)) AND (¬Empty(Q2)) do
    if Empty(Q1) then
        while ¬Empty(Q2) do
            x := Dequeue(Q2);
            Push(Q3, x);
        end
    else
        if Empty(Q2) then
            while ¬Empty(Q1) do
                x := Dequeue(Q1);
                Push(Q3, x);
            end
        end
    end
    x := head(Q1);
    y := head(Q2);
    if x ≤ y then
        x := Dequeue(Q1);
        Enqueue(x, Q3);
    else
        y := Dequeue(Q2);
        Enqueue(y, Q3);
    end
end
```

Algorithm 17: $Merge(Q_1, Q_2, Q_3)$: Q_1 and Q_2 contain sorted sequences in increasing order

```
Initialize(Q1);
Initialize(Q);
Push(r, Q1);
while ¬Empty(Q1) do
    x := Dequeue(Q1);
    Enqueue(x, Q);
    for each y ∈ Child_Nodes(x) do
        Enqueue(y, Q);
    end
end
```

Algorithm 18: $BFS(r, Q)$