# ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez
Electrical and Computer Engineering Department

Lecture 10 – September 20, 2001

## Alternative implementation of sets

- Allows for faster, constant time search and delete operation on the set.
  - Implementation for array require looking up all array.
- Alternative is called bit vector
- Use an array of unsigned char or unsigned int to represent sets of integers.
  - Each bit represents a number.
  - The bit at position k represents integer k.
  - If k is in the set, then the bit is set to 1, otherwise it is set to 0.
- Names, places and other objects to which we can assign a number can also be represented this way.

## Example:

- Suppose we have array of size 2 of unsigned char
- Rightmost position is 0, numbers increase to left.
- Ex: 0100001100011100
  - Represents set {2,3,4,8,9,14}
  - Storage requirements: 2 bytes
  - Previous implementation would have required at 5 bytes.
  - In fact, with 2 bytes we can represent up to 16 numbers.

## Bit Vector organization

- Bit vector contains 1 or more elements of type either:
  - unsigned char (1byte – 8bits)
  - unsigned short (2 bytes – 16bits)
  - unsigned long (4 bytes – 32bits)
- Each element is a segment of the bit vector.
- To find an element we need to its position within the bit vector.
- Position is given by segment number and its position within the segment:
  - Segment is given by: element / segment size
  - Position is given by: element % segment size

## Example 1

- Given 1 byte bit-vector 00011100
- Where is element 1 located?
  - Segment 0, position 1
- Where is element 3 located?
  - Segment 0, position 3
- Where is element 8 located?
  - Cannot be represented due to lack of space.
  - Bit vector n bytes can only represent (n * 8) – 1 elements

## Example 2

- Given 2 byte bit-vector 0100001100011100
- Where is element 1 located?
  - Segment 0, position 1
- Where is element 4 located?
  - Segment 0, position 4
- Where is element 8 located?
  - Segment 1, position 0
- Where is element 10 located?
  - Segment 1, position 2
- Where is element 19 located?
  - Cannot be represented due to lack of space.
  - Again: Bit vector n bytes can only represent (n * 8) – 1 elements

## Searching the bit vector

- To search for an element we need to inspect bits in the bit vector.
- This is done via Boolean Algebra Operation and Bit-shifting
- Boolean Algebra operations:
  - AND, OR, XOR
- Shifting operations
  - Left shift
  - Right shift

# AND Operator

- Expression in C++: A & B
- Truth Table:

B

A

| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Examples:

```
01001101    01010101    11111111    10100111
    &           &           &           &
11001100    10101010    00000000    00000100
--------
01001100    00000000    00000000    00000100
```

## OR Operator

- Expression in C++: A | B
- Truth Table:

|  |  | B |  |
|---|---|---|---|
|  | OR | 0 | 1 |
| A | 0 | 0 | 1 |
|  | 1 | 1 | 1 |

Examples:

```
 01001101        01010101    11111110
     |               |           |
 11001100        10101010    00000001
 ________        ________    ________
 11001101        11111111    11111111
```

# XOR Operator

- Expression A ^ B
- Truth Table:

B

A

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Examples:

```
 01001101        00000000     11111111
     ^               ^            ^
 11001100        11111111     11111111
 ________        ________     ________
 10000001        11111111     00000000
```

## Left Shift

- Operation:
  - move all bits n spots to the left
  - discard elements that pass beyond last position.
  - add zero at first position of bit vector

- Examples:
  - 0101 << 1 = 1010 (shift one)
  - 0101 << 3 = 1000 (shift three)
  - 11111111 << 2 = 11111100 (shift two)
  - 11110011 << 2 = 11001100 (shift two)
  - 11111111 << 8 = 00000000 (shift eight)

## Right Shift

- Operation:
  - move all bits n spots to the right
  - discard elements that pass beyond first position.
  - add zero at last position of bit vector

- Examples:
  - 0101 >> 1 = 0010 (shift one)
  - 0101 >> 3 = 0001 (shift three)
  - 11111111 >> 2 = 00111111 (shift two)
  - 11110011 >> 2 = 00111100 (shift two)
  - 11111111 >> 8 = 00000000 (shift eight)

## Finding an element

- Given a bitset b, and an element k we want to support: b.find(k)
  - True if the element is present
  - False otherwise
- Example: b = 01010011, k = 2.
  - Then, b.find(2) should be false.
- How do we do it? Inspect bit at position 2.
  - If set to 1, 2  is in the set, return true.
  - If set to 0, 2 is not in the set, return false.
- Solution: Use shift to “mask” all other bits, then use **AND** operator to determine if the bit is set or not.

# Finding an element (sketch)

- b = 01010011
- mask = 00000001
- mask = mask << 2 = 00000100
- Now use and operator to inspect bit:

```
01010011
    &
00000100
00000000
```

- Since result was 00000000, which is equal to number 0, then we now, that element was not present.

## Finding an element (many segments)

- If the bit vector has many segments, then we must locate segment, and then position within segment.
- Suppose the bit-vector is composed of 8-bit segments.
- Consider b = 0001000111000011, and b.find(12).
- We need to first find the segment and the position of 12 within the segment:
  - Segment = 12 / 8 = 1 -> 00010001
  - Position = 12 % 8 = 4
- Now, determine if bit 4 of 00010001
  - mask = 00000001 << 4 = 00010000
  - AND operation: 00010001 & 00010000 = 00010000
  - Since result is different from zero, element 12 is in the bit set

## Inserting an element

- To insert an element to the bit set, all we need to do it to turn the bit on.
- Consider: b = 01010011, k = 3, b.insert (3)
  - Result should be: 01011011
- How do we do it?
  - Use shift to move a 1 to bit 3
  - Use **OR** to add bit into the bit set

```
mask = 00000001 << 3 = 00001000
b = b | mask =>       01010011
                          |
                      00001000
                      --------
                      01011011
```

## Deleting an element

- To delete an element is somewhat trickier.
- We need to first write the element (if not already there) and delete it by making an XOR operation.
- Consider b = 01010011, k = 1, b.delete(k)
  - Result should be: 01011001
- How do we do it?
  - Use shift to move a 1 to bit 1
  - Use **OR** to add bit at pos 1 into the bit set (although already there) and then use **XOR** to delete it.

mask = 00000001 << 1 = 00000010

b = b | mask => 01010011

$\boxed{\text{First step}}$

$$\begin{array}{r} 01010011 \\ | \\ 00000010 \\ \hline 01011011 \end{array} \longrightarrow$$

b = b ^ mask => 01010011

$\boxed{\text{Second step}}$

$$\begin{array}{r} 01010011 \\ \wedge \\ 00000010 \\ \hline 01011001 \end{array}$$

## Union operation

- Consider two bit vectors
  - b1 = 00110011
  - b2 = 01010001
- If we take the OR operation between b1 and b2:

  00110011

  |

  01010001

  01110011
- We get all the bits that are set in either bit set: UNION!
- The union operation is obtained by making an OR on the bit sets. Corresponding segments in each bit set must be united.
  - Ex: 0010001000000010 | 1010000000000001 = 1010001000000011

## Intersection operation

- Consider two bit vectors
  - b1 = 00110011
  - b2 = 01010001
- If we take the AND operation between b1 and b2:

  00110011

  &

  01010001

  00010001
- We get all the bits that are set in both bit sets: Intersection!
- Intersection operation is obtained by making an AND on the bit sets. Corresponding segments in each bit set must be intersected.
  - Ex: 0010001000000010 & 1010000000000001 = 0010000000000000

## Difference operation

- Consider two bit vectors
  - b1 = 00110011
  - b2 = 01010001
- If we take the XOR operation between b1 and b2, and then we take the AND of this result with b1:

$$\begin{array}{lcl} 00110011 & & 00110011 \\ \wedge & \longrightarrow & \& \\ 01010001 & & \boxed{01100010} \\ \boxed{01100010} & & 00100010 \end{array}$$

- Get all the bits that are set in the first but not in the second: Difference!

## Difference Operation

- The difference operation is obtained by making an AND operation and then an XOR on the bit sets. Corresponding segments in each bit set must have the difference operator applied to each.
  – Ex: 0010001000000010 - 1010000000000001 =

    0010000000000010