>>

Abstract Data Types

^ >>

❖ Abstract Data Types

A data type is ...

- a set of values (atomic or structured values)
- a collection of operations on those values

An abstract data type is ...

- an approach to implementing data types
- separates interface from implementation
- users of the ADT see only the interface
- builders of the ADT provide an implementation

E.g. do you know what a (`FILE *`) looks like? do you want/need to know?

<< ^ >>

❖ DTs, ADTs, GADTs

We want to distinguish ...

DT = (non-abstract) data type (e.g. C strings)

- internals of data structures are visible (e.g. `char s[10];`)

ADT = abstract data type (e.g. C files)

- can have multiple instances (e.g. `Set a, b, c;`)

GADT = generic (polymorphic) abstract data type

- can have multiple instances (e.g. `Set<int> a, b, c;`)
- can have multiple types (e.g. `Set<int> a; Set<char> b;`)
- not available natively in the C language

<< ^ >>

❖ Interface/Implementation

ADT interface provides

- a user-view of the data structure (e.g. **`FILE*`**)
- function signatures (prototypes) for all operations
- semantics of operations (via documentation)
- a contract between ADT and its clients

ADT implementation gives

- concrete definition of the data structures
- definition of functions for all operations

<< ^ >>

❖ Collections

Many of the ADTs we deal with ...

- consist of a collection of items
- where each item may be a simple type or an ADT
- and items often have a key (to identify them)

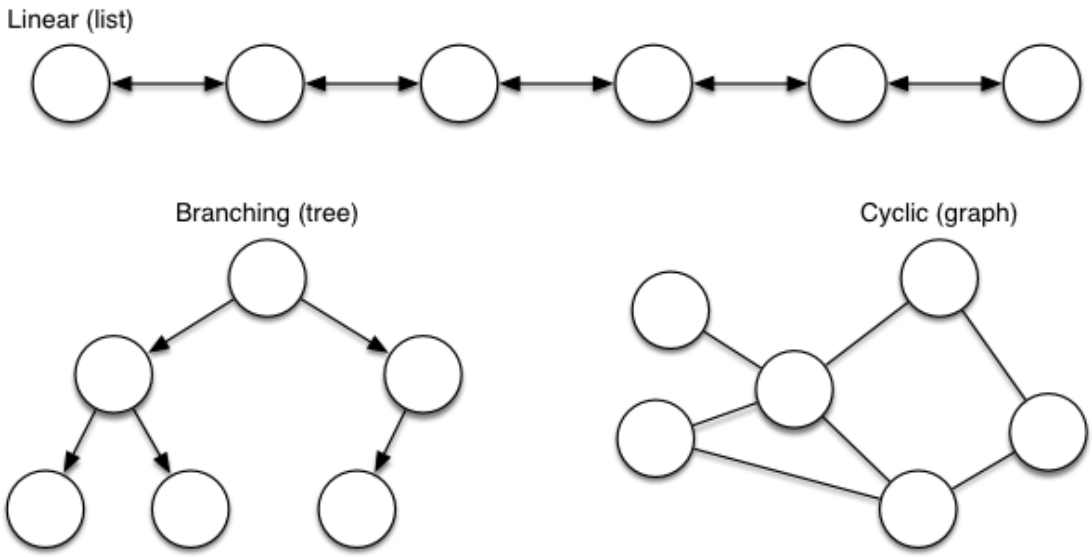
Collections may be categorised by ...

- structure: linear (list), branching (tree), cyclic (graph)
- usage: set, matrix, stack, queue, search-tree, dictionary, ...

<< ^ >>

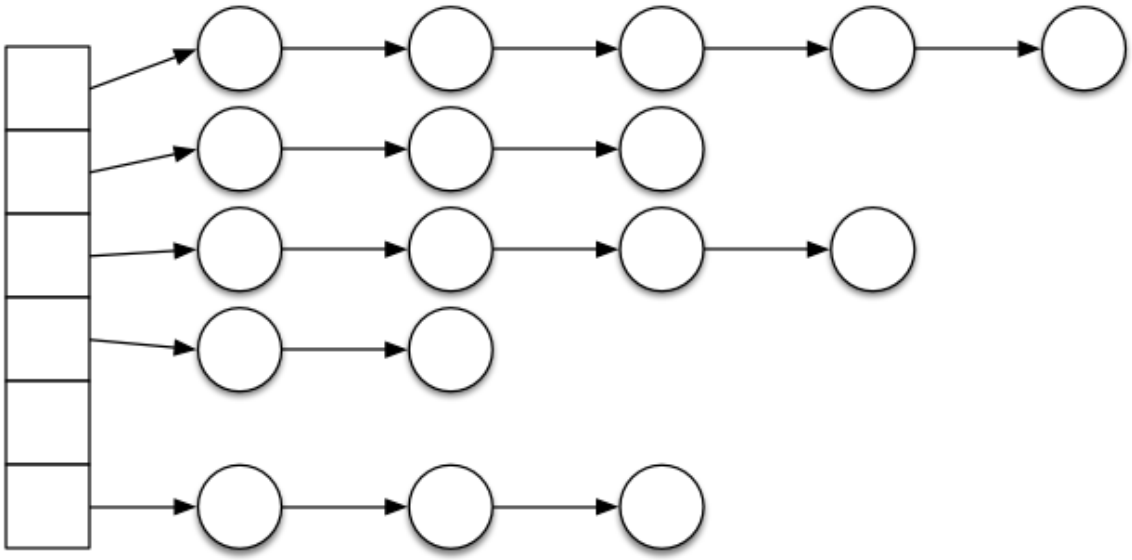
❖ ... Collections

Collection structures:



❖ ... Collections

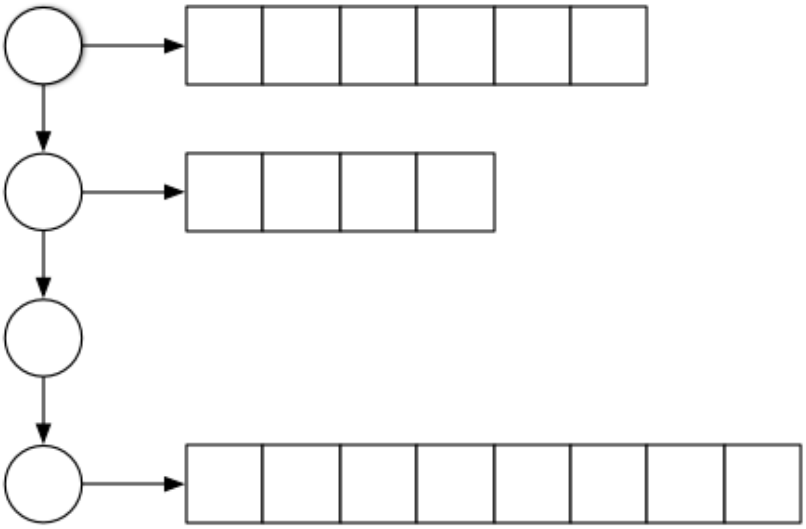
Or even a hybrid structure like ...



<< ^ >>

❖ ... Collections

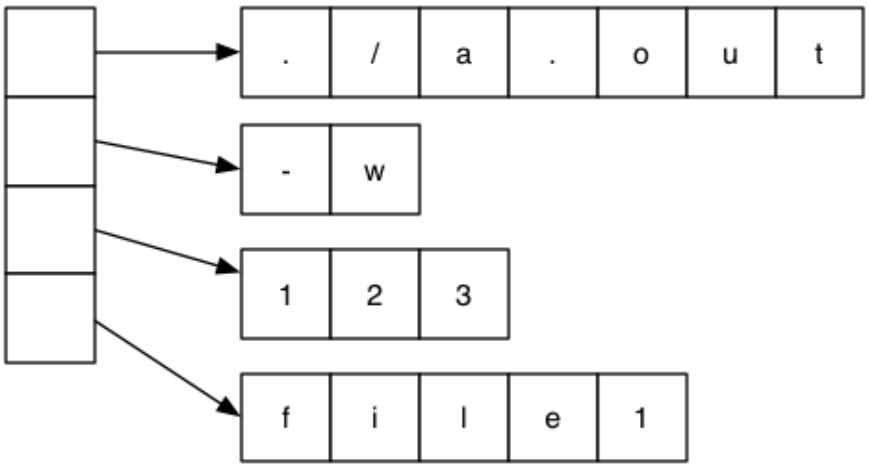
Or this ...



<< ^ >>

❖ ... Collections

Or this ...



<< ^ >>

❖ ... Collections

Typical operations on collections

- create an empty collection
- insert one item into the collection
- remove one item from the collection
- find an item in the collection
- check properties of the collection (size,empty?)
- drop the entire collection
- display the collection

❖ Example: Set ADT

Set data type: collection of unique integer values.

"Book-keeping" operations:

- **`Set newSet()`** ... create new empty set
- **`void dropSet(Set)`** ... free memory used by set
- **`void showSet(Set)`** ... display as **`{1,2,3...}`**

Assignment operations:

- **`void readSet(FILE*,Set)`** ... read+insert set values
- **`Set SetCopy(Set)`** ... make a copy of a set

<< ^ >>

❖ ... Example: Set ADT

Data-type operations:

- **`void SetInsert(Set,int)`** ... add number into set
- **`void SetDelete(Set,int)`** ... remove number from set
- **`int SetMember(Set,int)`** ... set membership test
- **`Set SetUnion(Set,Set)`** ... union
- **`Set SetIntersect(Set,Set)`** ... intersection
- **`int SetCard(Set)`** ... cardinality (#elements)

Note: union and intersection return a newly-created **`Set`**

<< ^ >>

❖ Set ADT Interface

```
// Set.h ... interface to Set ADT

#ifndef SET_H
#define SET_H

#include <stdio.h>
#include <stdbool.h>

typedef struct SetRep *Set;

Set newSet();                // create new empty set
void dropSet(Set);           // free memory used by set
Set SetCopy(Set);            // make a copy of a set
void SetInsert(Set,int);     // add value into set
void SetDelete(Set,int);     // remove value from set
bool SetMember(Set,int);     // set membership
Set SetUnion(Set,Set);       // union
Set SetIntersect(Set,Set); // intersection
int SetCard(Set);            // cardinality
void showSet(Set);           // display set on stdout
void readSet(FILE *, Set); // read+insert set values

#endif
```

<< ^ >>

❖ ... Set ADT Interface

Example set client: set of small odd numbers

```
#include "Set.h"
...
Set s = newSet();
for (int i = 1; i < 26; i += 2)
    SetInsert(s,i);
showSet(s); putchar('\n');
```

Outputs:

```
{1,3,5,7,9,11,13,15,17,19,21,23,25}
```

❖ Set Applications

Example: eliminating duplicates

```
#include "Set.h"
...
// scan a list of items in a file
int item;
Set seenItems = newSet();
FILE *in = fopen(FileName,"r");
while (fscanf(in, "%d", &item) == 1) {
   if (!SetMember(seenItems, item)) {
      SetInsert(seenItems, item);
      process item;
   }
}
fclose(in);
```

<< ^ >>

❖ Set ADT Pre/Post-conditions

Each **`Set`** operation has well-defined semantics.

Express these semantics in detail via statements of:

- what conditions need to hold at start of function
- what will hold at end of function (assuming successful)

Could implement condition-checking via **`assert()`**s

But only during the development/testing phase

- **`assert()`** does not provide useful error-handling

At the very least, implement as comments at start of functions.

<< ^ >>

❖ ... Set ADT Pre/Post-conditions

If **x** is a variable of type **T**, where **T** is an ADT

- $ptr(x)$ is the pointer stored in **x**
- $val(x)$ is the abstract value represented by ***x**
- $valid(T, x)$ indicates that
  - the collection of values in ***x**
    satisfies all constraints on "correct" values of type **T**
- x' is an updated version of x (note: $ptr(x')$ == $ptr(x)$)
- res is the value returned by a function

Can also use math/logic notation as used in pseudocode.

<< ^ >>

❖ ... Set ADT Pre/Post-conditions

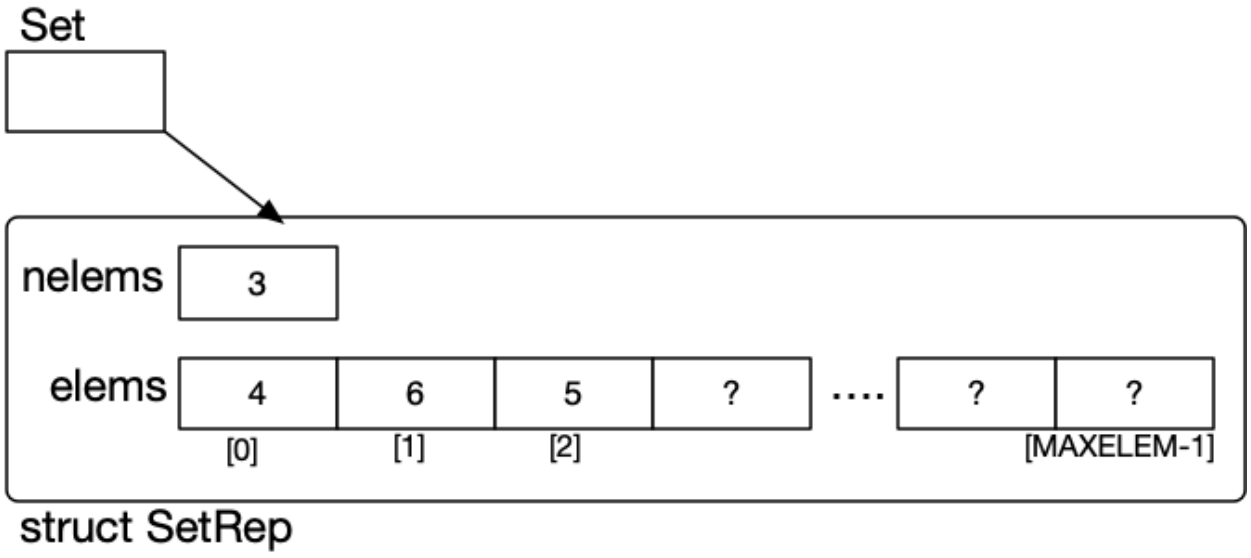
Examples of defining pre-/post-conditions:

```
// pre:  true
// post: valid(Set,res) and res = {}
Set newSet() { ... }

// pre:  valid(Set,s) and valid(int n)
// post: n ∈ s'
void SetInsert(Set s, int n) { ... }

// pre:  valid(Set,s1) and valid(Set,s2)
// post: ∀ n ∈ res, n ∈ s1 or n ∈ s2
Set SetUnion(Set s1, Set s2) { ... }

// pre:  valid(Set,s)
// post: res = |s|
int SetCard(Set s)  { ... }
```

<< ^ >>

❖ Sets as Unsorted Arrays

Concrete data structure:



<< ^ >>

❖ ... Sets as Unsorted Arrays

Concrete data structure (in C):

```
#define MAXELEMS 1000

// concrete data structure
struct SetRep {
    int elems[MAXELEMS];
    int nelems;
};
```

Need to set upper bound on number of elements

Could do statically (as above) or dynamically

```
Set newSet(int maxElems) { ... }
```

❖ ... Sets as Unsorted Arrays

Set creation:

```
// create new empty set
Set newSet()
{
    Set s = malloc(sizeof(struct SetRep));
    if (s == NULL) {
        fprintf(stderr, "Insufficient memory\n");
        exit(EXIT_FAILURE);
    }
    s->nelems = 0;
    // assert(isValid(s));
    return s;
}
```

❖ ... Sets as Unsorted Arrays

Checking membership:

```
// set membership test
int SetMember(Set s, int n)
{
    // assert(isValid(s));
    int i;
    for (i = 0; i < s->nelems; i++)
       if (s->elems[i] == n) return TRUE;
    return FALSE;
}
```

<< ^ >>

❖ ... Sets as Unsorted Arrays

Costs for set operations on unsorted array:

- card: read from struct; constant cost *O(1)*
- member: scan list from start; linear cost *O(n)*
- insert: duplicate check, add at end; linear cost *O(n)*
- delete: find, copy last into gap; linear cost *O(n)*
- union: copy s1, insert each item from s2; quadratic cost *O(nm)*
- intersect: scan for each item in s1; quadratic cost *O(nm)*

Assuming: s1 has *n* items, s2 has *m* items

❖ Sets as Sorted Arrays

Same data structure as for unsorted array.

Differences in

- membership test ... can use binary search
- insertion ... binary search and then shift up and insert
- deletion ... binary search and then shift down

<< ^ >>

❖ ... Sets as Sorted Arrays

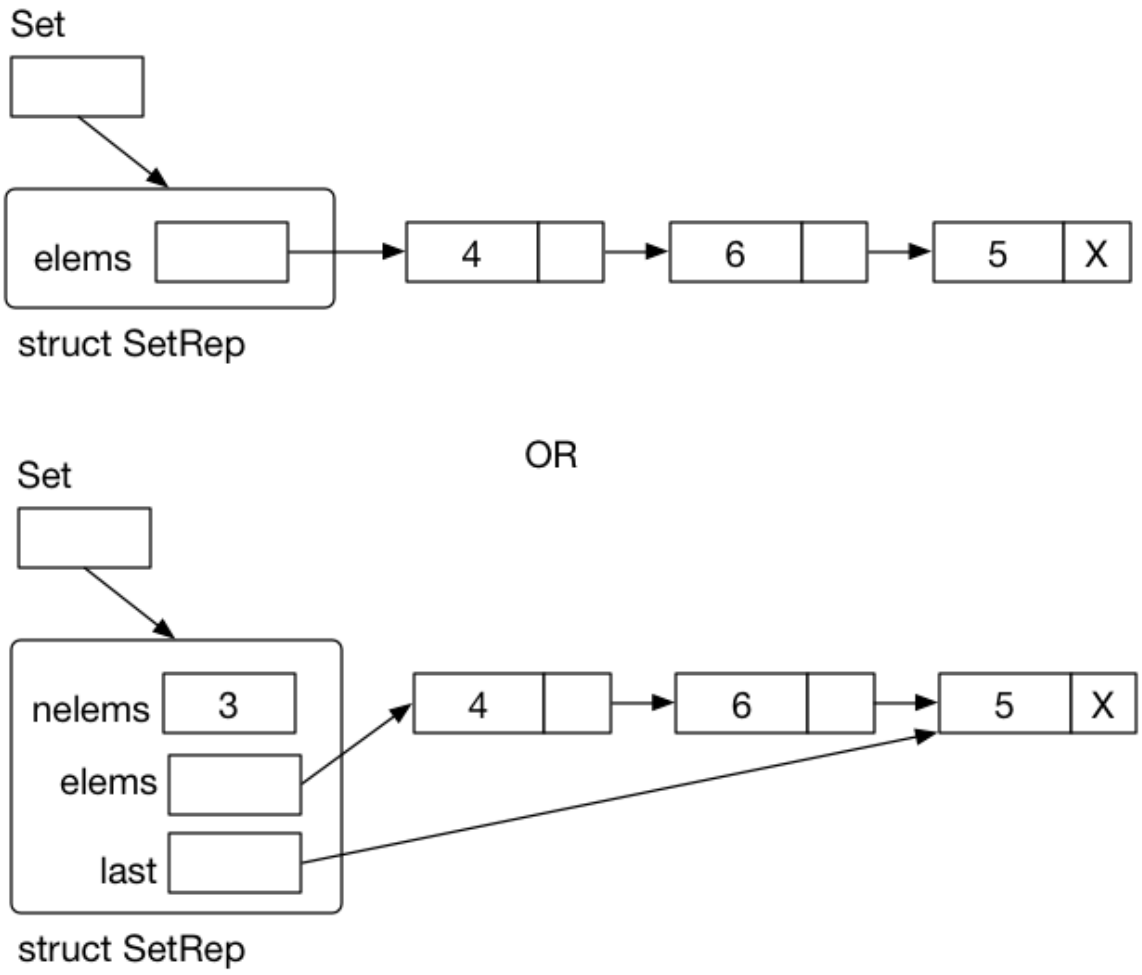
Costs for set operations on sorted array:

- card: read from struct; *O(1)*
- member: binary search; *O(log n)*
- insert: find, shift up, insert; *O(n)*
- delete: find, shift down; *O(n)*
- union: merge = scan s1, scan s2; *O(n)* (technically *O(n+m)*)
- intersect: merge = scan s1, scan s2; *O(n)* (technically *O(n+m)*)

<< ^ >>

❖ Sets as Linked Lists

Concrete data structure:



❖ ... Sets as Linked Lists

Concrete data structure (in C):

```
typedef struct Node {
   int  value;
   struct Node *next;
} Node;

struct SetRep {
   Node *elems;   // pointer to first node
   Node *last;    // pointer to last node
   int nelems;    // number of nodes
};
```

❖ ... Sets as Linked Lists

Set creation:

```
// create new empty set
Set newSet()
{
    Set s = malloc(sizeof(struct SetRep));
    if (s == NULL) {...}
    s->nelems = 0;
    s->elems = s->last = NULL;
    return s;
}
```

<< ^ >>

❖ ... Sets as Linked Lists

Checking membership:

```
// set membership test
int SetMember(Set s, int n)
{
    // assert(isValid(s));
    Node *cur = s->elems;
    while (cur != NULL) {
        if (cur->value == n) return true;
        cur = cur->next;
    }
    return false;
}
```

<< ^ >>

❖ ... Sets as Linked Lists

Costs for set operations on linked list:

- insert: duplicate check, insert at head; *O(n)*
- delete: find, unlink; *O(n)*
- member: linear search; *O(n)*
- card: lookup; *O(1)*
- union: copy s1, insert each item from s2; *O(nm)*
- intersect: scan for each item in s1; *O(nm)*

Assume *n* = size of *s1*, *m* = size of *s2*

If we don't have **`nelems`**, card becomes *O(n)*

<< ^ >>

❖ Sets as Bit-strings

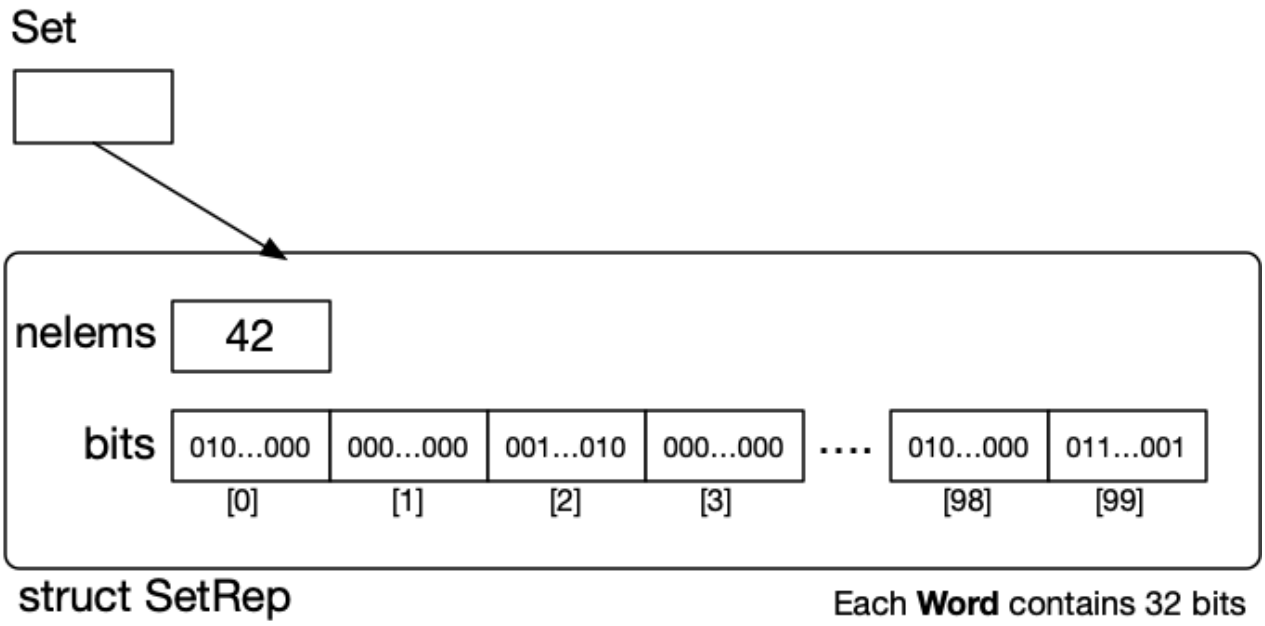
Set is a very long bit-string, typically an array of **`words`**.

Restrict possible values that can be stored in the Set

- typically restricted to 0..N-1, (where N%32 == 0)
- represent each value by position in large array of bits
- insertion means set a bit to 1 (**`bit|1`**)
- deletion means set a bit to 0 (**`bit&0`**)
- bit position for value **`i`** is easy to compute

❖ ... Sets as Bit-strings

Concrete data structure:



❖ ... Sets as Bit-strings

Concrete data structure (in C):

```
#define NBITS 1024
#define NWORDS (NBITS/32)
typedef unsigned int Word;
typedef Word Bits[NWORDS];

struct SetRep {
   int nelems;
   Bits bits;  // Word bits[NWORDS]
};
```

Sets defined like this can hold values in range 0..1023

❖ ... Sets as Bit-strings

Implementation as bit-strings requires extra functions:

- **`getBit(Bits b, int i)`** ... get value of i'th bit, 0 or 1
- **`setBit(Bits b, int i)`** ... ensure i'th bit is set to 1
- **`unsetBit(Bits b, int i)`** ... ensure i'th bit is set to 0

Can be implemented efficiently, e.g.

```
getBit(Bits b, int i) {
   int whichWord = i / 32;
   int whichBit  = i % 32;
   Word mask = (1 << whichBit)
   return (b[whichWord] & mask) >> whichBit;
}
```

❖ Setting and unsetting bits

Setting and unsetting bits by **&** and **|**

```
unsigned char x, y, z;
```

x 00000111	y 10000001	z = x & y;	z 00000001
x 00000111	y 10000001	z = x | y;	z 10000111
x 00000011		z = x & 0xFF;	z 00000011
x 00000001		z = x | 0xFF;	z 11111111
x 00000000		z = x | (1 << 2);	z 00000100
x 11111111		z = x & ~(1 << 2);	z 11111011

The last two switch on/off bit 2

❖ ... Setting and unsetting bits

Powers of two by bit-shifting - don't use **`pow(...)`** from **`math.h`**!

x		x'
00000001	x = x << 1	00000010
00000001	x = x << 2	00000100
00000001	x = x << 99	00000000
00000010	x = x >> 1	00000001
00011000	x = x >> 2	00000110
11111111	x = x >> 99	00000000

`x = x << n`

is

$x = x * 2^n$

assume:
`unsigned char x;`

`x = x >> n`

is

$x = x / 2^n$

<< ^

❖ Performance of Set Implementations

Performance comparison:

Data Structure	insert	delete	member	∪, ∩	storage
unsorted array	*O(n)*	*O(n)*	*O(n)*	*O(n.m)*	*O(N)*
sorted array	*O(n)*	*O(n)*	$O(log_2 n)$	*O(n+m)*	*O(N)*
unsorted linked list	*O(n)*	*O(n)*	*O(n)*	*O(n.m)*	*O(n)*
sorted linked list	*O(n)*	*O(n)*	*O(n)*	*O(n+m)*	*O(n)*
bit-maps	*O(1)*	*O(1)*	*O(1)*	*O(N)*	*O(N)*

n,m = #elems, *N* = max #elems,

Produced: 6 Jun 2020