Set ADTs implementation via:

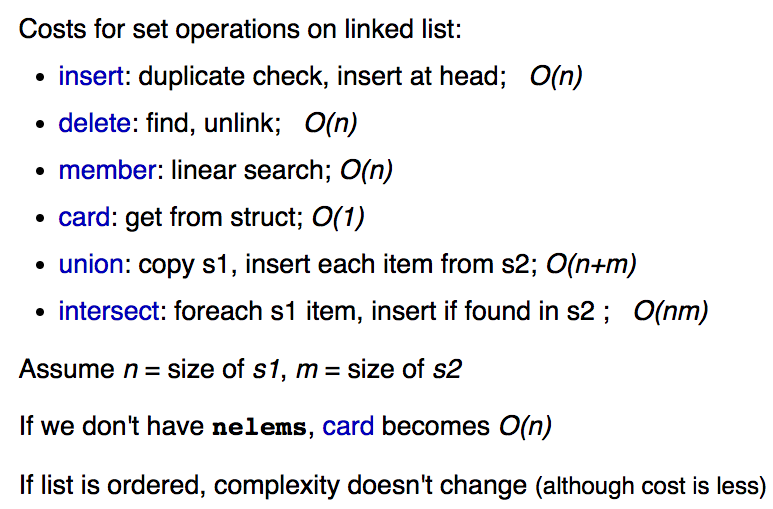
* // fixed size array, max # elements  
  **struct SetRep { int nelems; int elems[MAXELEMS]; }**
* // flexible array via. realloc()  
  **struct SetRep { int nelems; int \*elems; }**

**Sets as Linked Lists**

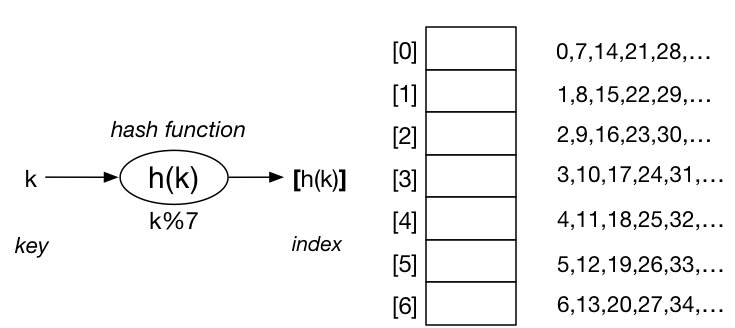
**Cardinality operations** are common to store # of elements, so you don’t need to do an O(n) traversal to count elts.

* However, this means higher space complexity / higher storage costs.
* This also means there will be extra work in incrementing / decrementing during other operations to update the counter

Efficient to have a single ptr to the **start AND end of the list**



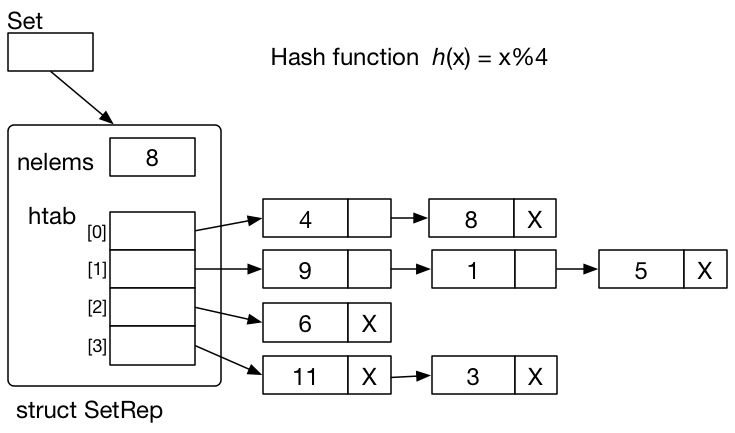
**Sets as Hash Tables**



**Hash Tables** = a way of indexing based off key values, which can be arbitrary values (numbers, strings etc.).

* Taking a key value and running it through a function that gives us an index into the Hash Table, telling us where the object belongs in the Hash Table.

If there are a large number of objects and a relatively small hash table, several objects with different key values could end up coming up with the same Hash Value, therefore going into the same spot in the Hash Table (which is bad).



E.g. insert val 4 into hash table.

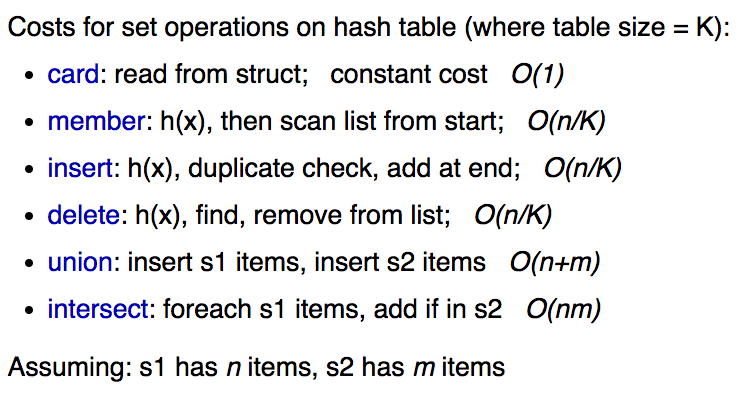
* Compute h(4) = 4%4 = 0
* 4 belongs into htab[0] 🡪 added to list

E.g. insert val 12 into hash table.

* H(12) = 12%4 = 0
* 12 belongs into htab[0] 🡪 added to list

**STEP 1: Take key value  
STEP 2: Compute index into hash table  
STEP 3: Insert object into hash table elt**

**Real world example**: We have 1 million elements, but our Hash Function is extremely complex, so Hash Table elements may only contain 1 object each. This would have similar property to accessing an array.



**Exercise: What would be the cost of a “perfect” Hash Table?**

(Achieved with a large table and a good hash function)

Costs:

* **The cost of every operation will be O(1)**

**Sets as Bit Strings**

**Bits: 0 or 1**

**Bit strings: 00010001**

Int / chars waste space. If we just use one bit, then we can save a lot of space complexity.

Implementation via. “presence” indicators (1 or 0). I.e. simply set things as “IN SET” / “NOT IN SET”

* **int inset[MAXVALUE]**;
* **int SetInsert(Set s, int i) { s->inSet[i] = 1; }**
* **int SetDelete(Set s, int i) { s->inSet[i] = 0; }**
* **int SetMember(Set s, int) { return s->inSet[i]; }** // simply return 0 or 1 to show if value is in set or not

**Linear Data Structures**

**Linear Data Structures** are collections of items where each item has **one predecessor** and **one successor**

Example: Linked lists and Arrays

Possible characteristics of linear data structures:

* Are items supposed to be in order?
* Is there a protocol for adding / removing items? (e.g. adding to front or end, allowing duplicates or not?)
* Flexible or fixed sized (Linked lists can be flexible, arrays too with realloc() function)

**Stacks**

**Stacks** are **LIFO (Last In First Out)**

Main Operations:

* Push(Stack s, Item it) // add item to stack
* Item it = pop(Stack s) // remove item from stack

Other Operations

* isEmpty(Stack s); // is the stack empty?
* nItems(Stack s); // how many items in the stack?
* show(Stack s); // display stack on Stdout
* Stack s = newStack(); // create new empty stack
* dropStack(Stack s); // release stack data.