Lecture 26 (3-25-26)

Bitsets, Linked Lists (C)

Motivating Questions:

- Using a **bitset**
- Using a **linked list**

Type sizes (in bytes)

- `char` = 1
- `int` = 4
- `double` = 8
- `void *` = 8
- `long` = 8
- `short` = 2
- `int32_t` = 4
- `uint64_t` = 8
- `size_t` = 8
- remember signed and unsigned only change how the bytes are interpreted, they don't change the size

---

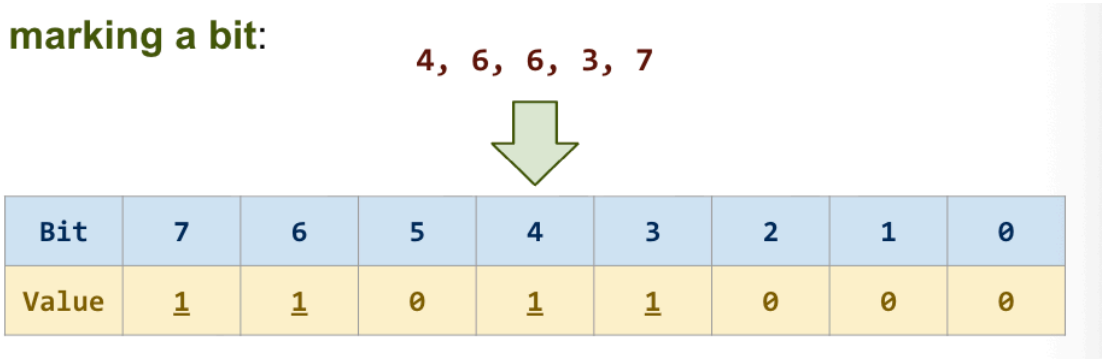
Review Data Structures

- Data structure:
  - Sequence:
    - C: n/a
    - Python: list
  - Fixed:
    - C: array
    - Python: tuple
  - Associative:

    - C: n/a
    - Python: dict
  - Membership:
    - C: n/a
    - Python: set

- Review : Set
  - `add(value)` : add value to set
  - `contains(value)` : determine if a value is in the set
  - `remove(value)` : remove value from set
  - We can implement this abstract data type using a variety of data structures
    - Bitset
    - Linked List
    - Hash Table

- Bitset: Overview
  - Instead of storing **values** in a sequence container, we can simply store whether or not the **value** is in the **set** by **marking a corresponding bit:**



  - Use **integers** as as **sets**
  - use **bit masking / operations** to set and clear individual **bits**
- **Bitset: Bitmask / Shifting**
  - To represent a particular **value**, we must produce a **bitmask** which contains a 1 at the particular bit 0's everywhere else. To make this **bitmask**, we can use **bit shifting:**

shifting:

Value: 0, Shift: 1<<0

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	1

Value: 4, Shift: 1<<4

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	1	0	0	0	0

- **Bitset: Add (Bitwise OR)**
  - to add a **value** to a **bitset**, we can use the **bitwise OR (|)** operation on the **bitset** and the **bitmask** that represents the **value:**

Bitset: 1, Bitmask: 1<<3

Bit	7	6	5	4	3	2	1	0
Bitset	0	0	0	0	0	0	0	1
Bitmask	0	0	0	0	1	0	0	0
Result	0	0	0	0	1	0	0	1
					0 | 1 = 1			1 | 0 = 1

- **Bitset: Remove (Bitwise AND)**

Bit	7	6	5	4	3	2	1	0
Bitset	0	0	0	0	1	0	0	1
Bitmask	1	1	1	1	0	1	1	1
Result	0	0	0	0	0	0	0	1
					1 & 0 = 0			1 & 1 = 1

`bitset.c`

```
/* bitset.c */
```

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

/* Type Definitions */

typedef int64_t Bitset;

/* Functions */

void     bitset_add(Bitset *b, int value) {
     // TODO
    *b = *b | (1<<value);
}

bool     bitset_contains(Bitset *b, int value) {
    // TODO
    return *b & (1<<value)
}

/* Main Execution */

int main(int argc, char *argv[]) {
    int numbers[] = {4, 6, 6, 3, 7, -1};
    Bitset bitset = 0;

    printf("Bitset: %ld\n", bitset);

    // TODO: Add numbers to bitset
    for (int *p = numbers; *p >= 0; p++){
         bitset_add(&bitset, *p)
    }

    printf("Bitset: %ld\n", bitset);

    // TODO: Check bitset for numbers
     for (int i = 0; i<10; i++){
          if(bitset_contains(&bitset, i)){
               printf("%d\n", i);
          }
     }
    return 0;
}
```

Linked List: Overview

- A **linked list** is a **data structure** that consists of a **sequence** of connected **nodes**. Each **node** stores a **value** and a **pointer** to the next **node** in the **sequence.**

`list.c`

```c
/* list.c */

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

/* Node Structure */
typedef Node Node;
// TODO
struct Node{
     int data;
     Node *next;
};

Node *   node_create(int data, Node *next) {
    // TODE
    Node *n = calloc(1, sizeof(Node));
    if (n){
         n->data = data;
         n->next = next;
    }
    return n;
}

void     node_delete(Node *n, bool recursive) {
    // TODO
    if (!n){
         return;
    }
    //free next
    node_delete(n->next, recursive);
    //free node
    free(n);
}

/* List Structure */
typedef Struct {
     Node *head;
```

```
} list;

// TODO

bool    list_contains(List *l, int value) {
    // TODO
    for (Node *curr = l->head; curr; curr= curr->next){
         if (curr->data == value){
              return true;
         }
    }
    return false;
}

void     list_add(List *l, int value) {
    // TODO
    if (list_contains(l,value)) return;

    l->head = node_create(value, l->head);

}

void     list_dump(List *l) {
    // TODO
    for (Node *curr = l->head; curr; curr->next){
         printf("%d\n", curr->data);
     }
}

/* Main Execution */

int main(int argc, char *argv[]) {
    int numbers[] = {4, 6, 6, 3, 7, -1};

    // TODO: Add numbers to list

    // TODO: Check list for numbers

    return 0;
}
```