Bryan Melanson

How to Learn Data Structures

When your brain doesn't work

2021 $\boxed{\mathcal{BM}}$

Contents

1 Bitwise Operations

1.1 Operators

Shift Left <<

Shift Right >>

Or `|`

And `&`

Invert `~`

Exclusive Or `^`

1.2 Common Operations

1.2.1 Set Bit

To set the nth bit in value x, shift 1 (`0x000000001`) by n bits and `AND` them.

```
x & (1 << n)
```

1.2.2 Clear Bit

To clear the nth bit in value x, shift 1 (`0x000000001`) by n bits and invert the `AND`ed value.

```
x & ~(1 << n)
```

1.2.3 Flip Bit

To flip the nth bit in value x, shift 1 (`0x000000001`) by n bits and `XOR` them.

```
x^(1 << n)
```

1.2.4 Clear

To clear all values, `AND` the value with `0xFFFF`.

```
x & 0xFFFF
```

1.2.5 Little Endian to Big Endian

A little-endian system, in contrast, stores the least-significant byte at the smallest address.

Binary (Decimal: 149)	1	0	0	1	0	1	0	1
Bit weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bit position	MSB							LSB

In the case of the value `0x12345678`, the **least significant byte** of this value `0x78` is stored in the lowest little Endian address, and subsequent bytes are stored in the next locations. The least significant byte can be considered the byte with the lowest value, when evaluated in typical bitwise fashion: `0x78563412`

```
0x78 (0x004000)

0x56 (0x004001)

0x34 (0x004002)

0x12 (0x004003)
```

1.2.6 Big Endian to Little Endian

In a Big Endian representation of `0x12345678`, the **most significant byte** (`0x12`) is stored at the lowest memory address: `0x12345678`

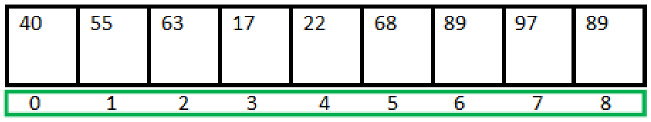
```
0x12 (0x004000)

0x34 (0x004001)

0x56 (0x004002)

0x78 (0x004003)
```

To reverse these positions, the bytes can be isolated and bit-shifted by the appropriate number of bits to form the appropriate order.

```
(0x12345678 & 0x000000FF) << 24

(0x12345678 & 0x0000FF00) << 16

(0x12345678 & 0x00FF0000) << 8

(0x12345678 & 0xFF000000) >> 24
```

2 Array

Arrays are collections of same-type data items stored in a contiguous memory location. Knowing the data type, each element is located at an offset based on that data size. In other words, for `int data[]` the data at `data[1]` is located at `data[0] + sizeof(int)`.



2.1 Arrays in C

There is no index out-of-bound checking in C, so if access goes beyond the index boundaries of the array (`0-(n-1)`) there will be undefined behavior.

2.2 Static vs. Dynamic Arrays

Pointer and array accesses can be treated the same way in C, either by accessing the values by using the [] operator, or by incrementing the value of the pointer.

2.2.1 Static Arrays

In C, the size of an array should be decided at compile time by defining the array size either by declaring an array with size constraints such as `arr[10]` or by using `malloc` to define the required size. At run time this size will be used to allocate the required memory space.

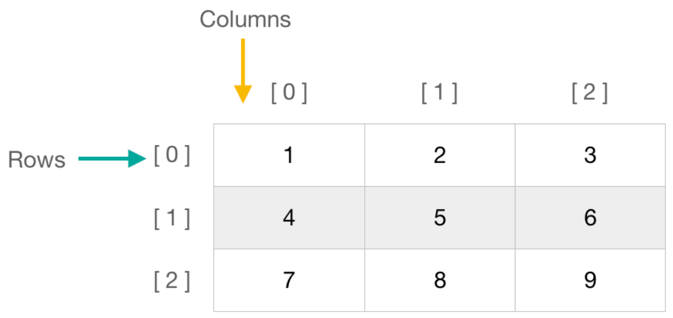
2.2.2 Dynamic Arrays

In C++ an array can be passed a variable and the size can be determined for the memory allocation at run time.

2.3 C Strings

A C string is a pointer to an array of `char` data items which is terminated by the `NULL` character at the size limit. So, a `char` array of `n` items will contain data at elements `0-(n-1)`, while the value at `n` will be `NULL`. This allows for string operations to check for the boundary of memory space for the string object.

3 Matrix

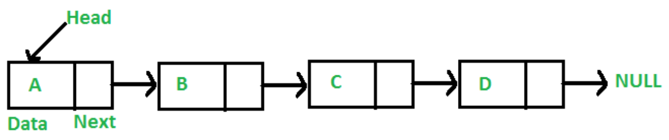


3.1 Normal Declaration

3.2 Dynamic Declaration

4 Linked List

Unlike arrays, Linked Lists are dynamic and can grow by pointing to non-contiguous locations in memory using pointers. Extra memory is required when representing each pointer in the linked list, but it allows for easily inserting and deleting objects in the linekd list.



Each node in the linked list consists of its data, and a pointer to the next object in the linked list. In the case that the head of the linked list is null.

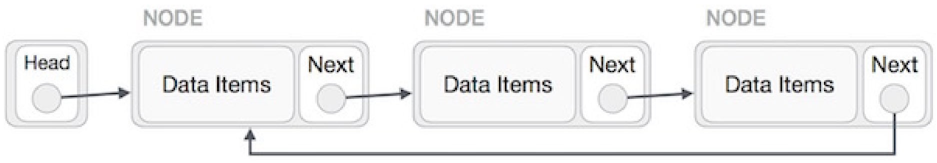
4.1 Inserting

4.2 Deleting

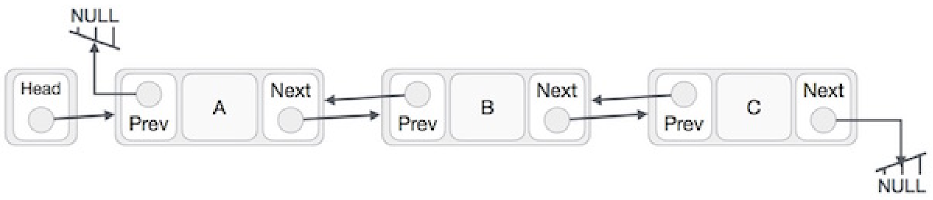
5 Circular Linked List

5.1 Inserting

5.2 Deleting



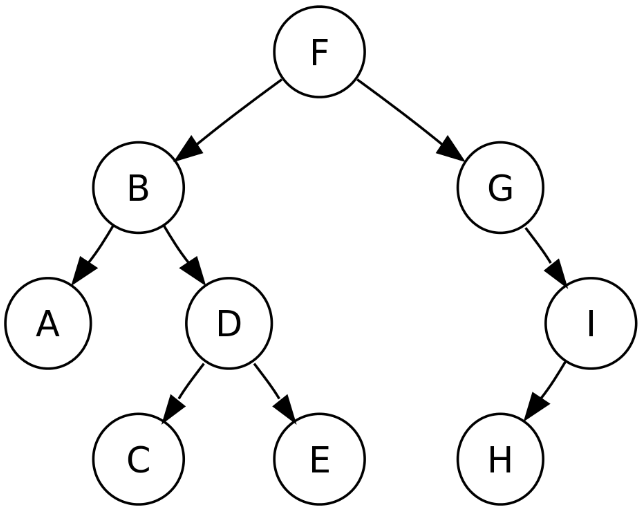
6 Doubly Linked List



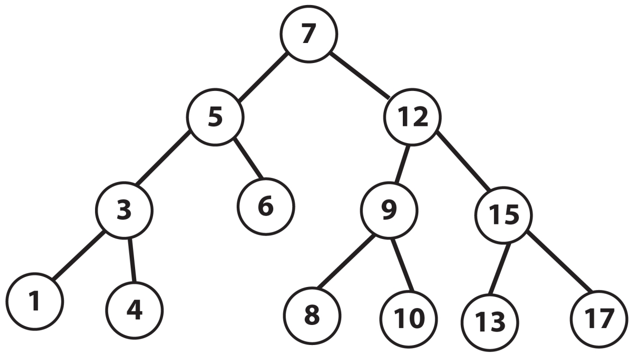
6.1 Inserting

6.2 Deleting

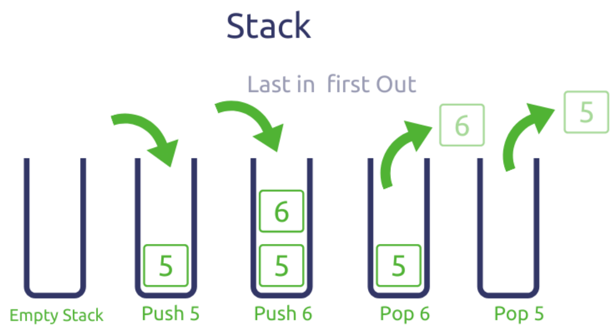
7 Binary Tree



8 Binary Search Tree



9 Stack



9.1 Stack Array

A struct can be created which tracks the index of the top, and holds the allocated array. When popping a value, the array index should be wiped with a `null` value and top decremented. If the top index is out of range, it should throw an error to indicate a stack underflow.

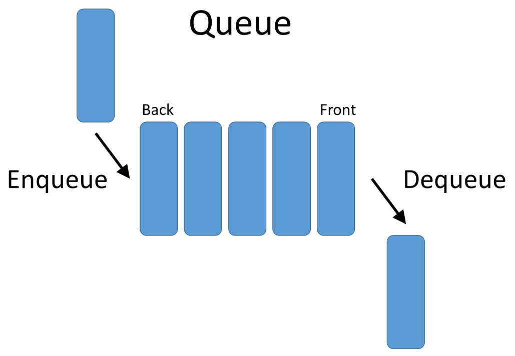
```
/*
* Stack implementation using array in C
*/

#defined CAPACITY 100

typedef struct stack {
    int top;
    arr[CAPACITY];
} stack;

// Function to push a new element in stack
void push(stack* s, int val)
{
    s->arr[s->top++] = val;
}

// Function to pop element from top of stack
int pop(stack* s)
{
    if (top < 0) return -1;
    return s->arr[s->top--];
}
```

9.2 Dynamic Stack Using Linked List

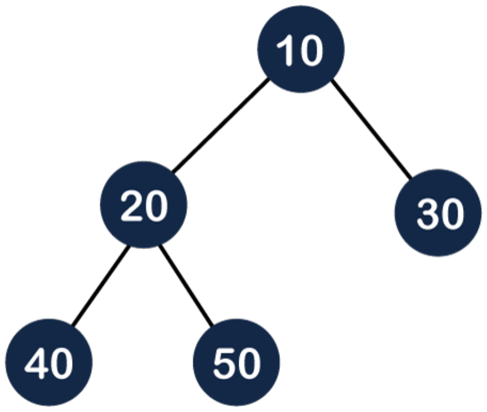
When the size of the array is not known and allocating a static array of its capacity may be wasteful, Linked Lists can be used to connect the items on the stack. When popping an object, the node pointer should be stored in a temporary pointer, then free'd once the new top is the old node's next pointer.

```c
/*
* Stack implementation using linked list in C
*/

// Define stack node structure
// The variable also instantiates this as a global
struct stack {
    int data;
    struct stack *next;
} *top;

// Function to push a new element in stack.
void push(int element)
{
    // Create a new node and push to stack
    struct stack* newNode = malloc(sizeof(struct stack));

    // Assign data to new node in stack
    newNode->data = element;

    // Next element after new node should be current top element
    newNode->next = top;

    // Make sure new node is always at top
    top = newNode;
}

// Function to pop element from top of stack.
int pop()
{
    // Check stack underflow
    if (!top)
    {
        printf("Stack is empty.\n");
        return -1;
    }
    // Hold pointer to node to be removed
    stack* old = top;

    int data = 0;

    // Copy data from stack's top element
    data = old->data;

    // Move top to its next element
    top = old->next;

    free(old);

    return data;
}
```

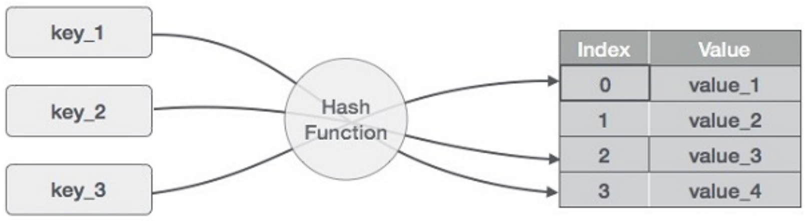
10 Queue



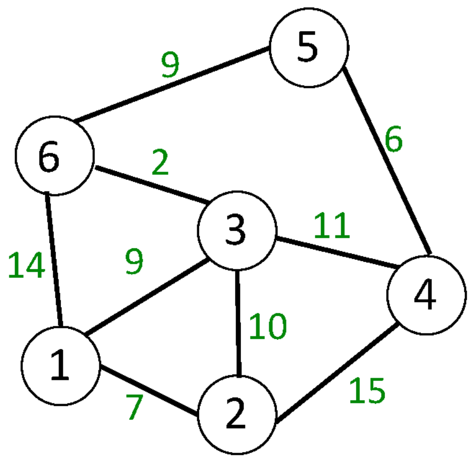
11 Heap



12 Hash Tables



13 Graph



9
5
6
2
6
3
11
14
9
4
10
1
7
2
15