Learning C by example

C programs that I have found useful during my studies

Category Archives: Common C interview questions

These are from my experience some of the common questions engineers are asked about C languange

print diamond shape in C

```
#include <stdio.h>
#include <stdlib.h>

void diamond1(int lines) {
    // make sure number lines is odd
    if (lines%2 == 0) lines++;

    int l, i;
    /* upper part (and central line)
       space_left + starts + space_right = lines
       space_left = space_right = space
       2*space + starts = lines
       starts = 2*line_no -1
       space = (lines - 2*line_no +1)/2
             = (lines+1)/2 - 2*line_no
    */
    for (l = 1; l<=(lines+1)/2; l++) {
        for (i = 1; i<=(lines-2*l+1)/2; i++)
            printf(" ", i);
        // starts
        for (i = 1; i<=2*l-1; i++)
```

```c
            printf("*");
        printf("\n");
    }
    /* bottom part
       2*space + starts = lines
       starts = 2*(lines - line_no) + 1
       space = (lines - 2*lines + 2*line_no - 1)/2
             = (2*line_no - lines -1)/2
             = line_no - (lines+1)/2
    */
    for( ; l<=lines; l++) {
        for (i = 1; i <= (2*l-lines-1)/2; i++)
            printf(" ", i);
        // starts
        for (i = 1 ; i<= 2*(lines-l) + 1; i++)
            printf("*");
        printf("\n");
    }
}

int diamond2(int lines) {
    if (lines%2==0) lines++;
    int l, c;
    int stars = 1;
    int spaces = (lines - stars)/2; // 2*spaces+stars=lines
    for (l = 1; l <= (lines+1)/2; l++) {
        for (c = 1; c <= spaces; c++)
            printf(" ");
        for (c = 1; c <=stars; c++)
            printf("*");
        stars+=2;
        spaces--;
        printf("\n");
    }
    spaces = 1;
    stars = (lines+1)/2+1;
    for ( ;l <= lines; l++) {
        for (c = 1; c <= spaces; c++)
            printf(" ");
        for (c = 1; c <=stars; c++)
            printf("*");
        stars-=2;
        spaces++;
        printf("\n");
    }
}


int main() {
    diamond1(7);
    diamond2(7);

    return 0;
}
```

```

   *
  ***
 *****
*******
 *****
  ***
   *
   *
  ***
 *****
*******
 *****
  ***
   *
```

Semaphores

This is a problem from an interview I found online. There are three lists (implemented as array in my solution), and three threads. Each thread access one list, and prints one number from the list. The threads must sync to print in this order: T1 -> T2 -> T3 -> T1 -> T2, …

The synchronization method used is semaphores. Semaphores are usually used to sync multiple threads. One thread can let another run by posting a semaphore for the other thread.

This is a solution with pthreads:

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>

#define N 10
sem_t sem1, sem2, sem3;
int a1[N], a2[N], a3[N];

void * thread1(void *arg) {
    int i;
    for (i = 0; i< N; i++) {
        sem_wait(&sem1);
        printf("%d, ", a1[i]);
        sem_post(&sem2);
    }
    pthread_exit(0);
}

void * thread2(void *arg) {
    int i;
    for (i = 0; i< N; i++) {
```

```
        sem_wait(&sem2);
        printf("%d, ", a2[i]);
        sem_post(&sem3);
    }
    pthread_exit(0);
}

void * thread3(void *arg) {
    int i;
    for (i = 0; i< N; i++) {
        sem_wait(&sem3);
        printf("%d, ", a3[i]);
        sem_post(&sem1);
    }
    pthread_exit(0);
}

int main() {
    pthread_t threads[3];
    int i;

   int no = 0;
    for (i = 0; i < N; i++) {
        a1[i] = no++;
        a2[i] = no++;
        a3[i] = no++;
    }

    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 0);
    sem_init(&sem3, 0, 0);

    pthread_create(&threads[0], NULL, thread1, NULL);
    pthread_create(&threads[1], NULL, thread2, NULL);
    pthread_create(&threads[2], NULL, thread3, NULL);

    for (i = 0; i < 3; i++)
        pthread_join(threads[i], NULL);
    printf("\n");

    sem_destroy(&sem1);
    sem_destroy(&sem2);
    sem_destroy(&sem3);

    return 0;
}


gcc -pthread threelists.c
./a.out
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
```

Reverse order of words within string

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define SWAP(a, b) (a^=b, b^=a, a^=b)

void reversestring(char *a, char *b) {
    assert(a != NULL && b != NULL);
    /* while (a++ < b++) runs the while loop with
     * parameters incremented, but check condition before
     * parameters are incremented
     * while (++a < ++b) increments parameters before
     * checking condition
     * both ( ;i<max ; i++) and ( ;i<max; ++i) run the
     * loop and then increment the parameter. they are both
     * identical */
    while (a < b) {
        SWAP(*a, *b);
        a++; b--;
    }
}
void reverseorderwords(char *str) {
    assert(str != NULL);
    char * ptra = str;
    char * ptrb = str;

    while (*ptrb != '\0') {
        while (*ptrb != '\0' && *ptrb != ' ') {
            ptrb++;
        }
        reversestring(ptra, ptrb-1);
        if (*ptrb != '\0') {
            ptrb++;
            ptra = ptrb;
        }
    }
    reversestring(str, --ptrb);
}


int main() {
    //char * str = "this is a test"; // static string read-only.
                                     // will segfault when changed
    char str[] = "this is a test";
    printf("%s\n", str);
    reverseorderwords(str);
```

```
    printf("%s\n", str);
    return 0;
}



./a.out
this is a test
test a is this
```

Linked List interview problems

Collection of linked list interview problems implemented in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


#define N 19 // hash table size
#define EMPTY -1
#define TRUE 1
#define FALSE 0

struct node {
    struct node * next;
    int value;
};
typedef struct node node_t;


int haskey(int table[N][5], int value) {
    int key = value % N;
    int i = 0;
    while (table[key][i] != -1) {
        if (table[key][i] == value)
            return TRUE;
        i++;
    }
    return FALSE;
}

void insertvalue(int table[][5], int value) {
    int key = value % N;
    int i = 0;
```

```c
    // checking if collision
    while (table[key][i] != -1) {
        i++;
        if (i == 5) // too many collisions
            return; // discard value
    }
    // insert value at key position
    table[key][i] = value;
    if (i < 4)
        table[key][++i] = -1;
}

/*
 * remove duplicates in unsorted list
 */
void remove_duplicates(node_t * list) {
    /* solution 1: sweep in two nested loos (O(N2))
       solution 2: put elements in buffer (O(N)) and
       sort elements (O(N*logN)) and sweep list (O(N)),
       checking if element in buffer (O(logN). Total O(NlogN)
       solution 3: use hash or bit array. O(N)
    */
    if (list == NULL || list->next == NULL)
        return;

    /* create hash table */
    int i;
    int hashtable[N][5]; // using array for collisions (max 5 key colli
    // for (i = 0; i < N; i++)
    //    hashtable[i][0] = EMPTY; // -1 indicates no value for the key
    memset(hashtable, EMPTY, sizeof(hashtable));

    /* sweep list */
    insertvalue(hashtable, list->value); // first element

    node_t * current = list->next;
    node_t * prev = list;
    node_t * duplicate;
    while (current != NULL) {
        if (haskey(hashtable, current->value)) {
            duplicate = current;
            /* skip node */
            prev->next = current->next;
            /* move current, prev doesnt move */
            current = current->next;
            /* free duplicate node */
            free(duplicate);
        } else {
            insertvalue(hashtable, current->value);
            /* move forward */
            prev = current;
            current = current->next;
        }
    }
```

```
}

void insert_beginning(node_t ** list, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode->value = value;
    // newnode->next = NULL;
    newnode->next = *list;
    *list = newnode;
}

void print_list(node_t * list) {
    while (list != NULL) {
        printf("%d, ", list->value);
        list = list->next;
    }
    printf("\n");
}

/*
 * Remove node of list having access to only that node
 */
void remove_node(node_t *node) {
    node_t *next = node->next;
    memcpy(node, node->next, sizeof(node_t));
    free(next);
}

node_t * nodefromend(node_t *list, int k) {
    node_t *first = list;
    node_t *second = list;
    int i = 0;
    while (first != NULL && i < k) {
        first = first->next;
        i++;
    }
    if (first == NULL)
        return NULL;
    while (first != NULL) {
        first = first->next;
        second = second->next;
    }
    return second;
}


/*
 * detect if list has loop
 */
int detectloop(node_t *list) {
    /* using slow ptr and fast ptr that
       advances two nodes at a time. If there is
       loop, they both will get into loop, and eventually
       fast ptr will point reach slow ptr and point to
       same node
```

```
     */
    if (list == NULL)
        return FALSE;


    node_t *slow = list;
    node_t *fast = list;
    while (fast->next->next != NULL) {
        fast = fast->next->next;
        slow = slow->next;
        if (fast == slow)
            return TRUE;
    }
    return FALSE;
}

int ispalindrome(node_t * list) {

    if (list == NULL || list->next == NULL)
        return FALSE;

    int queue[40];
    int index = 0;
    node_t * fast = list;
    node_t * slow = list;
    /* traverse list up to middle and put elements
       in a queue. use fast/slow method to stop at
       middle */
    while (fast != NULL && fast->next != NULL &&\
           fast->next->next != NULL) {
        queue[index++] = slow->value;
        slow = slow->next;
        fast = fast->next->next;
    }
    /* if list odd lenght, skip element in middle */
    if (fast->next != NULL)
        slow = slow->next;

    /* move slow to end, and compare with LIFO queue */
    while (slow != NULL) {
        if (slow->value != queue[--index])
            return FALSE;
        slow = slow->next;
    }
    return TRUE;
}

int main() {

    node_t * list = NULL;
    int a[8] = {4, 2, 7, 4, 3, 7, 9, 2};
    int i;
    for (i = 0; i < 8; i++) {
        insert_beginning(&list, a[i]);
```

```
        print_list(list);
    }
    /* remove duplicate nodes in unsorted list */
    remove_duplicates(list);
    print_list(list);

    /* return node k nodes from the end */
    node_t * node = nodefromend(list, 3);
    assert(node!=NULL);
    printf("%d\n", node->value);

    /* remove a node, give ptr to that node only */
    remove_node(node);
    print_list(list);

    /* detect if there is loop */
    if (detectloop(list))
        printf("Loop detected\n");
    else
        printf("Loop not detected\n");

    /* check if list is palindrome */
    if (ispalindrome(list))
        printf("List is palindrome\n");
    else
        printf("List is not palindrome\n");

    return 0;
}



4,
2, 4,
7, 2, 4,
4, 7, 2, 4,
3, 4, 7, 2, 4,
7, 3, 4, 7, 2, 4,
9, 7, 3, 4, 7, 2, 4,
2, 9, 7, 3, 4, 7, 2, 4,
2, 9, 7, 3, 4,
7
2, 9, 3, 4,
Loop not detected
List is not palindrome
```

all types of linked lists

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>


/* node for single linked list */
typedef struct node {
    struct node *next;
    int value;
} node_ts;

/* node for double linked list */
typedef struct noded {
    struct noded *next;
    struct noded *prev;
    int value;
} node_td;


/*
 * Routines for single linked list
 */
void insert_single(node_ts ** list, int value) {
    node_ts * new = malloc(sizeof(node_ts));
    new->value = value;
    new->next = NULL;

    /* empty */
    if (*list == NULL) {
        *list = new;
        return;
    }
    /* element becomes first. can be merged with case above */
    if ((*list)->value > value) {
        new->next = *list;
        *list = new;
        return;
    }

    node_ts *prev = *list;
    node_ts *ptr = (*list)->next;
    while (ptr != NULL && ptr->value < value) {
        prev = ptr;
        ptr = ptr->next;
    }
    prev->next = new;
    new->next = ptr;
}

void print_single(node_ts *list) {
    while (list != NULL) {
        printf("%d, ", list->value);
        list = list->next;
```

```c
    }
    printf("\n");
}

void reverse_single(node_ts **list) {

    /* empty or one element */
    if (*list == NULL || (*list)->next == NULL) {
        return;
    }

    node_ts * prev = NULL;
    node_ts * current = *list;
    node_ts * next = (*list)->next;

    while (next != NULL) {
        current->next = prev;
        /* move ptrs forward */
        prev = current;
        current = next;
        next = next->next;
    }
    /* current points to last element */
    current->next = prev;
    *list = current;
}

/*
 * Routines for circular single linked list
 */
void insert_singlecircular(node_ts ** list, int value) {

    node_ts * new = malloc(sizeof(node_ts));
    new->value = value;
    new->next = NULL;

    /* empty list */
    if (*list == NULL) {
        *list = new;
        new->next = new;
        return;
    }

    /* place at first location */
#ifdef TRAVERSE
    if ((*list)->value > value) {
        /* if list has only one node */
        if ((*list)->next == *list) {
            new->next = *list;
            (*list)->next = new;
            *list = new;
            return;
        }
        /* if list has more nodes, get last node */
```

```c
        node_t *ptr = (*list)->next;
        while (ptr->next != *list) {
            assert(ptr->next != NULL);
            ptr = ptr->next;
        }
        /* insert node */
        ptr->next = new;
        new->next = *list;
        *list = new;
        return;
    }
#else
    /* another way to insert at beginning
       without having to traverse list */
    if ((*list)->value > value) {
        memcpy(new, *list, sizeof(node_t));
        (*list)->value = value;
        (*list)->next = new;
        return;
    }
#endif


    node_ts *prev = *list;
    node_ts *ptr = (*list)->next;
    while (ptr != *list && ptr->value < value) {
        prev = ptr;
        ptr = ptr->next;
    }
    prev->next = new;
    new->next = ptr;
}

void print_singlecircular(node_ts * list) {
    node_ts * ptr = list;
    do {
        printf("%d, ", ptr->value);
        ptr = ptr->next;
    } while (ptr != list);
    printf("\n");
}

void reverse_singlecircular(node_ts ** list) {

    if (*list == NULL || (*list)->next == *list)
        return;

    node_ts * prev = *list;
    node_ts * current = (*list)->next;
    node_ts * next = current->next;

    while(current != *list) {
        current->next = prev;
        prev = current;
```

```
        current = next;
        next = next->next;
    }
    /* prev points to last element,
       current points to first */
    current->next = prev;
    *list = prev;
}



/*
 * Routines for double linked list
 */
void insert_double(node_td ** list, int value) {
    node_td * new = malloc(sizeof(node_td));
    new->value = value;
    new->prev = NULL;
    new->next = NULL;

    /* empty list */
    if (*list == NULL) {
        *list = new;
        return;
    }

    /* first place */
    if ((*list)->value > value) {
        new->next = *list;
        (*list)->prev = new;
        *list = new;
        return;
    }

    node_td *ptr = *list;
    /* looking one node ahead */
    while (ptr->next != NULL && ptr->next->value < value) {
        ptr = ptr->next;
    }

    /* place at end of list */
    if (ptr->next == NULL) {
        ptr->next = new;
        new->prev = ptr;
        return;
    }

    /* place new node */
    new->prev = ptr;
    new->next = ptr->next;
    /* adjust previous node */
    ptr->next = new;
    /* adjust next node */
    new->next->prev = new;
```

```c
}

void reverse_double(node_td **list) {
    /* empty or one element */
    if (*list == NULL || (*list)->next == NULL) {
        return;
    }

    node_td * ptr = *list;
    node_td * next = (*list)->next;

    while (next != NULL) {
        /* reverse prev and next pointers */
        ptr->next = ptr->prev;
        ptr->prev = next;
        /* move forward */
        ptr = next;
        next = next->next;
    }
    /* make last element the first one */
    ptr->next = ptr->prev;
    ptr->prev = NULL;
    *list = ptr;
}

void print_double(node_td * list) {
    while (list != NULL) {
        printf("%d, ", list->value);
        list = list->next;
    }
    printf("\n");
}


/*
 * Routines for double circular linked list
 */
void insert_double_circular(node_td ** list, int value) {
    node_td * new = malloc(sizeof(node_td));
    new->value = value;
    new->next = NULL;
    new->prev = NULL;

    /* empty list */
    if (*list == NULL) {
        *list = new;
        new->prev = new;
        new->next = new;
        return;
    }

    /* insert at beggining */
    if (value < (*list)->value) {
        node_td * prev = (*list)->prev;
```

```
        /* insert new node */
        new->next = *list;
        new->prev = (*list)->prev;
        /* adjust next (previoulsy first node) */
        (*list)->prev = new;
        /* adjust previous (previously last node) */
        prev->next = new;
        /* adjust list pointer */
        *list = new;
        return;
    }


    node_td * ptr = (*list)->next;
    node_td * prev;
    while (ptr != *list && ptr->value < value) {
        ptr = ptr->next;
    }
    prev = ptr->prev;

    prev->next = new;
    ptr->prev = new;
    new->next = ptr;
    new->prev = prev;
}

void print_double_circular(node_td * list) {

    if (list == NULL) {
        printf("\n");
        return;
    }
    node_td *ptr = list;
    do {
        printf("%d, ", ptr->value);
        ptr = ptr->next;
    } while (ptr != list);
    printf("\n");
}

void reverse_double_circular(node_td ** list) {

    if (*list == NULL || (*list)->next == *list)
        return;

    node_td *ptr = *list;
    node_td *next = ptr->next;
    while(next != *list) {
        /* switch next and prev pointers */
        ptr->next = ptr->prev;
        ptr->prev = next;
        /* move forward */
        ptr = next;
        next = next->next;
```

```c
    }
    /* last element becomes first */
    ptr->next = ptr->prev;
    ptr->prev = next;

    *list = ptr;
}

int main() {
    int a[5] = {3, 1, 6, 5, 9};
    int i;

    // --------
    printf("testing single linked list\n");
    node_ts * singlell = NULL;
    for (i = 0; i<5; i++) {
        insert_single(&singlell, a[i]);
        print_single(singlell);
    }
    reverse_single(&singlell);
    print_single(singlell);

    // ----------
    printf("testing double linked list\n");
    node_td * doublell = NULL;
    for (i = 0; i<5; i++) {
        insert_double(&doublell, a[i]);
        print_double(doublell);
    }
    reverse_double(&doublell);
    print_double(doublell);

    // ---------
    printf("testing single circular linked list\n");
    node_ts * singlec = NULL;
    for (i = 0; i<5; i++) {
        insert_singlecircular(&singlec, a[i]);
        print_singlecircular(singlec);
    }
    reverse_singlecircular(&singlec);
    print_singlecircular(singlec);

    // ----------
    printf("testing double circular linked list\n");
    node_td * doublecircularll = NULL;
    for (i = 0; i<5; i++) {
        insert_double_circular(&doublecircularll, a[i]);
        print_double_circular(doublecircularll);
    }
    reverse_double_circular(&doublecircularll);
    print_double_circular(doublecircularll);


    return 0;
```

```
}



testing single linked list
3,
1, 3,
1, 3, 6,
1, 3, 5, 6,
1, 3, 5, 6, 9,
9, 6, 5, 3, 1,
testing double linked list
3,
1, 3,
1, 3, 6,
1, 3, 5, 6,
1, 3, 5, 6, 9,
9, 6, 5, 3, 1,
testing single circular linked list
3,
1, 3,
1, 3, 6,
1, 3, 5, 6,
1, 3, 5, 6, 9,
9, 6, 5, 3, 1,
testing double circular linked list
3,
1, 3,
1, 3, 6,
1, 3, 5, 6,
1, 3, 5, 6, 9,
9, 6, 5, 3, 1,
```

Semaphores

The problem below synchronizes two threads, one prints odd numbers, and the other prints even numbers. In this solution each thread posts a semaphore to wake up the other thread when a number is printed. Since the same lock is acquire by a thread (wait), and released (post) by a different thread that did not own it, mutex cannot be used, since for a mutex the same thread must acquire it and release it.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

/* two threads, one prints odd numbers,
the other even numbers, up to 10
```

```
*/

sem_t odd_lock;
sem_t even_lock;

void * even(void *arg) {
int a = 1;
while (a < 10) {
sem_wait(&even_lock);
printf("number: %d\n", a);
sem_post(&odd_lock);
a+=2;
}
pthread_exit(0);
}

void * odd(void *arg) {
int a = 2;
while (a < 10) {
sem_wait(&odd_lock);
printf("number: %d\n", a);
sem_post(&even_lock);
a+=2;
}
pthread_exit(0);
}

int main() {
pthread_t threads[2];
sem_init(&odd_lock, 0, 0);
sem_init(&even_lock, 0, 1);
pthread_create(&threads[0], NULL, even, NULL);
pthread_create(&threads[1], NULL, odd, NULL);
pthread_join(threads[0], NULL);
pthread_join(threads[1], NULL);
sem_destroy(&odd_lock);
sem_destroy(&even_lock);
return 0;
}

gcc -pthread sem.c

./a.out
number: 1
number: 2
number: 3
number: 4
number: 5
number: 6
number: 7
number: 8
number: 9
```

malloc memory at n-byte memory boundary

I was asked in a interview once to write a c function that uses malloc, but aligns the pointer to n-byte boundary. And then to write a function to free the memory for that pointer. The solution I am posting here calls malloc,, then aligns the pointer to n-byte memory boundary, and then it stores the padding used for alignment at the first bytes. In this way, the free function can retrieve the address for the original memory allocated by malloc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>  // for uintptr_t type, needed for boolean
                       // operations on pointers


/* Aligning pointer to  nbyte memory boundary
   padding = n - (offset & ( -1)) = -offset & (n-1)
   aligned offset = (offset + n-1) & ~(n-1)
 */
void * mallocaligned(size_t size, int align) {
    if (align < sizeof(int))
        align = sizeof(int);
    /* allocate pointer with space to store original address at top and
     * to move to align-byte boundary */
    void *ptr1 = malloc(size + align + align - 1);
    printf("%d bytes of memory allocated at %p\n", size+2*align-1, ptr1)
    /* align pointer to align-byte boundary */
    void *ptr2 = (void *)(((uintptr_t)ptr1 + align - 1) & ~(align-1));
    /* store there the original address from malloc */
    *(unsigned int *)ptr2 = (unsigned int)ptr1;
    /* move pointer to next align-byte boundary */
    ptr2 = ptr2 + align;
    printf("aligned memory at %p\n", ptr2);

    return ptr2;
}

void freealigned(void *ptr, int align) {
    /* move pointer back align bytes */
    ptr = (void *)((uintptr_t)ptr - align);
    /* retreive from there the original malloced pointer */
    ptr = (void *)(*(unsigned int *)ptr);
    printf("free memory at address %p\n", ptr);
    /* free that pointer */
    free(ptr);
}

int main() {
```

```
    void *ptr = mallocaligned(1000, 64);
    printf("allocated pointer at %p", ptr);
    freealigned(ptr, 64);
    return 0;
}
```

Linked list remove all duplicates

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    struct node *next;
    int value;
};
typedef struct node node_t;

void removeallduplicates(node_t *head) {
    node_t *ptr1 = head;
    node_t *current;
    node_t *prev;
    node_t *deleteit;
    while (ptr1 != NULL) {
        current = ptr1->next;
        prev = ptr1;
        while (current != NULL) {
            if (current->value == ptr1->value) {
                deleteit = current;
                prev->next = current->next;
                current = current->next;
                free(deleteit);
            } else {
                prev = current;
                current = current->next;
            }
        }
        ptr1 = ptr1->next;
    }
}
```

```
void push(node_t **head, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->value = value;
    newnode->next = *head;
    *head = newnode;  // newnode becomes the new head
}

void printlist(node_t *head) {
    node_t *ptr = head;
    while (ptr != NULL) {
        printf("%d, ", ptr->value);
        ptr = ptr->next;
    }
    printf("\n");
}

int main() {

    node_t * headptr = NULL;
    int numbs[] = {1,2,3,4,5,1,5,2,4,3};
    int i;
    for (i = 0; i < 10; i++)
        push(&headptr, numbs[i]);
    printlist(headptr);

    removeallduplicates(headptr);
    printlist(headptr);

    return 0;
}


>  gcc linkedlistremovedup.c  -g
> ./a.out
3, 4, 2, 5, 1, 5, 4, 3, 2, 1,
3, 4, 2, 5, 1,
```

Linked list functions with dummy head and without it

Some linked list routines need to modify the head node (first node) of the list, if for example the head node is a new node, or the head node needs to be removed. This means that these routines need to modify the pointer to the head node, to point to a different node. In this cases, these routines can either return the new head node pointer, so that the function calling can update it, or get a pointer to the head node pointer, so that they can modify the head node pointer without any problems.

A different approach for this is to use a dummy head, which next pointer points to the head node. In this way the list functions can just pass a pointer to the dummy head, instead of a pointer to the head node pointer.

The following program illustrates both approaches:

```
#include &amp;lt;stdio.h&amp;gt;
#include &amp;lt;stdlib.h&amp;gt;

struct node {
    struct node *next;
    int value;
};
typedef struct node node_t;

void insert_ordered_dummy(node_t * dummy, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->value = value;
    newnode->next = NULL;
    if (dummy->next == NULL) {
        dummy->next = newnode;
        return;
    }
    node_t * current = dummy->next;
    node_t * prev = dummy;
    while (current != NULL && current->value < value) {
        prev = current;
        current = current->next;
    }
    prev->next = newnode;
    newnode->next = current;
}

void insert_ordered_nodummy(node_t ** list, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode->next = NULL;
```

```
    newnode->value = value;

    // *list->value is equivalent to *(list->value)
    if (*list == NULL || (*list)->value > value) {
        newnode->next = *list;
        *list = newnode;
        return;
    }

    node_t * prev = *list;
    node_t * current = (*list)->next;
    while (current != NULL && current->value < value) {
        prev = current;
        current = current->next;
    }
    prev->next = newnode;
    newnode->next = current;
}
void insertend_nodummy(node_t **head, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = NULL;

    if (*head == NULL) {
        *head = newnode; // if list empty, newnode is head
        return;
    }

    node_t *ptr = *head;
    while (ptr-&amp;gt;next != NULL) { // move ptr to last node
        ptr = ptr-&amp;gt;next;
    }
    ptr-&amp;gt;next = newnode; // insert after last node
}

void insertbeginning_withdummy(node_t *dummyhead, int value) {
    node_t * newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = dummyhead-&amp;gt;next;
    dummyhead-&amp;gt;next = newnode;  // now dummy head points to new node
}
void insertend_withdummy(node_t *dummyhead, int value) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode-&amp;gt;value = value;
    newnode-&amp;gt;next = NULL;
```

```
    if (dummyhead-&amp;gt;next == NULL) {
        dummyhead-&amp;gt;next = newnode;
        return;
    }

    node_t *ptr = head;
    while (ptr-&amp;gt;next != NULL) { // move ptr to last node
        ptr = ptr-&amp;gt;next;
    }
    ptr-&amp;gt;next = newnode; // insert after last node
}

void printlist(node_t *head) {
    node_t *ptr = head;
    while (ptr != NULL) {
        printf(&amp;quot;%d, &amp;quot;, ptr-&amp;gt;value);
        ptr = ptr-&amp;gt;next;
    }
    printf(&amp;quot;\n&amp;quot;);
}

int main() {

    /* without dummy node: the first node is the head. main keeps a pointer
      to the first node (head). If a function needs to modify the head of
      the list (say a new node replaces the head node), then the head pointe
      needs to be modified, which means that a pointer to the head pointer n
      to be passed to the function. Or otherwise the function could return t
      new head pointer */
    node_t * headptr = NULL; // dont forget NULL!!
    int numbs[] = {1,2,3,4,5};
    int i;
    for (i = 0; i &amp;lt; 5; i++) {
        insertbeginning_nodummy(&amp;amp;headptr, numbs[i]);
        printlist(headptr);
    }
    for (i = 0; i &amp;lt; 5; i++) {
        insertend_nodummy(&amp;amp;headptr, numbs[i]);
        printlist(headptr);
    }


    /* with dummy node: in this case you can just pass pointer to
       dummy head to any function, and the function will be able to
       modify the next pointer in the dummy head. So no need for
       double pointers */
```

```
    node_t dummy_head;
    dummy_head.next = NULL;
    dummy_head.value = -1;
    for (i = 0; i &amp;lt; 5; i++) {
        insertbeginning_withdummy(&amp;amp;dummy_head, numbs[i]); // pass pt
        printlist(dummy_head.next);
    }
    for (i = 0; i &amp;lt; 5; i++) {
        insertend_withdummy(&amp;amp;dummy_head, numbs[i]);
        printlist(dummy_head.next);
    }

    return 0;
}
```

```
&amp;gt;  gcc linkedlist.c  -g
&amp;gt; ./a.out
1,
2, 1,
3, 2, 1,
4, 3, 2, 1,
5, 4, 3, 2, 1,
5, 4, 3, 2, 1, 1,
5, 4, 3, 2, 1, 1, 2,
5, 4, 3, 2, 1, 1, 2, 3,
5, 4, 3, 2, 1, 1, 2, 3, 4,
5, 4, 3, 2, 1, 1, 2, 3, 4, 5,
1,
2, 1,
3, 2, 1,
4, 3, 2, 1,
5, 4, 3, 2, 1,
5, 4, 3, 2, 1, 1,
5, 4, 3, 2, 1, 1, 2,
5, 4, 3, 2, 1, 1, 2, 3,
5, 4, 3, 2, 1, 1, 2, 3, 4,
5, 4, 3, 2, 1, 1, 2, 3, 4, 5,
```

Function to check if singly linked list is palindrome

This problem comes from the 'Cracking the coding interview' book. It uses the slow/fast approach to traverse a linked list, and a stack to check if list is palindrome.

```
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

struct node {
    struct node *next;
    int key;
};
typedef struct node node_t;

void insertendlist(node_t *head, int key) {
    node_t *newnode = malloc(sizeof(node_t));
    newnode->key = key;
    newnode->next = NULL;

    node_t *ptr = head;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = newnode;
}

void printlist(node_t *head) {
    node_t *ptr = head->next;
    while (ptr != NULL) {
        printf("%d, ", ptr->key);
        ptr = ptr->next;
    }
    printf("\n");
}

int islistpalindrome(node_t *head) {
```

```
    int stack[20];
    int index = 0;
    node_t *fast = head->next;  // fast goes two nodes at a time
    node_t *slow = head->next;  // when fast at end, slow at middle
    // load stack with half list
    while (fast != NULL && fast->next != NULL) {
        stack[index++] = slow->key;
        fast = fast->next->next;
        slow = slow->next;
    }
    // check second half of link lised
    while (slow != NULL) {
        if (stack[--index] != slow->key)
            return FALSE;
        slow = slow->next;
    }
    return TRUE;
}

int main() {
    int i;

    node_t list1;
    list1.next = NULL;
    int nopalindrome[] = {2, 6, 3, 7, 5, 6};
    for (i = 0; i < 6; i++)
        insertendlist(&list1, nopalindrome[i]);
    printlist(&list1);
    if (islistpalindrome(&list1))
        printf("List is palindrome\n");
    else
        printf("List is not palindrome\n");



    node_t list2;
    list2.next = NULL;
    int palindrome[] = {2, 6, 4, 4, 6, 2};
    for (i = 0; i < 6; i++)
        insertendlist(&list2, palindrome[i]);
    printlist(&list2);
    if (islistpalindrome(&list2))
        printf("List is palindrome\n");
    else
        printf("List is not palindrome\n");
```

```
    return 0;
}
```

```
gcc llpalindrome.c -g
./a.out
2, 6, 3, 7, 5, 6,
List is not palindrome
2, 6, 4, 4, 6, 2,
List is palindrome
```

Reverse linear linked list

Function to reverse a single linear linked list

```
struct node {
        struct node *next;
        void *data;
};
void reverse(struct node *head) {
        // if list empty or one element, nothing to reverse
        if (head->next == NULL || head->next->next == NULL)
                return;
        struct node *temp;
        struct node *current = head->next->next;
        struct node *prev = head->next;
        prev->next = NULL; // first (after head) becomes last
        while (current != NULL) {
                temp = current->next;
                current->next = prev; // reverse list
                prev = current;
                current = temp;
        }
        head->next = prev; // last becomes first
}
```

Multhreading and pthreads API cheatsheet

Pthread creation:

- pthread_t threads[N]
- pthread_create(&threads[i], NULL, start_routine, void *args)
- pthread_join(threads[i])

Mutex:

- pthread_mutex_t mutex;
- pthread_mutex_init(&mutex);
- pthread_mutex_lock(&mutex);
- pthread_mutex_unlock(&mutex);
- pthread_mutex_destroy(&mutex);

Semaphore:

- sem_t sem;
- sem_init(&sem, 0, initial) -> initial = 0: lock, initial > 0: unlocked
- sem_wait(&sem) -> sem = 0: wait, sem > 0 decrement and go
- sem_post(&sem) -> increment value
- sem_destroy(&sem)

Condition variable:

- pthread_cond_t cond
- pthread_cond_init(&cond)
- pthread_cond_wait(&cond, &mutex) -> unlock mutex and wait on cond
- pthread_cond_signal(&cond) -> wake up threads waiting on cond
- pthread_cond_destroy(&cond)

Common condition variable usage:

- pthread_mutex_lock(&mutex);
- while(isnotready()) pthread_cond_wait(&cond, &mutex);
- critical section
- pthread_mutex_unlock(&mutex);
- pthread_cond_signal(&cond2);

Process virtual address space in Linux:

- Text: image of program (instructions), read-only
- Data: static and global variables initialized by programmer

- BSS: static variables uninitialized (initialized to zero)
- Stack: local variables, function calls, and function metadata
- Heap: memory dynamically allocated

reverse linked list

Function to reverse circular single linked list:

```
struct node {
    void * element;
    struct node *next;
};

typedef struct node node_t;

void reverselist(node_t *head) {
    node_t *current = head->next;
    node_t *prev = head;
    node_t *preprev;
    while (current != head) {
        preprev = prev;
        prev = current;
        current = current->next;

        prev->next = preprev;
    }
    head->next = prev;
}
```

Function to reverse circular double linked list:

```
struct node {
    void *element;
    struct node *next;
    struct node *prev;
};
typedef struct node node_t;


void reverse(node_t *head) {
    node_t *ptr = head->next;
    node_t *temp;
    while (ptr != head) {
        temp = ptr->next;

        ptr->next = ptr->prev;
        ptr->prev = temp;

        ptr = temp;
    }
    ptr = head->next;
    head->next = head->prev;
    head->prev = ptr;
}
```

Binary search tree operations

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    struct node *left;
    struct node *right;
    int key;
};
typedef struct node node_t;
```

```
node_t * alloc_node(int key) {
    node_t *node = malloc(sizeof(node_t));
    node->left = NULL;
    node->right = NULL;
    node->key = key;
    return node;
}

node_t * init_tree(int key) {
    node_t *root = alloc_node(key);
    return root;
}

void insert(node_t *root, int key) {
    node_t *newnode = alloc_node(key);
    node_t *ptr = root;
    while (ptr != NULL) {
        if (ptr->key == key) {
            printf("key %d is already in tree\n", key);
            free(ptr);
            return;
        }
        if (ptr->key > key) {
            if (ptr->left == NULL) {
                ptr->left = newnode;
                return;
            }
            ptr = ptr->left;
        }
        else if (ptr->key < key) {
             if (ptr->right == NULL) {
                 ptr->right = newnode;
                 return;
             }
             ptr = ptr->right;
        }
    }
    printf("Failed to insert key %d\n", key);
}

int find_key(node_t *root, int key) {
    node_t *ptr = root;
    while (ptr != NULL) {
        if (ptr->key == key)
            return ptr->key;
        else if (ptr->key > key)
```

```
            ptr = ptr->left;
        else
            ptr = ptr->right;
    }
    printf("cannot find key %d\n", key);
    return 0;
}

void traverse_inorder(node_t *node) {
    if (node == NULL)
        return;
    traverse_inorder(node->left);   // left
    printf("%d, ", node->key);      // current
    traverse_inorder(node->right);  // right
}

void traverse_preorder(node_t *node) {
    if (node == NULL)
        return;
    printf("%d, ", node->key);       // current
    traverse_preorder(node->left);   // left
    traverse_preorder(node->right); // right
}

void traverse_postorder(node_t *node) {
    if (node == NULL)
        return;
    traverse_postorder(node->left);    // left
    traverse_postorder(node->right);   // right
    printf("%d, ", node->key); // current
}

int maxdepth(node_t *node) {
    if (node == NULL)
        return 0;
    int nleft = maxdepth(node->left);
    int nright = maxdepth(node->right);
    if (nleft>nright)
        return 1 + nleft;
    return 1 + nright;
}

int mindepth(node_t *node) {
    if (node == NULL)
        return 0;
```

```
    int nleft = mindepth(node->left);
    int nright = mindepth(node->right);
    if (nleft>nright)
        return 1 + nright;
    return 1 + nleft;
}

int isbalanced(node_t *root) {
    int min = mindepth(root);
    int max = maxdepth(root);
    if (max - min > 1)
        return 0;
    else
        return 1;
}


int main(int argc, int **argv) {
    int i;
    int keys[] = {5, 4, 8, 6, 1, 43, 6};
    node_t *tree = init_tree(3);
    /* inserting keys */
    for (i = 0; i < 7; i++) {
        printf("Inserting key %d\n", keys[i]);
        insert(tree, keys[i]);
    }
    /* finding keys */
    printf("Finding key %d: %d\n", 43, find_key(tree, 43));
    printf("Finding key %d: %d\n", 15, find_key(tree, 15));

    /* traversing tree */
    traverse_inorder(tree); printf("\n");
    traverse_preorder(tree); printf("\n");
    traverse_postorder(tree); printf("\n");

    /* depth */
    printf("Depth of shortest branch: %d\n", mindepth(tree));
    printf("Depth of longest branch: %d\n", maxdepth(tree));
    if (isbalanced(tree) == 0)
        printf("Tree unbalanced\n");
    else
        printf("Tree balanced\n");

    /* check if tree A is subtree of tree B */
    // just put A's keys in array inorder, and same for B
```

```
    // if A is subtree, inorder keys must be subarray of B



   return 0;
}
```