CS1010E Lecture 12
Additional Material
Non-Examinable

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
www.comp.nus.edu.sg/~joxan
cs1010e@comp.nus.edu.sg

Semester II, 2016/2017

Lecture Outline

- Blackjack Simulation
  An interesting example
- A Contacts Database
  Your first database
- Dynamic Memory Allocation
  Flexible use of memory (and the main motivation for pointers)
- Linked Lists
  Premier example of a Dynamic Data Strucure

Blackjack - A Simulation Example

The very basic rules:

- Players vs one “Bank”
  each gets 2 cards, all cards open except one of the Bank
- Objective: get as close to 21 as possible by taking up to 3 more cards;
  “bust” if exceed 21
- Value of card: 2-9 is face value; 10,J,Q,K is 10, ace is 1 OR 11 (once)
- One player at a time, chooses “hit” (one more card) or “sit”
- When player busts, he loses immediately
- Bank plays last; must sit when reaches 17;
  if bust, banks pays remaining players

Blackjack - Main Program

```
#define DECKS 1
#define HAND 5
#define N 52*DECKS
#define TRIALS 999999
#define MONEY 1000000
int deck[N], bank[HAND], me[HAND];
int decktop, banktop, mytop;
int money = MONEY;

main() { // only one player vs Bank
    int i, r, tmp, n;
    double d;
    srand(5335);
    shuffle();
    for (i = 0; i < TRIALS; i++) {
        if (decktop > N - 10) {
            shuffle();
            decktop = 0;
        }
        deal();
        play();
    }
    d = (double) (money - MONEY) / TRIALS;
    printf("Final: money %d percent %g\n", money, d * 100.0);
}
```

Blackjack - The Shuffle and Deal

```
shuffle() {
    int i, r, tmp;
    for (i = 0; i < N; i++) deck[i] = (i % 13) + 1;
    for (i = 0; i < N; i++) {
        r = i + (rand() % (N - i)); // Random remaining position.
        tmp = deck[i]; deck[i] = deck[r]; deck[r] = tmp;
    }
}

deal() {
    me[0] = deck[decktop++];
    bank[0] = deck[decktop++];
    me[1] = deck[decktop++];
    bank[1] = deck[decktop++];
    banktop = mytop = 2;
}
```

Blackjack - Value of a Hand

```
int hardvalue(int hand[], int n) {
    int i, v = 0;
    for (i = 0; i < n; i++) {
    v += (hand[i] > 10 ? 10 : hand[i]);
    if (v > 21) return -1;
    }
    return v;
}

int softvalue(int hand[], int n) {
    int i, v = 0, ace = 0;
    if ((v = hardvalue(hand, n)) == -1) return -1;
    for (i = 0; i < n; i++) {
        if (hand[i] == 1) ace = 1;
        break;
    }
    if (ace && v <= 10) v += 10;
    return v;
}
```

Note: returns -1 if busted

Blackjack - Strategies

```
int hitstrategy1() { // simplest, sit at 17
    int i, j;
    if ((i = softvalue(me, mytop)) >= 17) return 0;
    return 1;
}

int hitstrategy2() { // aggressive if bank has good card
    int i, j;
    if ((i = softvalue(me, mytop)) >= 18 ) return 0;
    if (i <= 13) return 1;
    if (i <= 16 && (j = softvalue(bank, 1)) == 10 || j == 1)
        return 1;
    return 0;
}
```

More interesting strategies are based on *counting*.
That is, hit if certain counts have been seen.
(This is technically illegal.)

Blackjack - Play!

```
play() {
    int i, j;
    for (i = 3; i <= HAND; i++) {
        if (hitstrategy()) me[mytop++] = deck[decktop++];
        else break;
        if ((j = softvalue(me, mytop)) == -1) {
            money--;
            return;
        }
    }
    while ((i = softvalue(bank, banktop)) < 17 && i != -1)
        bank[banktop++] = deck[decktop++];
    if (i == -1) money++;
    else money += (j <= i ? (j < i ? -1 : 0) : 1);
}
```

Sample Runs with Money = $1000000, $1 per play, Trials = 999999:

- Strategy 1: money left $825761 (-17.4%)
- Strategy 2: money left $884855 (-11.5%)

Contacts - A First Database

```
#define MAXCONTACTS 10000
#define MAXNAME 256
#define MAXBUDDIES 200

struct contact {
    char name[MAXNAME];
    int mobile;
    int buddies[MAXBUDDIES];
}   contacts[MAXCONTACTS];

int numcontacts;
```

Notes:

`contacts[i].name[0] == '\0'` indicates that this contact space not used;

`contacts[i].buddy[j] == -1` indicates that this buddy space not used;

Contacts - Example

```
struct contact {
    char name[MAXNAME];
    int mobile;
    int buddies[MAXBUDDIES];
} contacts[MAXCONTACTS] = {
    {"joxan", 9123888, { 2, 4, -1}}, // two buddies "beng" "vijay"
    {"smith", 9124777, {-1, -1, -1}},
    {"beng", 9125666, { 0, -1, -1}}, // buddy "joxan"
    {"ahmad", 9126555, {-1, -1, -1}},
    {"vijay", 9127444, { 0, -1, -1}} // buddy "joxan"};
int numcontacts = 5;
```

After deleting “beng”, database should look like

```
{"joxan", 9123888, { -1, 4, -1}}, // now one buddy "vijay"
{"smith", 9124777, {-1, -1, -1}},
{"\0", 9125666, {0, -1, -1}}, // this entry is now nullified
{"ahmad", 9126555, {-1, -1, -1}},
{"vijay", 9127444, { 0, -1, -1}} // still has buddy "joxan"
```

Contacts - The Front End

```
main() {
    for (;;) action();
}

action() {
    int c;
    printf("Action [a, d, b] ? ");
    while ((c = getchar()) == '\n') ;
    switch (c) {
        case 'a' : addcontact(); break;
        case 'd' : deletecontact(); break;
        case 'b' : addbuddy(); break;
        default: exit(0);
    }
}
```

Contacts - Adding a contact

```
addcontact() {
    char name[MAXNAME];
    int m, i;

    printf("Input new Name and Mobile: ");
    scanf("%s %d", name, &m);
    strcpy(contacts[numcontacts].name, name);
    contacts[numcontacts].mobile = m;
    for (i = 0; i < MAXBUDDIES; i++)
        contacts[numcontacts].buddies[i] = -1;
    numcontacts++;
}

int findname(char *name) {
    int i;
    for (i = 0; i < numcontacts; i++)
        if (strcmp(name, contacts[i].name) == 0) break;
    return i;
}
```

Contacts - Adding a Buddy

```
addbuddy() {
    char name[MAXNAME], buddy[MAXNAME];
    int i, j, k;

    printf("Input Name and new Buddy: ");
    scanf("%s %s", name, buddy);
    i = findname(name);
    for (j = 0; j < MAXBUDDIES; j++)
    if (contacts[i].buddies[j] == -1) break;
    k = findname(buddy);
    contacts[i].buddies[j] = k;
    for (j = 0; j < MAXBUDDIES; j++)
        if (contacts[k].buddies[j] == -1) break;
    contacts[k].buddies[j] = i;
}
```

Contacts - Deleting a Contact

```
deletecontact() {
    int i, j, k, l;
    char name[MAXNAME], m;

    printf("Input Name to Delete: ");
    scanf("%s", name);
    for (i = 0; i < numcontacts; i++)
        if (strcmp(name, contacts[i].name) == 0) break;

    if (i >= numcontacts)
        printf("Error: %s not in contacts\n", name);
    contacts[i].name[0] = '\0';

    // now delete name from buddies
    for (j = 0; j < MAXBUDDIES; j++) {
        k = contacts[i].buddies[j];
        for (l = 0; l < MAXBUDDIES; l++)
            if (contacts[k].buddies[l] == i)
                contacts[k].buddies[l] = -1;
    }
}
```

Contacts - Some Inefficiencies

- *Static size:*
  the `contacts` array is in use all the time;
  the `buddies` array for each contact is also always in use.
  (In the example, `sizeof(struct contact)` is 1060, so 10Mb in total.)

- *Fragmentation:*
  Space allocated to deleted contacts not re-used (but can easily fix)

- *Search*
  is limited to *linear* search

Dynamic Memory Allocation

- Allocate memory during *execution time* as opposed to compile time.
- Importance: when the size of memory needed is known only at execution time.
- The C constructs from `stdlib.h`:

```
void *malloc(size_t m);
void *calloc(size_t n, size_t size);
```

  where size_t is is system dependent, but usually `unsigned int` or `unsigned long`.
  For `malloc`, `m` denotes the number of *bytes* required, and for `calloc`, `n * size` denotes the number of bytes required.
- `calloc` differs from `malloc` essentially because it guarantees that the bytes returned are initialized to zero.
- Returns NULL if memory request cannot be fulfilled.

Dynamic Memory Allocation

- First, use `sizeof` to determine the number of bytes of the data type of interest.
- To request memory for 200 integers:
- ```
  int *p;
  p = (int *) malloc(n * sizeof(int));
  ```
- ```
  int *p;
  p = (int *) calloc(200, sizeof(int));
  ```
- To *free* memory: use the prototype

  ```
  void free(void *ptr);
  ```

  Note that the argument pointer needs to have been previously allocated memory in order to now be freed.

Dynamic Memory Allocation

```
#include <stdlib.h>
#define UNIT 50000000

int main(void) {
    int k = 1, *ptr;
    ptr = (int *) malloc(k * UNIT * sizeof(int));
    while (ptr != NULL) {
        printf("Obtained %d integers so far ...\n", k*UNIT);
        free(ptr);
        k++;
        ptr = (int *) malloc(k * UNIT * sizeof(int));
    }
    printf("\nMaximum contiguous memory available: \n");
    printf("%d integers\n", (k - 1) * UNIT);
}
```

Dynamic Memory Allocation

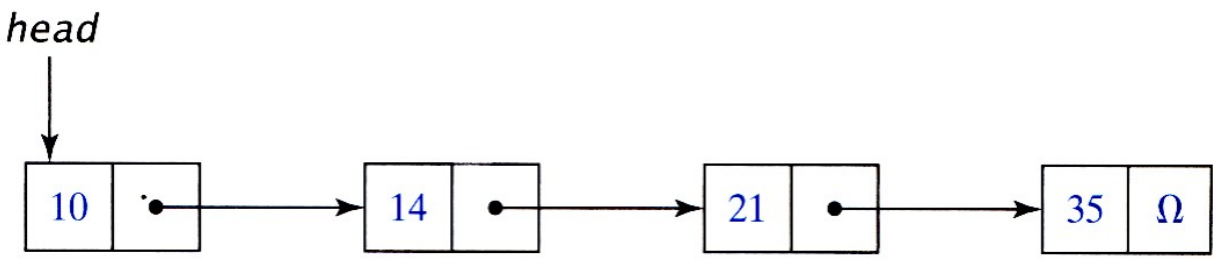
```
OUTPUT:

Obtained 50000000 integers so far ...
Obtained 100000000 integers so far ...
Obtained 150000000 integers so far ...
Obtained 200000000 integers so far ...
Obtained 250000000 integers so far ...
Obtained 300000000 integers so far ...

Maximum contiguous memory available:
300000000 integers
```

Dynamic Data Structures: Linked List

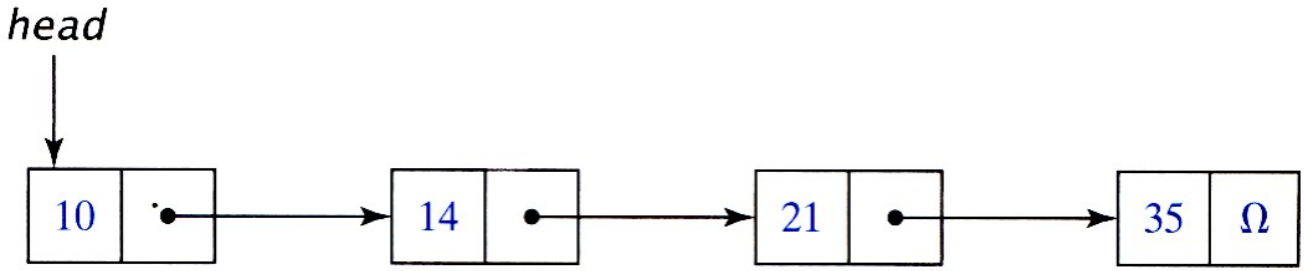
- Data structures that can *grow* and *shrink* during execution.
- Use only memory that is needed
- Linked List: the premier example
- A linked list is a group of *nodes* that are connected by *pointers*.
  A node consists of *data*, and a pointer to the next node.
  Data can be a simple data type (eg. integer) or a more complex type
  (such as defined by a structure).



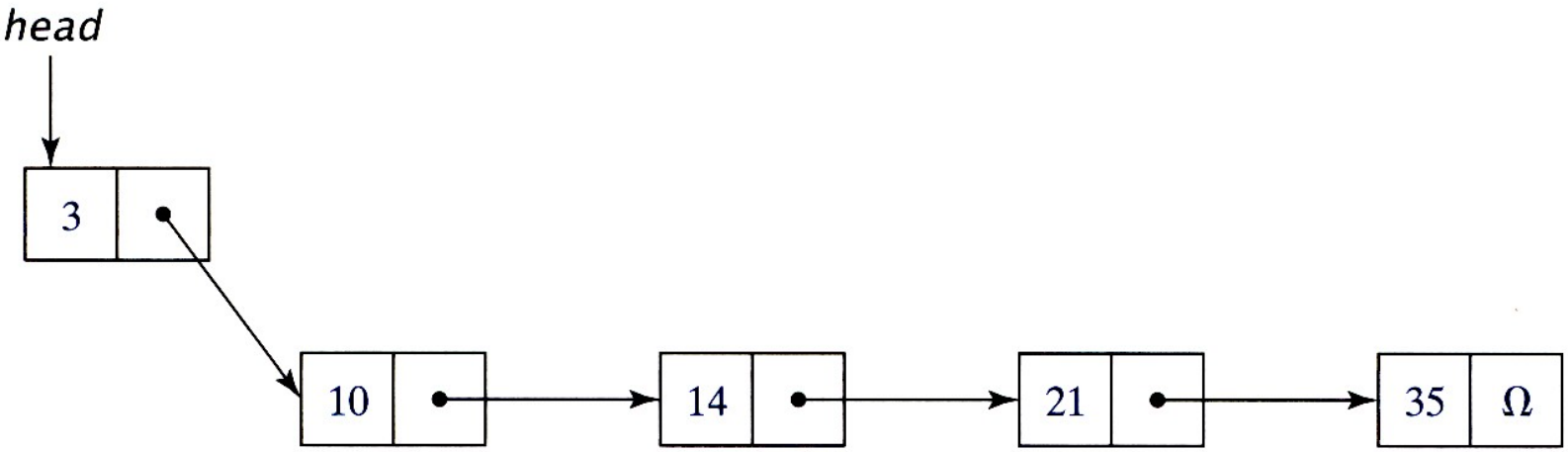
- A linked list serves to store, in some order, a collection of data.
  What we consider next is various operations on a list.

Insertions to a Linked List

Original List:



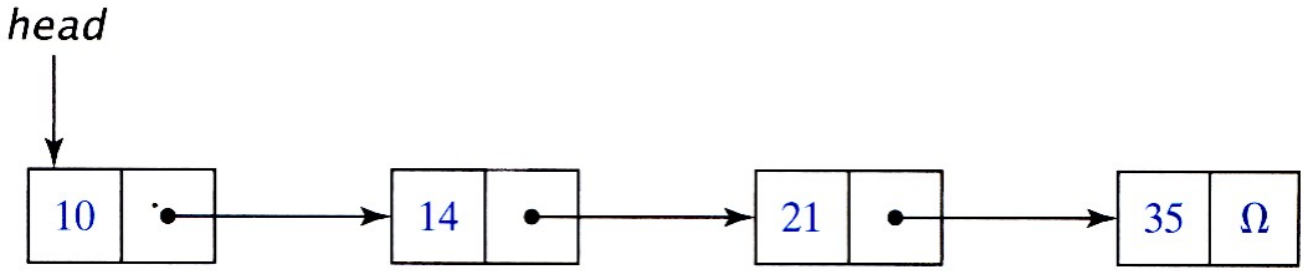
Insert a new data element 3 into the *front* of the list:



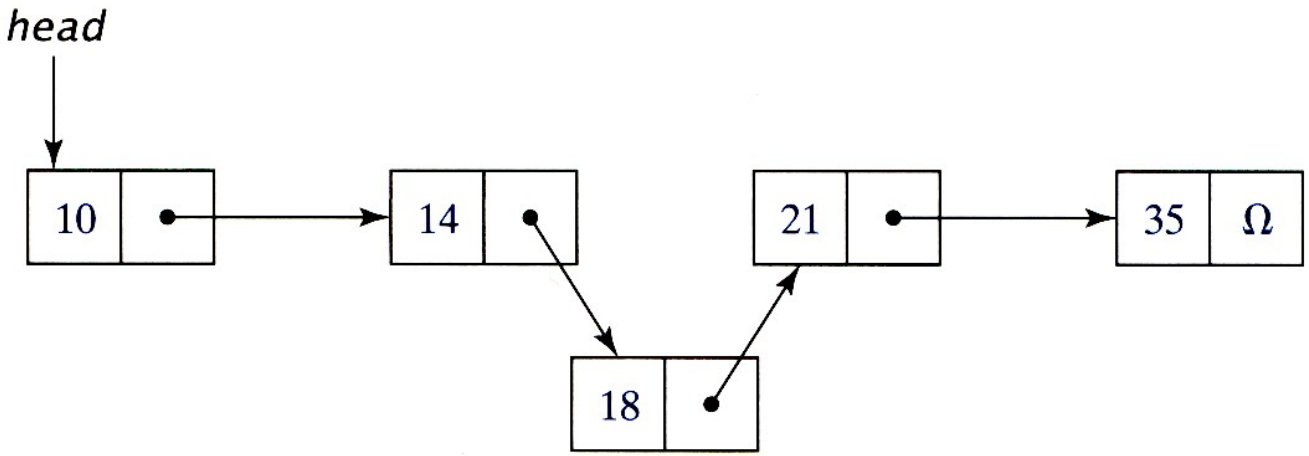
Note that the pointer `head` has been modified.

Insertions to a Linked List

Original List:



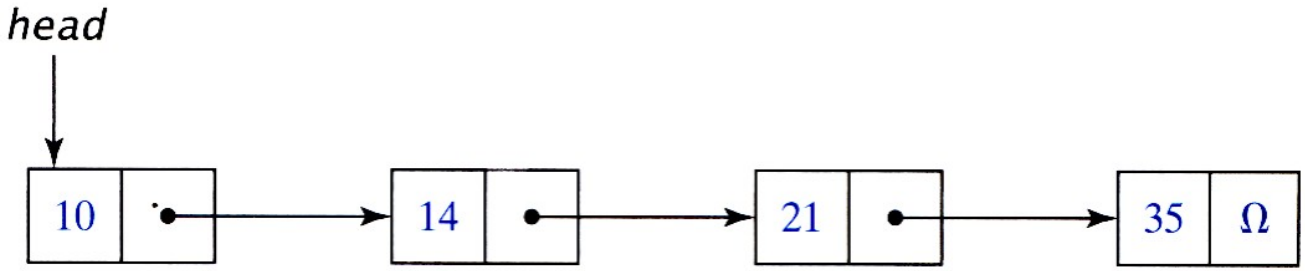
Insert a new data element 18 *in between* two nodes of the list:



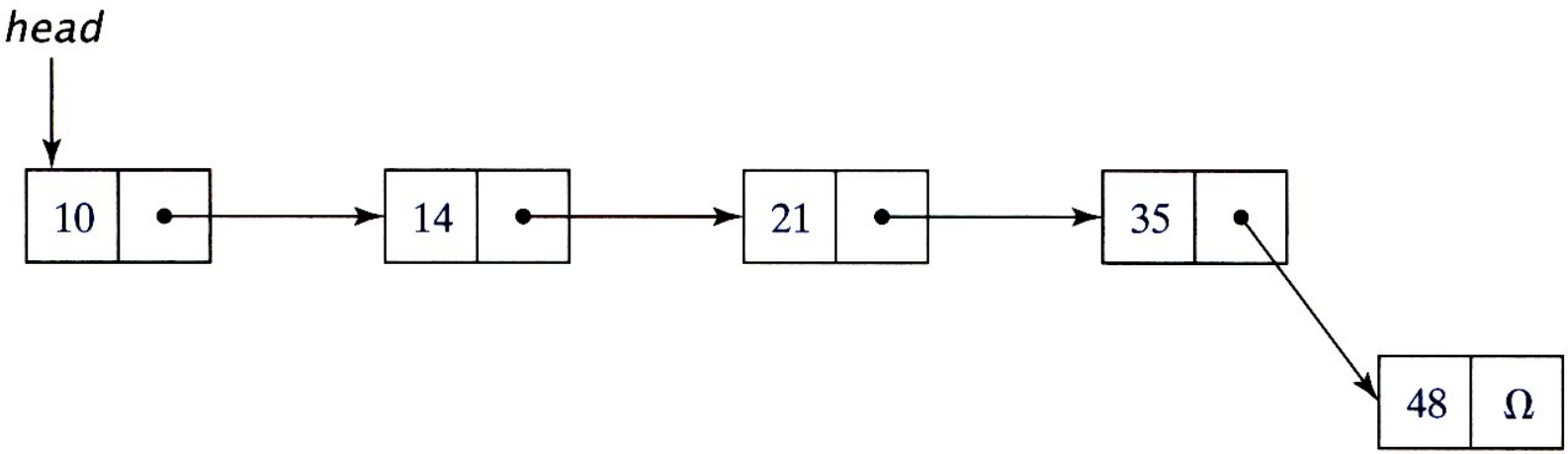
Note that the pointer `head` has not been modified.

Insertions to a Linked List

Original List:



Insert a new data element 48 *at the end* of the list:



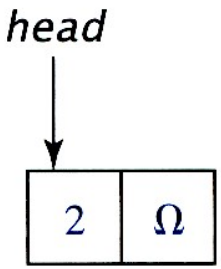
Note that the pointer `head` has not been modified.

Insertions to a Linked List

Original List:

```
Head = NULL
```

Insert a new data element 2:



Note that the pointer `head` has been modified.

Empty List?

```
struct node {
   int data;
   struct node *link;
};
int empty(struct node *head);
```

```
/*-------------------------------------------------------------*/
/*  This function returns a value of one if the linked list    */
/*  is empty.                                                  */

int empty(struct node *head)
{
   /*  Declare variables.  */
   int k=0;

   /*  Determine if the list is empty.  */
   if (head == NULL)
      k = 1;

   /*  Return integer.  */
   return k;
}
/*-------------------------------------------------------------*/
```

Print a Linked List

```
/*-------------------------------------------------------------*/
/*  This function prints a linked list.                        */

void print_list(struct node *head)
{
   /*  Declare variables.  */
   struct node *next;

   /*  Print linked list.  */
   if (empty(head))
      printf("Empty list \n")
   else
   {
      printf("List Values: \n");
      next = head;
      while (next->link != NULL)
      {
         printf("%d \n",next->data);
         next = next->link;
      }
      printf("%d \n",next->data);
   }

   /*  Void return.  */
   return;
}
/*-------------------------------------------------------------*/
```

Linked List - Main Program

```
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int empty(struct node *head);
void print_list(struct node *head);
void insert(struct node **ptr_to_head, struct node *nw);
void remove(struct node **ptr_to_head, int old);
```

Linked List - Main Program

```
int main(void) {
    int k = 0, old, value;
    struct node *head, *next, *previous, *nw, **ptr_to_head = &head;
    head = (struct node *) malloc(sizeof(struct node));
    getlist(head);
    print_list(head);
    while (k != 2) {
        printf("Enter 0 to delete node, 1 to add node, 2 to quit.\n");
        scanf("%d", &k);
        if (k == 0) {
            printf("Enter data value to delete:\n");
            scanf("%d", &old);
            remove(ptr_to_head, old);
            print_list(head);
         } else if (k == 1) {
            printf("Enter data value to add:\n");
            scanf("%d", &value);
            nw = (struct node *) malloc(sizeof(struct node));
            nw->data = value;
            nw-> link = NULL;
            insert(ptr_to_head, nw);
            print_list(head);
        }
    }
}
```

Linked List - Create an Initial List

```
void getlist(struct node *head) {
    struct node *next, *previous;
    int k;
    next = head;
    for (k = 1; k <= N; k++) {
        next->data = k * 5;
        next->link = (struct node *)
            malloc(sizeof(struct node));
        previous = next;
        next = next-> link;
    }
    previous->link = NULL;
}
```

Linked List - Insert

```
void insert(struct node **ptr_to_head, struct node *nw) {
    struct node **next;
    if (empty(*ptr_to_head)) *ptr_to_head = nw;
    else {
        next = ptr_to_head;
        while (((*next)->data < nw->data) &&
               ((*next)->link != NULL))
            next = &(*next)->link;
        if ((*next)->data == nw->data)
            printf("Node already in list.\n");
        else if ((*next)->data < nw->data)
            (*next)->link = nw;
        else {
            nw->link = *next;
            *next = nw;
        }
    }
}
```

Linked List - Delete

```
void remove(struct node **ptr_to_head, int old) {
    struct node *next, *last, *hold, *head;
    head = *ptr_to_head;
    if (empty(head)) printf("Empty List.\n");
    else {
        if (head->data == old) {
            hold = head; *ptr_to_head = head->link; free(hold);
        } else {
            next = head->link;
            last = head;
            while ((next->data < old) && (next->link != NULL)) {
                last = next;
                next = next->link;
            }
            if (next->data == old) {
                hold = last;
                last->link = next->link;
                free(next);
            } else printf("Value %d not in list.\n", old);
        }
    }
}
```

Linked List - Sample Run

```
5 10 15 20 25
Enter 0 to delete node, 1 to add node, 2 to quit.
0
Enter data value to delete:
20
Found 20, last->data is 15
5 10 15 25
Enter 0 to delete node, 1 to add node, 2 to quit.
1
Enter data value to add:
555
5 10 15 25 555
Enter 0 to delete node, 1 to add node, 2 to quit.
0
Enter data value to delete:
10
Found 10, last->data is 5
5 15 25 555
Enter 0 to delete node, 1 to add node, 2 to quit.
1
Enter data value to add:
333
5 15 25 333 555
Enter 0 to delete node, 1 to add node, 2 to quit.
2
```

That’s all Folks !