Linked lists

- Much like Java's objects with references to objects of that class, C structures can have pointers to structures of the same type

```c
typedef struct node {
    int val;
    struct node *next;
} Node;
```

- Both `next` and `head` are of the same type
**Linked list in memory**

| head | 5     | 8     | 12    | 17     | NULL |

**Searching through a linked list**

```c
Node *find(Node *start, int target) {
    while (start != NULL) {
        if (start->val == target)
            return start;
        start = start->next;
    }
    return NULL;
}

int find(Node *start, int target) {
    while (start != NULL && start->val != target)
        start = start->next;
    return start != NULL;
}
```

**Inserting into an ordered linked list**

| head | 3     | 5     | 8     | 12     | NULL |

**Tracing insertion**

```c
int insert(Node *head, int new_value) {
    Node *current = head, *prev = NULL, *new_item;
    while (current != NULL && current->val < new_value) {
        prev = current;
        current = current->next;
    }
    new_item = malloc(sizeof(*new_item));
    if (new_item == NULL)
        return 0;
    new_item->val = new_value;
    new_item->next = current;
    if (prev == NULL)
        head = new_item;
    else
        prev->next = new_item;
    return 1;
}
```
Tracing insertion - fixed

```c
int insert(Node **head, int new_value) {
    Node *current = *head, *prev = NULL, *new_item;
    while (current != NULL && current->val < new_value) {
        prev = current;
        current = current->next;
    }
    new_item = malloc(sizeof(*new_item));
    if (new_item == NULL)
        return 0;
    new_item->val = new_value;
    new_item->next = current;
    if (prev == NULL)
        *head = new_item;
    else
        prev->next = new_item;
    return 1;
}
```

Tracing deletion

```c
int delete(Node **head, int value) {
    Node *prev = NULL, *current = *head;
    while (current != NULL && current->data != value) {
        prev = current;
        current = current->next;
    }
    if (current == NULL)
        return 0;  /* not found */
    if (prev != NULL)
        prev->next = current->next;
    else
        *head = current->next;  /* deleted first item */
    free(current);
    return 1;
}
```

Deleting from a linked list

Doubly linked lists

- We can have references in both directions
- Allows use of only one pointer when performing some operations

```c
typedef struct dl_node {
    struct dl_node *prev;
    struct dl_node *next;
    int val;
} DL_node;
```
Deletion from a doubly linked list

```c
int delete(DL_node **root, int target) {
    DL_node *current;
    for (current = *root; current != NULL; current = current->next)
        if (current->val == target)
            break;
    if (current == NULL)
        return 0;
    if (current == *root)
        *root = current->next;
    else
        current->prev->next = current->next;
    if (current->next != NULL)
        current->next->prev = current->prev;
    free(current);
    return 1;
}
```

Assembly language

- The CPU uses machine language to perform all its operations.
- Assembly is a much more readable translation of machine language, and it is what we work with if we need to see what the computer is doing.
- Many different kinds of assembly languages; we'll focus on the Y86 language defined in the text (with minor modifications).

An example of Y86 assembly

- The summation of all integers from 1 to 1000:
  ```assembly
  irmovl $0,%eax    # sum = 0
  irmovl $1,%ecx    # num = 1
  Loop: addl %ecx,%eax    # sum += num
  irmovl $1,%edx    # tmp = 1
  addl %edx,%ecx    # num++
  irmovl $1000,%edx  # lim = 1000
  subl %ecx,%edx    # if lim - num >= 0
  jge Loop          # loop again
  wrint %eax         # printf("%d", sum)
  irmovl $10,%edx    # ch = 'n'
  wrch %edx          # printf("\n", ch)
  halt
  ```
Assembly language concepts

• Registers
  – Fast-access locations that the CPU uses, rather than storing all variables in memory
  – In the code we'll be examining, these are named by three-letter codes, and begin with `%
  – Y86 has 8 registers, listed on pg. 168 in Figure 3.2

• Memory
  – Program text is here, along with any data which cannot fit in the registers
  – Since there are usually many different pieces of data involved in a program, and a limited number of registers, program variables are stored in memory when not in immediate use

Assembly language instructions

• These typically have an instruction name, a list of registers, and sometimes have a constant value too

• Examples:
  - `irmovl $1000,%edx`
  - `subl %ecx,%edx`
  - `jge Loop`

• Specify which operation to perform, which registers to act upon, and any values that might be needed

Assembling assembly

• Machine code (pure numbers) is generated by translating each instruction into binary numbers that the CPU uses
• This process is called "assembling"; conversely, we can take assembled code and disassemble it into (mostly) human readable assembly language

Labels

• When the assembler encounters a label (e.g., `Loop:`), the address of the labeled instruction is used by the assembler to replace all references to that label in the program
• Labels do not need to be declared before use
• In Y86, instructions have variable lengths, and are not aligned; most take 6 bytes, but some use fewer

Label translation

```
irmovl $0, %eax
irmovl $1, %ecx
Loop:  addl  %ecx, %eax
irmovl $1, %edx
addl  %edx, %ecx
irmovl $1000, %edx
subl  %ecx, %edx
jge   Loop
wrint %eax
irmovl $10, %edx
wrch  %edx
halt
```

```
0x00:  irmovl $0, %eax
0x06:  irmovl $1, %ecx
0x0c:  addl  %ecx, %eax
0x0e:  irmovl $1, %edx
0x14:  addl  %edx, %ecx
0x16:  irmovl $1000, %edx
0x1c:  subl  %ecx, %edx
0x1e:  jge   Loop
0x23:  wrint %eax
0x25:  irmovl $10, %edx
0x2b:  wrch  %edx
0x2d:  halt
```