Today’s Goals

- Self-referential Structures
- Linked Lists
  - single
  - double
  - general purpose
  - memory issues

Self-referential Structures

- A basic data type (building block) for complex data structures such as trees and linked lists.
- Structure tags (i.e. `tnode`, `lnode`) are required for self-referential structure declarations.

```c
typedef struct lnode {
    int x;
    struct lnode *next;
} Listnode;
```

Linked Lists

- A linked list stores a list of items (`struct`s).
- Linked lists are typically unbounded, that is, they can grow infinitely.
- An array is a single consecutive piece of memory, a linked list is made of many pieces.
- A linked list offers quick insertion, deletion and reordering of the items

```c
typedef struct tnode {
    int x;
    struct tnode *left;
    struct tnode *right;
} Treenode;
```

Singly and Doubly Linked Lists

- A singly linked list has each `struct` containing only one pointer to the next.
- A doubly linked list has each `struct` containing both a pointer to the previous as well as the next `struct` in the list.

```c
Node *makenode (int x) {
    Node *new;
    if ( (new = (Node *) malloc( sizeof(Node) ) ) != NULL) {
        new->num = x;
        new->next = NULL;
    } else {
        printf("Out of memory!
");
        exit(0);
    }
    return new;
}
```
append

```c
void append (Node *p) {
    if (head == NULL) {
        head = p;
        tail = p;
    } else {
        tail->next = p;
        tail = p;
    }
}
```

delete

```c
void delete (Node *p) {
    Node *tmp, *prev;
    if ((p == head) && (p == tail))
        head = tail = NULL;
    else if (p == head)
        head = p->next;
    else {
        for(tmp=head, prev=NULL; tmp!=p; prev=tmp, tmp=tmp->next);
        if (p == tail)
            tail = prev;
        prev->next = p->next;
    }
}
```

insert_after

```c
/* insert a node p after p2 */
void insert_after (Node *p, Node *p2) {
    p->next = p2->next;
    if (p2 == tail)
        tail = p;
    p2->next = p;
}
```

print/search

```c
void print () {
    Node *tmp;
    for (tmp = head; tmp != NULL; tmp = tmp->next)
        printf("%d ", tmp->num);
    printf("\n");
}

Node *search (int x) {
    Node *tmp;
    for (tmp = head; tmp != NULL; tmp = tmp->next)
        if (tmp->num == x)
            return tmp;
    return NULL;
}
```

main

```c
int main () {
    Node *tmp, *tmp2;
    for (i = 0; i < 10; i++) {
        tmp = makenode (i);
        append (tmp);
        print ();
        tmp2 = makenode (9);
        insert_after (tmp, head->next->next);
        delete (head->next);
        print ();
    }
}
```

clear

```c
void clear () {
    Node *tmp, *tmp2;
    for (tmp = head; tmp != NULL; tmp = tmp2) {
        tmp2 = tmp->next;
        free (tmp);
    }
    head = tail = NULL;
}
```

• Note that this only works if structure Node does not contain any other pointers to memory
General Purpose Linked Lists

- **void** *
  - Generic pointer – just a memory address
  - Can be casted to any type

```c
struct llist_node {
    void *data;
    struct llist_node *prev;
    struct llist_node *next;
};
typedef struct llist_node Lnode;
```

Linked List in Java

- In Java, a linked list and the nodes in the list are of different types.
- This can be simulated in C.
- However, there are advantages to keeping the same type.

```java
class ListNode {
    int item;
    ListNode next;
    ...
};
class List {
    ListNode head;
    ...
}
```

Avoid Memory Leaks Like Plagues

- Whenever dynamically allocated storages are in use, memory leaks are plentiful
- The problem is more evident when complicated data structures are used
  - Mixing: list of trees, trees of lists, etc
  - Nesting: list of lists of lists
- When implementing complex data structures, plan your `clear/release` functions very carefully

Shallow and Deep Copying

- There are two ways to make a copy of a linked list
- Shallow copy:
  - The new list consist of duplicated pointers only
  - ![Shallow Copy Diagram]

- Deep copy:
  - The new list consist of duplicated data as well as pointers
  - ![Deep Copy Diagram]

Summary

- Linked lists are the most commonly used data structure in programming
- Learning how to implement a proper linked list is essential
- Watch out for memory leaks!
- Explore general purpose linked lists