Today's Goals

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

Section 1

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.

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

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

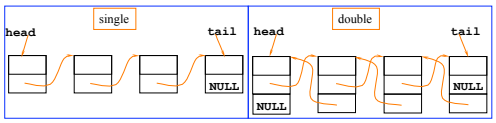

Section 2

Linked Lists

- A linked list stores a lists 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

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.



`struct node`

singlelist.c

```
struct node {
  int num;
  struct node *next;
};
typedef struct node Node;
// typedef Node *Nodeptr;
Node *head = NULL;
Node *tail = NULL;
```


`makenode`

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

append

```
void append (Node *p) {

  if (head == NULL) {
    head = p;
    tail = p;
  }
  else {
    tail->next = p;
    tail = p;
  }
}
```


delete

```
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

```
/* 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

```
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

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


clear

- Note that this only works if structure **Node** does not contain any other pointers to memory

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

Section 3

General Purpose Linked Lists

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

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


General Purpose **`makenode`**

```
Lnode *makenode (void *data) {
  Lnode *new = NULL;
  if ((new = (Lnode *) malloc(Lnode)) != NULL) {
    new->prev = NULL;
    new->next = NULL;
    new->data = data;
  }
  return new;
 }
```


Section 4

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.

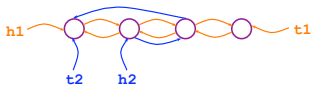
```
class ListNode{
  int item;
  ListNode next;
  ...
};
class List {
  ListNode head;
  ...
}
```


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



- Deep copy:
  - The new list consist of duplicated data as well as pointers

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