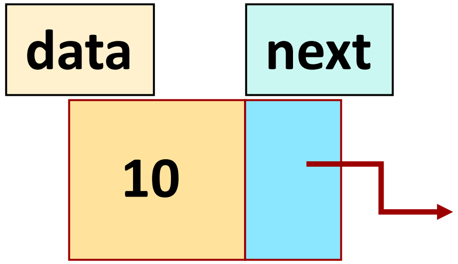
Using linked lists

ESC101: Fundamentals of Computing

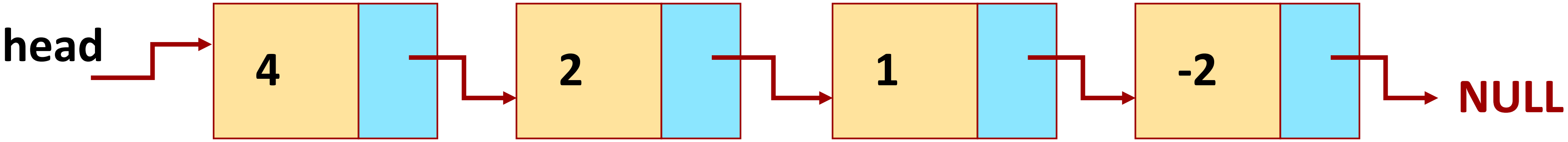
Nisheeth

Linked List

```
struct node {
    int data;
    struct node *next;
};
```



struct node



Use of typedef

Define a new type Listnode as struct node *

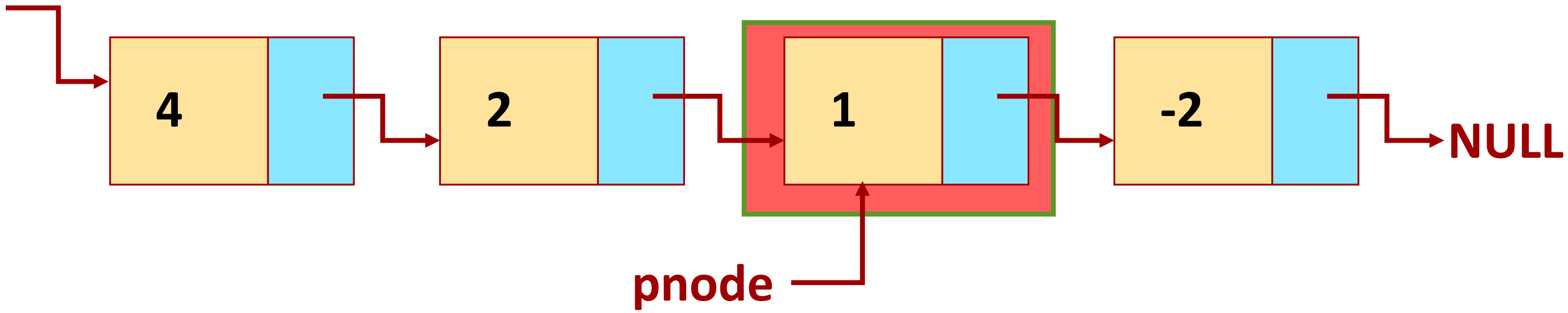
```
typedef struct node * Listnode;
```

Listnode is a type. It can be used for struct node * in variables, argument, return type, etc..

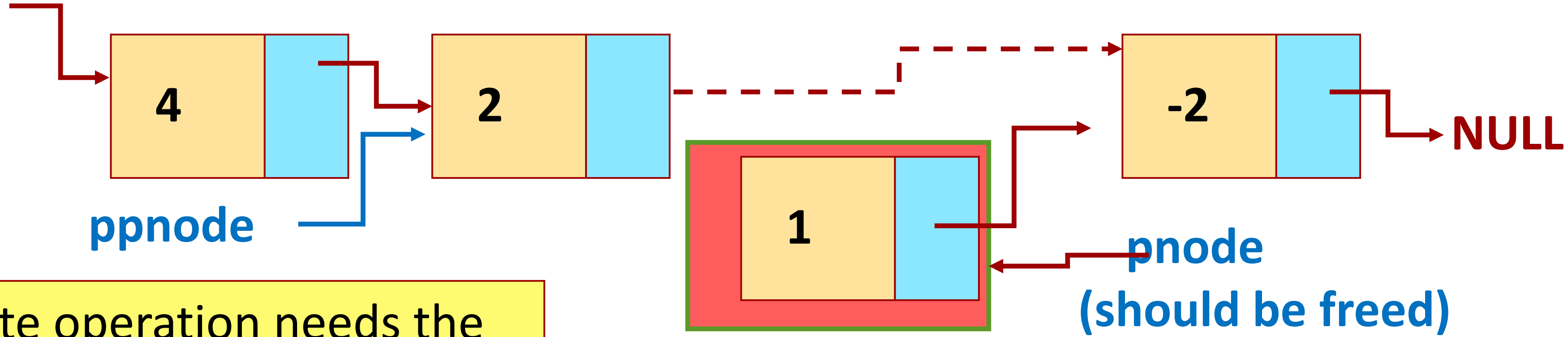
```
Listnode head, curr;
 /* search in list for key */
Listnode search(Listnode list, int key);
/* insert the listnode n in front of listnode list */
Listnode insert_front(Listnode list, Listnode n);
 /* insert the listnode n after the listnode curr */
Listnode insert_after(Listnode curr, Listnode n);
```

Deletion in linked list

Given a pointer pnode. How do we delete the node pointed by pnode?



After deletion, we want the following state



Delete operation needs the pointer to previous node to pnode to adjust pointers.

call free() to release storage for deleted node.

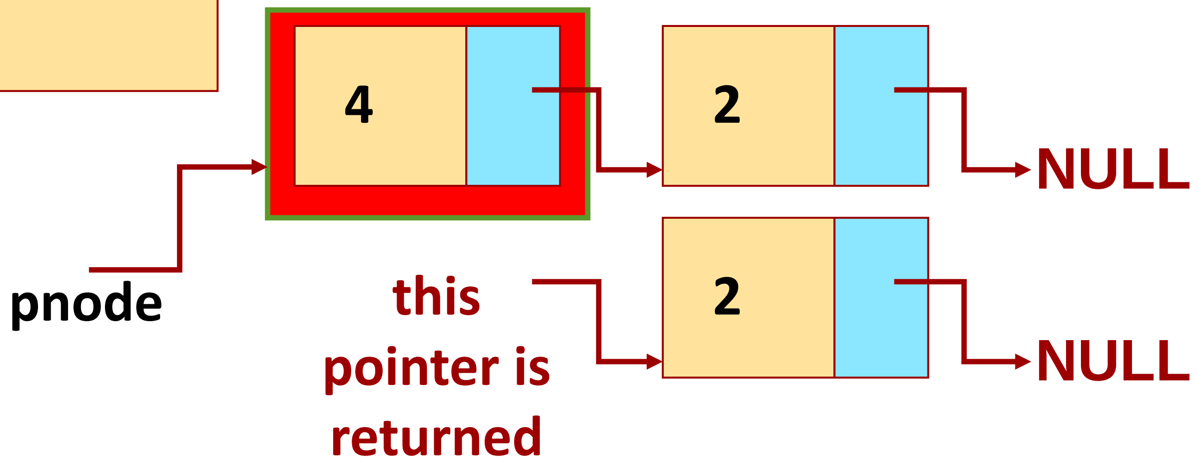
delete(Listnode pnode, Listnode ppnode);

```
Listnode delete(Listnode pnode, Listnode ppnode) {
   Listnode t;
   if (ppnode)
      ppnode->next = pnode->next;
   t = ppnode ? ppnode : pnode->next;
   free (pnode);
   return t;
}
```

Delete the node pointed by pnode. ppnode: pointer to the node before pnode, if it exists; otherwise NULL.

Function returns ppnode if it is non-null, else returns the successor of pnode.

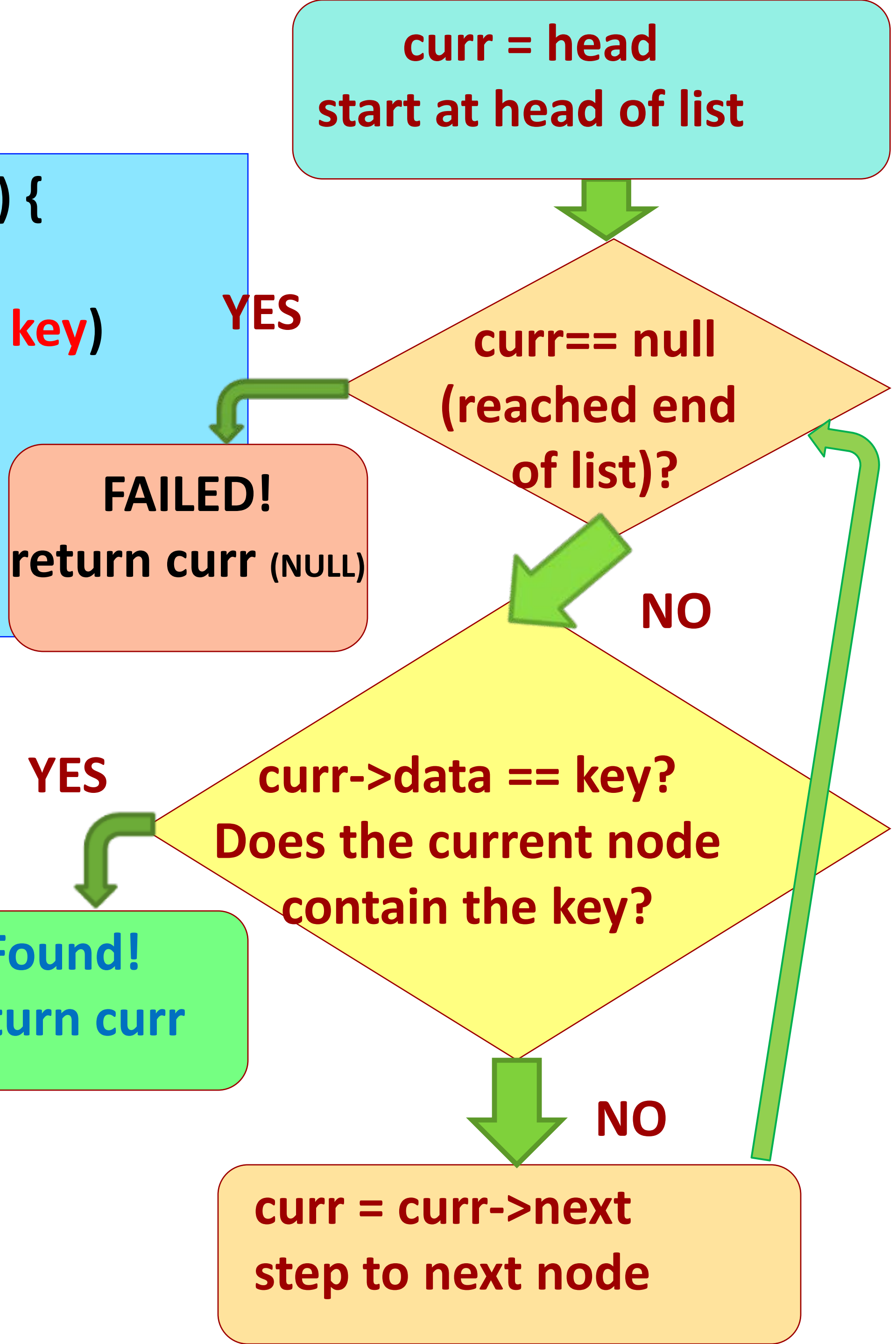
The case when pnode is the head of a list. Then ppnode == NULL.



Searching in LL

```
Listnode search(Listnode head, int key) {
  Listnode curr = head;
  while  (curr != NULL && curr->data != key)
      curr = curr->next;

   return curr;
}
```



search for key in a list pointed to by head.
Return pointer to the node found or else return NULL.

Disadvantage:
Sequential access only.

Linked List: A useful application

- Customer information can be defined using a struct

```
struct cust_info {
    int Account_Number;
    int Account_Type;
    char *Customer_Name;
    char* Customer_Address;
    bitmap Signature_scan; // user defined type bitmap
} ;
```

- A customer can have more than 1 accounts
  – Want to keep multiple accounts for a customer together for easy access

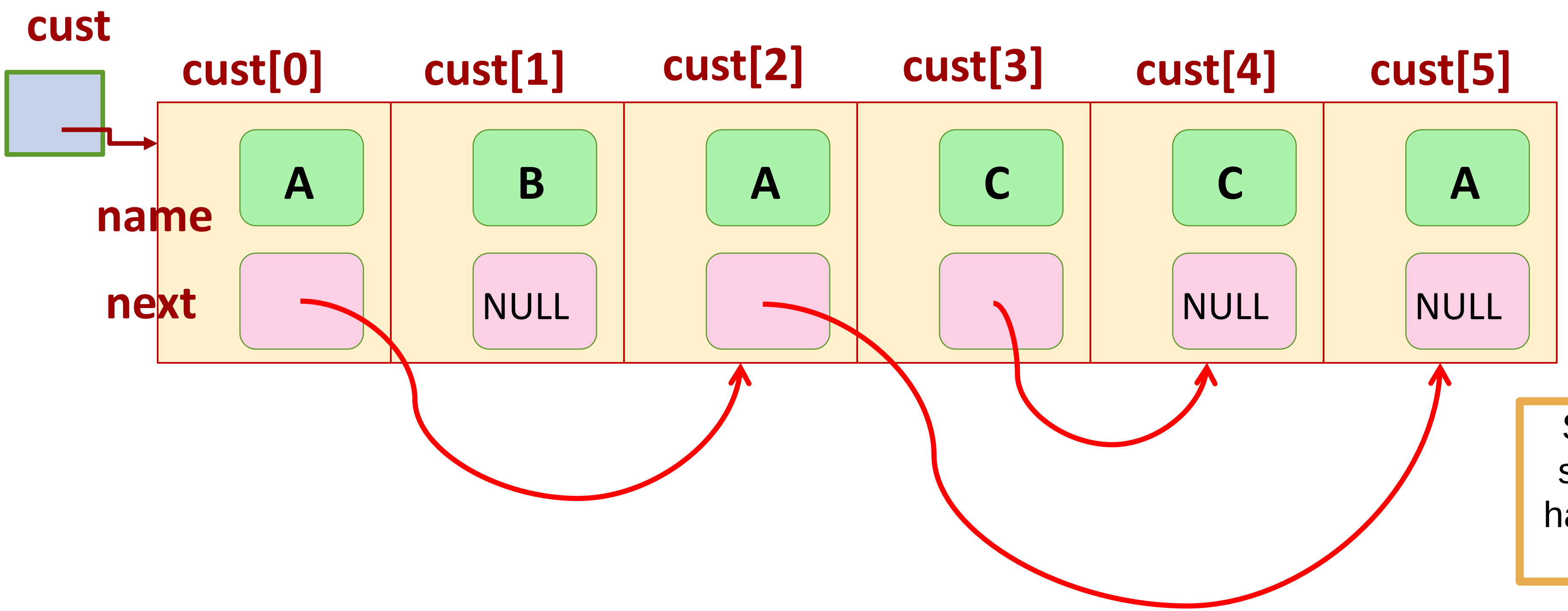
Linked List: A useful application

- "Link" all the customer accounts together using a "chain-of-pointers"

```
struct cust_info {
    int Account_Number;
    int Account_Type;
    char *Customer_Name;
    char* Customer_Address;
    bitmap Signature_scan; // user defined type bitmap
    struct cust_info* next_account;
 } ;
```

- So each customer can be defined by a linked list (and each such linked lists can have one or more nodes)

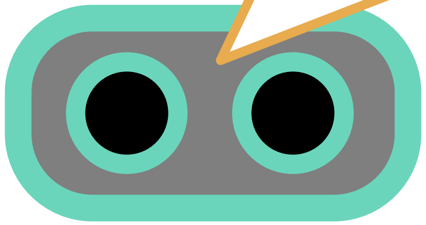
Linked List: A useful application



Some lists have a single node, some have more than one node

Can think of this as an array of singly linked lists

cust[i].next, cust[i].next->next, cust[i].next->next->next etc., when **not NULL**, points to the "other" records of the same customer



Reminder: Why linked lists, not arrays?

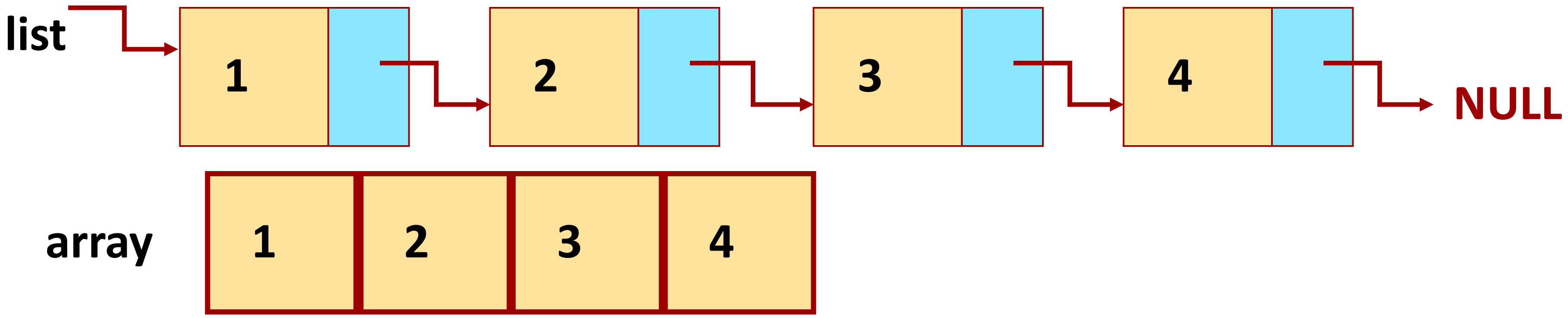
➢ **A list of things can be represented in an array. So, where is the advantage with linked list?**

1. **Insertion and deletion are inexpensive, only a few "pointer changes".**
2. **To insert an element at position k in array:**
   create space in position k by shifting elements in positions k or higher one to the right.
3. **To delete element in position k in array:**
   compact array by shifting elements in positions k or higher one to the left.

Disadvantages of Linked List

➢ **Direct access to kth position in a list is expensive (time proportional to k) but is fast in arrays (constant time).**

Linked Lists: the pros and the cons



Operation	Singly Linked List	Arrays
Arbitrary Searching.	sequential search (linear-time)	sequential search (linear-time)
Searching in a sorted structure.	Still sequential search. Cannot take advantage.	Binary search possible (logarithmic-time)
Insert key after a given point in structure.	Very quick (constant-time)	Shift all array elements at insertion index and later one position to right. Make room, then insert. (linear-time)

Will see later

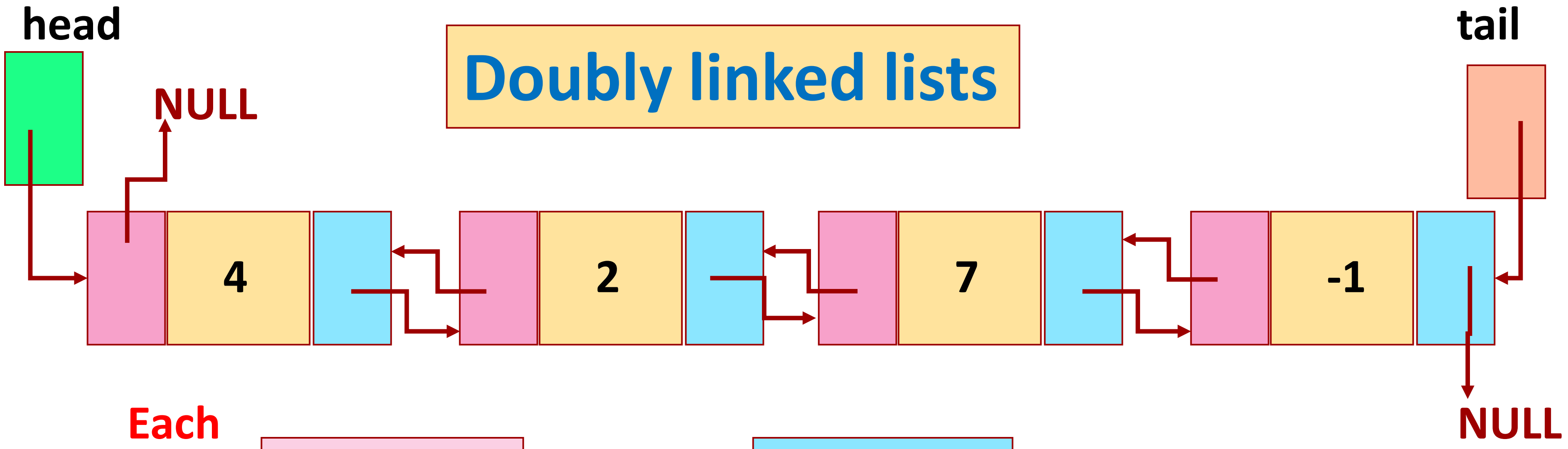
Singly Linked Lists

Operations on a linked list. For each operation, we are *given a pointer to a current node* in the list.

Operation	Singly Linked List
Find next node	Follow next field
Find previous node	Can't do !!
Insert before a node	Can't do !!
Insert in front	Easy, since there is a pointer to head.

Principal Inadequacy: Navigation is one-way only from a node to the next node.

Doubly linked lists



Each node has 3 fields

(i) pointer to previous node

(ii) data

(iii) pointer to next node

Defining *node* of Doubly linked list and the *Dllist* itself.

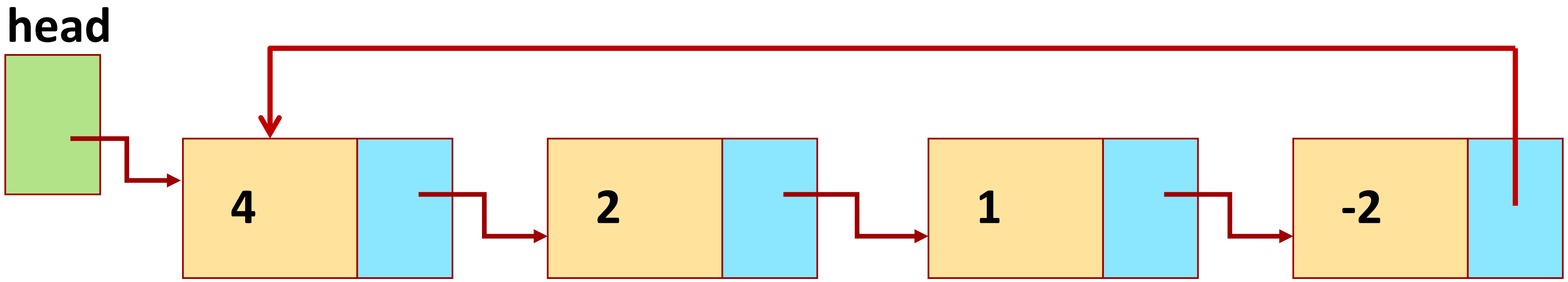
```
struct dlnode {
     int data;
     struct dlnode *next;
     struct dlnode *prev;
};
typedef struct dlnode *Ndptr;
```

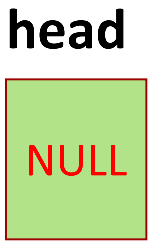
```
struct dlList {
   Ndptr head; /*ptr to first node */
   Ndptr tail;   /* ptr to last node */
};
typedef struct dlList *DlList;
```

Circular Linked List

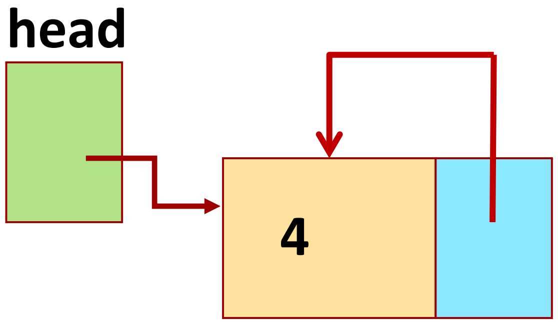
So far, we were modeling a singly linked list by a pointer to the first node of the list.
Let us make the following change:

Make the list circular: next pointer of last node is not NULL, it points to the head node.





An empty circular list



A circular list with a single node

Why circular linked list

- Round robin scheduling
- Board games
- Processes on CPU

Linked Lists to construct other data structures

Stack

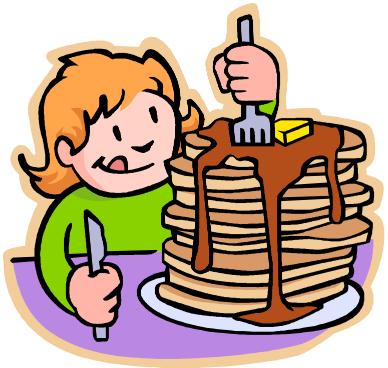
Queue

Tree

Stack

- A linear data structure where addition and deletion of elements can happen only at one of the ends of the data structure
  - Last-in-first-out (LIFO).
  - Only the top-most element is accessible at any point of time.
- Some operations:
  - Push: Add an element to the top of the stack.
  - Pop: Remove the topmost element.
  - IsEmpty: Checks whether the stack is empty or not.

Can implement a stack using arrays or using linked lists (we will see both approaches)



Stack using (statically allocated) arrays

- Uses an array and a marker.

```
#include<stdio.h>
#define MAX 100 // global

int stack[MAX]; // global (elements on the stack, each assumed integer)
int marker = -1; // global

int top_value();
void insert(int value);
int delete();

int full();
int empty();
```

Empty and full

```
int full() {
        if (marker == MAX-1) {
                return 1;
        else
                return 0;
}
int empty() {
        if (marker == -1)
                return 1;
        else
                return 0;
}
```

Insert (push)

```
void insert(int value) {
	if (full()) {
		printf(“Stack is full, can’t insert value \n”);
	}
	else {
		marker = marker + 1;
		stack[marker] = value;
		printf(“%d inserted at %d \n”, value, marker);
	}
}
```

Delete (pop)

```
int delete() {
	int top = -1;
	if (empty()) {
		printf(“Stack is empty, can’t delete value \n”);
	}
	else {
		top = stack[marker];
		marker = marker - 1;
		printf(“%d deleted from %d \n”, top, marker);
	}
	return top;
}
```

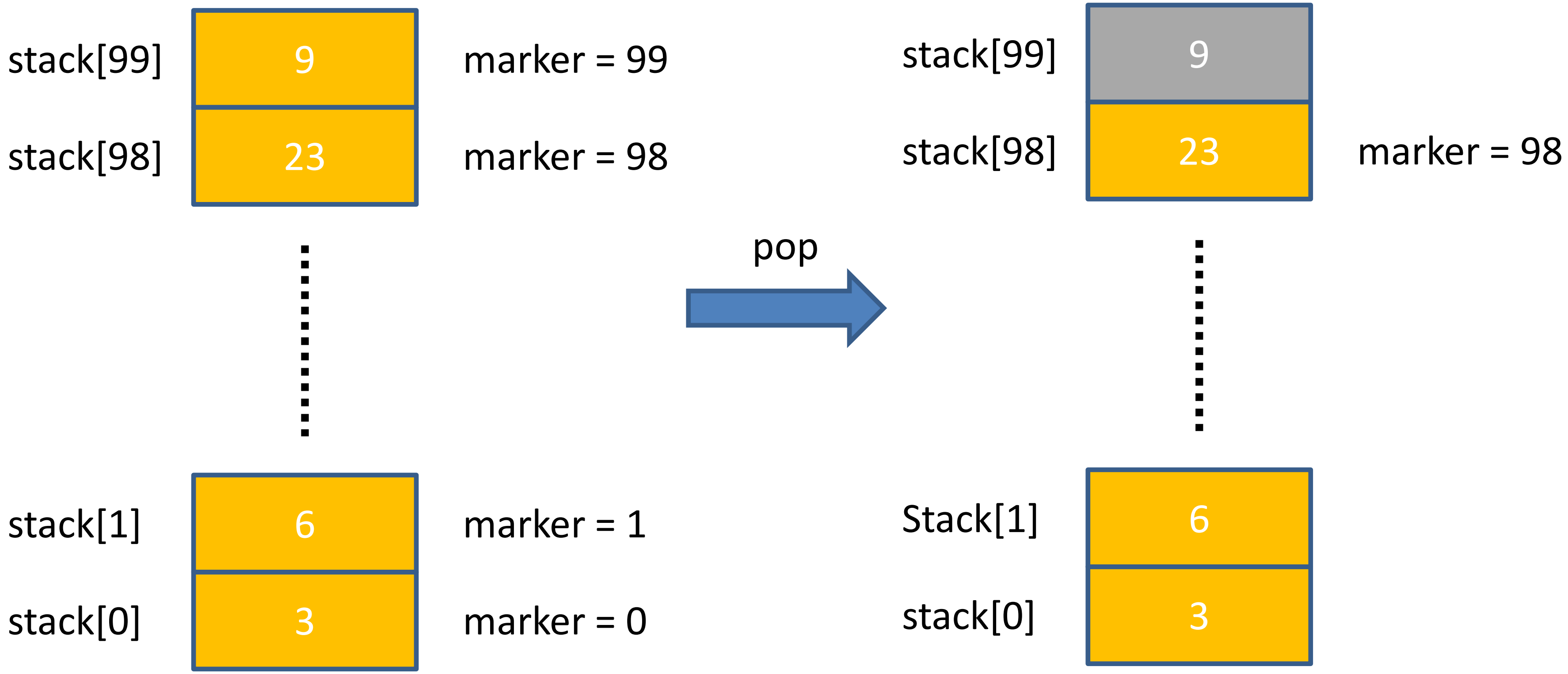
top_value and main

- Writing the top_value function is given as a simple exercise ☺

```
int main() {
        insert(20);
        insert(10);
        delete();
        insert(100);
        if (delete() == -1) {
                printf("element can't be deleted \n");
        }
        return 0;
}
```

An issue with (statically allocated) array based approach

- delete/pop doesn’t actually remove the elements from the array; it simply changes the index (marker) of the top element



Stack using arrays

- The array based approach we saw is just one of the ways
- We kept the array fixed (didn't shift the indices of elements after delete/pop) and simply moved the marker
- We can use arrays in many other ways too, to implement a stack
  - Can also shift the indices of elements in the array upon delete/pop
- .. and, of course, we can also use a linked list to implement a stack (next class)