Final LAB Exam duration: **2hrs 45mins.**

On Saturday, 7th Nov @ 2 PM

B1-6 (Mon/Tue Lab Batch).

Report at **New Core Labs** before 2pm**.**

On Sunday, 8th Nov @ 10 AM

B7-12 (Wed/Thu Lab Batch).

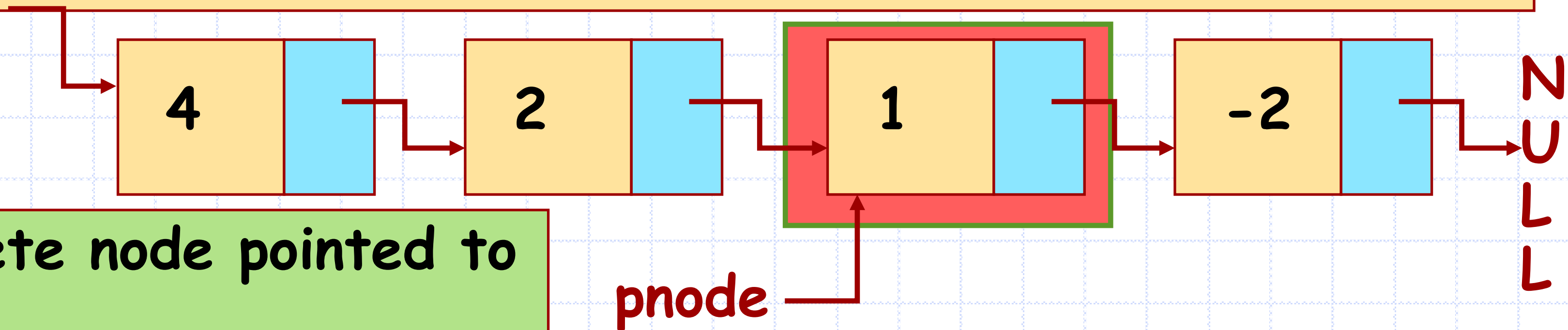
PH Category (all sections).

Report at **New Core Labs** before 10am.

Syllabus: Everything covered till Friday, 6th Nov.

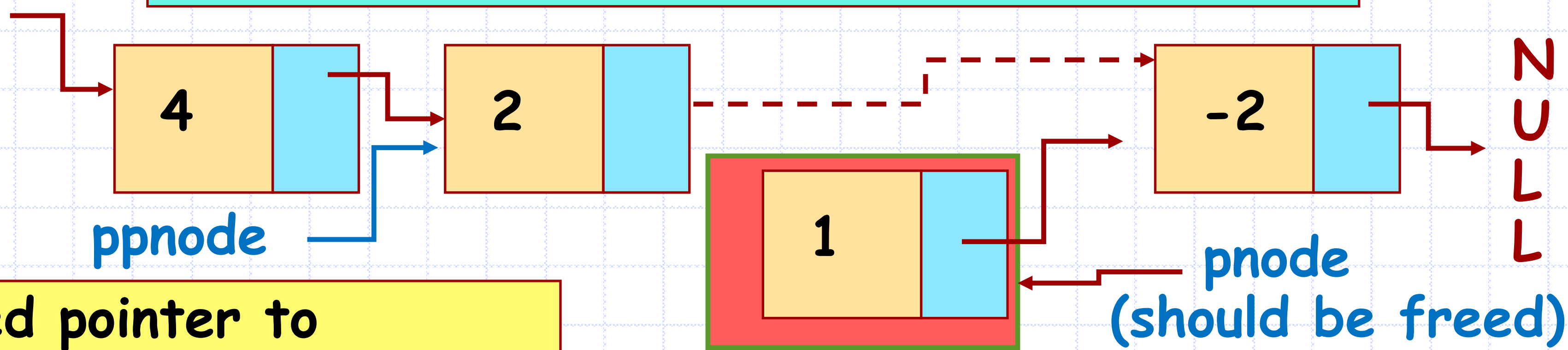
Deletion in linked list

Given a pointer to a node pnode that has to be deleted. Can we delete the node?



E.g, delete node pointed to by pnode

After deletion, we want the following state



Need pointer to previous node to pnode to adjust pointers.

call free() to release storage for deleted node.

prototype

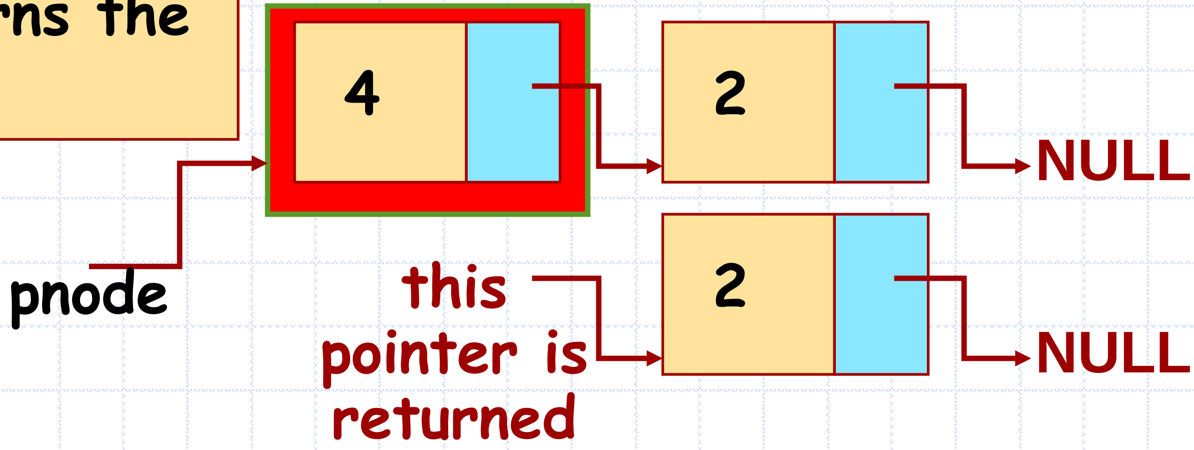
```
delete(Listnode pnode, Listnode ppnode)
```

```
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 to by pnode. ppnode is pointer to the node previous to pnode in the list, if such a node exists, otherwise it is 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.



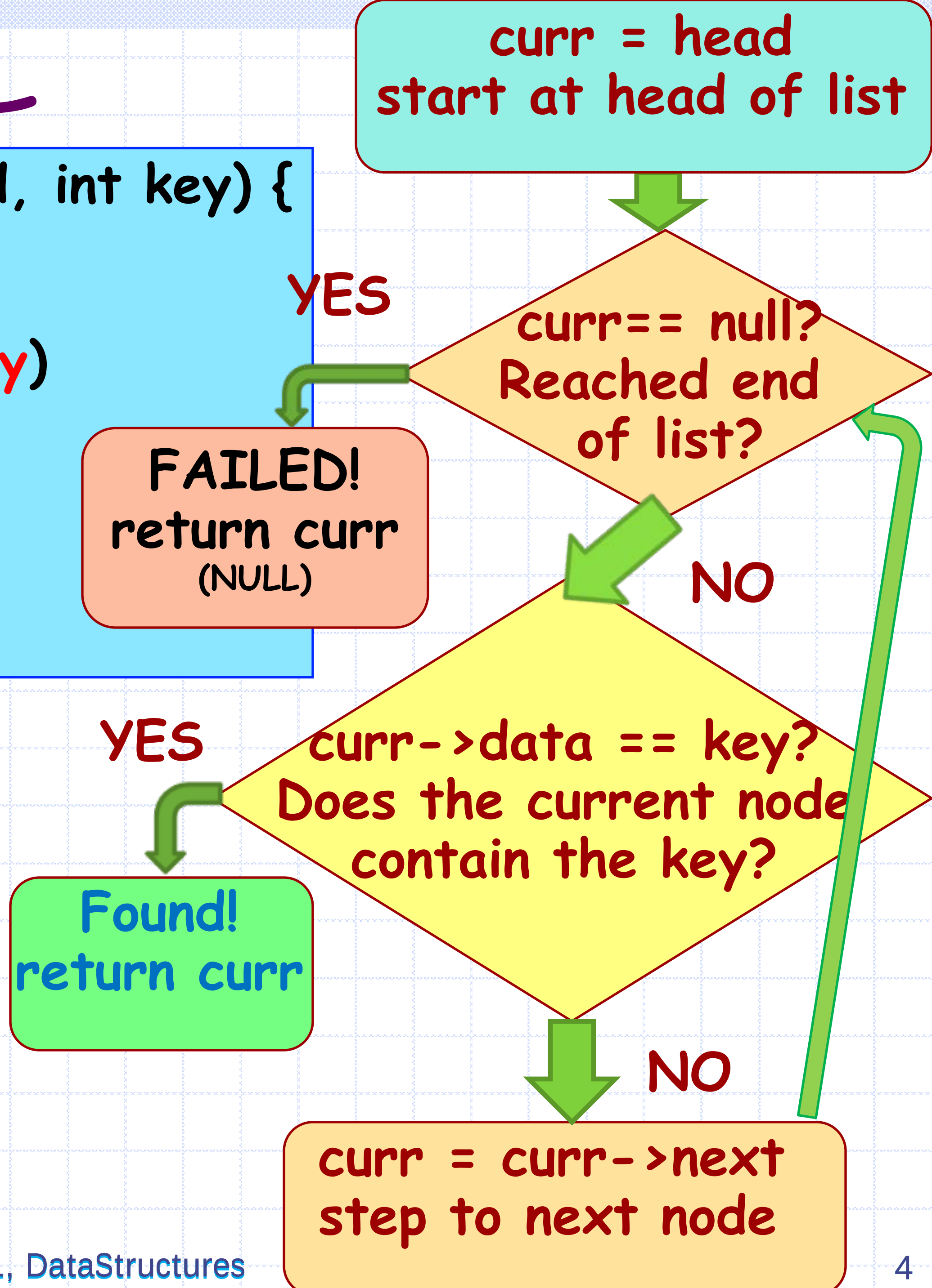
Searching in LL

```
Listnode search(Listnode head, int key) {
   Listnode curr = head;
    while
    (curr && 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.



Why linked lists

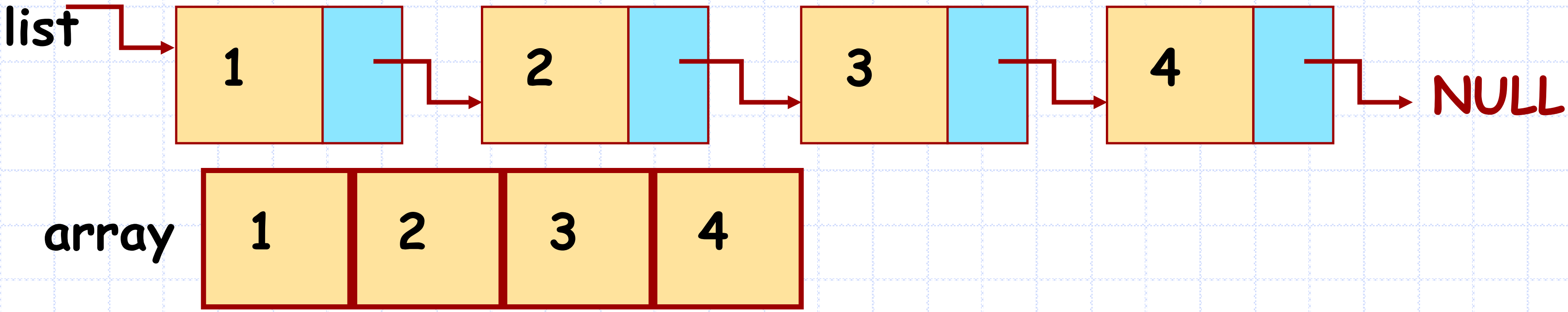
> The same numbers can be represented in an array. So, where is the advantage?

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)
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)

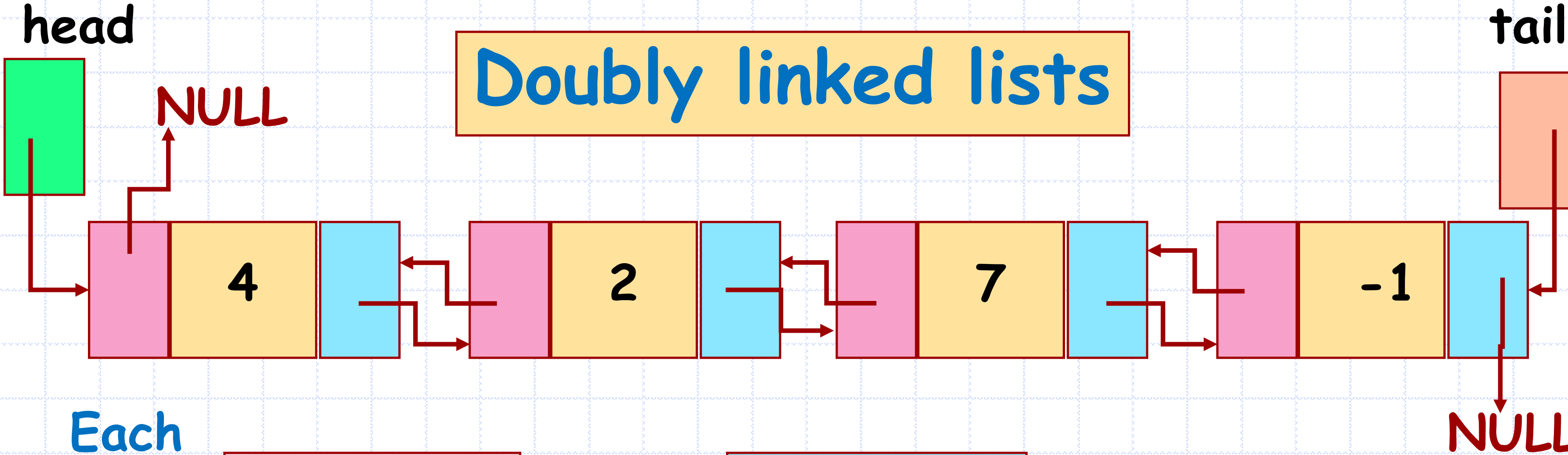
Singly Linked Lists

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

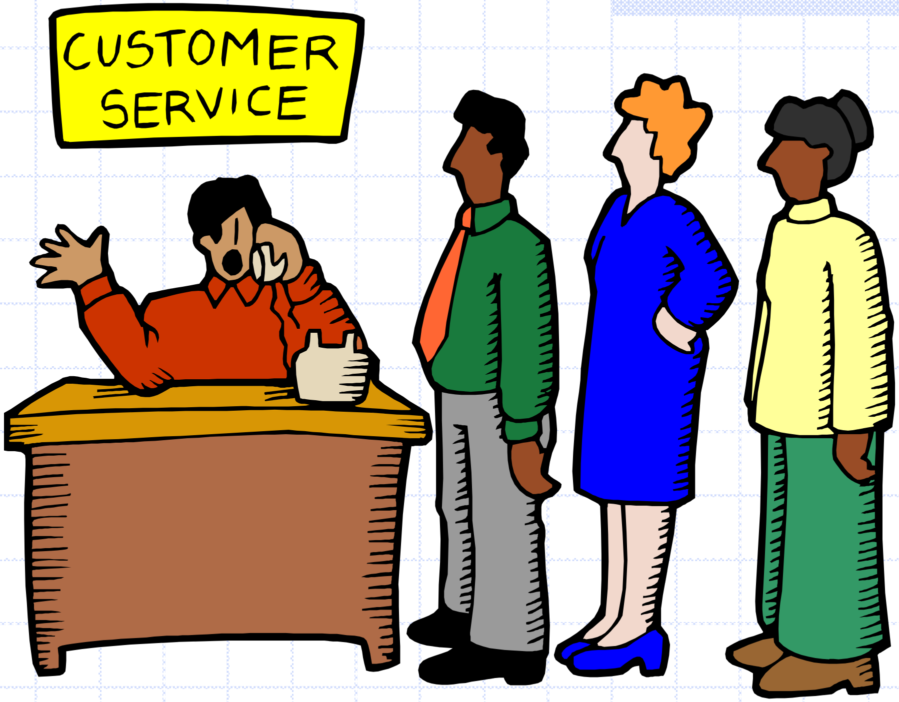
(ii) data

(iii) pointer to next node

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

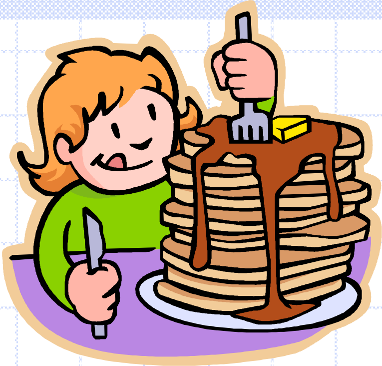
```
struct dlnode {
      int data;
      struct dlnode *next;
      struct dlnode *prev;
};
typedef struct dlnode *Ndptr;
```

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



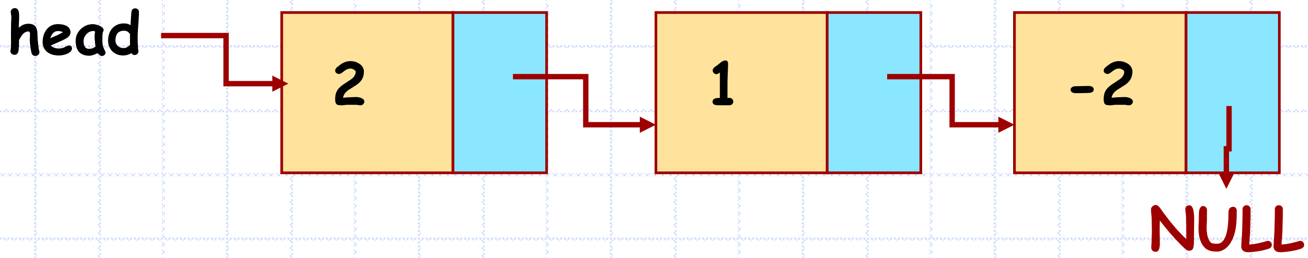
Data structures, Stack and Queue, can also be implemented using Linked Lists!

Stack

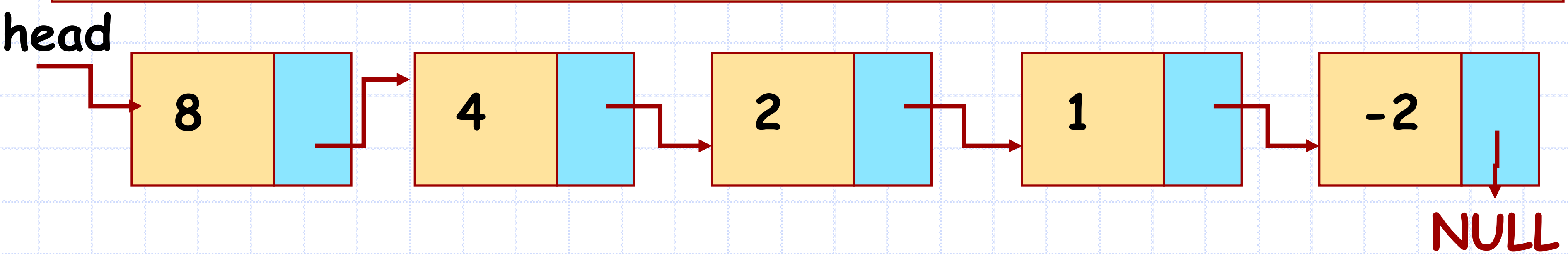


- A linear data structure where addition and deletion of elements can happen at one end of the data structure only.
  - Last-in-first-out.
  - Only the top most element is accessible at any point of time.
- Operations:
  - Push: Add an element to the top of the stack.
  - Pop: Remove the topmost element.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

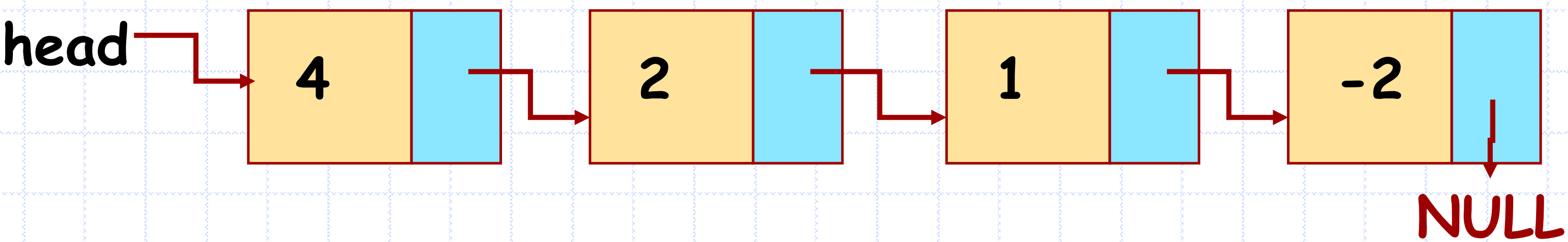
STACK



Push 4,8 in stack: *insert_front(4, head);*
insert_front(8, head);



Pop from stack: *val = head->data;*
delete(head,NULL);

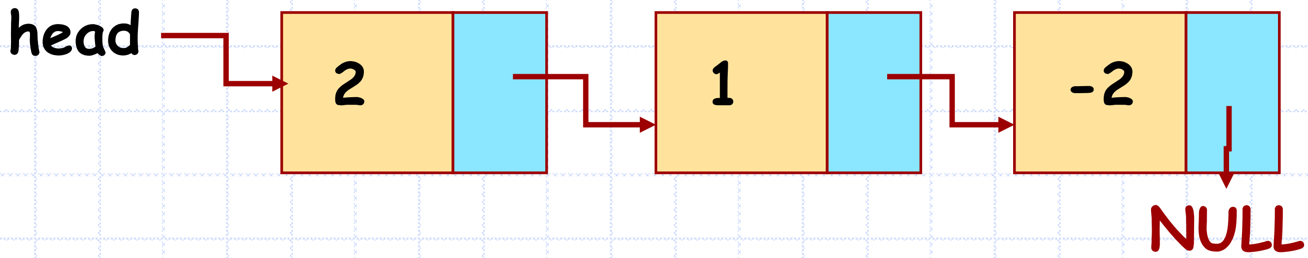


isEmpty function: *return !head ;*

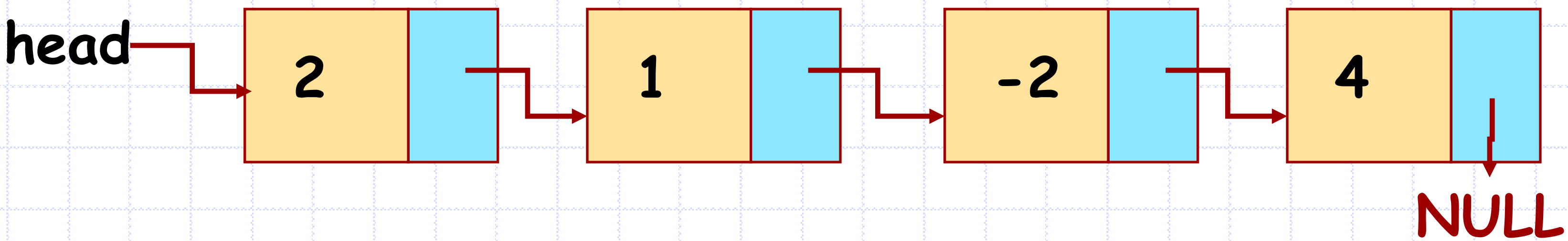
Queue

- A linear data structure where addition happens at one end (`back') and deletion happens at the other end (`front')
  - First-in-first-out
  - Only the element at the front of the queue is accessible at any point of time
- Operations:
  - Enqueue: Add element to the back
  - Dequeue: Remove element from the front
  - IsEmpty: Checks whether the queue is empty or not.

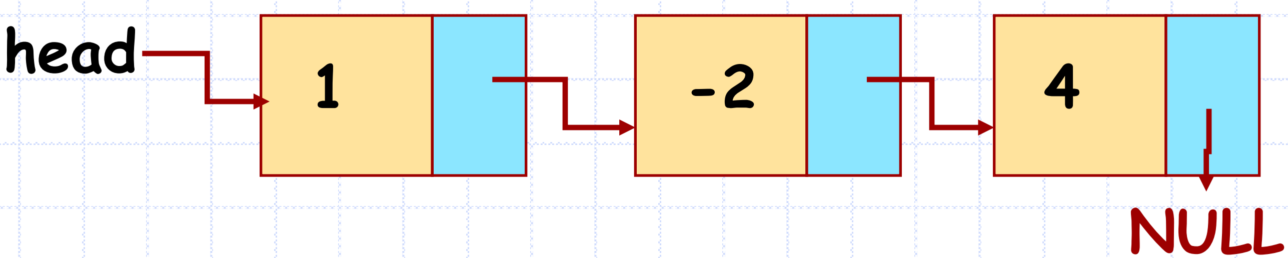
QUEUE



Enqueue 4: *//make a node pnew with data=4*
insert_after_node(tail, pnew);

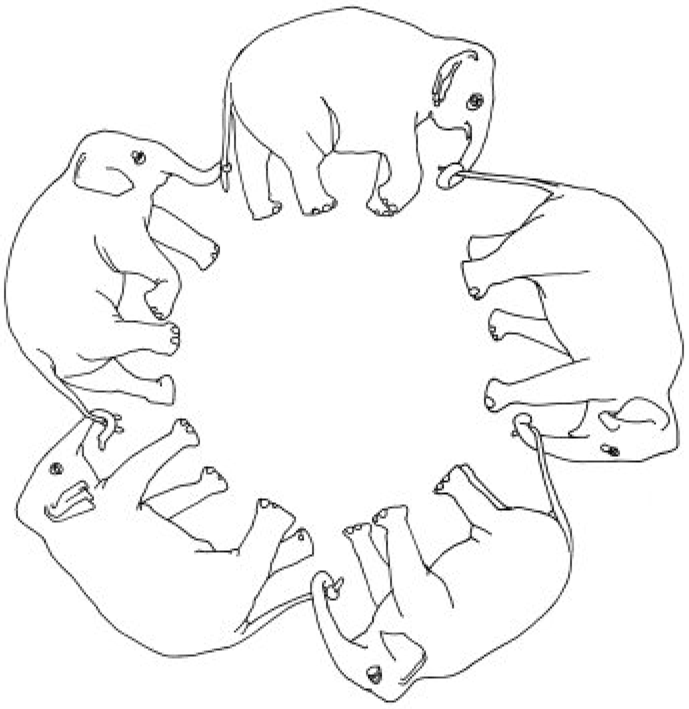


Dequeue: *val = head->data;*
delete(head,NULL);



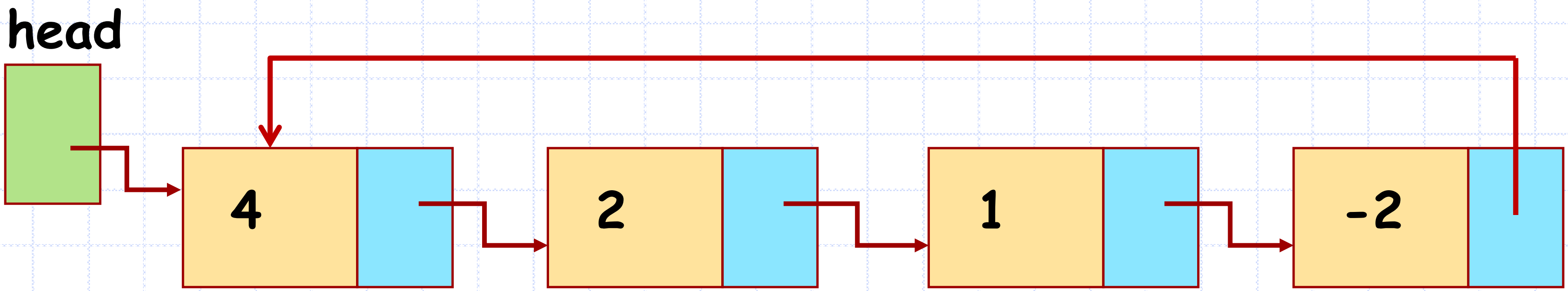
isEmpty function: *return !head ;*

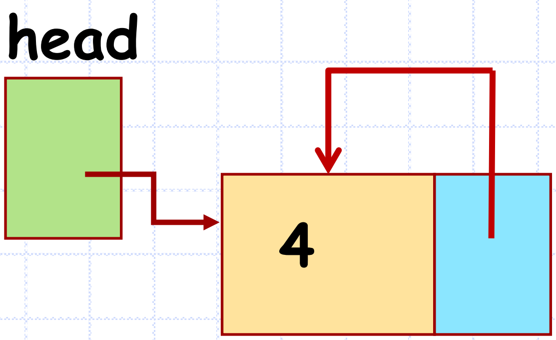
Circular Linked List

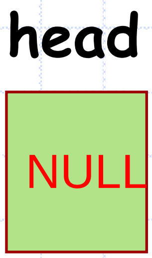


So far, we were modeling a singly linked list by a pointer to the first node of the list.
Let us make the following change:

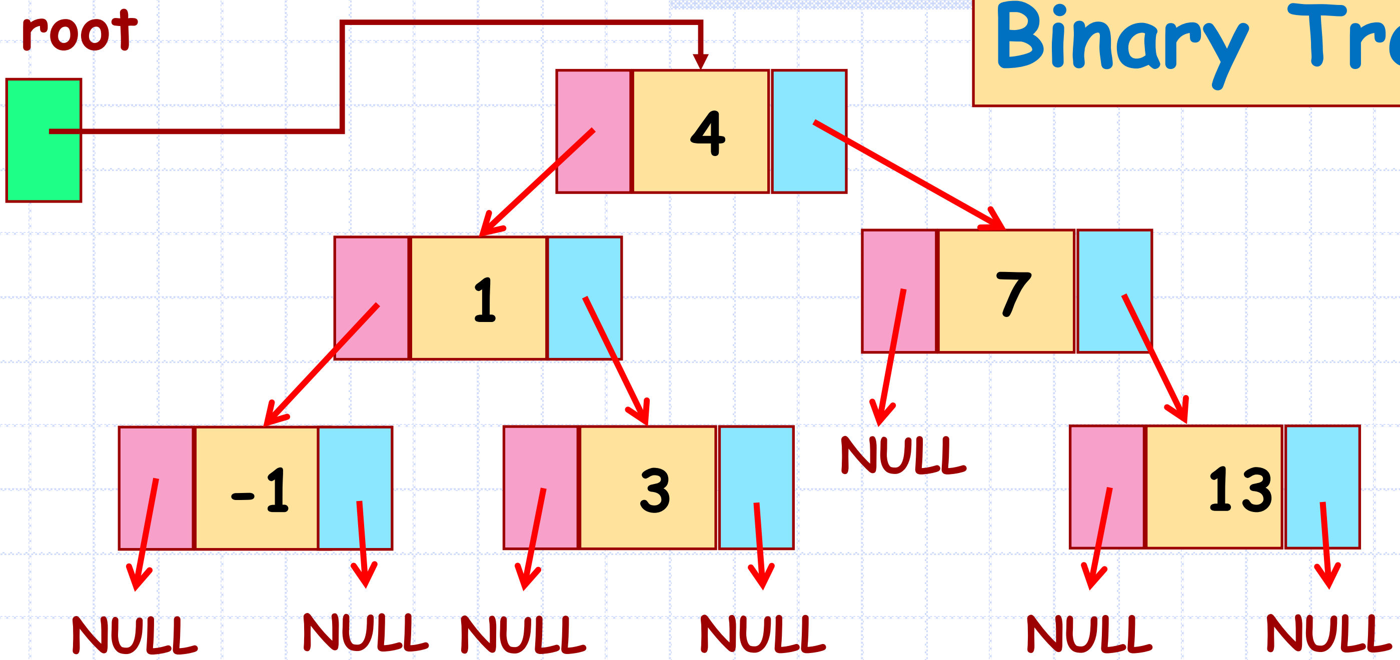
Make the list circular: next pointer of last node is not NULL, it points to the head node.







Binary Tree



Each node has 3 fields

(i) pointer to left child node	(ii) data	(iii) pointer to right child node

Defining Binary Tree

```
typedef struct _btnode *Btree;
struct _btnode {
        int data;
        Btree left;
        Btree right;
};
```

```
Btree root;
```

Traversing a Binary Tree

- Visit each node in the binary tree exactly once
- Easy to traverse recursively
- Three common ways of visit
  - inorder: left, root, right
  - preorder: root, left, right
  - postorder: left, right, root

```
void inorder(tree  t)
{
    if (t == NULL) return;
    inorder(t->left);
    process(t->data);
    inorder(t->right);
}
```

Recursion vs Iteration

```
void inorder(tree t) {
    stack s;
    push(s,t);
    while (!empty(s)) {
        curr = top(s);
        if (curr) {
            if (!curr->visited) {
                push(s,curr->left);
            } else {
                process(curr->data);
                pop(s);
                push(s,curr->right);
            }
        } else {
            pop(s);
            if (!empty(s))
                top(s)->visited = true;
        }
    }
}
```

```
void inorder(tree  t)
{
    if (!t) return;

    inorder(t->left);
    process (t->data);
    inorder(t->right);
}
```

Stack entries use an extra field – visited

*** Disclaimer: Code not tested!**