Using Linked Lists

IC-100

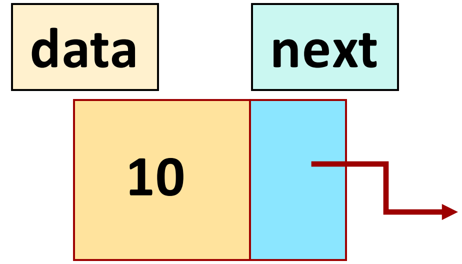
December, 2023

This Class

- Operations on Linked List
- Implementation of Common Data Structures using Linked List

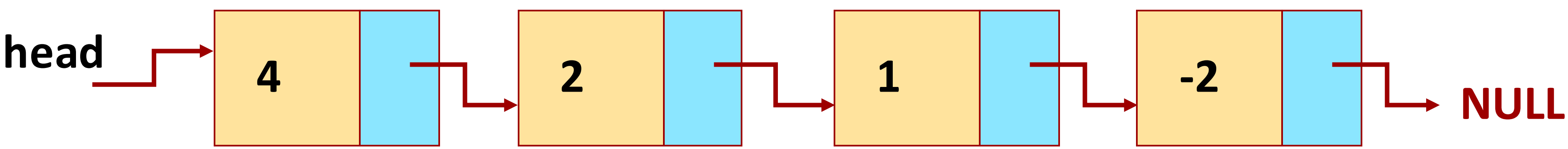
Linked List : A Self-Referential Structure

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



struct node

1. Defines **struct node**, used as a node (element) in the “linked list”.
2. Note that the field **next** is of type **struct node ***
3. **next** can’t be of type **struct node**,
   (recursive definition, of unknown or infinite size).

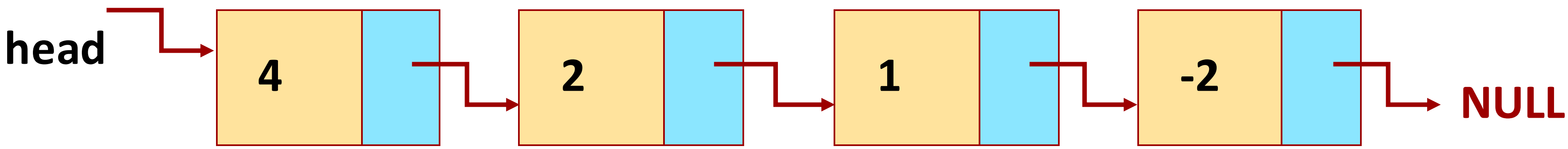


Only one link (pointer) from each node, hence “**singly linked list**”.

Linked Lists

List starts at node pointed to by **head**

next field == NULL pointer indicates the last node of the list



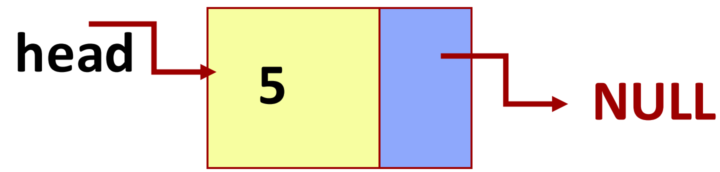
1. **The list is modeled by a variable (head): points to the first node of the list.**
2. **head == NULL implies empty list.**
3. **The next field of the last node is NULL.**
4. **Name head is just a convention – can give any name to the pointer to first node, but head is used most often.**

Generic Insertion in Linked List

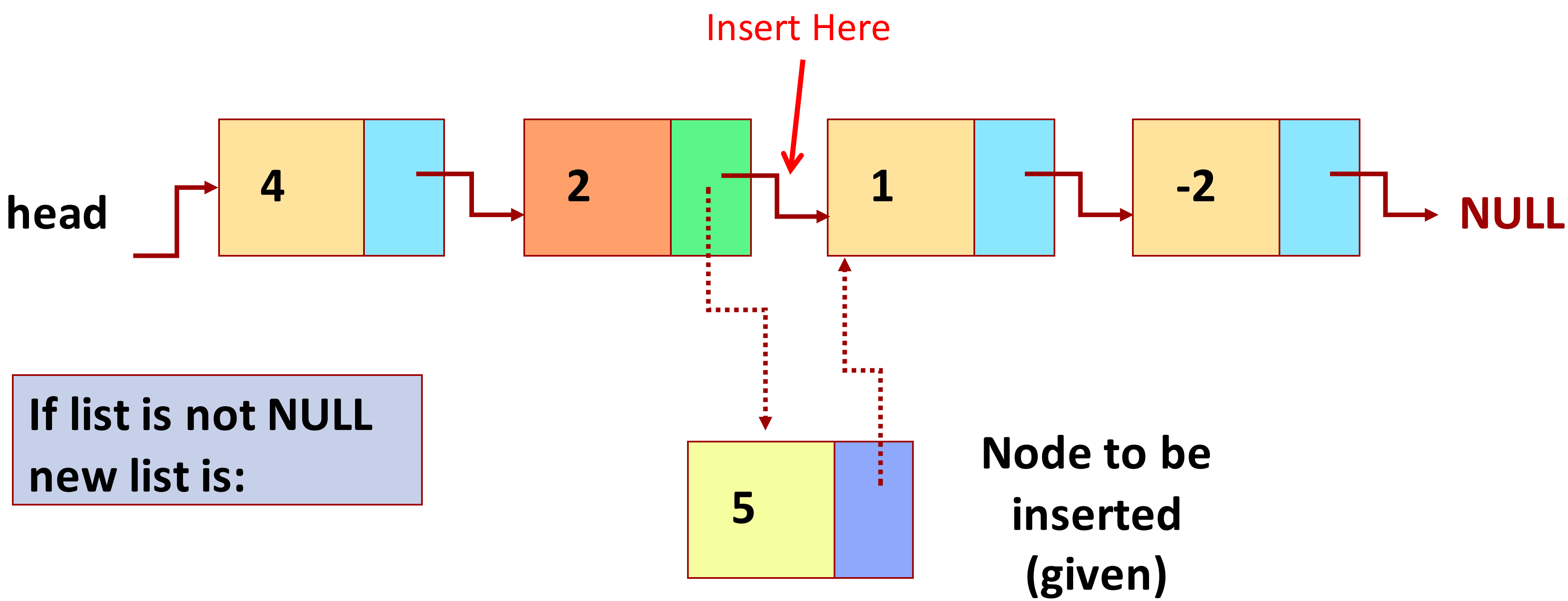
List Insertion

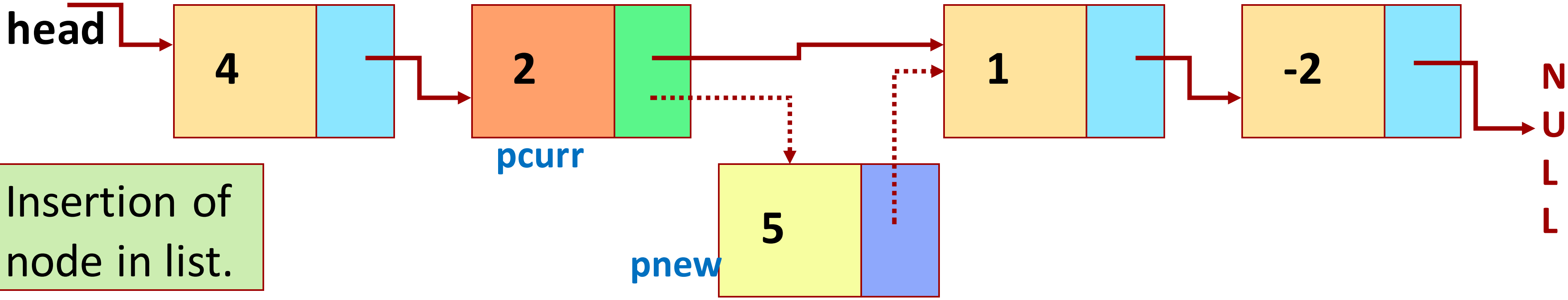
Given a node, insert it after a specified node in the linked list.

If list is NULL new list is:



If list is not NULL new list is:





Insertion of node in list.

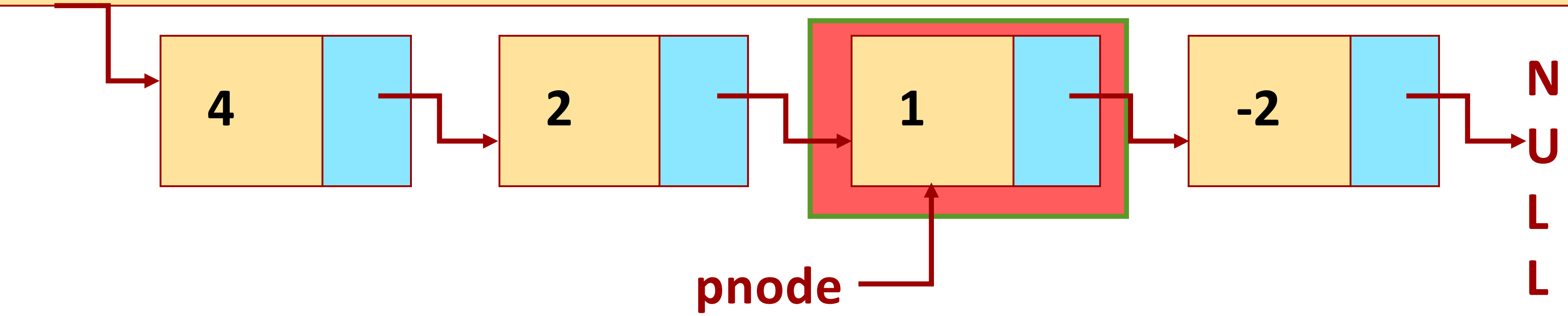
Given

pcurr: Pointer to node after which insertion to be made
pnew: Pointer to new node to be inserted.

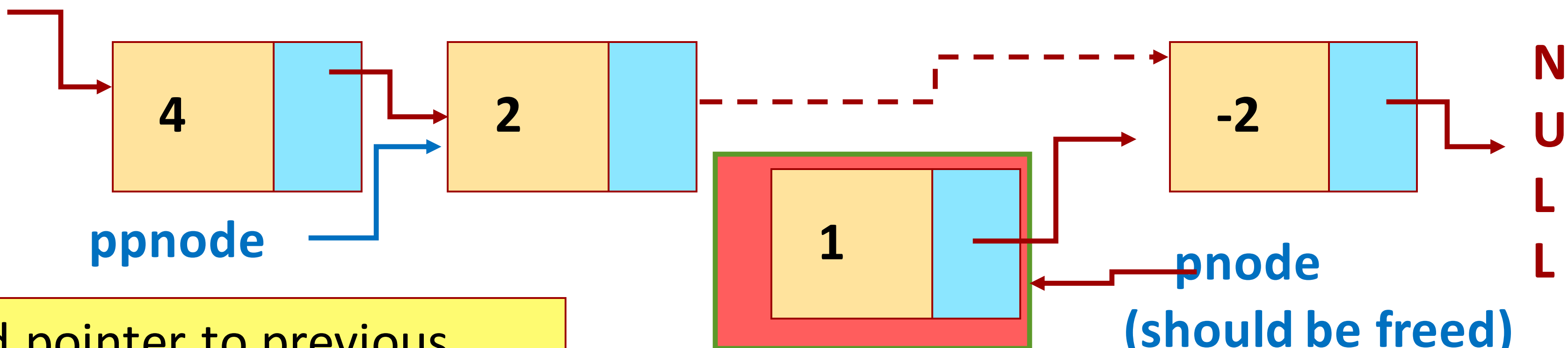
```
struct node *insert_after_node (struct node *pcurr,
                                struct node *pnew) {
  if (pcurr != NULL) {
    // Order of next two stmts is important
    pnew->next = pcurr->next;
    pcurr->next = pnew;
        return pcurr; // return the prev node
  }
  else return pnew; // return the new node itself
}
```

Deletion in Linked List

Given a pointer pnode. Can we delete the node pointer 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.

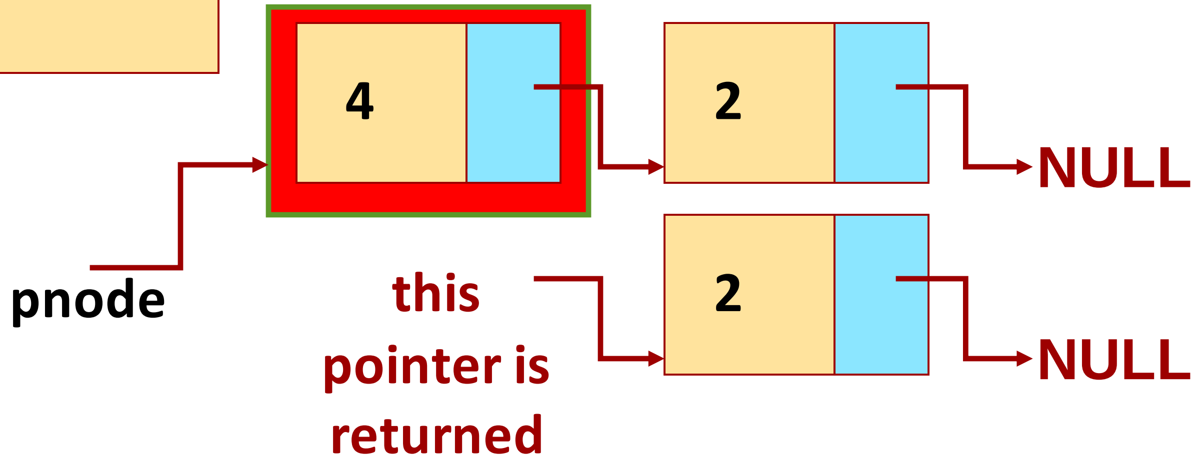
delete(Listnode pnode, Listnode ppnode)

```
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.

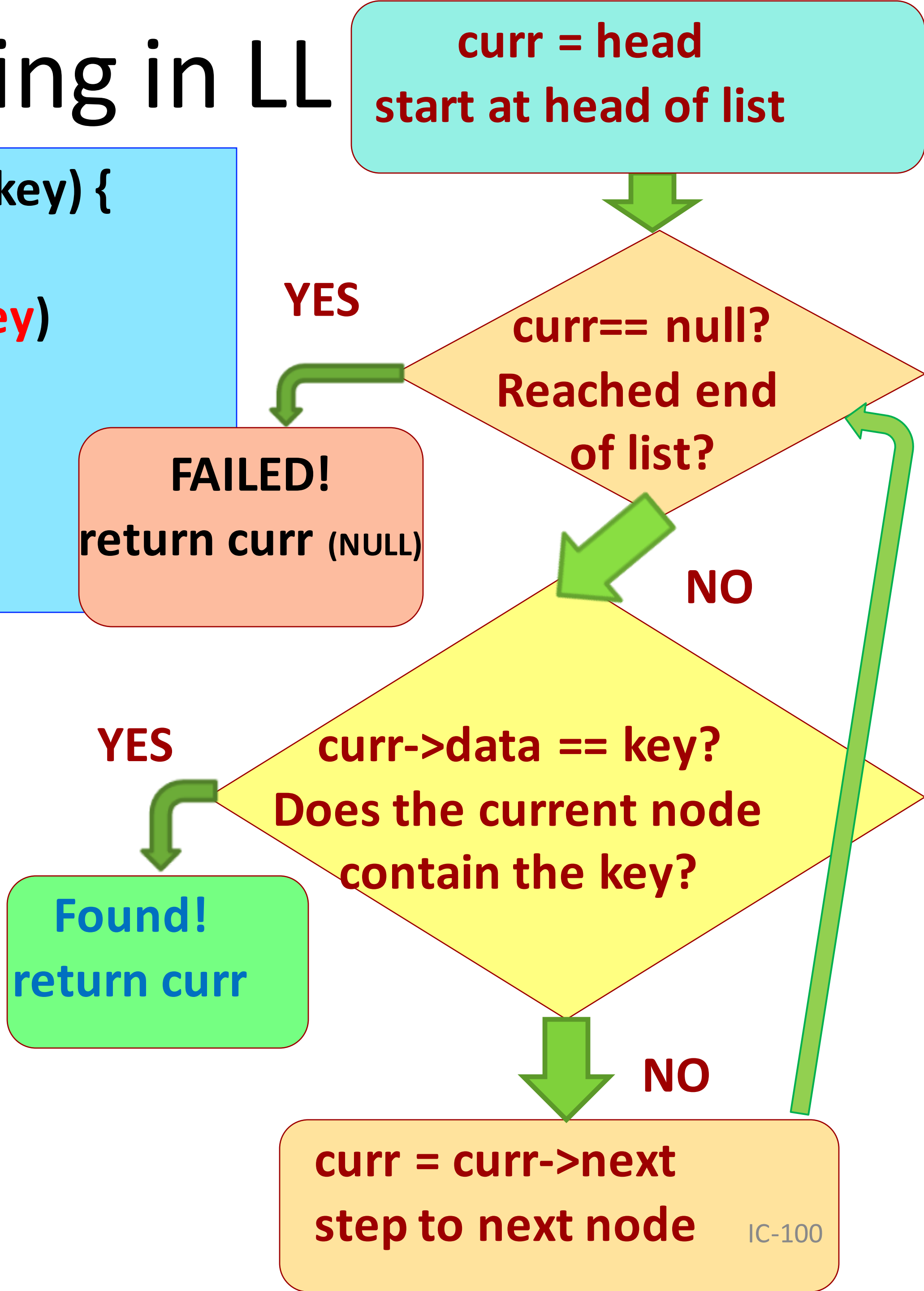
The case when pnode is the head of a list. Then ppnode == NULL.



Searching in LL

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

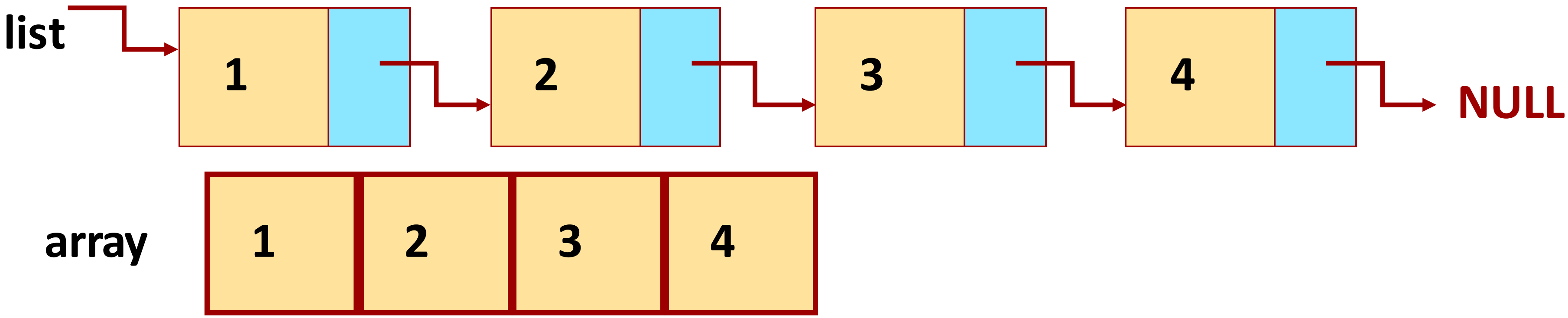
➢ **The same numbers can be represented in an array. So, where is the advantage?**

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)

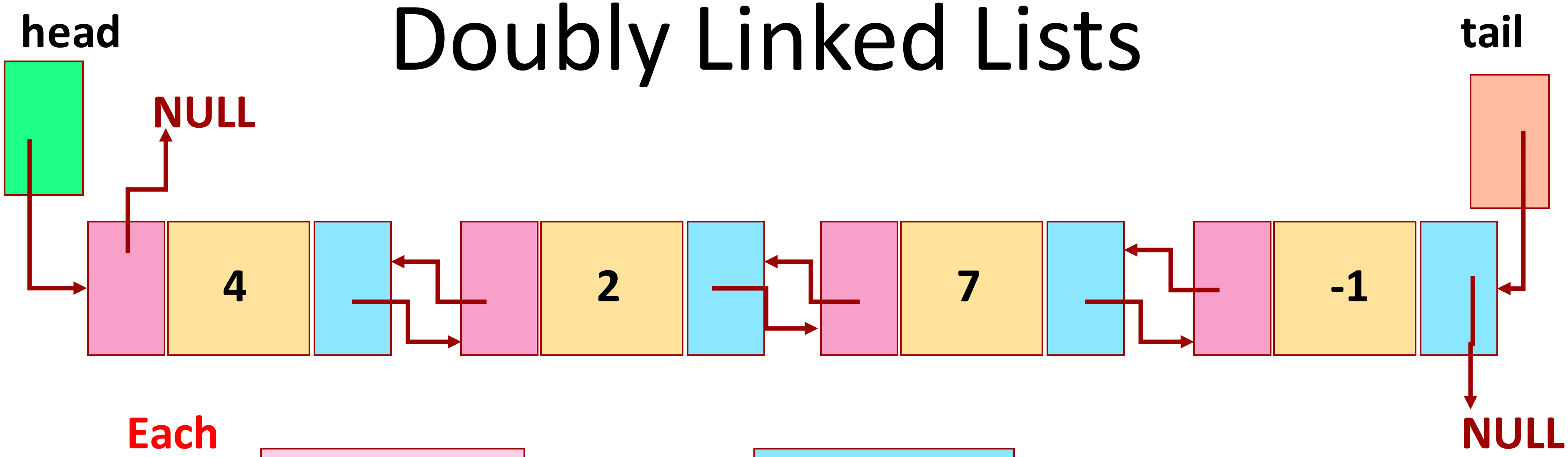
Singly Linked Lists

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

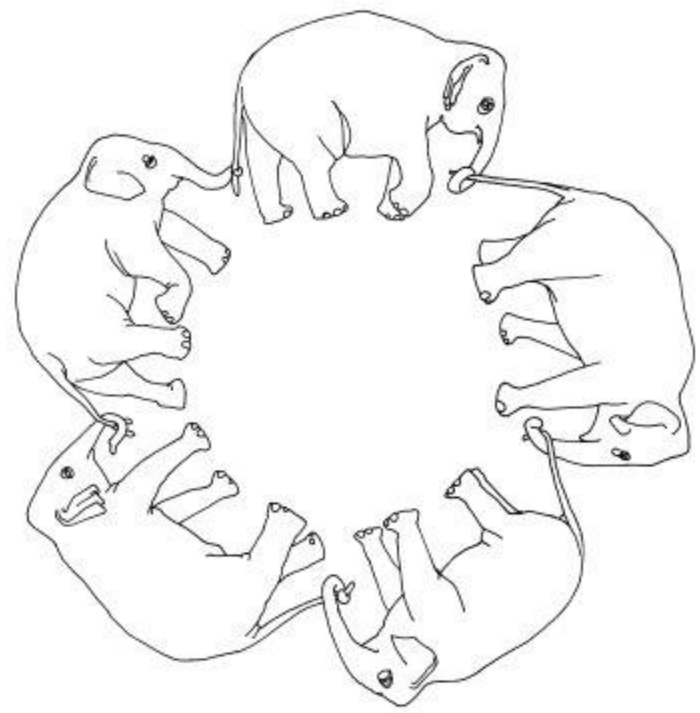
(iii) pointer to next node

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

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

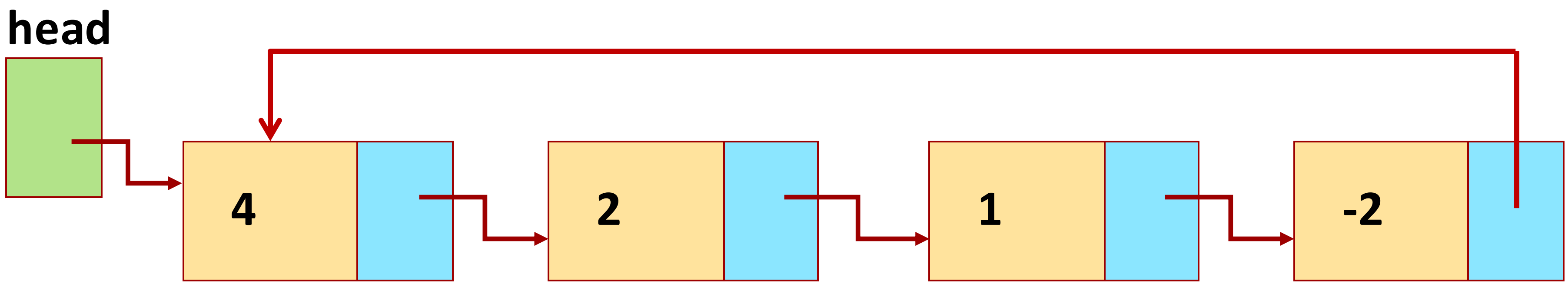
```
struct dlList {
  Ndptr head;/*first node */
  Ndptr tail; /* last node */
};
typedef struct dlList *DlList;
```

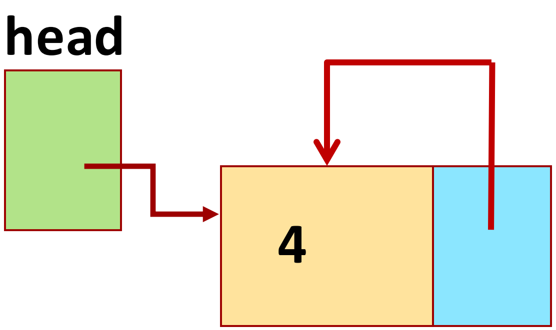
Circular Linked List

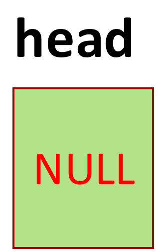


So far, we were modeling a singly linked list by a pointer to the first node of the list.
Let us make the following change:

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

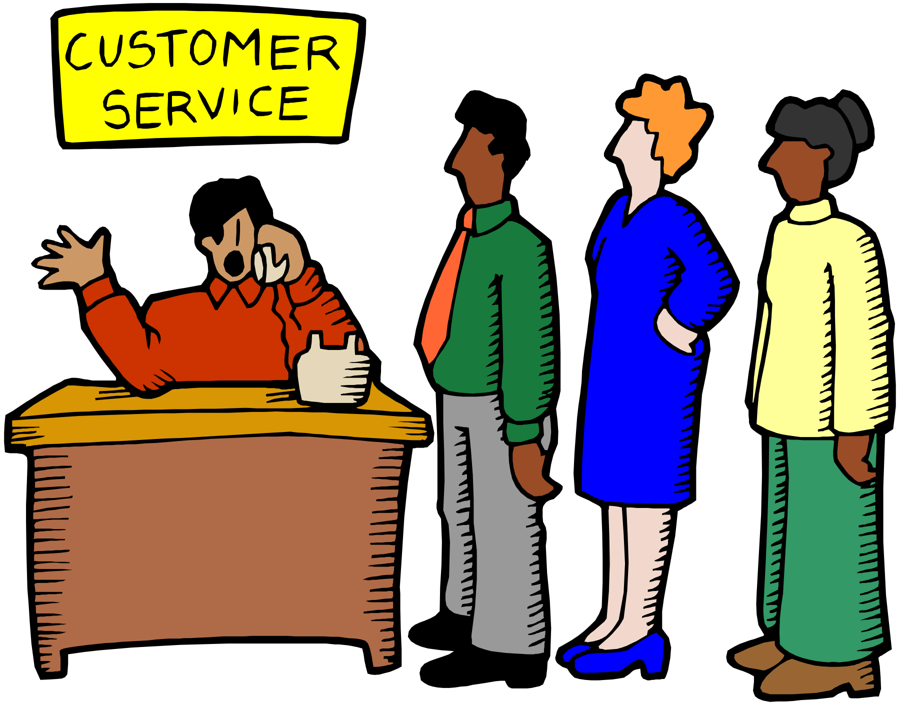






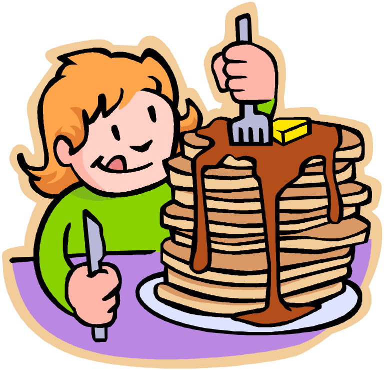
Why Circular Linked List

- Round robin scheduling
- Board games
- Processes on CPU



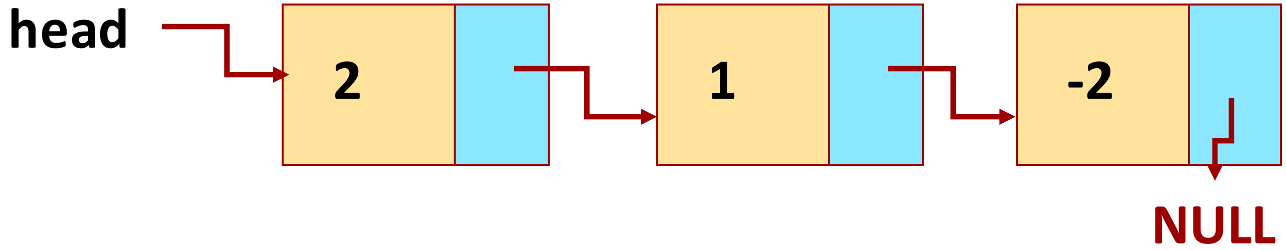
Data structures, Stack and Queue, can also be implemented using Linked Lists!

Stack



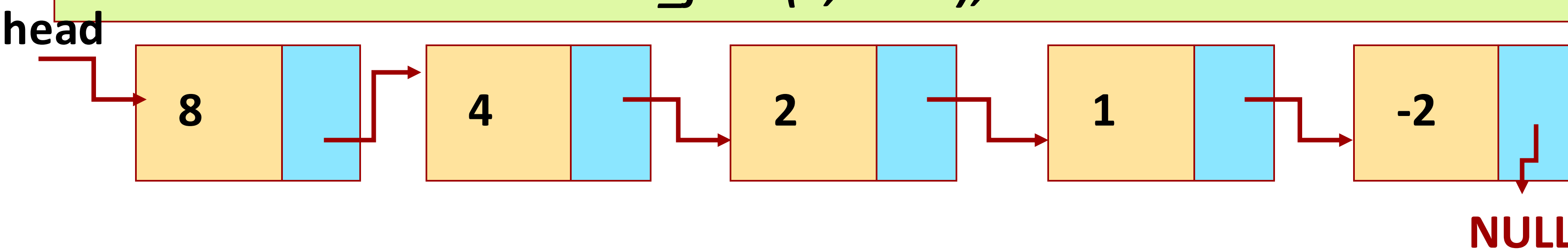
- A linear data structure where addition and deletion of elements can happen at one end of the data structure only.
  - Last-in-first-out.
  - Only the top most element is accessible any point of time.
- Operations:
  - Push: Add an element to the top of the stack.
  - Pop: Remove the topmost element.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

Stack



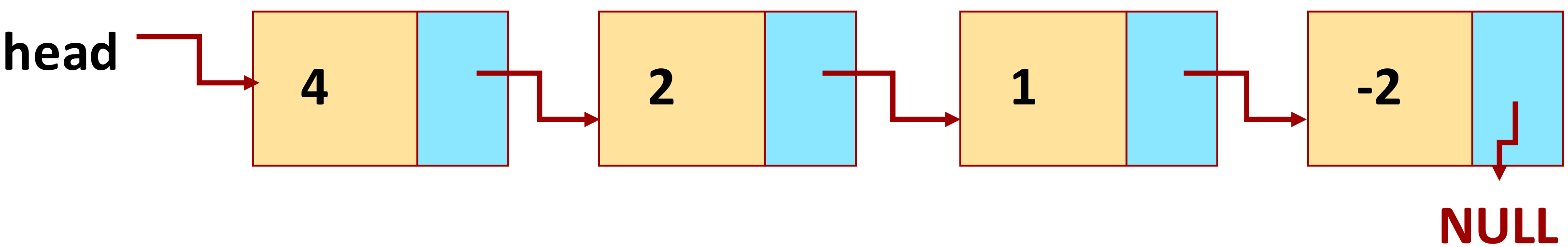
Push **4,8 in stack:** head = *insert_front(4, head);*

head = insert_front(8, head);



Pop **from stack: head1 = head; head1 = head1-> next;** ***val = head->data;***

delete(head,NULL); head = head1

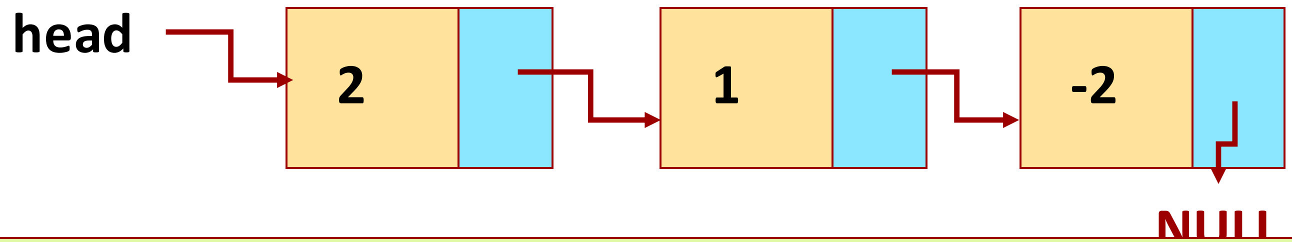


isEmpty **function:** ***return !head ;***

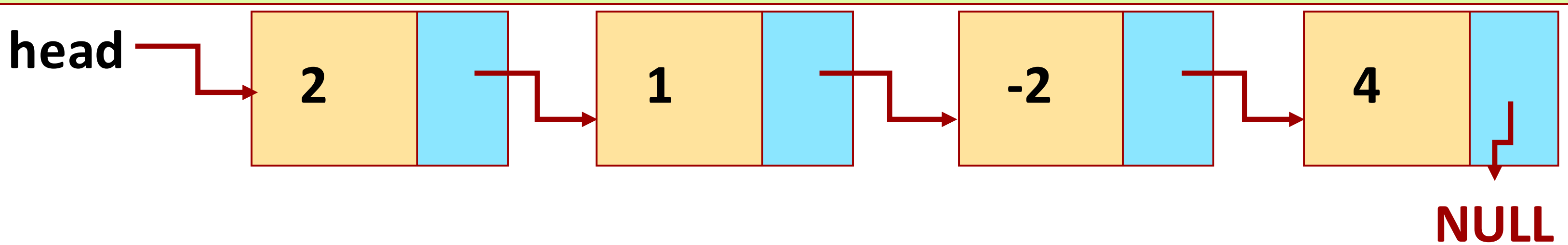
Queue

- 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.

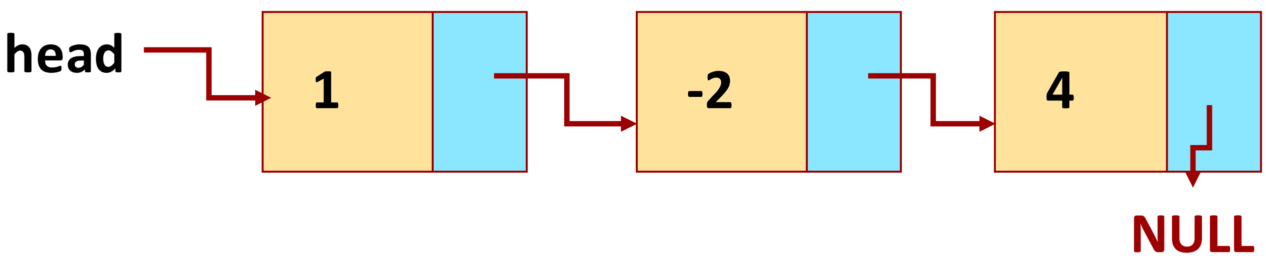
Queue



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

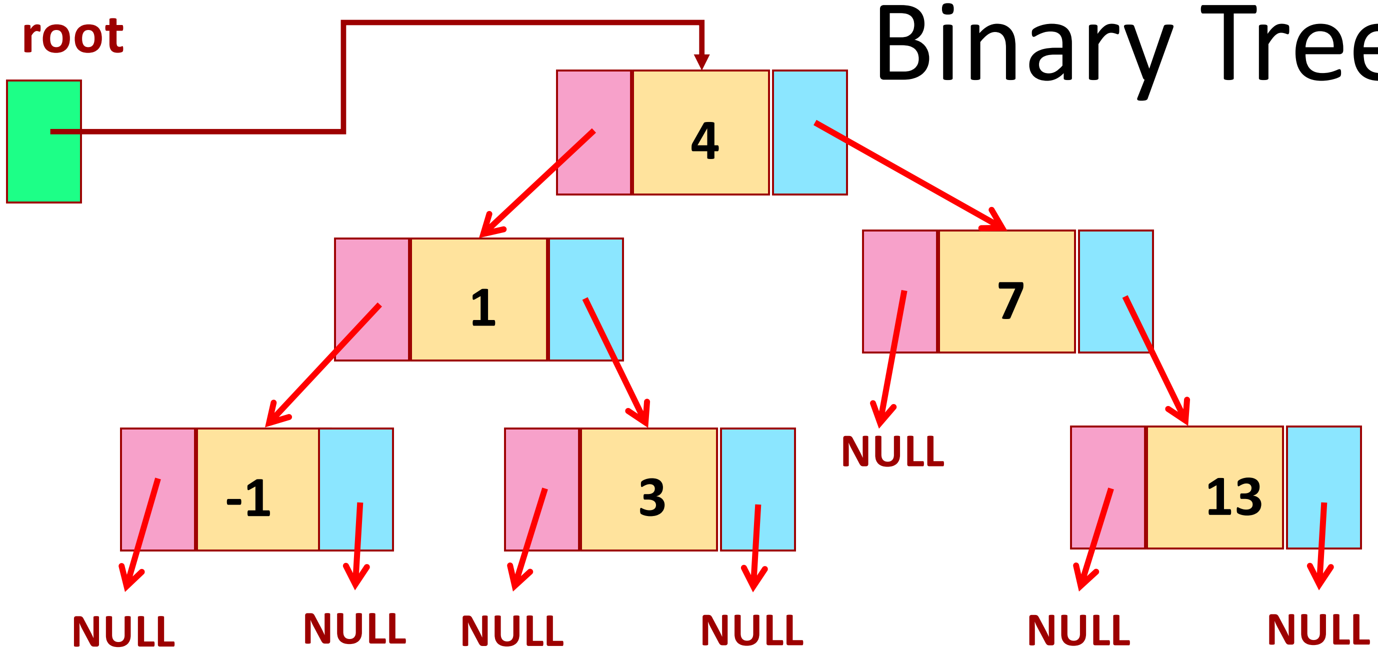


Dequeue: head1 = head; head1 = head1-> next; ***val = head->data;***
delete(head,NULL); head = head1;



isEmpty function: ***return !head ;***

Binary Tree



Each node has 3 fields

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

Defining Binary Tree

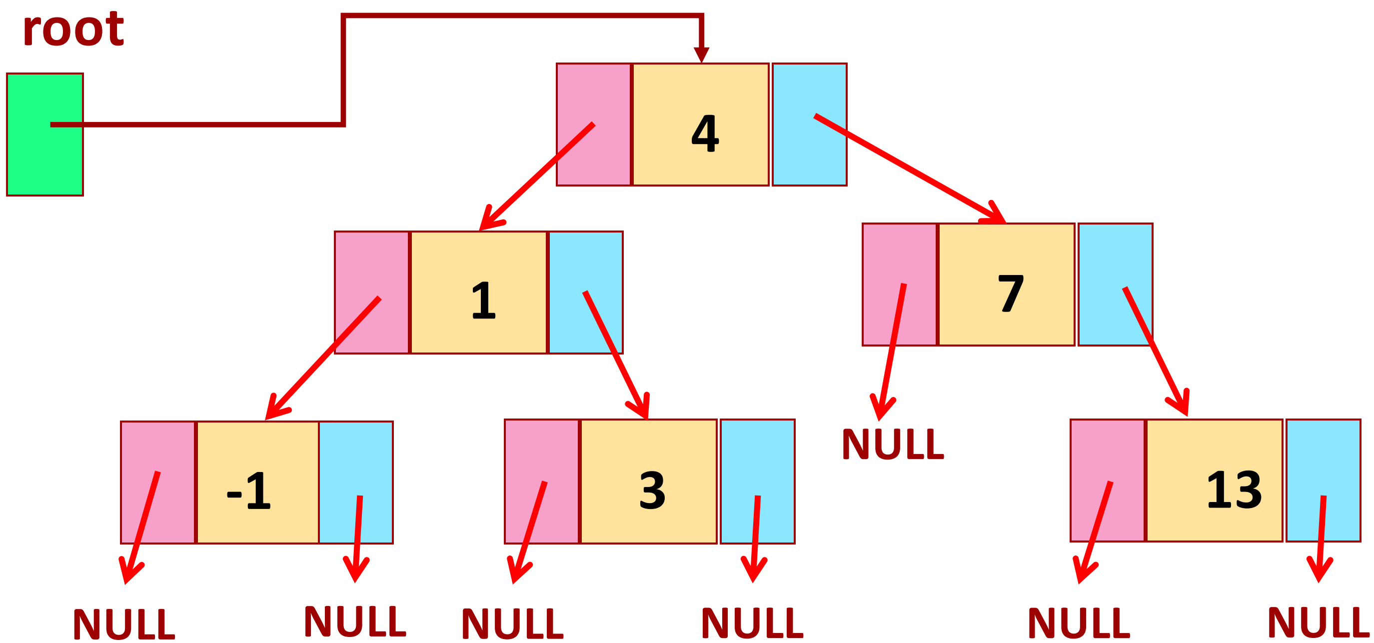
```
struct _btnode {
    int data;
    Btree left;
    Btree right;
};
typedef struct _btnode *Btree;
```

```
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

Inorder Traversal

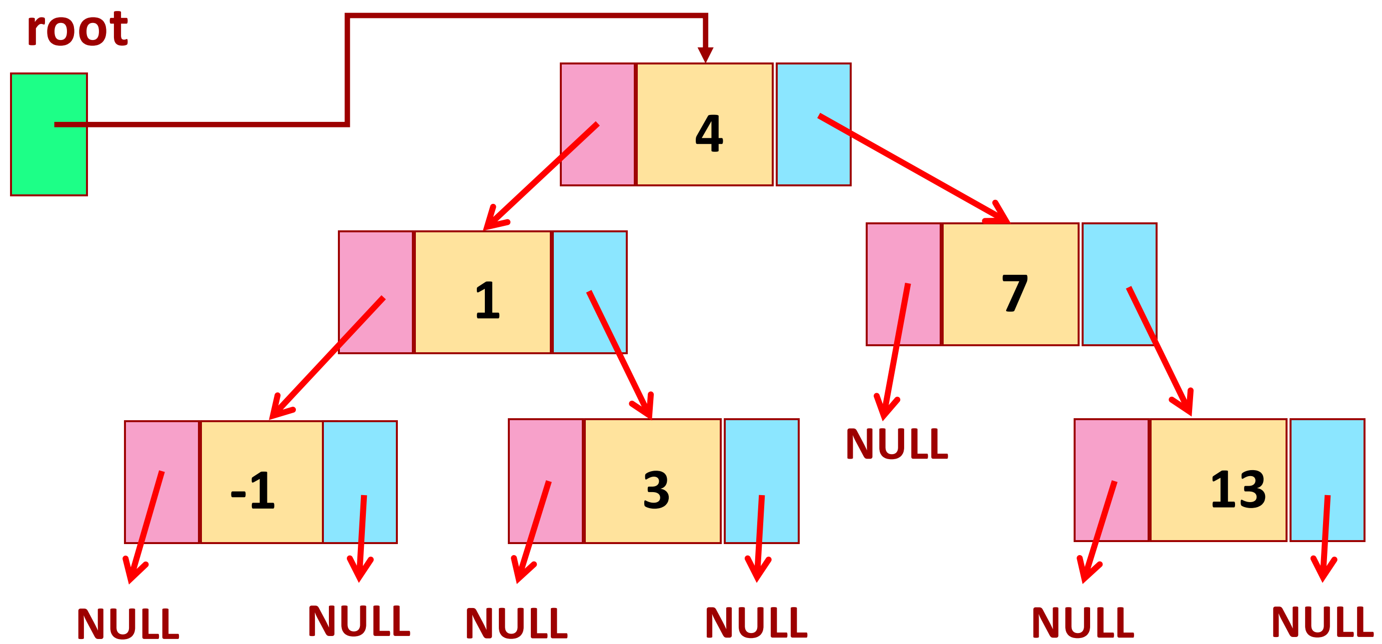



```
void inorder(tree  t)
{
   if (t == NULL) return;
   inorder(t->left);
   printf("%d ", t->data);
   inorder(t->right);
}
```

Result

-1 1 3 4 7 13

Preorder Traversal

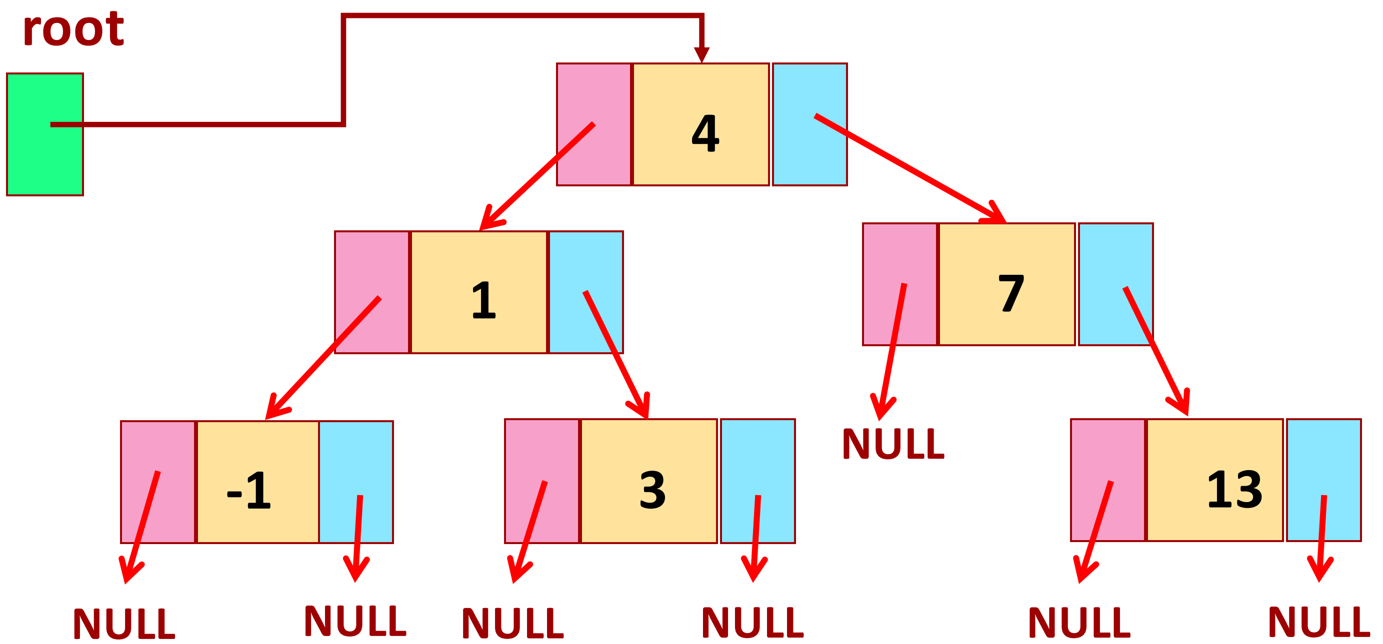



```
Void preorder(tree  t)
{
   if (t == NULL) return;
   printf(“%d “, t->data);
   preorder(t->left);
   preorder(t->right);
}
```

Result

4 1 -1 3 7 13

Postorder Traversal




```
Void postorder(tree  t)
{
   if (t == NULL) return;
   postorder(t->left);
   postorder(t->right);
   printf(“%d “, t->data);
}
```

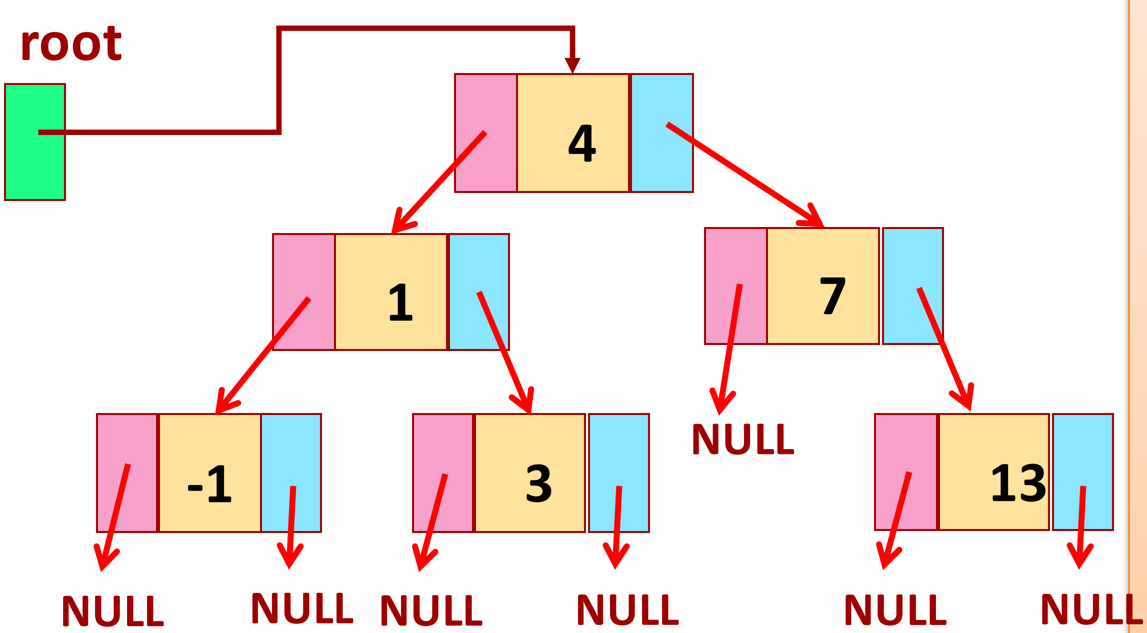
Result

-1 3 1 13 7 4

Inorder Traversal - Iterative

- Need a stack
- Push, pop, empty, top
- Another field: visited
- Process a node

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;
          }
     }
}
```

Next Classes

- Command Line Argument