Data Structures and Algorithms
(ES221)

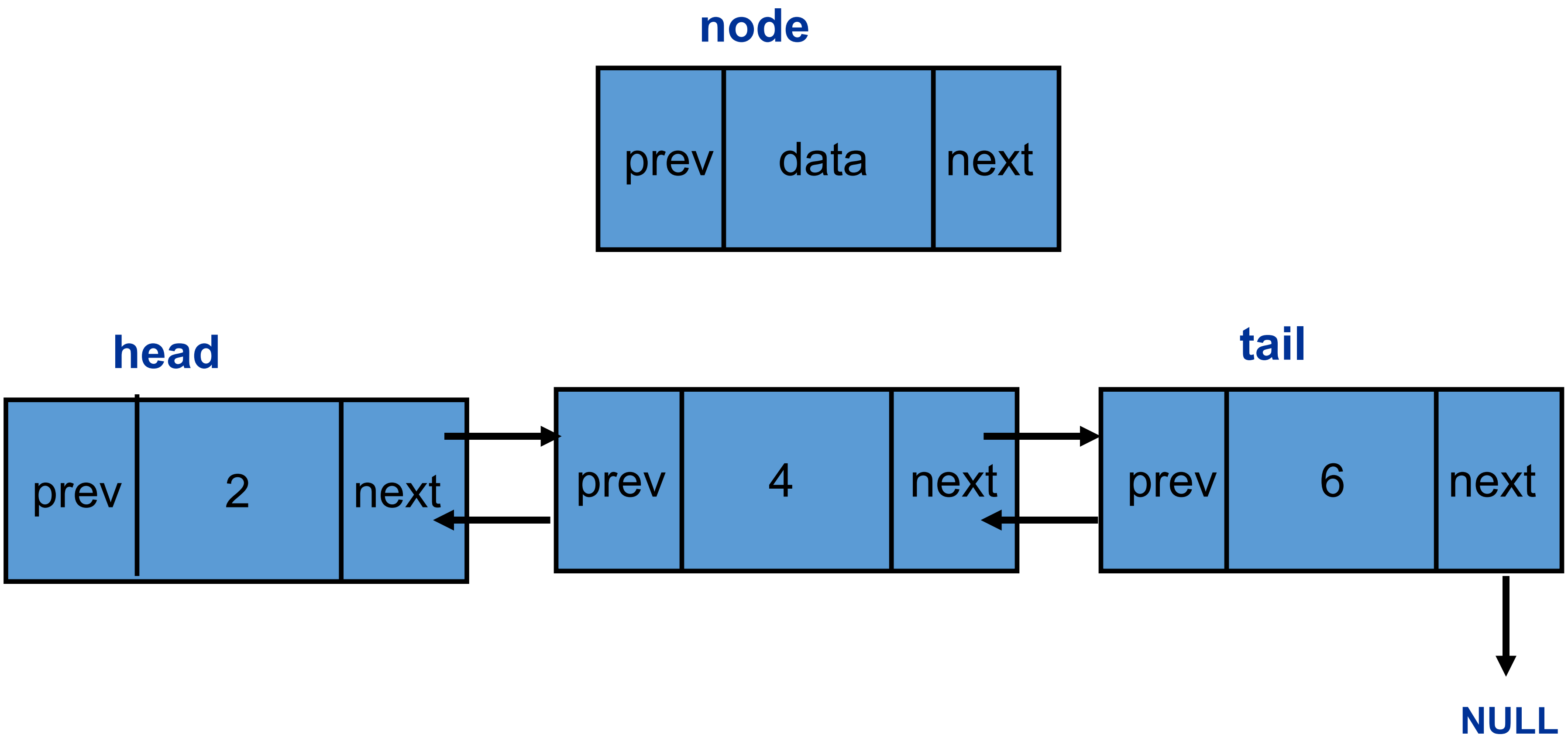
Double and Circular Linked List

Dr. Zubair Ahmad

- **Attendance?**

  - Active Attendance
  - **Dead Bodies.**
  - **Active Minds**
  - Mobiles in hands -> Mark as absent
  - 80% mandatory

Double Linked List



Double Linked List: Create the head node

```
struct node
  {
  int data;
  node *next;
  node *prev;
  };
node *head, *tail;
void main()
  {
  head = new node;
  head->next = NULL;
  head->prev = NULL;
  tail = head;       }
```

int data → Stores the actual value.

node *next → A pointer to the **next** node

node *prev → A pointer to the **previous** node.

head → Points to the first node and **tail** → Points to the last node of the list.

new node; → Dynamically allocates memory for a new node

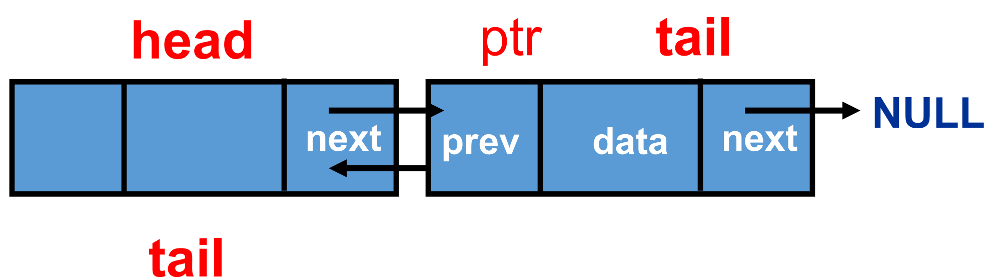
Since this is the **only node**, its next pointer is NULL.

No previous node, so prev is NULL

tail = head; → Since there's only one node, both head and tail point to it.

Double Linked List: Add an element after the head node

```
node *ptr = new node;
cin>> ptr ->data;
head->next = ptr;
ptr->prev = head;
ptr->next = NULL;
tail = ptr;
```



Complexity : O(*1*)

Double Linked List: Add an element at the head node

```
node *ptr  = new node;

cin>> ptr ->data;
head->next = ptr;
ptr->prev = head;
ptr->next = NULL;
tail = ptr;
```

A **new node** is created dynamically using new node;ptr is a pointer that stores the address of this newly allocated node.

The user **enters a value**, which gets stored in ptr->data.

The existing head node's next pointer is updated to point to ptr.

The new node's prev pointer is set to head, forming a **backward link**.

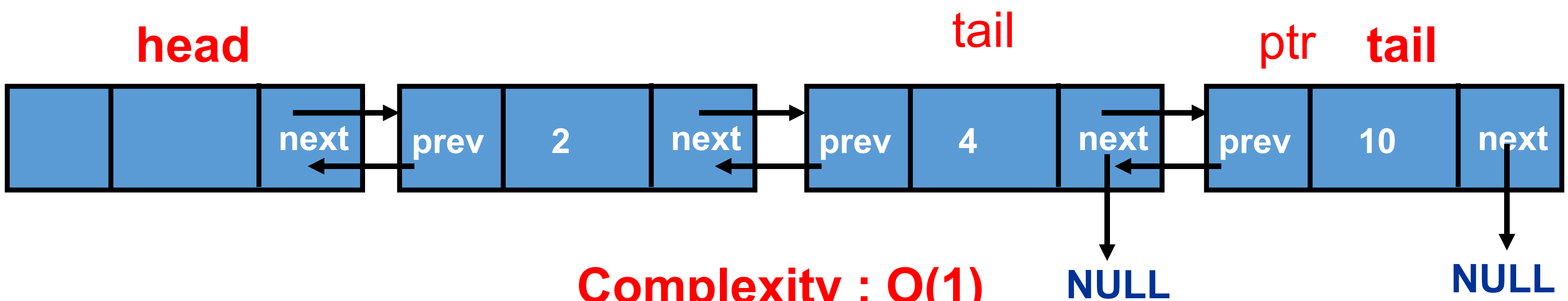
Since this new node is the **last node**, its next pointer is set to NULL.

Since the new node is now the **last node in the list**, tail is updated to point to ptr.

Double Linked List: Append a new element after the tail node

- Suppose you want to insert after the node with data = '*value*'

```
ptr = new node;
        cin>>ptr->data = // e.g ptr->data =10;
          if(value == tail->data ) {
               tail->next = ptr;
               ptr->prev = tail;
               ptr->next = NULL;
                tail = ptr;
      }
```

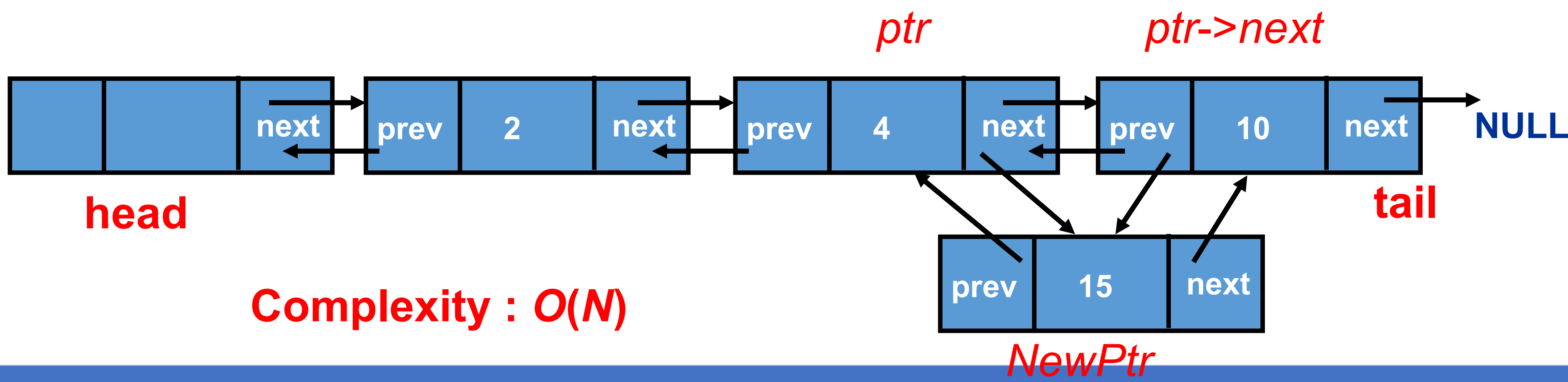


Complexity : O(1)

Double Linked List: Insert a new element after a specific node (other than the tail)

- Suppose you want to insert after the node with data = '*Aftervalue*'

```
node* NewPtr;
 for(node *ptr = head->next; ptr != NULL;ptr = ptr->next){
   if(ptr->data == AfterValue) {        e.g AfterValue =4
     NewPtr = new node;
     NewPtr->data = NewData;//      e.g NewData =15;
     NewPtr->next = ptr->next;
     (NewPtr->next)->prev = NewPtr;
     ptr->next = NewPtr;
     NewPtr->prev = ptr;
   } // end if
 } // end for
```

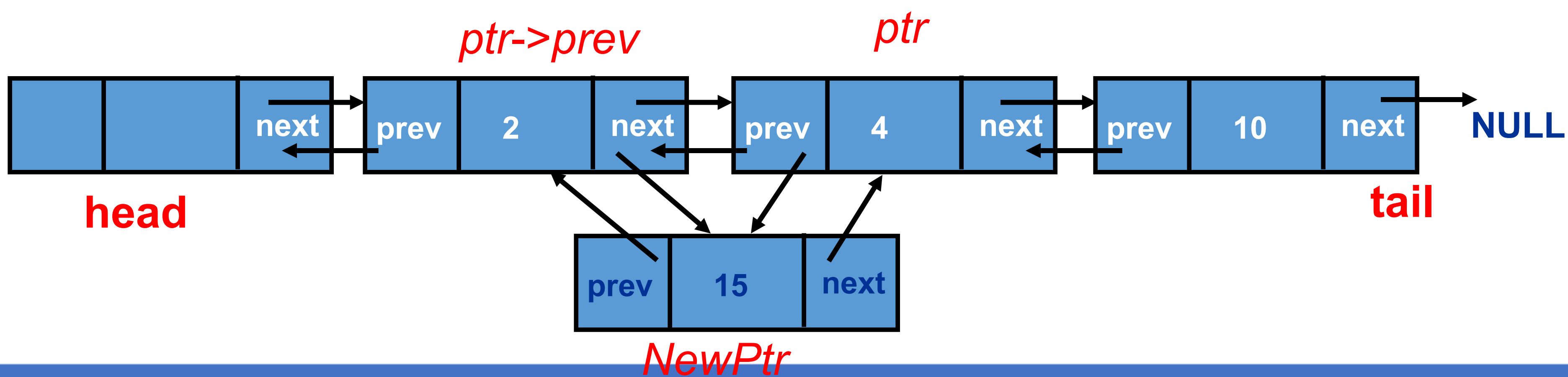


Complexity : *O*(*N*)

Double Linked List: Insert a new element before a specific node

- Suppose you want to insert after the node with data = ‘*Beforevalue*’

```
node* NewPtr;
 for(node *ptr = head->next; ptr != NULL;ptr = ptr->next){
   if(ptr->data == BeforeValue) {        e.g BeforeValue =4
     NewPtr = new node;
     NewPtr->data = NewData;//      e.g NewData =15;
     NewPtr->next = ptr;
     (ptr->prev)->next = NewPtr;
     NewPtr->prev = ptr->prev;
     ptr->prev = NewPtr;
     } // end if
  } // end for
```

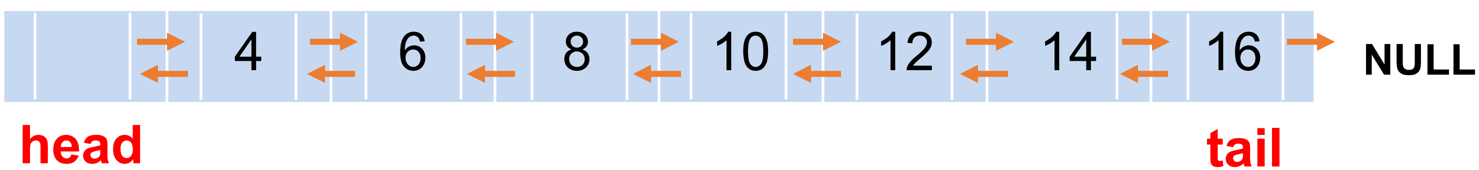


Complexity : O(*N*)

Double Linked List: Traverse through the linked list

Visit each element of the list
- e.g., print each element of the list on the screen

List data




```
for (node *ptr = head->next; ptr != NULL; ptr = ptr->next)
            cout<< ptr->data<<" ";
```

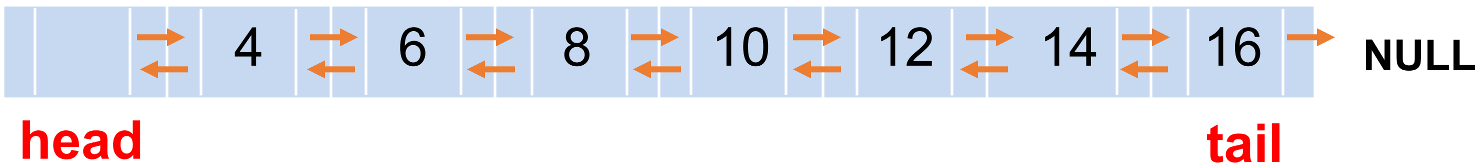
OUTPUT: 4 6 8 10 12 14 16

Complexity : O(*N*)

Double Linked List: Traverse through the linked list in the reverse order

- Visit each element of the list, starting for the last element, in the reverse order
  - e.g., print each element of the list, in the reverse order, on the screen

List data




```
for (node *ptr = tail; ptr != head; ptr = ptr->prev)
  cout<< ptr->data<<" ";
```

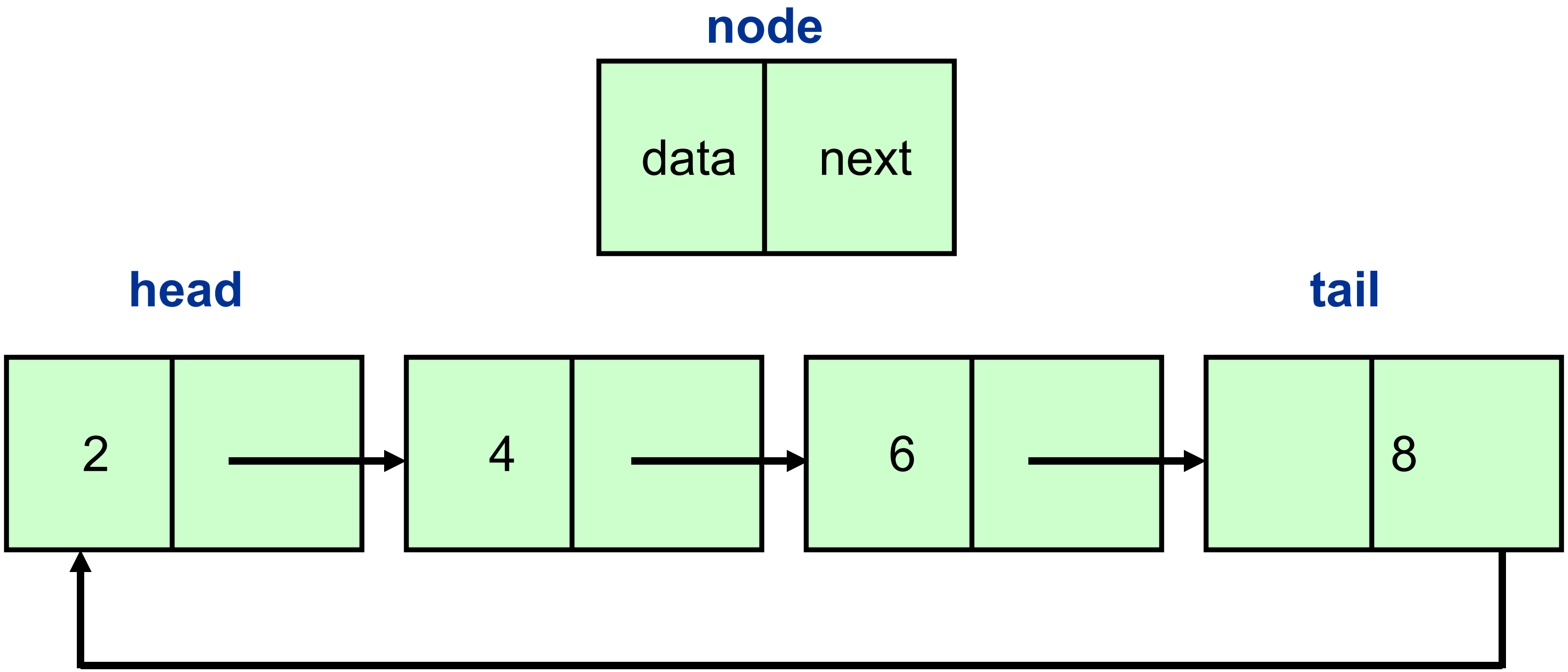
OUTPUT: 16 14 12 10 8 6 4

Complexity : O(*N*)

Question

- Can we traverse/print the elements of the a simple linked list (not the double linked) in the reverse order?

Circler Linked List



Circler Linked List

A **circular linked list (CLL)** is a variation of a linked list where:

- **The last node points back to the first node**, forming a **circular structure**.
- Unlike a normal linked list (which ends in NULL), a **circular linked list never ends**.

Circler Linked List

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

node *head, *tail;

void main() {
    head = new node;
    head->data = 10;
    head->next = head;
    head->prev = head;
    tail = head;
}
```

Summary

- Comparisons of
  - array implementation of static lists
  - array implementation of dynamic lists (dynamic array)
  - single linked list
  - double linked list
  - circular linked list

Summary

- Array implementation of static list vs dynamic list

Static array	Dynamic array
Need to know the memory requirements/ size of the list data in advance	Memory requirements/ size of the list can be adjusted at runtime
Very easy to handle and assign memory (at the beginning)	Complicated runtime memory reallocation procedure
Memory assigned in a single chunk at compile time	Memory assigned in multiple chunks at runtime
Insert/delete operations in the worst case scenario require moving/shifting the whole array elements	Insert/delete operations in the worst case scenario require moving/shifting the whole array elements

Summary

- Dynamic array vs dynamic linked list

Dynamic array	Dynamic linked list
Memory requirements/ size of the list can be adjusted at runtime	Memory requirements/ size of the list can be adjusted at runtime
Memory allocation may not be efficient: memory assigned in multiple chunks at runtime	Memory allocation is more efficient: memory assigned at runtime, one node at a time
Require contiguous memory allocation: frequent insertion/deletion operations of large size may cause fragmentation problem	Effectively addresses the fragmentation problem: the allocated memory is scattered around the available memory space(RAM)
Complicated runtime memory reallocation procedure : need to copy previous data into new memory for each memory re-allocation	Simple runtime memory reallocation procedure: Only the new node is assigned new memory.
Complicated Insert/delete procedures: Insert/delete operations in the worst case scenario require moving/shifting the whole array elements	Simple Insert/delete procedures: Insert/delete operations only require the knowledge/address of one/two nodes
Allow constant-time random access	• Allow only sequential access to elements • Singly linked lists can only be traversed in one direction. • This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly

Summary

- Single linked list vs Double linked list

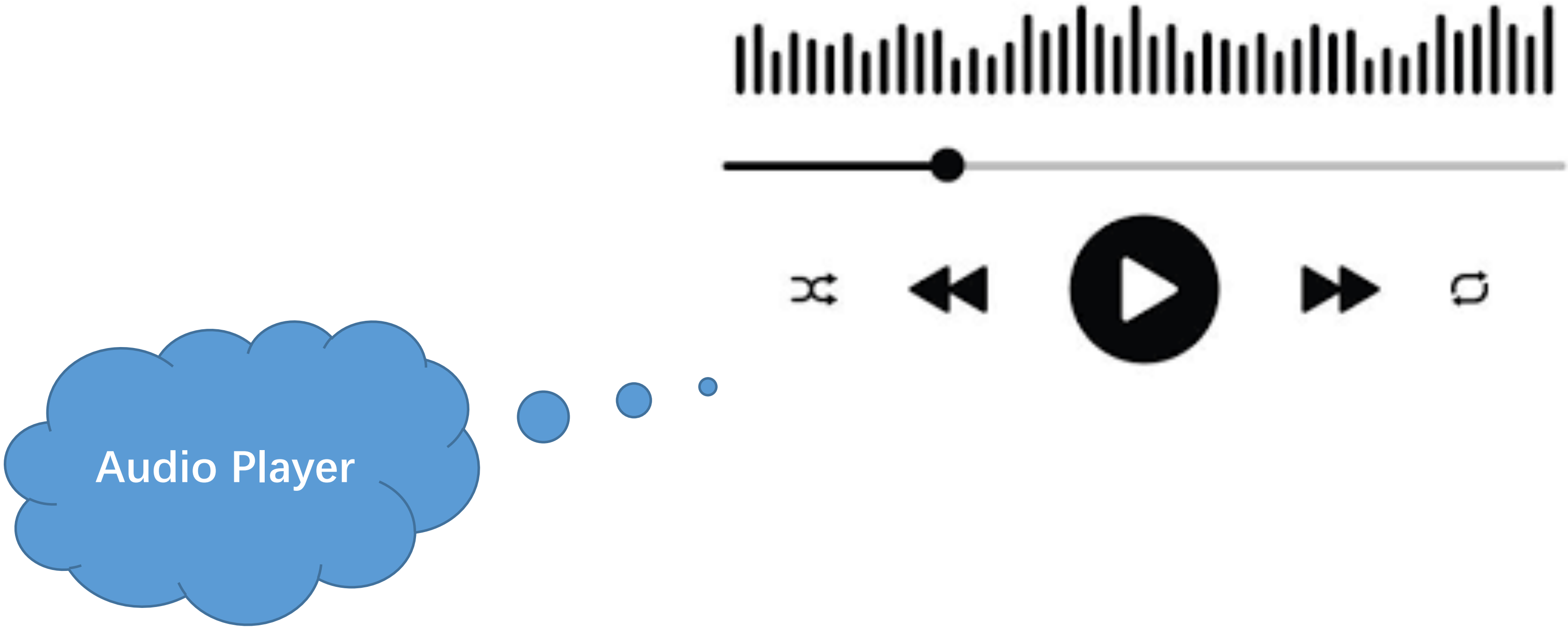
Single linked list	Double linked list
Allow only sequential access to elements	• Allow only sequential access to elements
Singly linked lists can only be traversed in one direction.	Doubly linked lists can be traversed in both directions
Insert/delete operations only require the knowledge/address of one/two nodes	Insert/delete operations require the address of only one node: i.e. the node after/before to insert OR node to be deleted

Summary

- simple linked list vs circular linked list

Simple linked list	circular linked list
Requires a NULL pointer implementation (The last nodes points to the NULL)	Simple implementation: no NULL pointers at the ends; no need to check whether pointers are NULL.
The program may crash: due to the presence of NULL pointer, a reference to the NULL pointer is possible	Safe programming practice: a node always has a non-NULL predecessor and successor, and this means that you can always safely dereference its previous and next pointers
May require special handling/different implementation of add-head, add-tail, insert-before and insert-after functions	Simple Implementation: all of add-head, add-tail, insert-before and insert-after functions, can be implemented through a single function
May require special handling/different implementation of delete-head, delete-tail, delete-anywhere functions	Simple Implementation: All of delete-head, delete-tail, delete-anywhere functions, can be implemented through a single function
Merging /Splitting operations of the linked lists are not so simple	A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. • The operation consists in swapping the contents of the link fields of those two nodes. • Applying the same operation to any two nodes in two distinct lists joins the two list into one

Linked List Applications



Linked List Applications

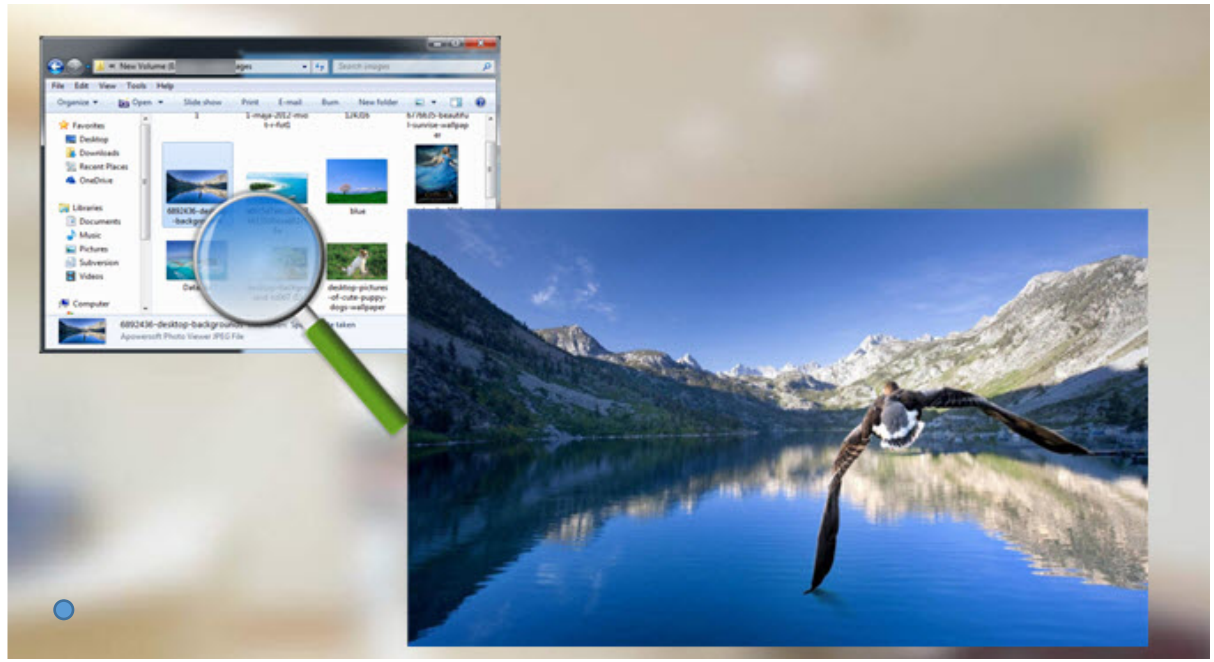


Image Viewer

Questions?

zahmaad.github.io