Data Structure & Algorithms

Lec 02: Linked Lists

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. Motivation & Background
2. Linked List Definition & Operations
3. Types of Linked-Lists
4. Linked List ADT Operations
5. Self Assessment

Learning Outcome

By the end of this class, you should

- Know the difference between arrays and linked lists
- Know pros and cons of linked lists and arrays
- Explain the different types of linked list
- Describe the basic operations on ADT linked list.

Think Carefully

Consider the following programing tasks:

> *Write a computer program in C/C++ that reads GPA of 100 students and return the maximum, minimum, and the average GPA for all the students.*
>
> *Write a computer program in C/C++ that reads GPA of 100 students and sort them in descending order.*

form a data structure point of view, what is the main difference between the two programming tasks above?

Arrays Again!

Instead of declaring individual variables, such as GPA variable for each student you just declare one array variable GPAs of float type and use GPAs[1], GPAs[2], ..., GPAs[100] to represent individual students GPAs

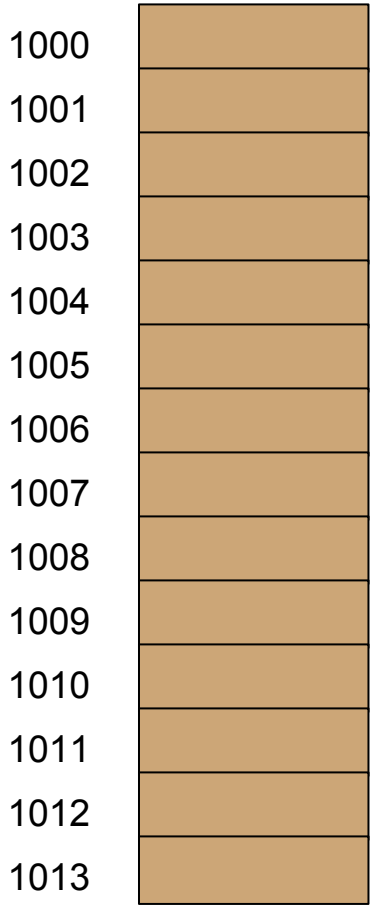
All arrays consist of contiguous memory locations.

What are the advantages of arrays?

- Arrays are simple and easy to use.
- Faster access time to the element O(1).

Now consider the computer memory on the right side could we create the following arrays?

- char Arr1[4];
- char Arr2[5];
- char Arr3[4];



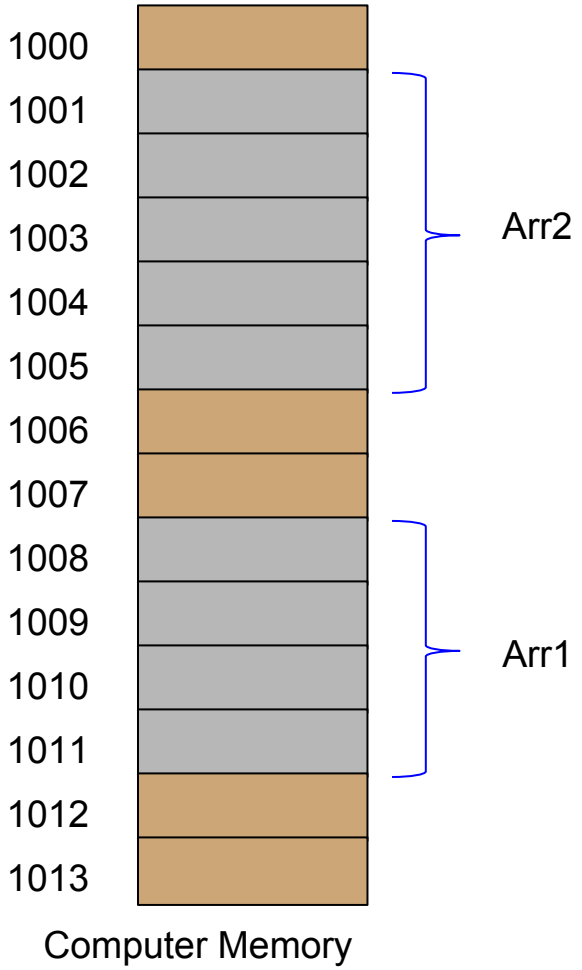
Arrays Again!

While arrays are simple and easy to use they are inefficient when it comes to allocating memory blocks.

What are the disadvantages of arrays?

- Pre-allocates memory up front (why is that bad thing?)
- Fixed Size.
- Contiguous memory allocation
- Insertion and deletion are in general expensive.

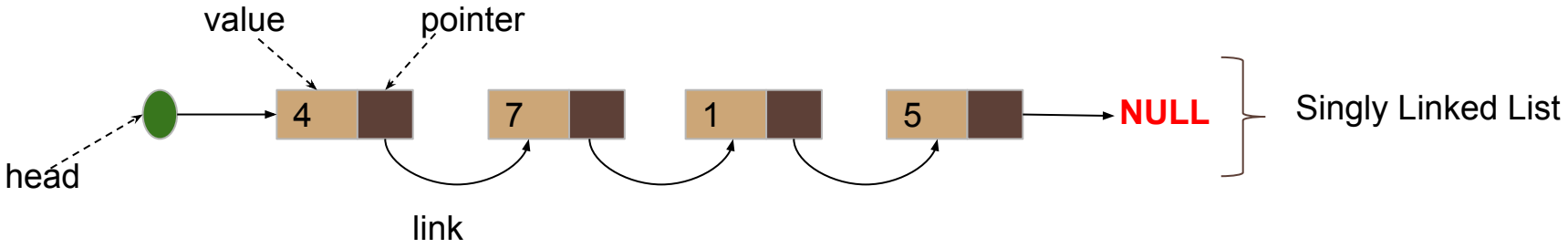
Could we use ***dynamic arrays***? No we need an efficient linear data structure



Linked List

What is a Linked List?

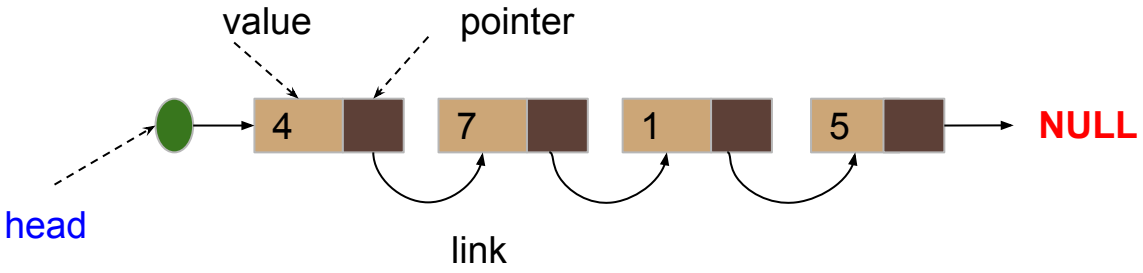
- Elements are located in a linear order.
- A collection of nodes where each node has a value and points to the next node in the list.

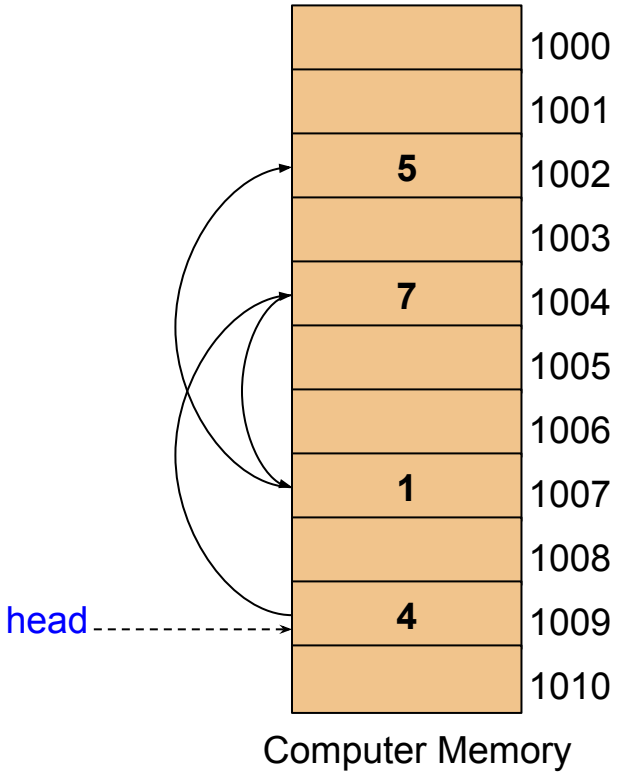


Linked List Properties

- Pointers connect successive elements.
- A Linked list has a dynamic size. It can grow or shrink during execution time.
- It can grow as much as we want or until the system run-out of memory.
- It does not waste memory space (really!!)
- The pointer of the last element in the list points to null
- It require a special pointer "head" that always point to the beginning of the list

How a Linked List stored in memory?





Advantages & Disadvantage of Linked Lists

Advantages:

- Memory allocation is dynamic the list could grow or shrink during execution time. In other words, preallocation is not required.
- An element can always be added to the list as long as there is enough memory to store it.

Disadvantages:

- Only provide *sequential access* to list elements *O(n)*
- Takes more memory to store pointer and reference information

Linked List as ADT

Linked List ADT = Linked List + Linked List Operations

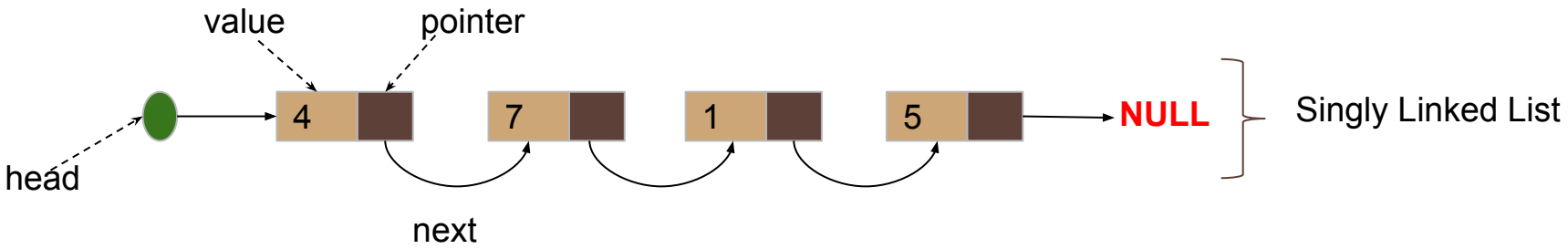
Linked List Operations:

1. Traverse: visit each node (element) in the list in a sequential order.
2. Insert: add new node to the list
3. Delete: delete an existing node from the list
4. Search: find the index of a node with given data value
5. Sort: sort the nodes in the list based on some order
6. Merge: combine two or more list into one list

Types Of Linked List: Singly Linked List

Singly Linked List: A linked list where each node contains a pointer (next) pointing to the next node in the list

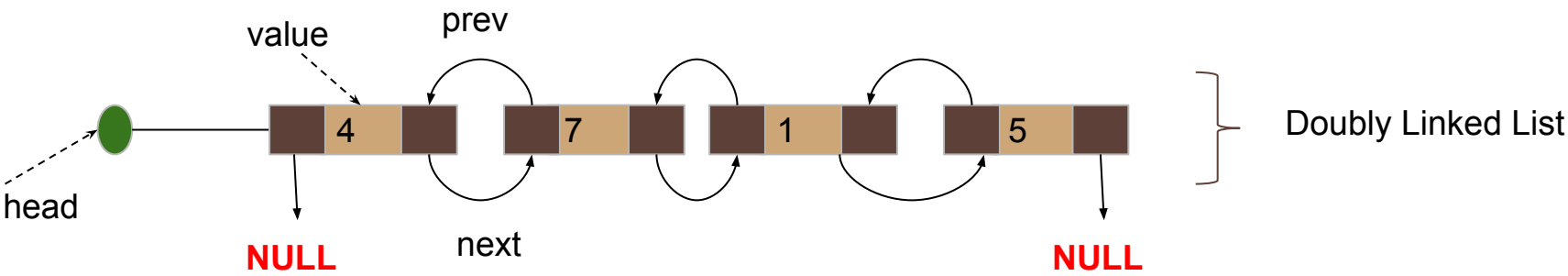
```
struct SinglyListNode {
    int data;
    struct SinglyListNode * next;
}
```



Types Of Linked Lists: Doubly Linked List

Doubly Linked List: A linked list where each node contains a pointer (next) pointing to the next node in the list and a second pointer (prev) pointing to previous node

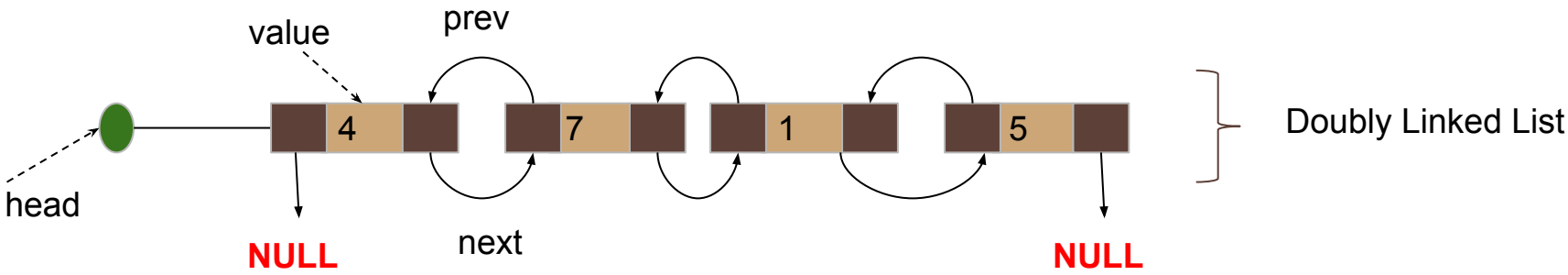
```
struct DoublyListNode {
    int data;
    struct DoublyListNode * next;
    struct DoublyListNode * prev;
}
```



Types Of Linked Lists: Doubly Linked List

Advantages: From any node in the middle of the list we can navigate in any direction.

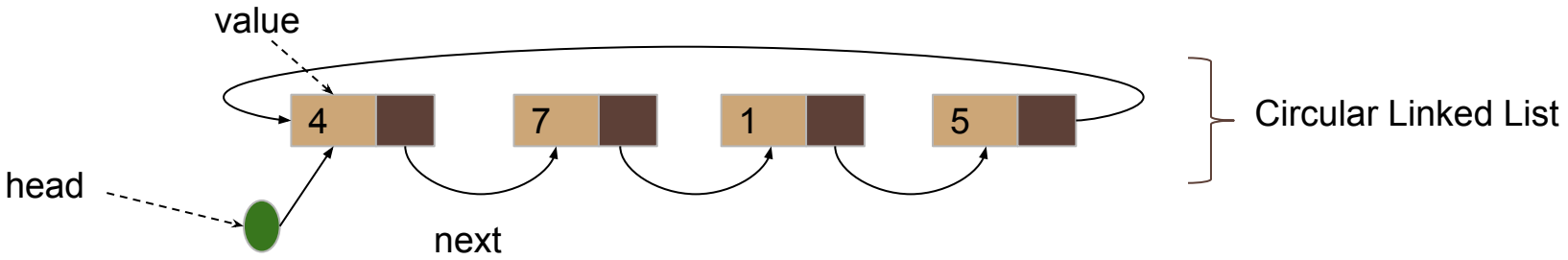
Disadvantages: Each node require extra memory space because the additional pointer. In addition, there are more pointer operations when inserting or deleting a node from the list



Types Of Linked List: Circular Linked List

Circular Linked List: A linked list where each node contains a pointer (next) pointing to the next node in the list. The pointer of the last node in the list point to the head node (first node in the list)

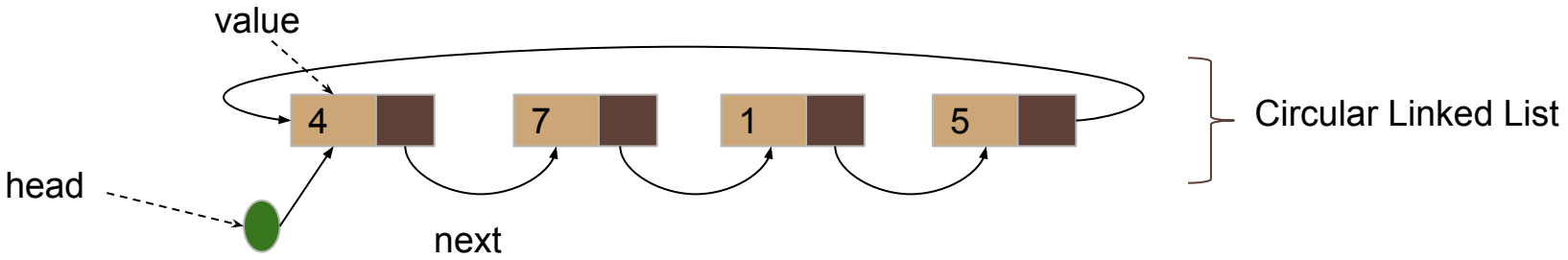
```
struct CircularListNode {
    int data;
    struct CircularListNode * next;
}
```



Types Of Linked List: Circular Linked List

Circular Linked List: A linked list where each node contains a pointer (next) pointing to the next node in the list. The pointer of the last node in the list point to the head node (first node in the list)

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

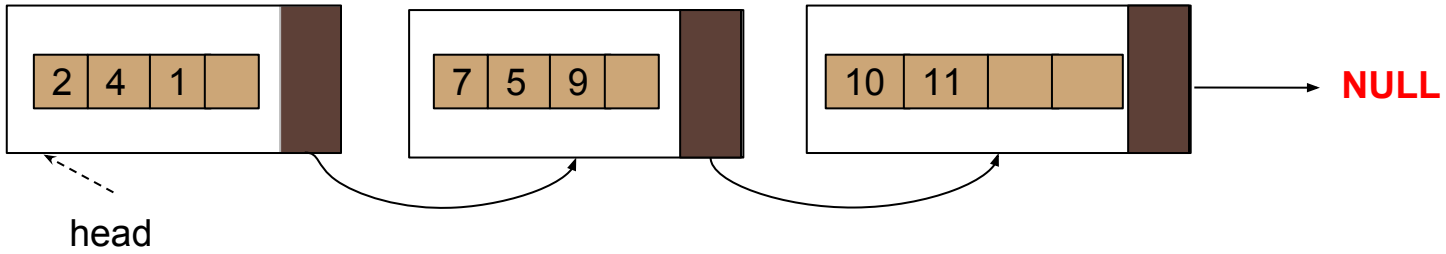


Types of Linked Lists: Skip List and Unrolled List

Unrolled Linked List: store multiple elements in each node. Improve the element access time.

Each node or block has an ID and store a range of elements

```
struct UnrolledListNode {
    int data;
    int elements [4];
    struct UnrolledListNode * next;
    int  blockID;
}
```

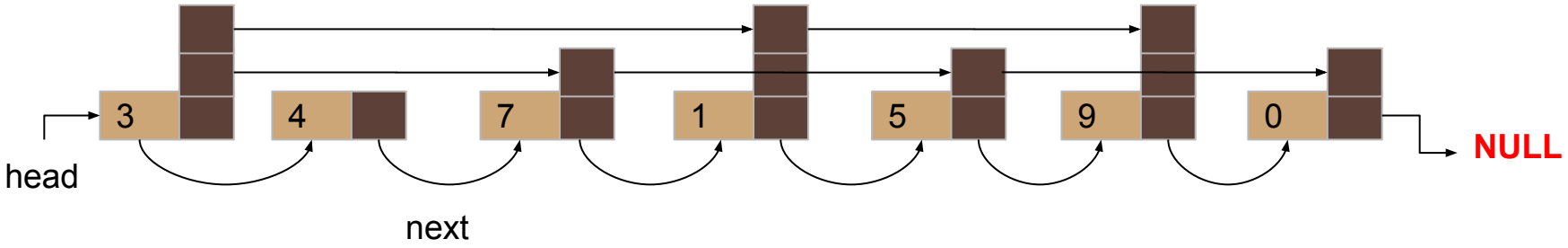


Types of Linked Lists: Skip List and Unrolled List

Skip Linked List: is built in layers. The bottom layer is a regular linked list. The each higher layer provide the ability to scan the list in bigger steps, so we can jump and skip elements

```
// what else we need to add to the
// struct to create a Skip List
struct SkipListNode {
        int data;
        struct ListNode * next;
        ???

}
```



Linked List ADT Operations: Traversing

Traversing a Linked List

1. Start from the head pointer
2. Follow the next pointers
3. Process every node as required (e.g. print the node date, count the size of the node, find the node with maximum data value, etc)
4. Stop when the pointer next points to NULL.

The time complexity is ***O(n)***

Linked List ADT Operations: Traversing (cont…)

Traversing a Linked List

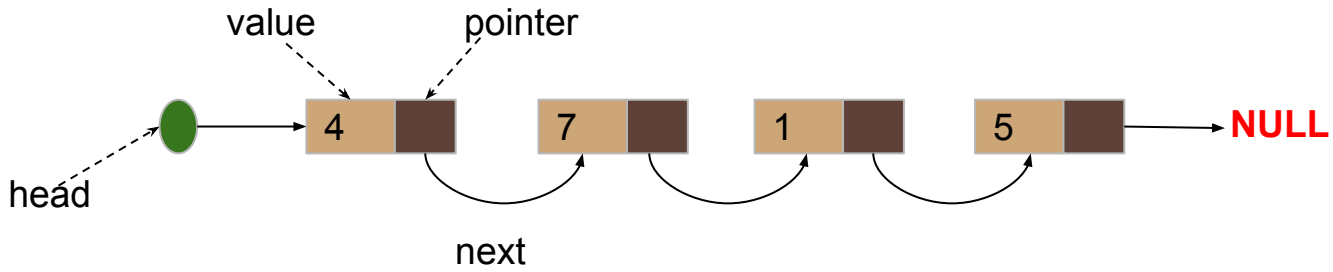
```
void printListData(struct SinglyListNode * head) {
        struct SinglyListNode * current = head;
        while(current != NULL) {
                cout<< current→data <<endl;
                current = current → next;
        }
}
```

How do we traverse a circular linked list?

Linked List ADT Operations: Inserting

When we are inserting a node into a linked list we have three cases:

1. Inserting a node at the beginning of the list
2. Inserting a node at the end of the list
3. Inserting a node at the middle of the list



Linked List ADT Operations: Inserting (cont...)

CASE 1: Inserting at the beginning:

1. Create the new node and set the data field
2. Temporary store the head pointer content in temp pointer
3. Set the next pointer of the new node to temp
4. Update the head pointer to point to the new node

What is the run time complexity of this step, is it O(n) or O(1)?

Linked List ADT Operations: Inserting (cont...)

CASE 1: Inserting at the beginning:

```
1.  struct SinglyListNode *temp, *newNode;
2.  newNode = (SinglyListNode *)malloc(sizeof(struct SinglyListNode));
3.  newNode → data = v;
4.  temp = *head;
5.  If (index == 1){
6.     newNode → temp;
7.     *head = newNode;
8.  }
```

Linked List ADT Operations: Inserting (cont...)

CASE 2: Inserting at the end:

1. Starting from the head pointer (first node) follow the pointers until you reach the last node (next = NULL)
2. Set the next pointer of the new node to NULL
3. Set the next pointer of the last node to points to the new node

What is the run time complexity of this step, is it O(n) or O(1)?

Linked List ADT Operations: Inserting (cont...)

CASE 3: Inserting at the middle:

1. Starting from the head pointer (first node) follow the pointers until you reach the last node (next = NULL)
2. To add an element at position ***i*** we need to stop at position ***i-1*** we will call the node at i-1 the base node.
3. Set the next pointer of the new node to point to the base node next pointer.
4. The based node next pointer points to the the new node.

What is the run time complexity of this step, is it O(n) or O(1)?

Linked List ADT Operations: Inserting (cont...)

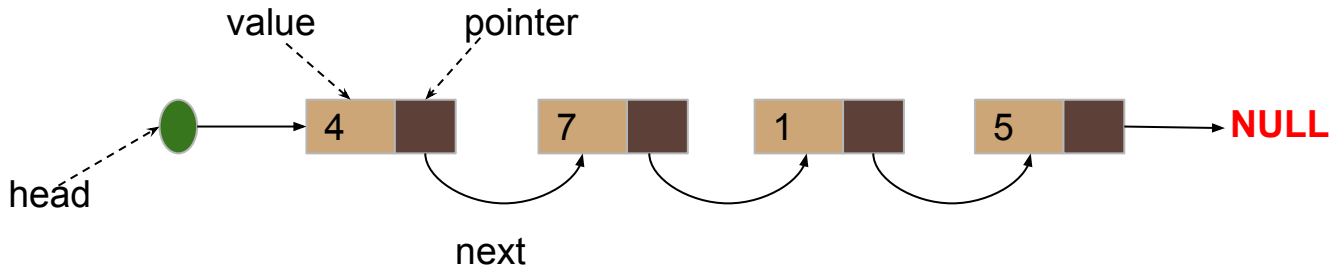
Inserting at the middle or at the end:

```
1.  struct SinglyListNode *current, *newNode, *temp;
2.  while (current!= NULL && cindex < index){
3.          cindex ++;
4.          temp = current;
5.          current = current→ next;
6.  }
7.  temp→ next = newNode;
8.  newNode → next = current;
```

Linked List ADT Operations: Deleting

When we are deleting a node from a linked list we have three cases:

1. Delete from the beginning (first node)
2. Delete from the end (delete the last node)
3. Delete from the middle.



Linked List ADT Operations: Deleting (cont...)

CASE 1: Deleting at the beginning

1. Create a temp pointer to point the first node (head node)
2. Set the head pointer to the next pointer of the first node
3. Finally free the first node using the temp pointer (this step optional or required ??)

What is the run time complexity of this step, is it O(n) or O(1)?

Linked List ADT Operations: Deleting (cont...)

CASE 1: Deleting at the beginning

```
1.  struct SinglyListNode *temp;
2.  temp = *head;
3.  If (index == 1){
4.      *head = (*head) → next;
5.      free(temp);
6.  }
```

Linked List ADT Operations: Deleting (cont...)

CASE 2: Deleting at the end

1. Starting from the beginning of the list follow the next pointer.
2. Maintain two pointer when point to the current node and one point to the previous node.
3. When the next point of the current node is NULL, set the next pointer of the previous node to NULL.
4. Free the current node.

Linked List ADT Operations: Deleting (cont...)

CASE 2: Deleting at the end

1. Starting from the beginning of the list follow the next pointer.
2. Maintain two pointer when point to the current node and one point to the previous node.
3. When the index of current node match the index we want to delete, set the next pointer of the previous node to next pointer of the current node.
4. Free the current node.

What is the run time complexity of this step, is it O(n) or O(1)?

Linked List ADT Operations: Deleting (cont...)

Deleting a node at the end or the middle of the list

```
1.  struct SinglyListNode *current, *prev;
2.  while (current!= NULL && cindex < index){
3.          cindex++;
4.          prev = current;
5.          current = current → next;
6.  }
7.  prev→ next = current→ next;
8.  free(current);
```

What is missing here?

Time Complexity Arrays vs. Linked List

Operation	Array	Linked List
Access an Element (indexing)	***O(1)***	***O(n)***
Insert at the beginning	***O(n)***	***O(1)***
Insert at the end	***O(1)***	***O(n)***
Insert at the middle	***O(n)***	***O(n)***
Delete at the beginning	***O(n)***	***O(1)***
Delete at the end	***O(1)***	***O(n)***
Insert at the middle	***O(n)***	***O(n)***

self-assessment

1. When to use an array and when to use a linked list?
2. The memory storage required to store an array of 100 integer values equal to the memory storage required to store them in a linked list (TRUE | FALSE)
3. We can find an element in a sorted array in O(log n), could we say the same for a sorted linked list?
4. Write an algorithm to find the nth from the end of a linked list.
5. Write an algorithm to traverse a circular linked list.