

# CSC230

# Outline



- **Singly Linked List**
- **Lab 5 discussion**

# Review: Arrays & ArrayList

## Benefits of Arrays

- Random access with index numbers
  - `int MyID = MyArray[2]`
- No waste in memory
  - The element stores as itself
- Fast sequential access
  - The elements are stored continuously in memory

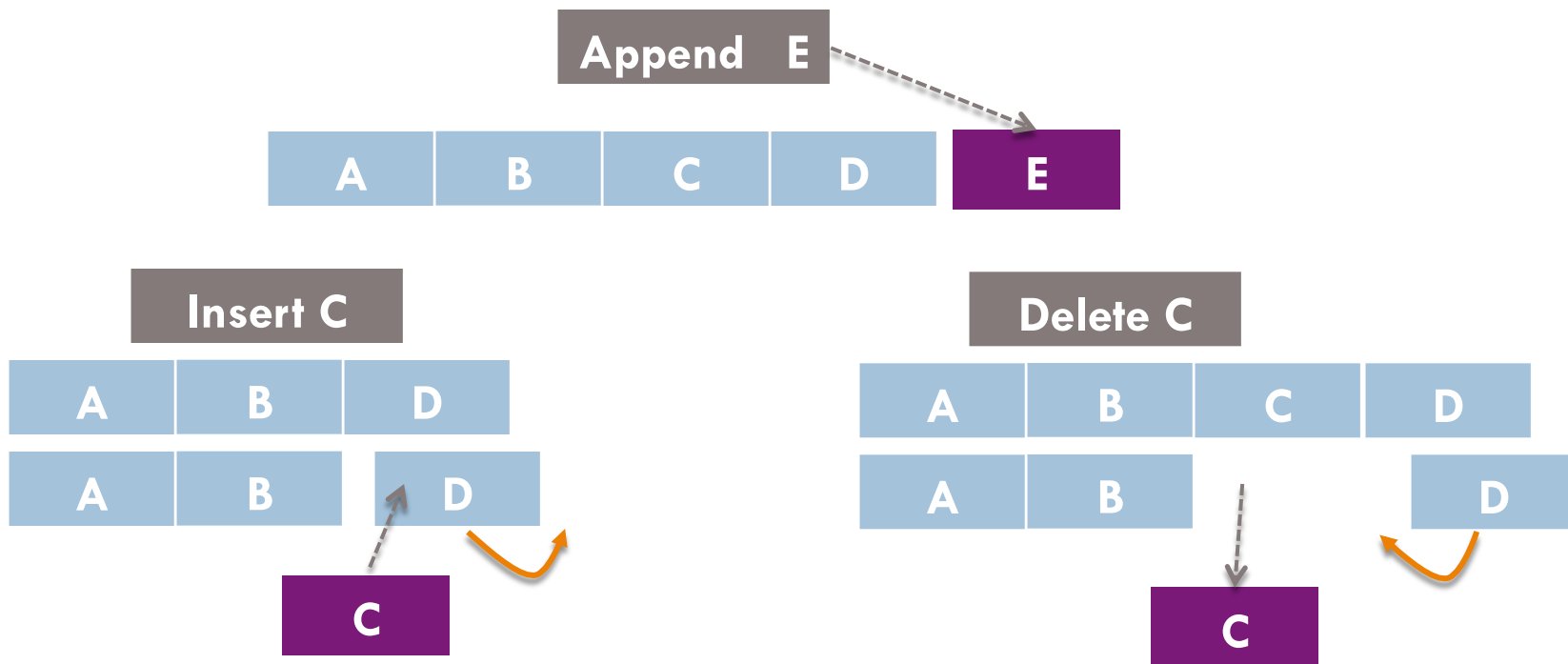
## Limitations of Arrays

- Need to know the size beforehand
  - `int MyID[5]`
- Can not change the size at run-time
  - Array is static data structure, fixed at compile time
- Difficult to insert / delete element
  - Insertion and deletion result in element shifting

# Review: Arrays & ArrayList

## Resizing Arrays : ArrayList

- ArrayList allows run-time resizing
- It is efficient to append an object at the end
- It enables insertion and deletion, however, elements still need to be shifted



# Alternative: Linked list

5

- The size of linked list can **easily grow** or **shrink** based on the number of items currently in the list
- Usually, **arrays** are allocated and de-allocated by **large chunk**
- Usually, **Linked lists** grow or shrink by **small chunk**.



Single item  
(linked list)

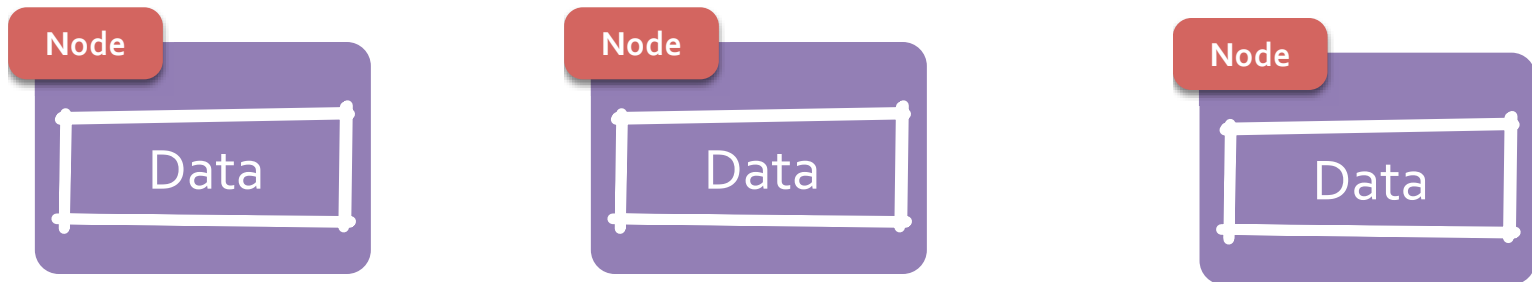


Bulk item  
(array)

# Introduction to Linked List

## What is Singly Linked List ?

- A singly linked list is a concrete data structure consisting of a **sequence of nodes**
- Is it a linked list ?



**It is a list of nodes**

**NOT a linked list of nodes**

# Linked List: a class implementation

7

- The data type of the item/node can be **structure or class**, depending on whether there is any operation associated with the item.
- The items are linked by **pointers**.

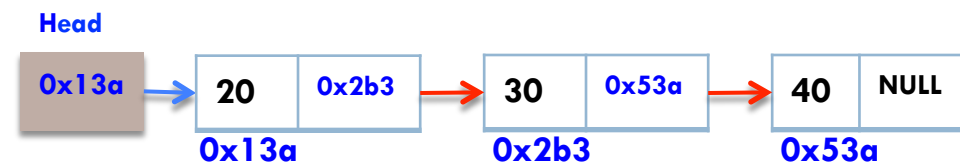
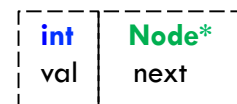
A list is

- **Arbitrarily** sized
- Can add any number of new values by dynamic memory allocation
- Supports typical list operations:
  - Append
  - At
  - Remove
  - Size
  - Empty
- Can define a List class

```
#include<iostream> using namespace std;
struct Node{
    int val;
    Node* next;
};

class List
{
public: List();
    ~List();
    void append(int v); ... private:
    Node* head;
};
```

Item :



# Linked List: without a class

8

To access a linked list, you need the address of the **some node/item**, which is pointed by a **pointer**.

Starting from the **head pointer**, we can access every node of the linked list.


- A **class** acts as a **wrapper** around the header pointer and the related operations.
- It is easy for user to manipulate the list.

If list is implemented without a class, each function will use **header pointer** as an argument.

```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node*& head, int v);

int main(){
    Node* head1 = NULL;
    Node* head2 = NULL;
}
```



Head is a **reference** (alias) to a **Node pointer** (address)



# Linked List: without a class

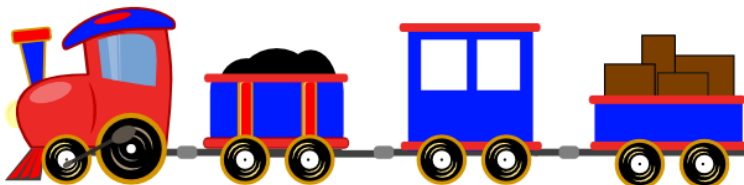
9

To access a linked list, you need the address of the **some node/item**, which is pointed by a **pointer**.

Starting from the **head pointer**, we can access every node of the linked list.

What if?

**Node \*ptr** You can't reassign a reference



Head      Node      Node      Node

```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node* &ptr, int v);

int main(){
    Node* headPtr1 = NULL;
    Node* headPtr2 = NULL;
}
```

ptr is a **reference**  
(alias) to a **Node**  
**pointer**

# Append a node to the linked list

10

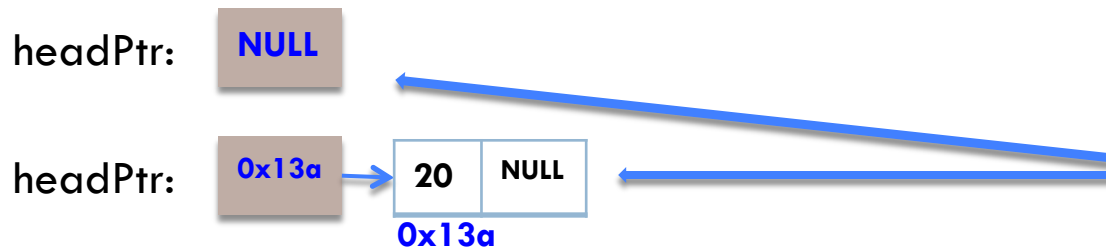
Where can we insert an element ?

- the **beginning** of the list,
- or the **end** of the list,
- or somewhere in the **middle** of the list.

Adding a node to the **end** of the list (**append**). There is two cases:

- The current list is **empty**
- The current list is **not empty**

List has ONE node, headPtr is **not** part of the list itself.



```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node*& head, int v){
    if(head == NULL){
        head = new Node;
        head->val = v;
        head->next = NULL;
    }
    else{

    }
}

int main(){
    Node* headPtr = NULL;
    append(headPtr, 20);
}
```

# Append a node to the linked list

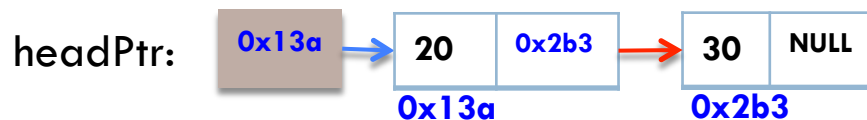
11

To **add** a new node to an existing linked list, you can add it to

- the **beginning** of the list,
- or the **end** of the list,
- or somewhere in the **middle** of the list.

Adding a node to the **end** of the list (**append**). There is two cases:

- The current list is **empty**
- **The current list is not empty**



List has TWO nodes, headPtr is **not** part of the list itself.

```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node*& head, int v){
    if(head == NULL){
        head = new Node;
        head->val = v;
        head->next = NULL;
    }
    else{
        ...
    }
}

int main(){
    Node* headPtr = NULL;
    append(headPtr, 20);
    append(headPtr, 30);
}
```

# Append a node to the linked list

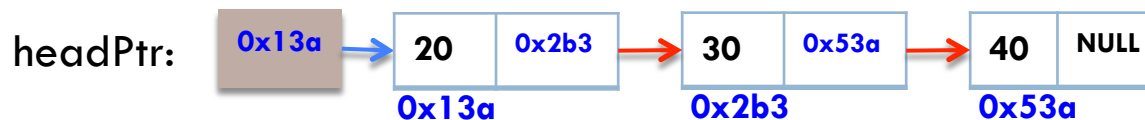
12

To **add** a new node to an existing linked list, you can add it to

- the **beginning** of the list,
- or the **end** of the list,
- or somewhere in the **middle** of the list.

Adding a node to the **end** of the list (**append**). There is two cases:

- The current list is **empty**
- **The current list is not empty**



List has THREE nodes, headPtr is **not** part of the list itself.

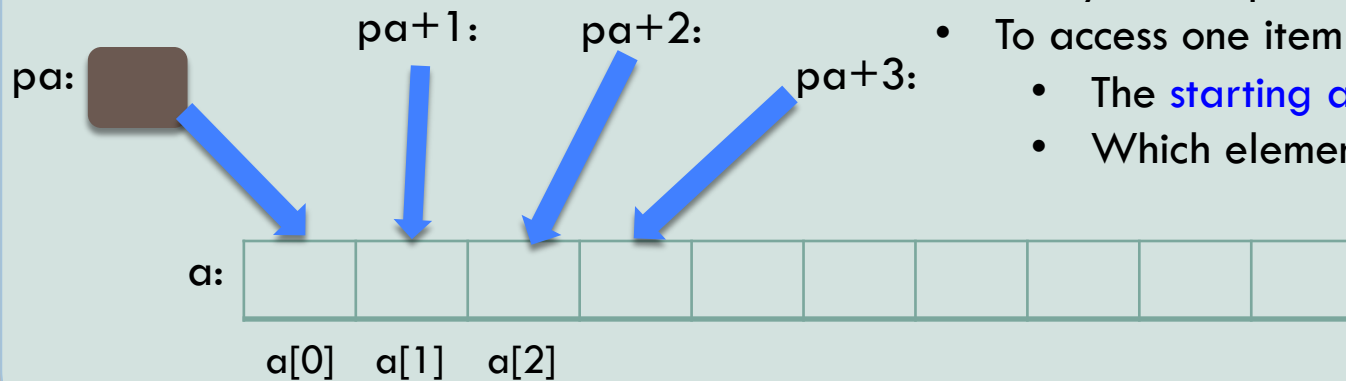
```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node*& head, int v){
    if(head == NULL){
        head = new Node;
        head->val = v;
        head->next = NULL;
    }
    else{
        ...
    }
}

int main(){
    Node* headPtr = NULL;
    append(headPtr, 20);
    append(headPtr, 30);
    append(headPtr, 40);
}
```

# Array vs. Linked List

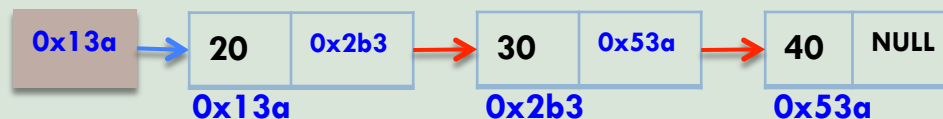
13



- Array uses a piece of **contiguous memory**.
- To access one item in an array, you need
  - The **starting address** (array name)
  - Which element we want (the index value)

`a[6] = 10;`

- Linked list is **NOT contiguous in memory**.
- The address of each node is explicitly stored.
- To access one node in a list, you need to
  - Have the **starting address**
  - **Iterate** the nodes before reaching the end or the intermediate node

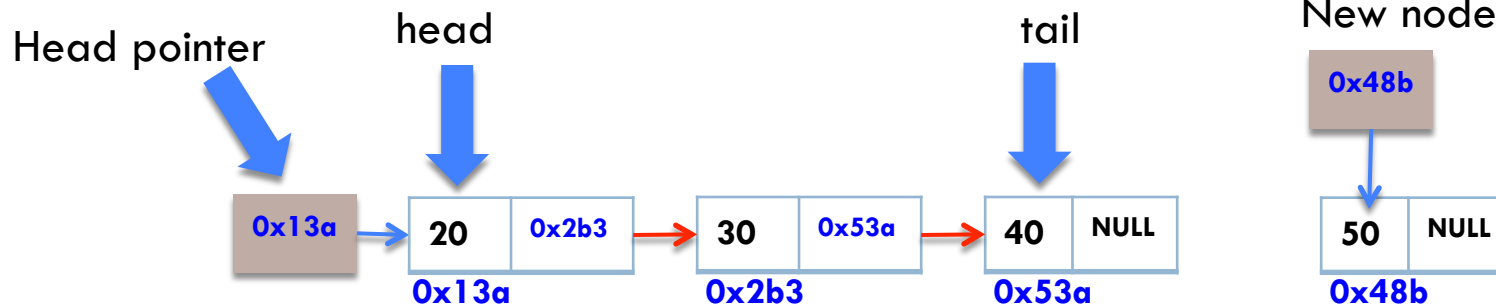


# Append()

14

Start from head and iterate to the end of the list.

- Create a new node, initialize it.
- Copy head pointer to temp pointer
- Use temp pointer to iterate through the list until reaching the tail
- Attach the new node to the tail

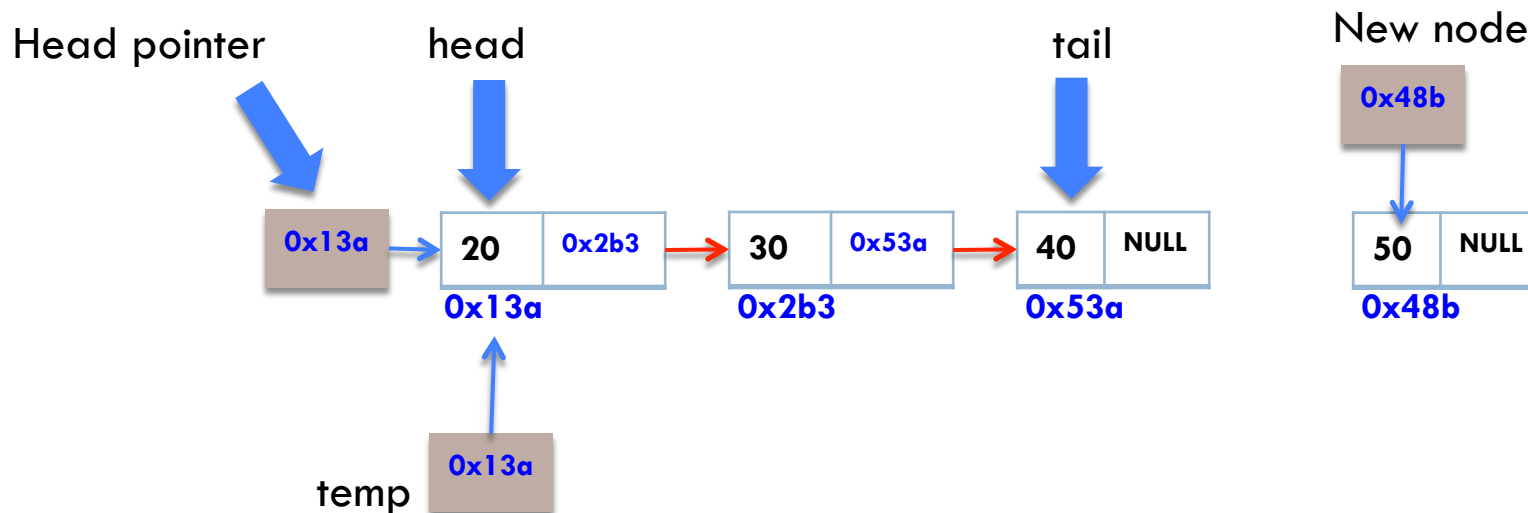


# Append()

15

Start from head and iterate to the end of the list.

- Create a new node, initialize it.
- Copy head pointer to temp pointer
- Use temp pointer to iterate through the list until reaching the tail
- Attach the new node to the tail

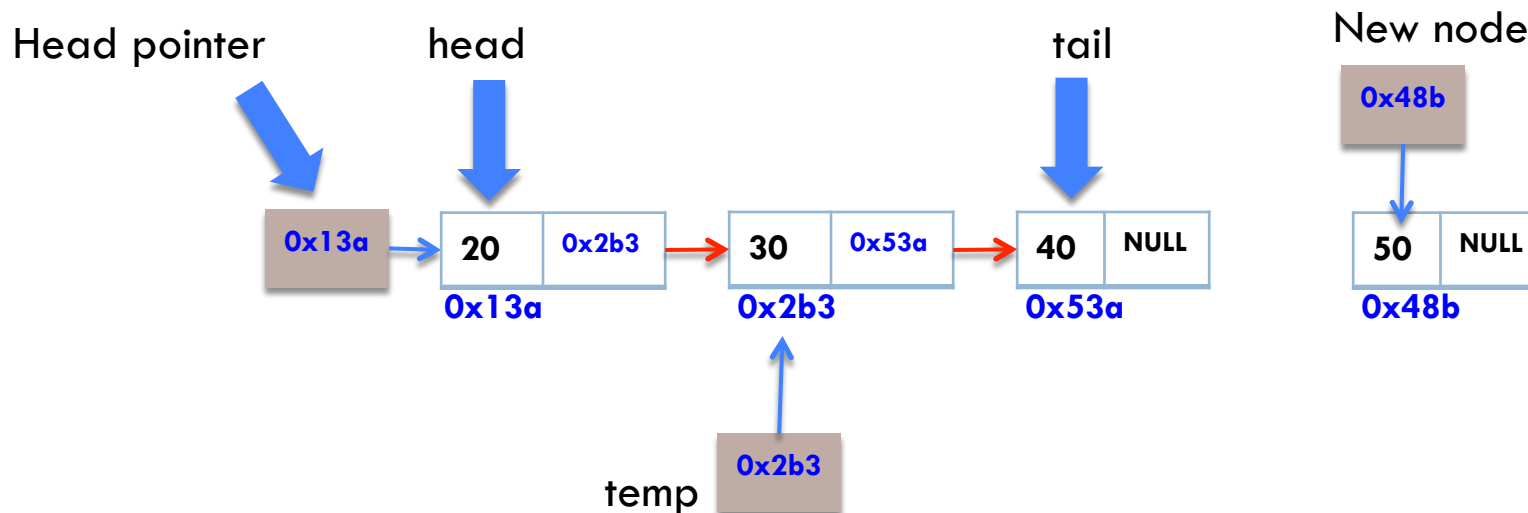


# Append()

16

Start from head and iterate to the end of the list.

- Create a new node, initialize it.
- Copy head pointer to temp pointer
- Use temp pointer to iterate through the list until reaching the tail
- Attach the new node to the tail



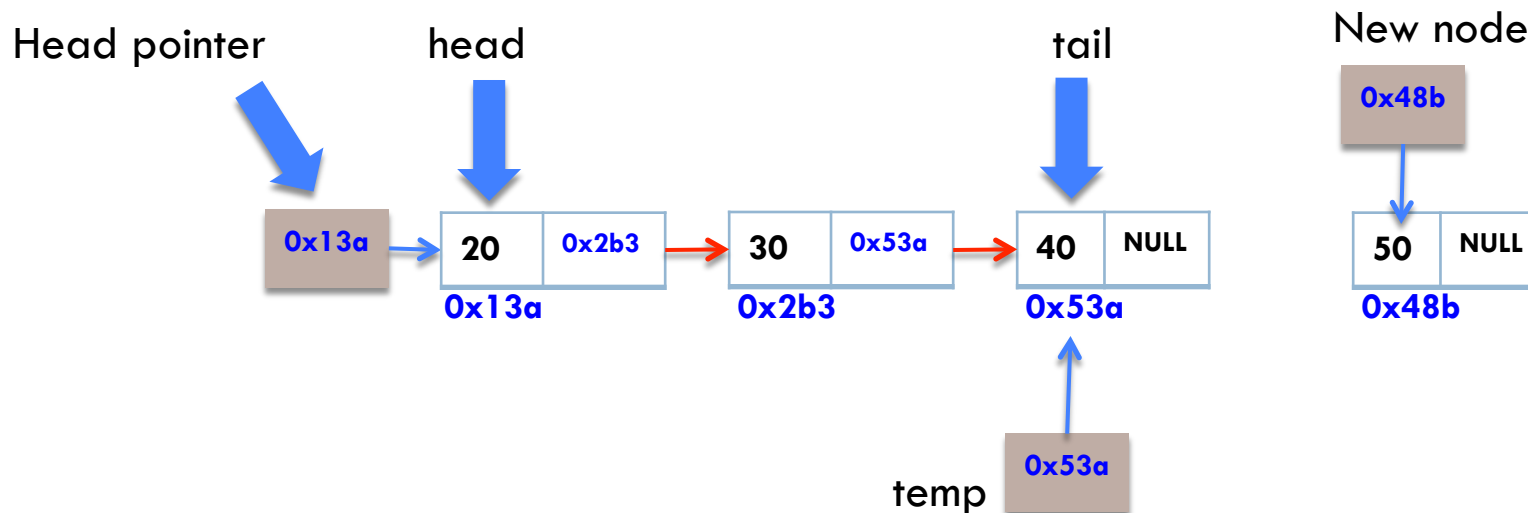


# Append()

17

Start from head and iterate to the end of the list.

- Create a new node, initialize it.
- Copy head pointer to temp pointer
- Use temp pointer to iterate through the list until reaching the tail
- Attach the new node to the tail

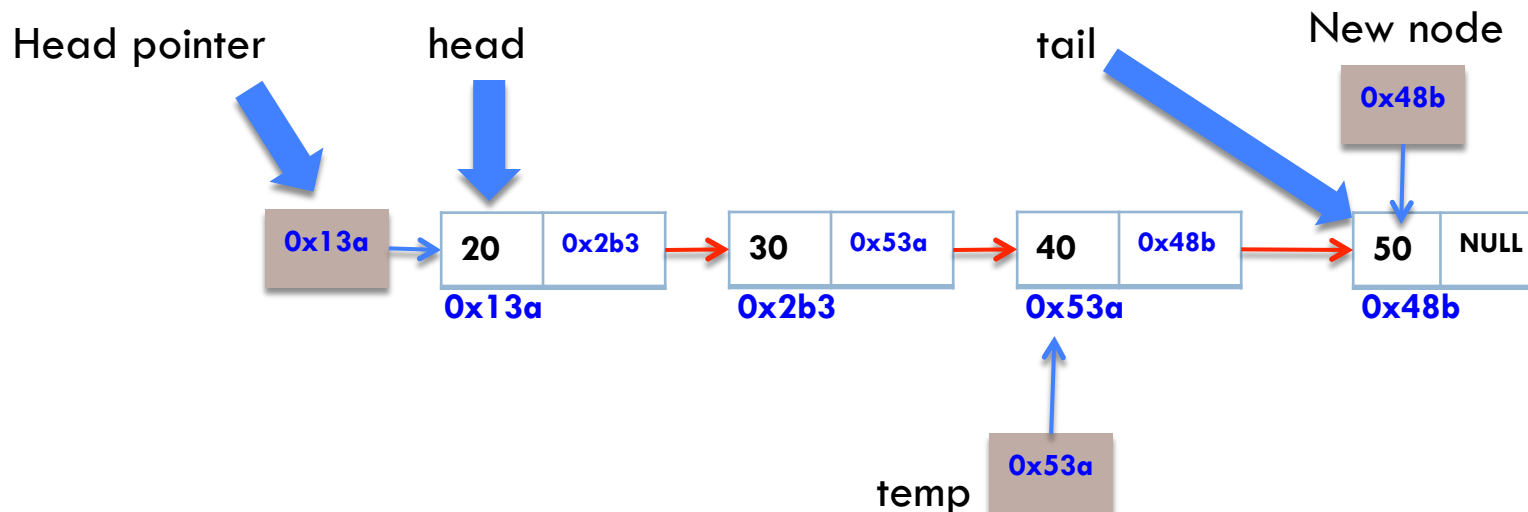


# Append()

18

Start from head and iterate to the end of the list.

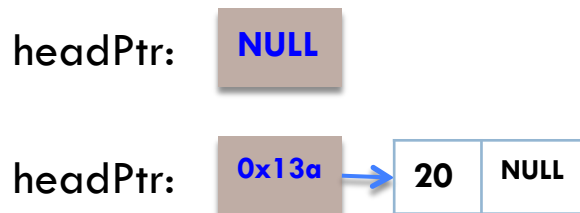
- Create a new node, initialize it.
- Copy head pointer to temp pointer
- Use temp pointer to iterate through the list until reaching the tail
- Attach the new node to the tail



# Append()

19

- We use **head pointer** to access the list
- The **head pointer** points to the **first node**
- If the **address** of the first node **changes**, the **head pointer** must points to the new address
- If a Node pointer is the parameter, the function will get the address of the first NODE, not head pointer itself.
- By using a reference to the Node pointer, we access head pointer



```
#include<iostream>
using namespace std;
struct Node{
    int val;
    Node* next;
};

void append(Node*& ptr, int v);

int main(){
    Node* headPtr1 = NULL;
    Node* headPtr2 = NULL;
}
```

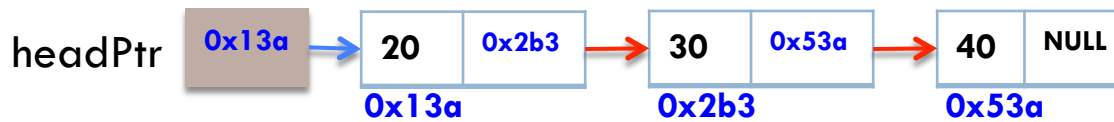


What if?  
**Node \*ptr**

ptr is a **reference**  
(alias) to a **Node**  
**pointer**

# Head pointer

20



- Head pointer is **NOT** a node
- Head pointer is a **variable**, which points to the first node
- What does `headPtr->next` points to ?
  - not point to the first node, it points to the second node.

# Append()

21

If the current is not empty.

- The head pointer should not be changed.
- Copy head pointer to a temp pointer, which will iterate through the list
- How to iterate through the list?

```
while(temp->next != NULL){  
    temp = temp->next;  
}
```

```
void append(Node*& head, int v){  
    Node * newPtr = new Node;  
    newPtr->val = v;  
    newPtr->next = NULL;  
    if(head == NULL){  
        head = newPtr;  
    }  
    else{  
        Node* temp = head;  
        while(temp->next != NULL){  
            temp = temp->next;  
        }  
        temp->next = newPtr;  
    }  
}
```

# Append()

22

```
void append(Node*& head, int v){  
    Node * newPtr = new Node;  
    newPtr->val = v;  
    newPtr->next = NULL;  
    if(head == NULL){  
        head = newPtr;  
    }  
    else{  
        Node* temp = head;  
        while(temp->next != NULL){  
            temp = temp->next;  
        }  
        temp->next = newPtr;  
    }  
}
```

Create a new node

If current list is empty

If current list is not empty

# While loop or for loop?

23

```
void append(Node*& head, int v){
    Node * newPtr = new Node;
    newPtr->val = v;
    newPtr->next = NULL;
    if(head == NULL){
        head = newPtr;
    }
    else{
        Node* temp = head;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = newPtr;
    }
}
```

```
void append(Node*& head, int v){
    Node * newPtr = new Node;
    newPtr->val = v;
    newPtr->next = NULL;
    if(head == NULL){
        head = newPtr;
    }
    else{
        for(temp = head;
            temp->next;
            temp = temp->next);
        temp->next = newPtr;
    }
}
```

for loop body is  
empty



# Print out the values in each node

24

```
void display(Node* head){  
    Node* temp = head;  
    while(temp != NULL){  
        cout << temp->val << endl;  
        temp = temp->next;  
    }  
}
```

```
void display(Node* head){  
    Node* temp;  
    for(temp = head;  
        temp;  
        temp = temp->next){  
        cout << temp->val << endl;  
    }  
}
```





# Tail pointer

25

To append a node to the linked list:

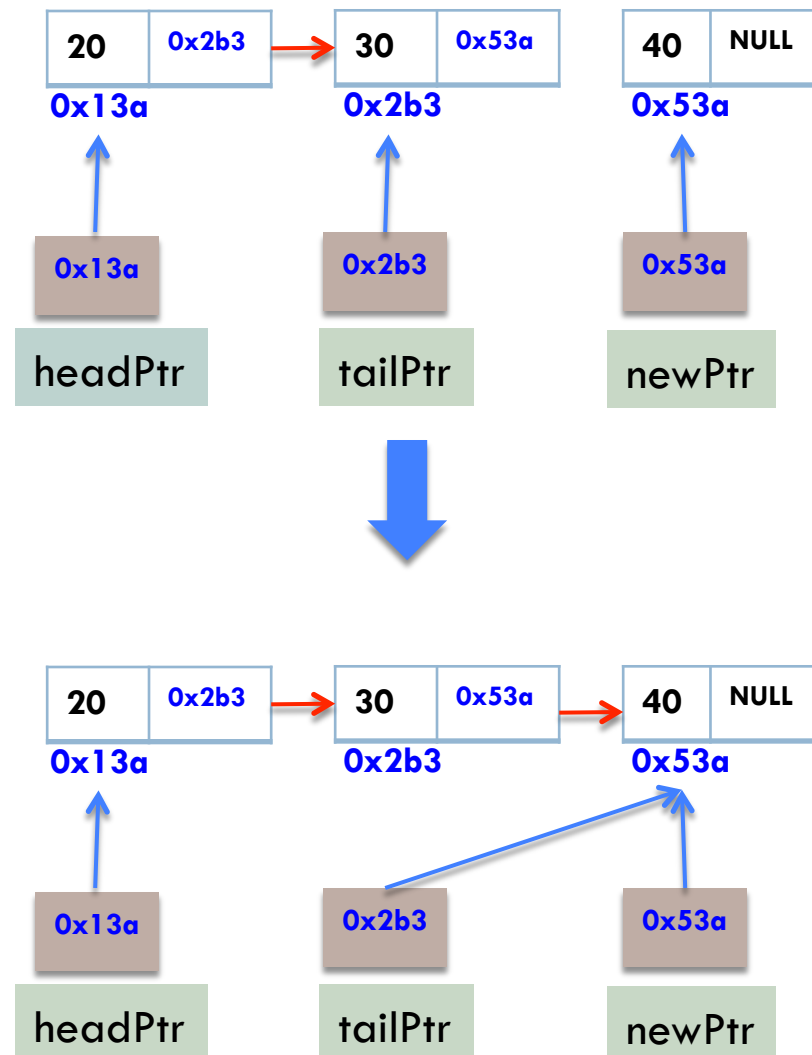
- Get a copy of the head pointer
- **Iterate** through to the end of the list

What if the list is **REALLY** long?

- Do you really want to **iterate** every node for every append operation?

Add a tail pointer to the linked list

- Every time appends a new node, just add the new node after the node pointed by tail pointer (**fast**)
- What if we need to remove the tail node? (**slow**)

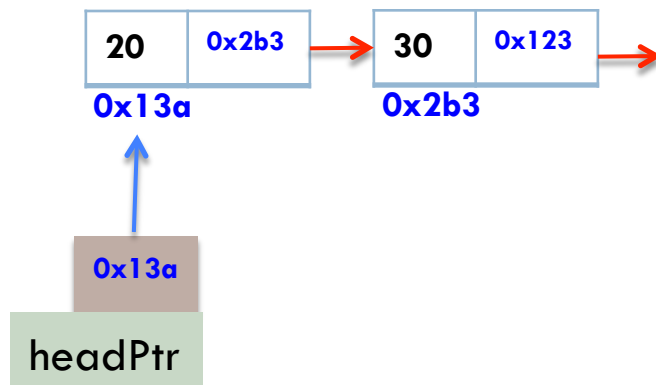


# Delete

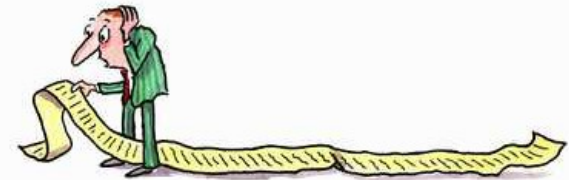
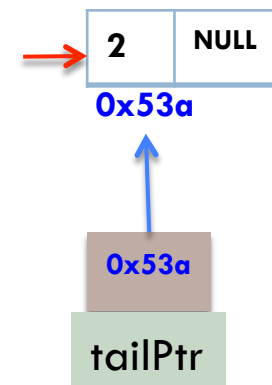
26

To **delete** the **last** node from the linked list.

- Start from the head
- Iterate through the **second last** node, why?
  - Change the next value of the second last node to be NULL
- It is **expensive** to delete one node in a long list
  - Doubly-linked list can make it less expensive



...



# Performance Analysis



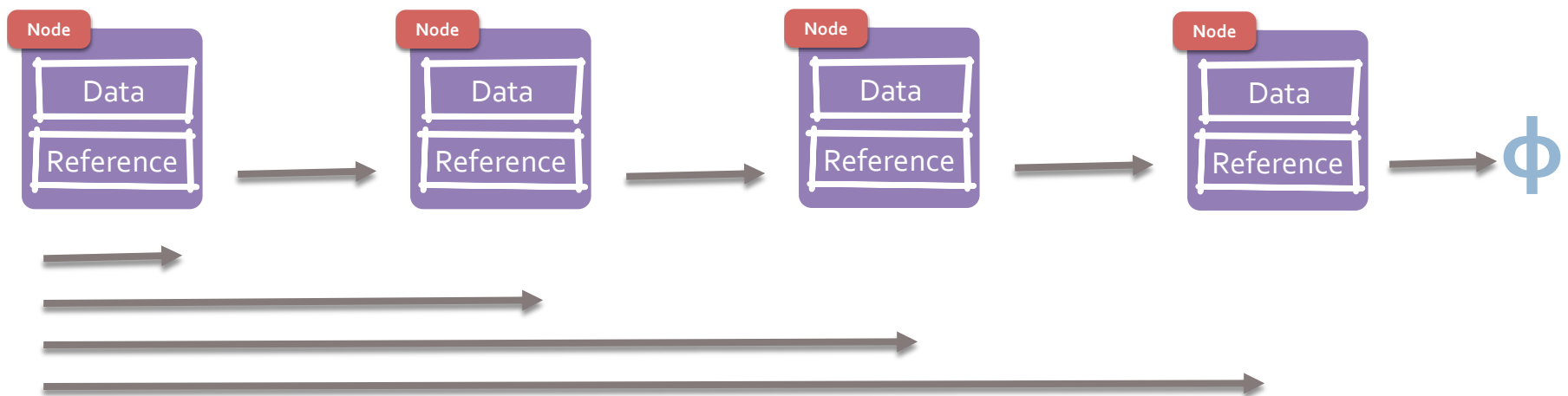
## **Performances of Linked List operations**

- Insertion & Deletion Tests
  - **Workload**
    - 5M numbers for add function
    - 10M numbers for insertion and deletion
    - 10K numbers for enumerating and iterating
  - **What has been tested ?**
    - Add one element at the tail
    - Insert one element at a given position
    - Delete one element at a given position
    - Traverse with get() and next() methods

# Performance Analysis

## Performances of Linked List operations

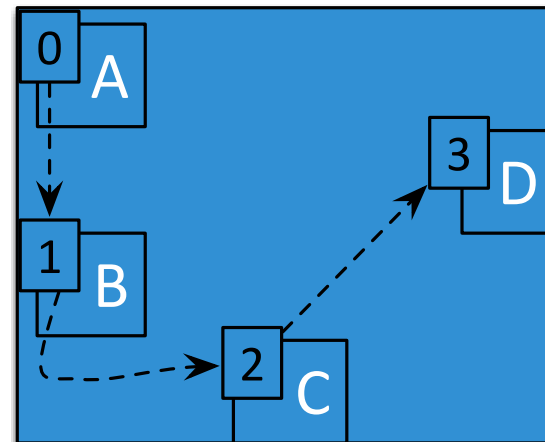
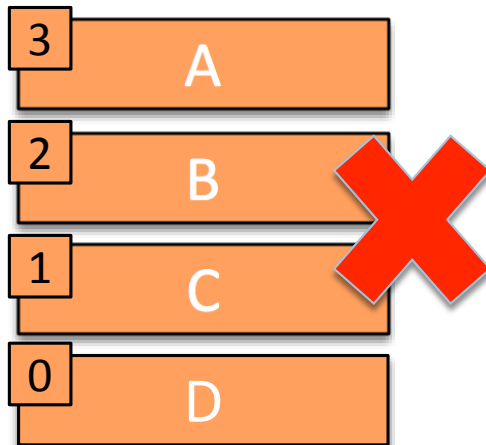
- Insertion & Deletion Tests
  - **Now we run the examples**
    - Q1: Why add an element is usually 0 ms ?
    - Q2: Why insertion at a position grows with the numbers?
    - Q3: Why deletion time cost grows ?
    - Q4: Why there is a huge difference in the last test ?



# Performance Analysis

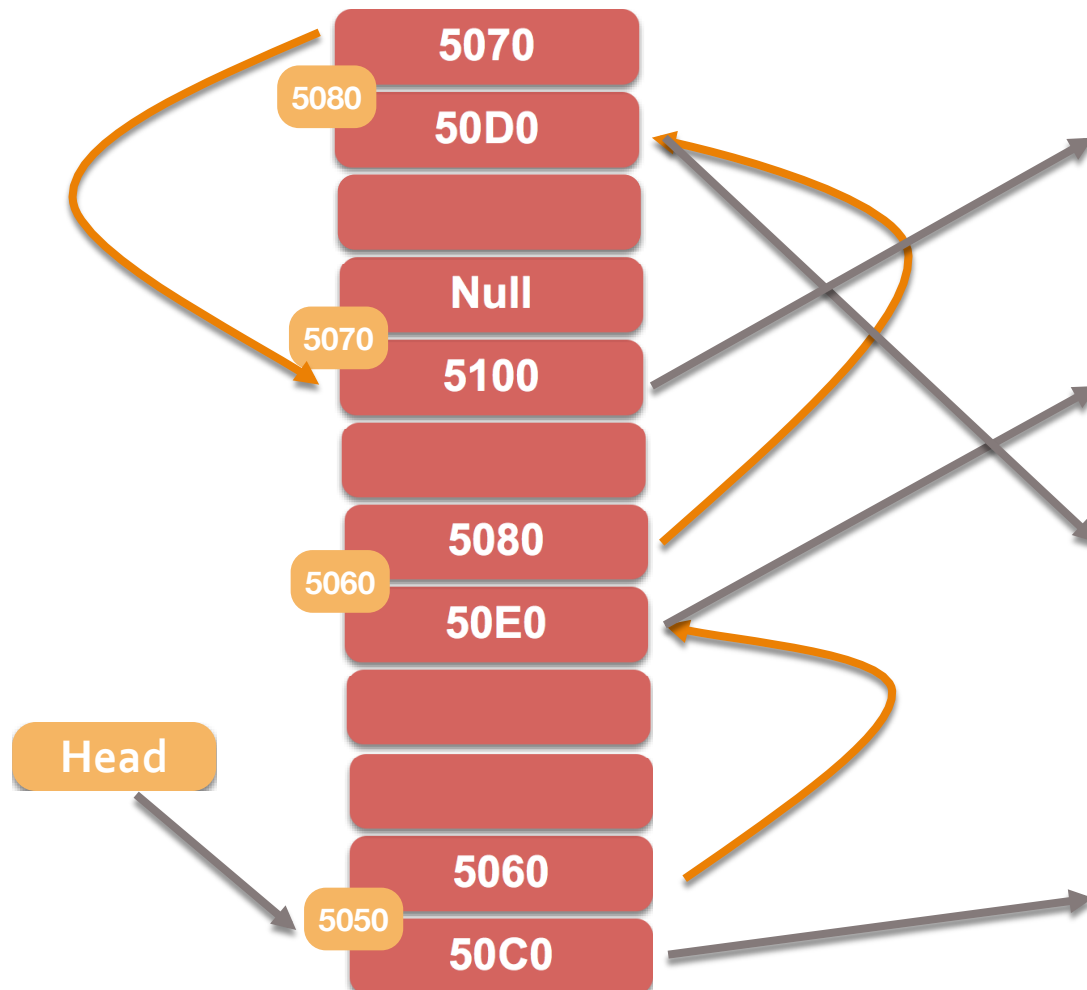
## Linked List in memory

- Each node contains a data block and a reference
- The position in memory may not be continuous.
- Demo with C++



# Performance Analysis

## Linked List in memory



# Lab discussion



- In lab 4, we read a file and store the data into a array;
- This week, we read a file and store the data into a singly linked list
  - Define the node with / without class
  - Implement the append function
  - Print out the singly linked list
  - Implement the search function