fit@hcmus

PROGRAMMING TECHNIQUES

Week 4: Dynamic Data Structures – Linked Lists

Nguyễn Hải Minh - 02/2024

Today content

- Review of pointers
  - New operator
  - Arrays
- Introduction to Linked Lists
- Linear Linked List
- Doubly Linked List
- Circular Linked List

REVIEW OF POINTERS

- new operator
- Arrays

Pointers

- What advantage do pointers give us?
- How can we use pointers and new to allocating memory dynamically?
- Why allocating memory dynamically vs. statically?
- Why is it necessary to deallocate this memory when we are done with the memory?

Pointers and Arrays

- Are there any **disadvantages** to a dynamically allocated array?
  - The benefit - of course - is that we get to wait until run time to determine how large our array is.
  - The drawback - however - is that the array is still **fixed size**.... it is just that we can wait until run time to fix that size.
  - And, at some point prior to using the array we must determine how large it should be.

Arrays Analysis

- Array characteristics:
  - Elements are arranged in a linear order.
  - Fixed element number.
  - Memory allocated in block (continuously).
  - The order is determined by the array indices.
- Operations – Analysis:
  - Access an element?
  - Update the array?
    - **Insert** a new element into the array?
    - **Delete** an element from the array?

Insert an Element to the Array

- Move all the elements 1 index to the right

18

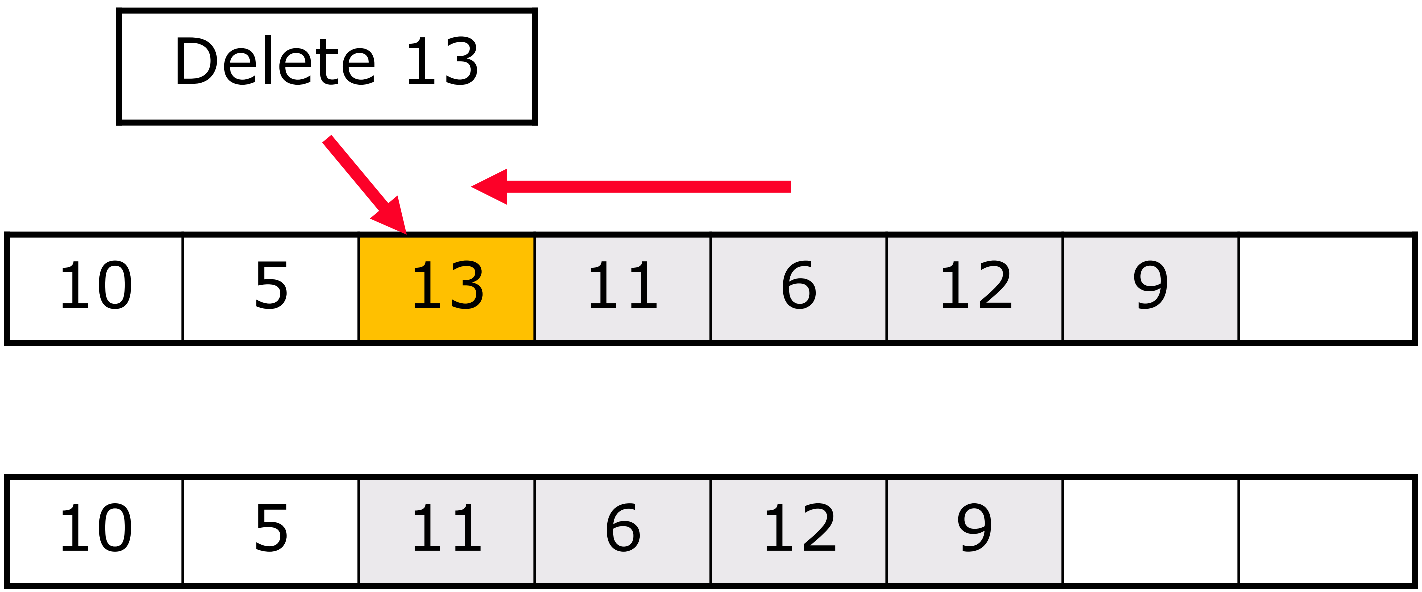
10	5	13	11	6	12	9	

- Then, insert the element to the slot

10	5	18	13	11	6	12	9

Delete an Element from the Array

- Move all the elements 1 index to the left



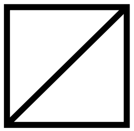
LINKED LISTS

Linked Lists

- Our solution to this problem is to use **linear linked lists** instead of arrays to maintain a “list”
- With a linear linked list, we can grow and shrink the size of the list as new data is added or as data is removed
- The list is ALWAYS sized exactly appropriately for the size of the list

Linked Lists

- A linear linked list starts out as empty
  - An empty list is represented by a null pointer
  - We commonly call this the **head** pointer



head

Linked Lists

- As we add the first data item, the list gets one **node** added to it
  - So, **head** points to a **node** instead of being null
  - And, a **node** contains the data to be stored in the list and a **next** pointer (to the next node...if there is one)

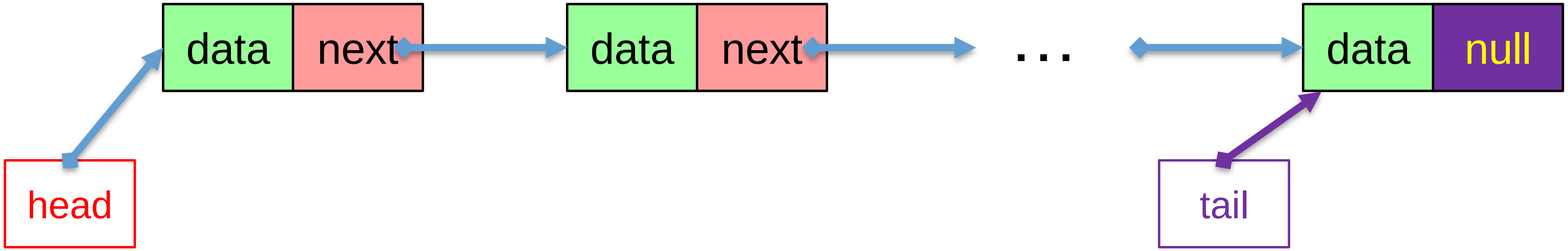
a dynamically allocated node

Linked Lists

- To add another data item we must first decide in what order
  - does it get added at the beginning
  - does it get inserted in sorted order
  - does it get added at the end
- This term, we will learn how to add in each of these positions.

Linked Lists

- Ultimately, our lists could look like:



- Sometimes we also have a tail pointer. This is another pointer to a node -- but keeps track of the end of the list.
- This is useful if you are commonly adding data to the end

Linked Lists

- So, how do linked lists differ than arrays?
  - An array is direct access; we supply an element number and can go directly to that element (through pointer arithmetic)
  - With a linked list, we must either start at the head or the tail pointer and **sequentially traverse** to the desired position in the list

Linked Lists

- In addition, linear linked lists (singly) are connected with just one set of **next** pointers.
  - This means you can go from the first to the second to the third to the forth (etc) nodes
  - But, once you are at the forth you can't go back to the second without starting at the beginning again.....

Linked Lists

- Besides linear linked lists (singly linked), there are other types of lists:
  - Circular linked lists
  - Doubly linked lists
  - Non-linear linked lists

LINEAR LINKED LIST

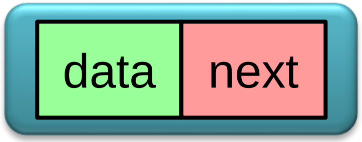
Or Singly Linked List

Linear Linked Lists

- We need to define both the head pointer and the node (by using struct)
- We'll start with the following:

```
struct video {      //our data
   char* title;
   char category[5];
   int quantity;
};
```

- Then, we define a **node** structure:




```
struct node {
   video data;
   node* pNext;  //a pointer to the next node
};
```

Linear Linked Lists

- Initialize a linked list:
  - At first, linked list has no element → head pointer points to NULL

```
node* pHead = NULL;
```

  - Or you can use a struct to store a linked list

```
struct mylist {
   node* pHead;
   int n; //number of element in linked list
};
```

Traversing

- To show how to traverse a linear linked list, let's spend some time with the DisplayAll function:

```
void DisplayAll(node* pHead) {
  node* pCurrent = pHead;
  if (pCurrent == NULL)
     cout << "Your list is empty" << endl;
  while (pCurrent != NULL) {
     cout << pCurrent->data.title << '\t'
          << pCurrent->data.category << '\t'
          << pCurrent->data.quantity << endl;
     pCurrent = pCurrent->pNext;
  }
}
```

Traversing – Step-by-step

- Why do we need a `pCurrent` pointer?
  - It is used to mark the position of the current node.
- Can we just use `pHead` like following?

```
while (pHead != NULL) {
    cout << pHead->data.title <<'\t' …
    pHead = pHead->pNext;
}
```

→ *Be careful! Otherwise, we could lose our list!!!*

Traversing – Step-by-step

- Why do we use the NULL stopping condition:

```
while (pCurrent != NULL) {
```

  - This implies that the very last node's next pointer must have a **NULL** value
    - so that we know when to stop when traversing
    - NULL is a `#define` constant for zero
    - So, we could have said:

```
while (pCurrent) {
```

Traversing – Step-by-step

- Now let's examine how we access the data's values:

```
cout << pCurrent->data.title << '\t'
     << pCurrent->data.category << '\t'
     << pCurrent->data.quantity << endl;
```

  - Since current is a pointer, we use the `->` operator (*indirect member access operator*) to access the "`data`" and the "`pNext`" members of the `node` structure
  - But, since "`data`" is an object (and not a pointer), we use the `.` operator to access the title, category, etc.

Traversing – Step-by-step

- If our node structure had defined data to be a pointer:

```
struct node {
  video* pData; //a pointer to data
  node* pNext;  //a pointer to the next node
};
```

- Then, we would have accessed the members via:

```
cout << pCurrent->pData->title << '\t'
     << pCurrent->pData->category << '\t'
     << pCurrent->pData->quantity << endl;
```

(And, when we insert nodes we would have to remember to allocate memory for a video object in addition to a node object...)

Traversing

- So, if `pCurrent` is initialized to the head of the list, and we display that first node
  - to display the second node we must **traverse**
  - this is done by:

```
pCurrent = pCurrent->pNext;
```

  - why couldn't we say:

```
pCurrent = pHead->pNext;
```

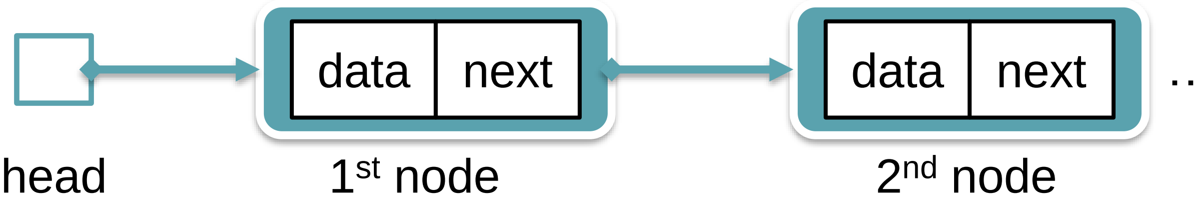
NO!!!

Building

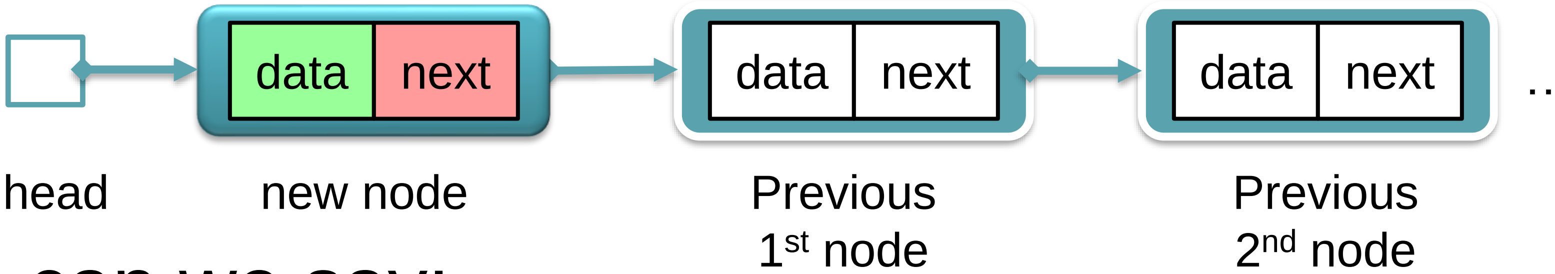
- Well, this is fine for traversal. But, you should be wondering at this point, how do I create (build) a linked list?
- So, let's write the algorithm to add a node to the **beginning** of a linked list

Inserting at Beginning

- We go from:



- To:



- So, can we say:

```
pHead = new node;
```

Inserting at Beginning

- If we did, we would lose the rest of the list!
- So, we need a temporary pointer to hold onto the previous head of the list

```
node* pCurr = pHead;  //copy and backup head to current
pHead = new node;  //create a new node
pHead->pData = new video;  //if data is a pointer
pHead->pData->title = new char[strlen(newtitle)+1];
strcpy(pHead->pData->title, newtitle);  //etc.
pHead->pNext = pCurr;   //reattach the list!!!
```

Inserting at End

- Add a node at the end of a linked list.
  - What is wrong with the following?

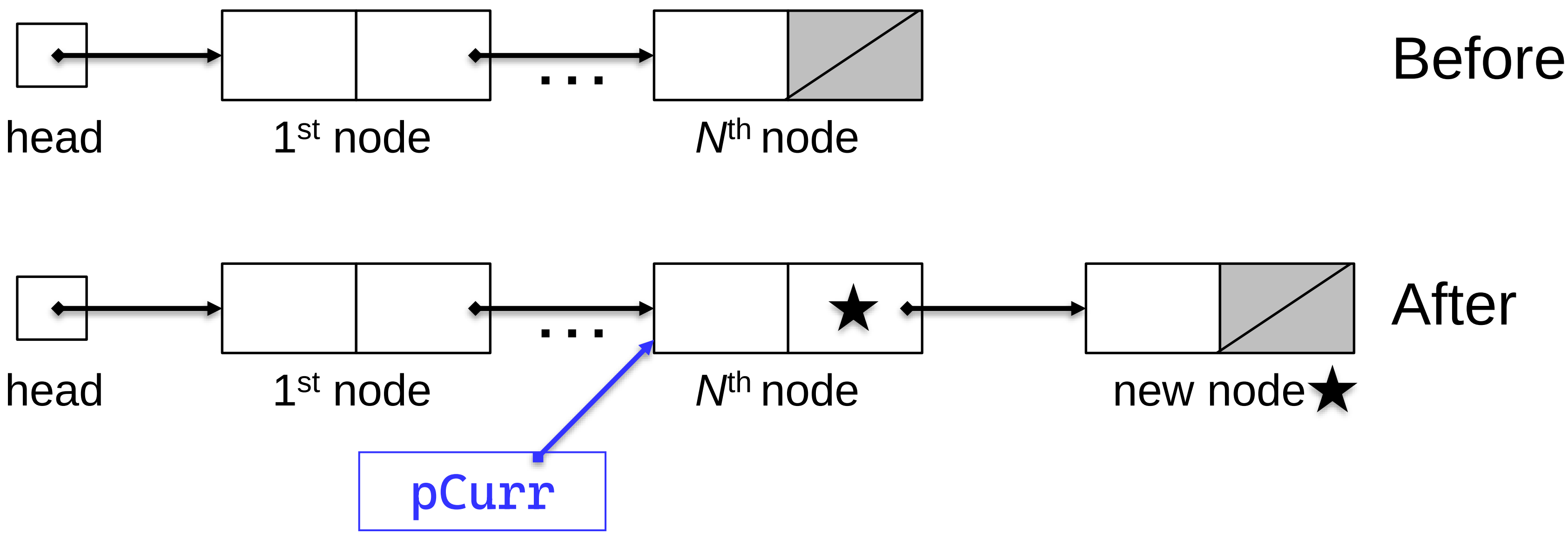
```
node* pCurr = pHead;
while (pCurr != NULL) {
  pCurr = pCurr->pNext;
}
pCurr = new node;
pCurr->pData = new video;
pCurr->pData = data_to_be_stored;
…
```

Inserting at End

- We need a temporary pointer because if we use the head pointer
  - we will *lose the original head* of the list and therefore ***all of our data***
- If our loop's stopping condition is `if pCurr is not null` -- then what we are saying is loop until current **IS** null
  - and, we will NOT be pointing to the last node!

Inserting at End

- Instead, think about the “before” and “after” **pointer diagrams:**



Inserting at End

- So, we want to loop until `pCurr->pNext` is not NULL!
- But, to do that, we must make sure `pCurr` isn't NULL
  - This is because if the list is empty, `pCurr` will be null and we'll get a segmentation fault by dereferencing the null pointer

```
if (pCurr) {
while (pCurr->pNext != NULL) {
   pCurr = pCurr->pNext;
}
```

Inserting at End

- Next, we need to connect up the nodes
  - Having the last node point to this new node

```
pCurr->pNext = new node;
```

  - Then, traverse to this new node and create data:

```
pCurr = pCurr->pNext;
pCurr->pData = new video;
```

  - And, set the next pointer of this new last node to null:

```
pCurr->pNext = NULL;
```

Inserting at End

- Lastly, in our first example for today, it was inappropriate to just copy over the pointers to our data
  - We allocated memory for a video and then immediately lost that memory with the following:

```
pCurr->pData = new video;
pCurr->pData = data_to_be_stored;
```

  - The correct approach is to allocate the memory for the data members of the video and physically copy each and every one

Removing at Beginning

- Now let’s look at the code to remove a node at the beginning of a linear linked list.
- Remember when doing this, we need to deallocate **all** dynamically allocated memory associated with the node.
- Will we need a temporary pointer?
  - Why or why not...

Removing at Beginning

- What is wrong with the following?

```
node* pCurr = pHead->pNext;
delete pHead;
pHead = pCurr;
```

  - everything? (just about!)

Removing at Beginning

- First, don't dereference the head pointer before making sure head is not NULL

```
if (pHead) {
    node* pCurr = pHead->pNext;
    …
```

  - If head is NULL, then there is nothing to remove!

- Next, we must deallocate all dynamic memory:

```
delete [] pHead->pData->title;
delete pHead->pData; //deallocate pointer pData
delete pHead;
pHead = pCurr;  //this was correct....
```

Removing at End

- Now take what you’ve learned and write the code to remove a node from the end of a linear linked list
- What is wrong with: (lots!)

```
node* pCurr = pHead;
while (pCurr != NULL) {
      pCurr = pCurr->pNext;
}
delete[] pCurr->pData->title;
delete pCurr->pData;
delete pCurr;
```

Removing at End

- Look at the stopping condition
  - if pCurr is null when the loop ends, how can we dereference pCurr? It isn’t pointing to anything
  - therefore, we’ve gone too far again

```
node* pCurr = pHead;
if (!pHead) return 0; //failure mode
while (pCurr->pNext != NULL) {
   pCurr = pCurr->pNext;
}
```

  - is there anything else wrong? (yes)

Removing at End

- So, the deleting is fine....

```
delete [] pHead->pData->title;
delete pHead->pData; //deallocate pointer pData
delete pCurr;
```

  - but, doesn't the previous node to this still point to this deallocated node?
  - when we retraverse the list -- we will still come to this node and access the memory (as if it was still attached).

Removing at End

- When removing the last node, we need to reset the new last node's next pointer to NULL
  - but, to do that, we must keep a pointer to the previous node
  - because we <u>do not</u> want to "retraverse" the list to find the previous node
  - therefore, we will use an additional pointer
    - we will call it *"previous"*

Removing at End

- Taking this into account:

```
node* pCurr = pHead;
node* pPrev = NULL;
if (!pHead) return 0;
while (pCurr->pNext) {
   pPrev = pCurr;
   pCurr = pCurr->pNext;
}
delete[] pCurr->pData->title;
delete pCurr->pData;
delete pCurr;
pPrev->pNext = NULL;
```

Can anyone see the remaining problem?

Removing at End

- Always think about what special cases need to be taken into account.
- What if...
  - there is only ONE item in the list?
  - `pPrev->pNext` won't be accessing the deallocated node (`pPrev` will be NULL)
  - we would need to reset `pHead` to NULL, after deallocating the one and only node.

Removing at End

- Taking this into account:

```
…
if (!pPrev) //only 1 node
   pHead = NULL;
else
   pPrev->pNext = NULL;
```

Now, put this all together as an exercise

Exercise

- Given struct video and a linked list as follow

```
struct video{
    char* title;
    char category[5];
    int quantity;
};

struct node{
     video data;
     node* pNext;
};
```

Exercise

- Write the following functions

```
1. bool IsListEmpty(node* lst);
2. int GetListLength(node* lst);
3. int AddItemToList(node* &lst, video item);
4. int InsertToList(node* &lst, int newpos, video item);
5. int RemoveFromList(node* &lst, int pos);
6. void ClearList(node* &lst);
7. video GetListEntry(node* lst, int pos);
8. void SetListEntry(node* &lst, int pos, video item);
```

Exercise

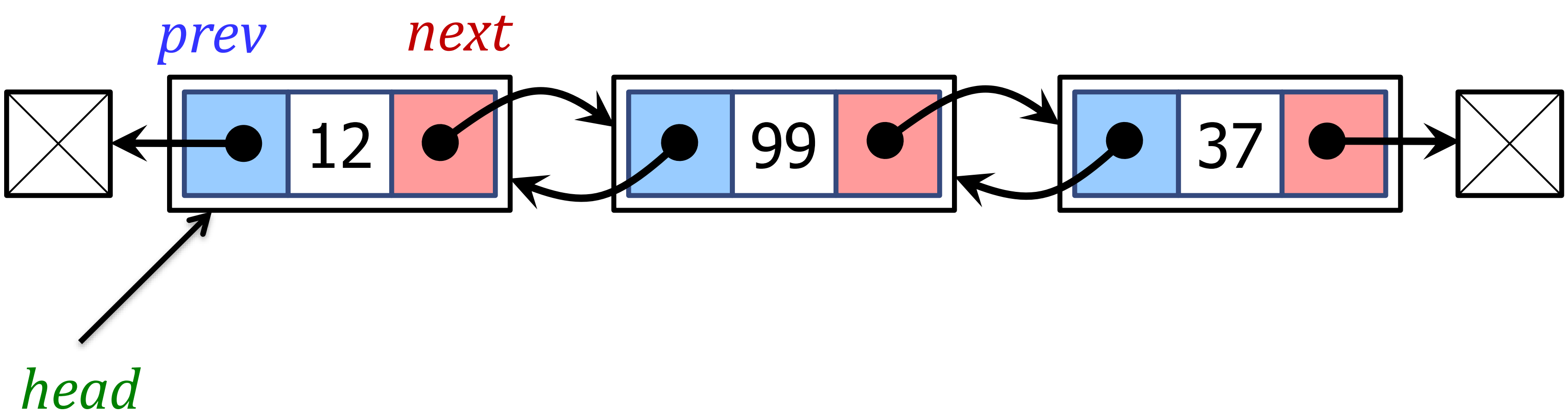
- Explanation:

1. Check if the list is empty or not (return true if `lst` is empty)
2. Return the list length
3. Add a new `item` to the **end** of the lst
4. Insert a video `item` into lst at the position `newpos`
5. Remove a video at the position `pos` in `lst`
6. Clear all videos in `lst`
7. Get the video at the position `pos` in `lst`
8. Set the video at the position `pos` in `lst` to `item`

DOUBLY LINKED LIST

Doubly Linked List

- Each node has 2 pointers:
  - 1 pointer to its successor (next pointer)
  - 1 pointer to its predecessor (previous pointer)



Doubly Linked List

- Let have a student data structure:

```
struct student {
    int ID;
    float GPA;
};
```

- Then, we define a node structure:

```
struct node {
    student data;
    node* pPrev; //a pointer to the previous
    node* pNext; //a pointer to the next
};
```

Doubly Linked List

- Now we can define a linked list of students using **struct** (you can also use **class**)

```
struct student_list{
    node* pHead;
    int nStudents; //number of students in this list
};
void Init(student_list& lst);
void Add(student_list& lst, const student& stu);
void Remove(student_list& lst, int ID);
void Display_all(student_list lst);
…
```

Same as singly linked list

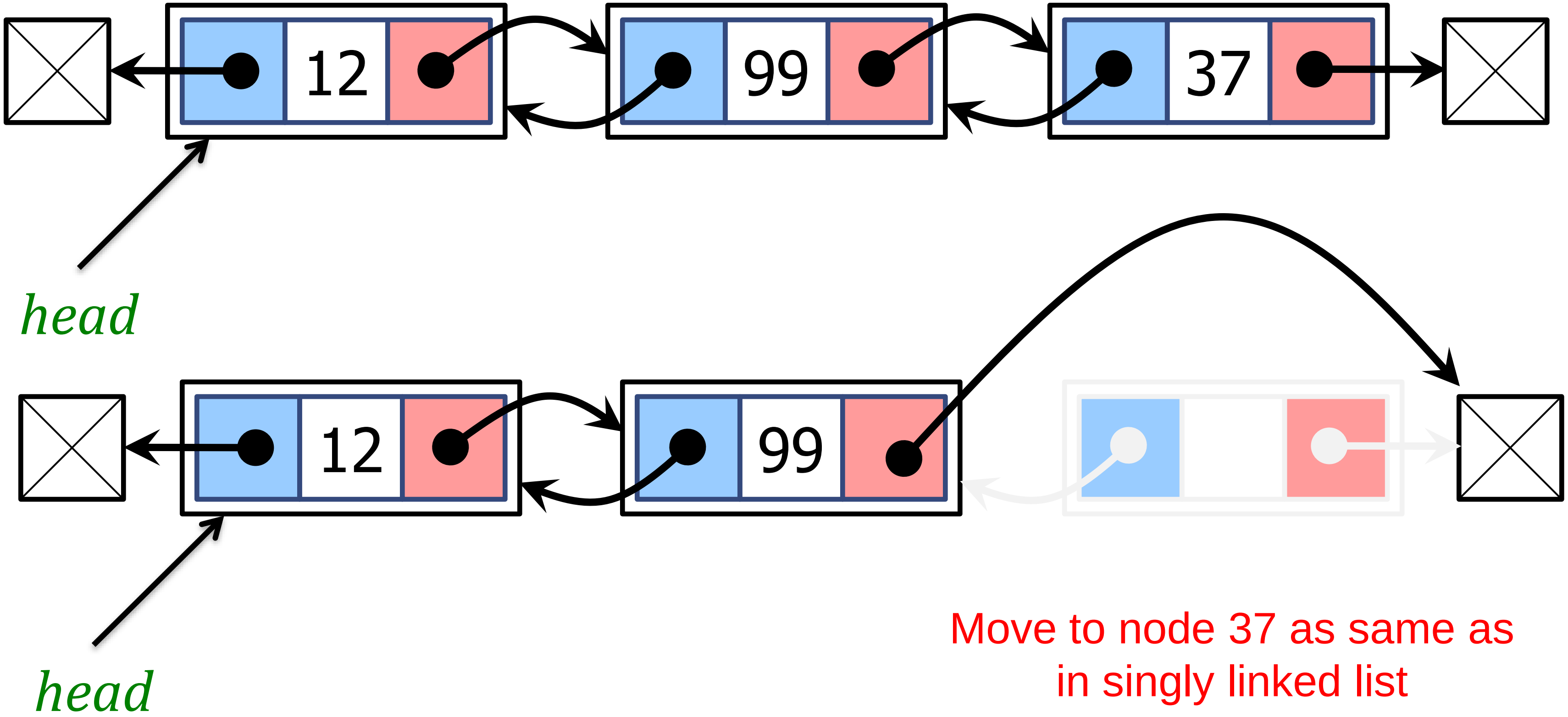
DLL – Inserting at Beginning

```
void Init(student_list& lst) //Initialize the linked list lst
{
   lst.pHead = NULL;
   lst.nStudents = 0;
}
void Add(student_list& lst, const student& stu)
{
   node* pCurr = lst.pHead;  //copy and backup head to pCurr
   lst.pHead = new node;
   lst.pHead->data.ID = …; //copy data
   lst.pHead->data.GPA = …;//copy data
   lst.pHead->pNext = pCurr;  //reattach the list!!!
   lst.pHead->prev = NULL;  //head->prev should always be NULL
   if (pCurr != NULL)
      pCurr->pPrev = lst.pHead;
   lst.nStudents++;
}
```

Pass by Reference

More links to change

DLL – Removing at End



Move to node 37 as same as in singly linked list

DLL – Removing at End

```
int RemoveAtEnd(student_list& lst)
{
   node* pCurr = lst.pHead;
   if (pCurr == NULL) //lst is empty
      return 0;
   //else: lst has 1 or more students
   while (pCurr->pNext != NULL) //traverse to the last student
      pCurr = pCurr->pNext;
   if (pCurr->pPrev == NULL) //lst has only 1 student
   {
      delete lst.pHead;
      lst.pHead = NULL;
      lst.nStudents--;
      return 1;
   }
   pCurr->pPrev->pNext = NULL;
   delete pCurr;
   lst.nStudents--;
   return 1;
}
```

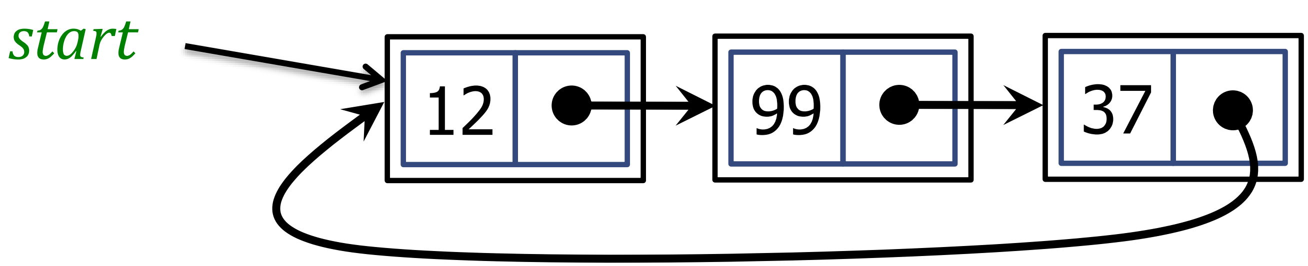
Doubly Linked List – Analysis

- Advantage:
  - Adding/removing are simpler and potentially more efficient for nodes other than first nodes
- Disadvantage:
  - Require changing more links than singly linked list when adding/removing a node

CIRCULAR LINKED LIST

Circular Linked List

- Nodes form a ring:
  - The first element point the next element, the last element points to the first element.
  - There is no NULL at the end!



- Can be used to traverse the same list again and again

Circular Linked List

- Circular linked list may be used to represent:
  - Arrays that are naturally circular, e.g. the corners of a polygon
  - A pool of buffers that are used and released in First in, first out order
- → A pointer to any node serves as a handle to the whole list

Circular Linked List

- Define a node structure:
  - Singly circular linked list

```
struct node {
   POINT data;
   node* pNext; //a pointer to the next
};
```

```
struct POINT{
   int x;
   int y;
};
```

  - Doubly circular linked list

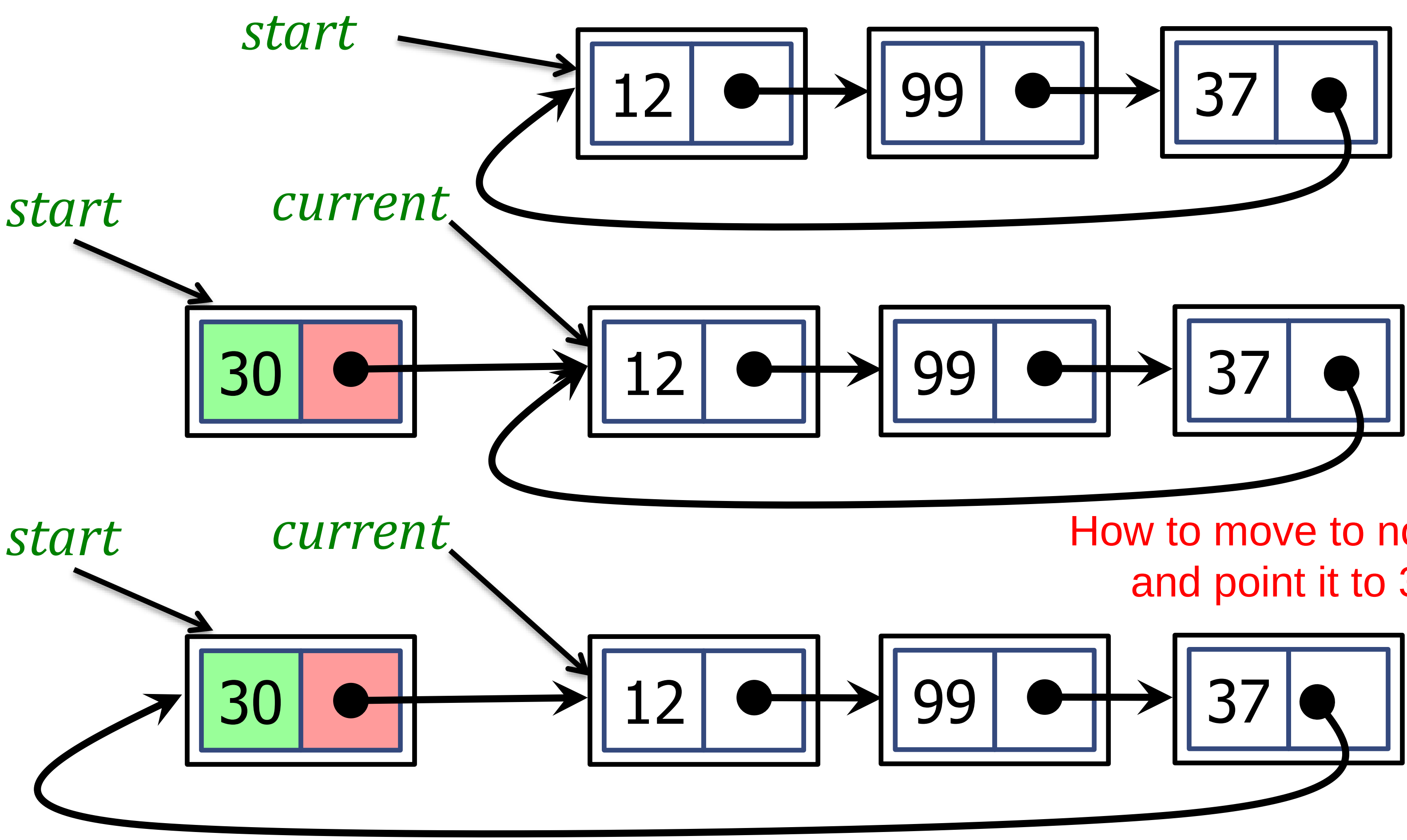
```
struct node {
   POINT data;
   node* pPrev; //a pointer to the previous
   node* pNext; //a pointer to the next
};
```

Circular Linked List

- Define a circular linked list to represent coordinates of a polygon:

```
struct polygon {
      node* pStart;
      int n;      //number of edges
};
void Init(polygon& poly);
void Add(polygon& poly, const POINT& p);
void Remove_Edge(polygon& poly);
void Display_all(polygon poly);
```

CLL – Inserting at Start



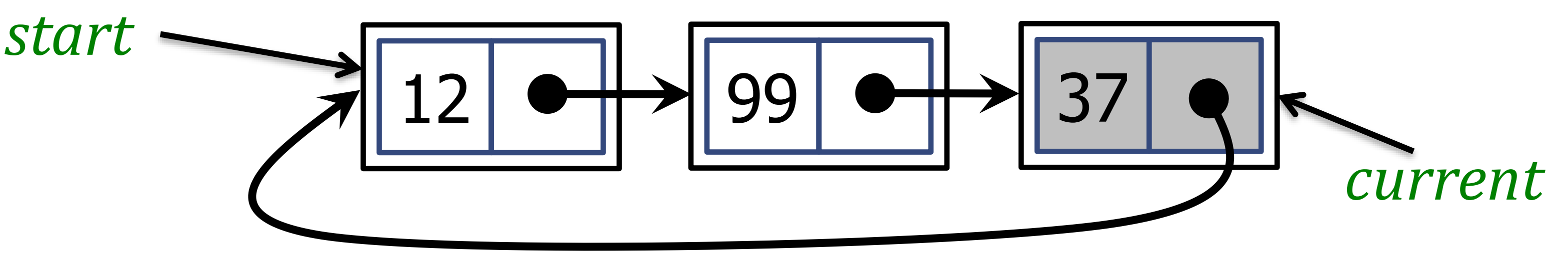
How to move to node 37
and point it to 30?

CLL – Inserting at Start

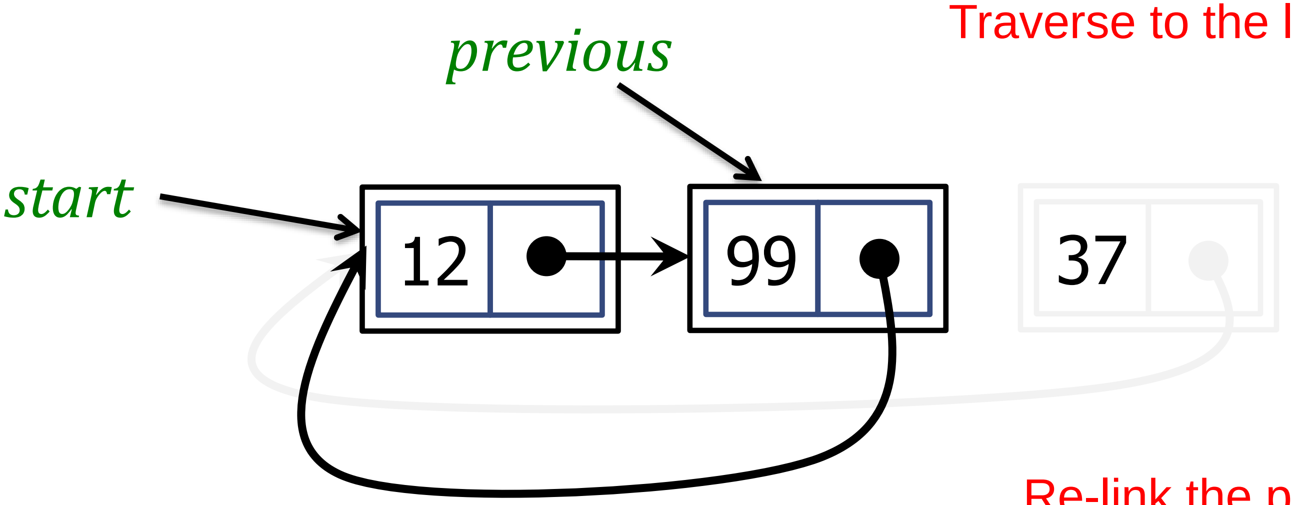
```
void Add(polygon& poly, const POINT& p)
{
   node* pCurr = poly.pStart;   //copy and backup head to current
   poly.pStart = new node;
   poly.pStart->data.x = p.x;
   poly.pStart->data.y = p.y;
   if (pCurr == NULL) //or poly.n == 0
      poly.pStart->pNext = poly.pStart;
   else
   {
      poly.pStart->pNext =pCurr;  //reattach the list!!!
      while (pCurr->pNext != poly.pStart->pNext)
         pCurr =pCurr->pNext;
      pCurr->pNext = poly.pStart;
   }
   poly.n++;
}
```

Notice this line!!!

CLL – Removing at Last



Traverse to the last node



Re-link the pointer

CLL – Removing at Last

```
int RemoveAtLast(polygon& poly)
{
  … //Handle special cases
  node* pCurr = poly.pStart;
  node* pPrev = NULL;
  do //find the last element
  {
    pPrev = pCurr;
    pCurr = pCurr->pNext;
  } while (pCurr->pNext != poly.pStart);
  pPrev->pNext = poly.pStart;
  delete pCurr;
  poly.n--;
  return 1;
}
```

Same as in singly linked list

Circular Linked List – Analysis

- Advantage:
  - Any node can be a starting point. We can traverse the whole list from any point
  - Useful for applications to repeatedly go around the list:
    - Applications in PC
    - Multiplayer games
    - Circular Queue

Circular Linked List – Analysis

- Disadvantage:
  - Finding the end of a list is more difficult (no NULL's to mark the beginning / end)
  - Add at beginning could be expensive to search for the last node (depending on the implementation)

Exercises

- Write the following functions for doubly linked list and circular linked list:

1. Add an element to the end of list
2. Remove the first element of the list
3. Delete the whole list
4. Search for an element in the list

Next week's topic

- Dynamic Structures: Stack & Queue
- Next week's quiz:
  - Review today topic (Linear Linked list)


Q&A