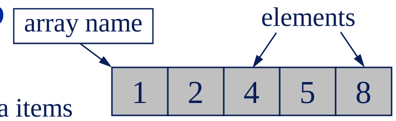
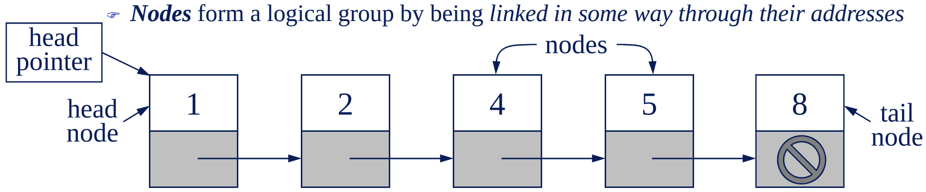
What is a Linked List?



- Similar to array...
  - Data structure for storing collection of data items
- Different from array...
  - Array
    - Consists of a *single block* of *contiguous* memory locations
    - ***Elements*** form a logical group by being *next to each other*
  - Linked list
    - Consists of as *many blocks* of *separate* memory locations as there are data items
    - ***Nodes*** form a logical group by being *linked in some way through their addresses*

allocated at compile time or at run time

usually allocated at run time



- For linked list shown above → called ***singly-linked list***:
  - Each node contains a *data item* and a *pointer to the next node*
  - Last node uses a special "end marker" → commonly the *null address*

not the only type possible, but the type we'll mostly concerned with

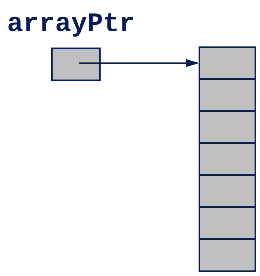
Why Linked List? ❶

- Sizing an array can be a tricky issue...
  - The size of a "*static*" array must be specified at compile time and is fixed
    - Memory space is wasted if most of the time only a small percentage of the allocated space is required and used
    - Code must be modified and recompiled whenever a larger-sized array is needed
  - With a little extra effort, the specification of array size can be delayed until run time (*dynamic* array), but after that the size remains fixed
    - It is possible to dynamically resize the array (as summarized in the next 4 slides) but that requires quite some real programmer and computational effort
- By design, a linked list ***grows and shrinks as required***...
  - Initially, a linked list has no nodes → an *empty list*
    - Usually indicated by setting the *head pointer* to *point to the null address*
  - A new node is created on-the-fly (memory space dynamically allocated) and linked to the list each time a new data item is to be added to the list
  - An existing node is detached from the list and destroyed (dynamically allocated memory space released) each time an existing data item is to be removed from the list

you've seen and done it

Resizing Dynamic Arrays

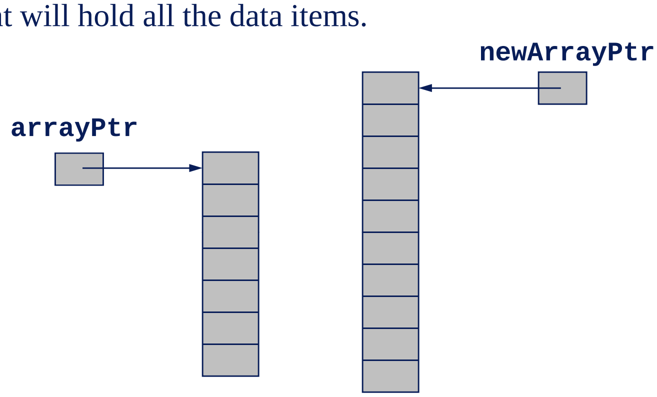
Assume the capacity of this dynamically allocated array is not sufficient to hold all of the data items we need to store



3

Resizing Dynamic Arrays

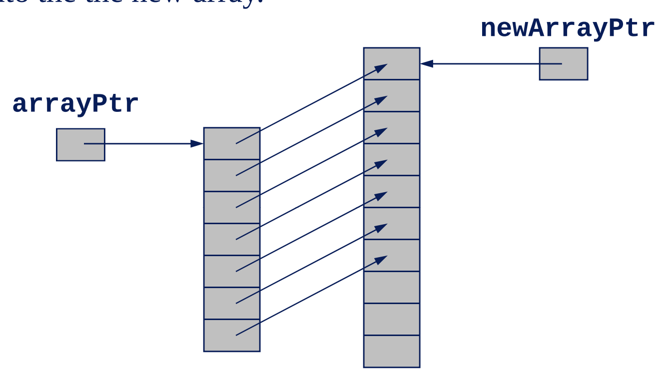
The first step is to dynamically allocate a new, larger array that will hold all the data items.



4

Resizing Dynamic Arrays

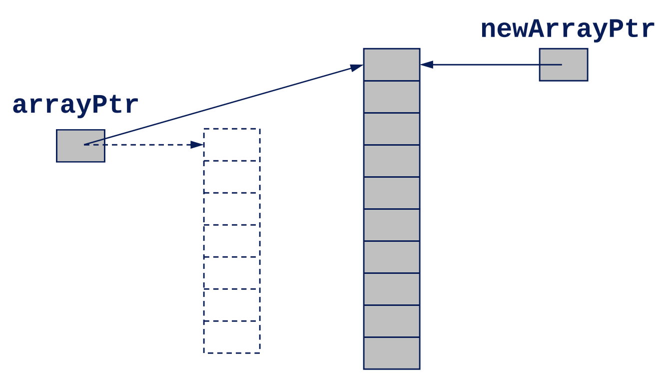
The next step is to copy all the data items from the old array into the the new array.



5

Resizing Dynamic Arrays

Then we want to release the old array to the free store, since we don't need this memory anymore.
Finally, we set **`arrayPtr`** to point to the new, larger array.



6

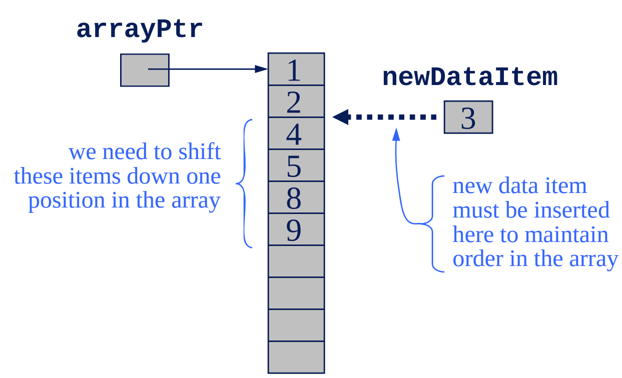
Why Linked List? ❷

- Inserting a new data item into, and deleting an existing data item from, an ***ordered*** list implemented with an array can be *expensive*...
  - Some if not all existing data items (in general) must be shifted to make room for inserting the new data item
  - Some if not all existing data items (in general) must be shifted to fill the void created by the deletion of an existing data item
  - (Next 4 slides summarize the situation)
- Inserting a new data item into, and deleting an existing data item from, an ***ordered*** list implemented with a linked list is *cheap*...
  - Only need to change a few pointers
  - (We shall see in due time)

7

Inserting Data into Ordered Arrays

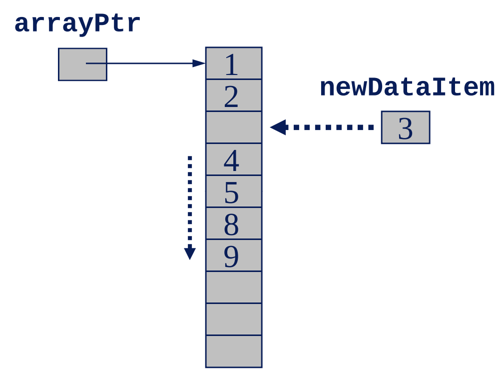
If we want to insert a new data item into an ordered array, we must first make room for the new data item (unless it happens to become the new last item of the array).



8

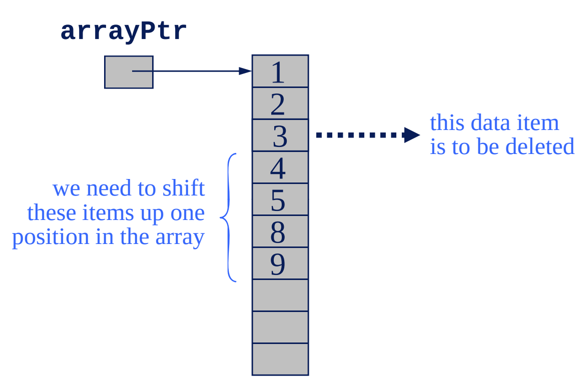
Inserting Data into Ordered Arrays

If we want to insert a new data item into an ordered array, we must first make room for the new data item (unless it happens to become the new last item of the array).



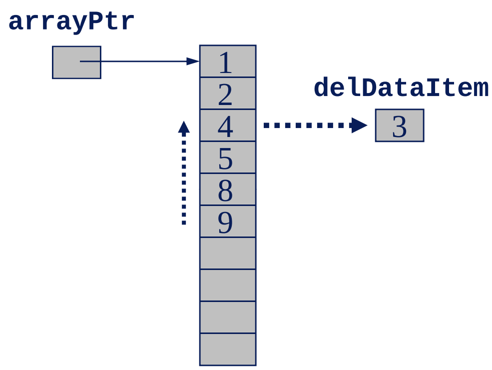
Deleting Data from Ordered Arrays

If we want to delete an existing data item from an ordered array, we must fill the void created by the deleted data item (unless it happens to be the last item of the array).



Deleting Data from Ordered Arrays

If we want to delete an existing data item from an ordered array, we must fill the void created by the deleted data item (unless it happens to be the last item of the array).

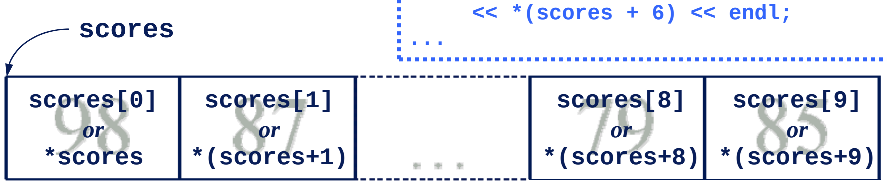


Why NOT Linked List?

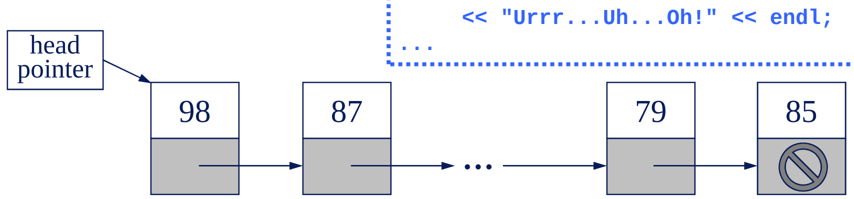
- Random access to data items (with known sequential locations) in an array is fast/cheap... *e.g.*, to retrieve the value stored in the 1234th element of the array
  - We need only to supply the index of the element, which specifies the sequential location of the element → in C/C++ specifically...
    - The index gives the number of elements that must be offset (relative to the first array element) in order to be at the desired element
    - The array name is a pointer constant containing the starting address of the array and array elements can be randomly accessed using the indexed variable notation or pointer notation
  - This is so because an array consists of a single block of contiguous memory locations
- Random access to data items (with known sequential locations) in a linked list is slow/expensive... *e.g.*, to retrieve the value stored in the 1234th node of the linked list
  - A linked list consists of nodes each of which is a separate block of memory locations
  - We only have the pointer to the first node (head pointer)
    - Perhaps also the pointer to the last node (tail pointer) if we choose to also store it (which of course requires more programming effort and system resource)
  - The address of any node, end node(s) excepted, is stored in its immediate neighbor(s)
    - To get to a particular node, we must first get to its neighbor(s) to obtain the node's address
    - But we face the same problem getting to its neighbor(s), unless the first node (or last node if we also keep a tail pointer) happens to be a neighbor
    - In general, we must start at the first node (or last node) of the list and methodically *traverse* down (or up) the list to get to a particular node on the list

Random Access
(Array vs Linked List)

```
...
int scores[] = {98,87,...,79,85};
...
cout << "Score in the 7th element is "
     << *(scores + 6) << endl;
...
```




```
...
cout << "Score in the 7th node is "
     << "Urrr...Uh...Oh!" << endl;
...
```



13

Node Specification

- We can specify the structure of a node using **struct**
- For our illustrations...
  - We will consider the very simply case where the linked list is to be used for storing a list of integer values
    - In other words, the data item in each node consists of only an integer
    - (In real applications, the data item will usually be of some user-defined type that can be very complex)
  - We will also consider only the ***singly-linked list***
    - In a singly-linked list, each node contains only *one* linking pointer → pointing to the next node
    - (In a ***doubly-linked list***, each node contains *two* linking pointers → one pointing to the next node and one pointing to the previous node)
- We will name the node data type **Node**, the data item **data** and the pointer to next node **link**
- The specification:

```
struct Node
{
   int   data;
   Node *link;
};
```

14

Creating an Empty List

- (**Repeat**) By design, a linked list ***grows and shrinks as required***...
  - Initially, a linked list has no nodes → an *empty list*
    - Usually indicated by setting the *head pointer* to *point to the null address*
  - A new node is created on-the-fly (memory space dynamically allocated) and linked to the list each time a new data item is to be added to the list
  - An existing node is detached from the list and destroyed (dynamically allocated memory space released) each time an existing data item is to be removed from the list
- Using the usual convention (head pointer points to the null address when the list is empty), an empty linked list is created by the declaration statement...

```
Node *headPtr = 0;
```

```
struct Node
{
   int   data;
   Node *link;
};
```

15

Creating New Nodes, Linking Nodes to List, Traversing List, and Destroying List

- (**Repeat**) By design, a linked list ***grows and shrinks as required***...
  - Initially, a linked list has no nodes → an *empty list*
    - Usually indicated by setting the *head pointer* to *point to the null address*
  - A new node is created on-the-fly (memory space dynamically allocated) and linked to the list each time a new data item is to be added to the list
  - An existing node is detached from the list and destroyed (dynamically allocated memory space released) each time an existing data item is to be removed from the list
- The next slide shows a rather "silly" but complete C++ program that...
  - Creates an empty list
  - Creates first new node to contain 10
  - Links first new node to the list
  - Traverses list and displays list content
  - Creates second new node to contain 20
  - Links second new node to the list
  - Traverses list and displays list content
  - Creates third new node to contain 30
  - Links third new node to the list
  - Traverses list and displays list content
  - Destroys the list

- To ***traverse*** a linked list means to visit all the nodes in the list, once per node, one by one starting from the first node.
- Note how the "end marker" (*null address*) is used as a *sentinel* during list traversals.
- Note how the ***common linked list idiom***

```
somePtr = somePtr->link;
```

is used to make `somePtr` point to the next node down the list (relative to the node that it is currently pointing to).

16

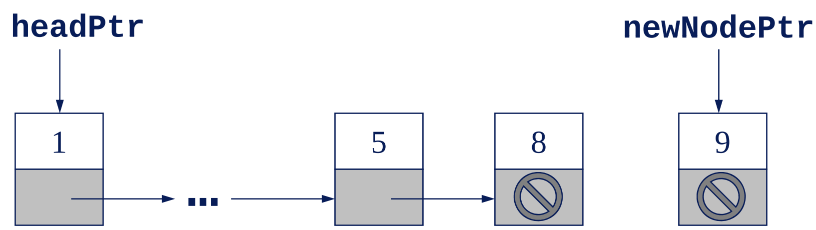
```cpp
#include <iostream>
#include <stdlib>
using namespace std;

struct Node
{
   int   data;
   Node *link;
};

void DisplayList(Node *headPtr)
{
   Node *cursor = headPtr;
   cout << "List content: ";
   while (cursor != 0)
   {
      cout << cursor->data << ' ';
      cursor = cursor->link;
   }
   cout << endl;
}

void DestroyList(Node*& headPtr)
{
   Node *cursor = headPtr;
   while (cursor != 0)
   {
      headPtr = headPtr->link;
      delete cursor;
      cursor = headPtr;
   }
}
```

```cpp
int main()
{
   // create empty list
   Node *headPtr = 0;

   // set up/display 1-node list
   Node *newNodePtr = new Node;
   newNodePtr->data = 10;
   newNodePtr->link = 0;
   headPtr = newNodePtr;
   DisplayList(headPtr);

   // set up/display 2-node list
   newNodePtr = new Node;
   newNodePtr->data = 20;
   newNodePtr->link = 0;
   headPtr->link = newNodePtr;
   DisplayList(headPtr);

   // set up/display 3-node list
   newNodePtr = new Node;
   newNodePtr->data = 30;
   newNodePtr->link = 0;
   headPtr->link->link = newNodePtr;
   DisplayList(headPtr);

   // destroy list and end program
   DestroyList(headPtr);
   cout << "Hit Enter when ready...";
   cin.get();

   return EXIT_SUCCESS;
}
```


Summary Note for Remaining Slides on Basic Linked List Operations

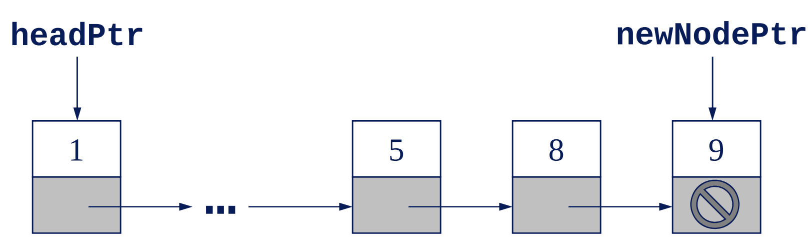
- The previous program is rather "silly" because linked lists are not created and used like that in real applications
  - However, it serves well to illustrate in a very basic and uncluttered way some key features of linked lists
- When using linked lists in more serious programs, it is useful and convenient to develop a ***linked list toolkit*** → essentially a *library of functions* implementing the common linked list operations
  - (We'll be studying and using such a library ← described in textbook)
- In the remaining slides, the basic operations of adding nodes to, and removing nodes from, a singly-linked list are sketched
  - Be sure to make full use of a **LinkedListConceptsPractice** partial program (posted under ***DrillsAndChallenges***) as a hands-on familiarization tool
  - To enhance your appreciation of linked-list fundamentals and gain valuable background experience working with linked lists
    - (Important both for examinations and for the understanding of textbook's more involved implementation of linked lists using `class`)

Appending New Node



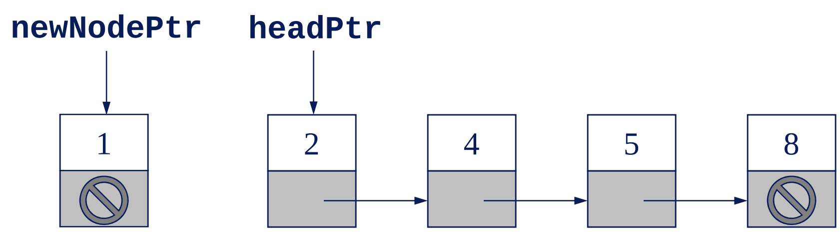
First, create a new node with **`new`**
Set new node's **`link`** to be the end marker
Set new node's **`data`** to the value to be stored

Appending New Node



Set the last node of the current list to point to the new node
We're all done!

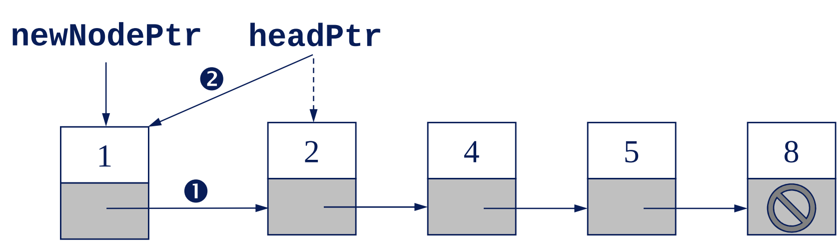
Inserting New Node as Head Node



Create a new node with the new data value

21

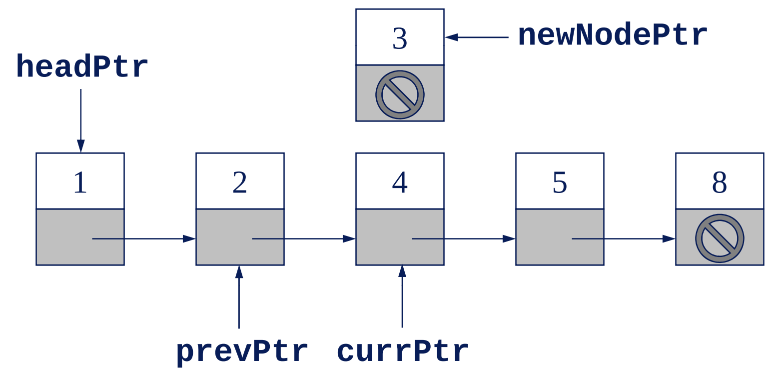
Inserting New Node as Head Node



Set new node to point to the head node of the current list
Set the head pointer to point to the new node

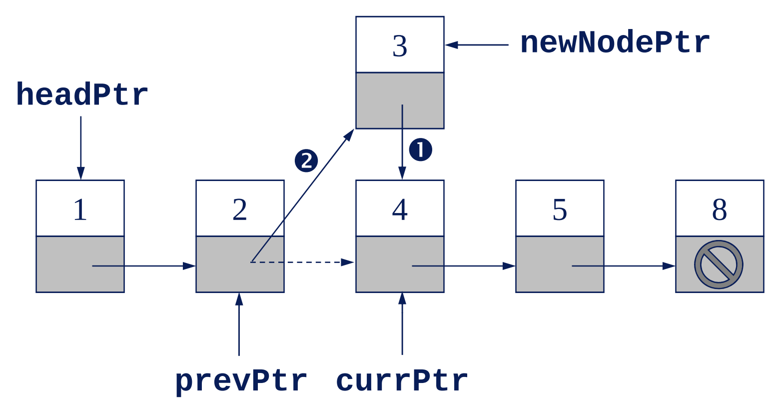
22

Inserting New Node In Between Nodes



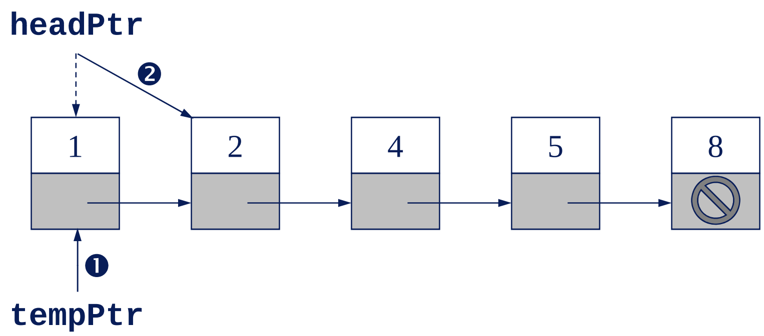
Create a new node with the new data value
Find the node in the list that the new entry will follow

Inserting New Node In Between Nodes



Set the new node to point to the node that previous node points to
Set the previous node to point to the new node

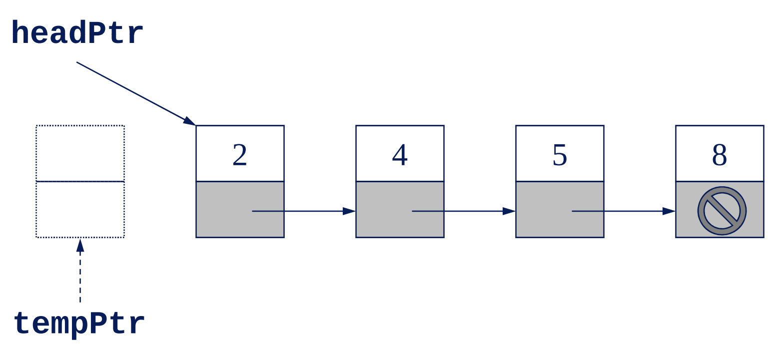
Deleting First Node



Set a temporary pointer to point to the current head node
Set the head pointer to point to the next node in the list

25

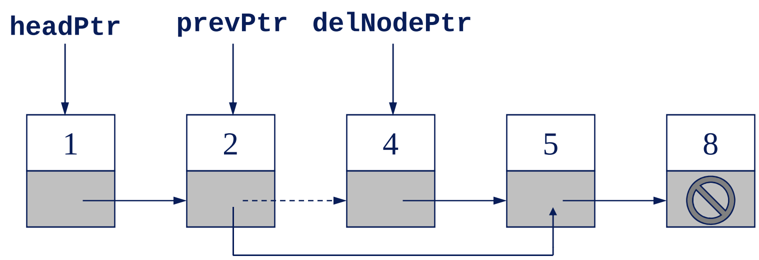
Deleting First Node



`delete` the node pointed to by the temporary pointer
We're done!

26

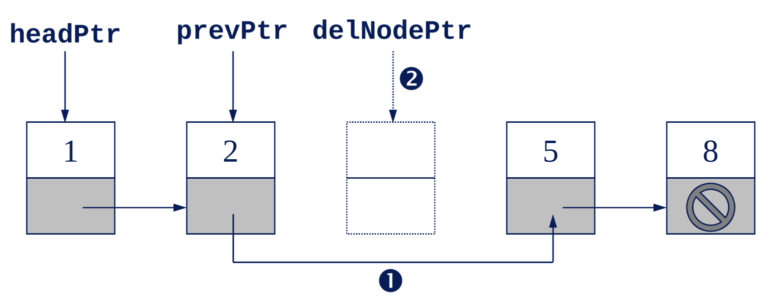
Deleting Other Node (Ordered List)



Find the node previous to the node to be deleted
Set the previous node to point to the node after the node to be deleted

27

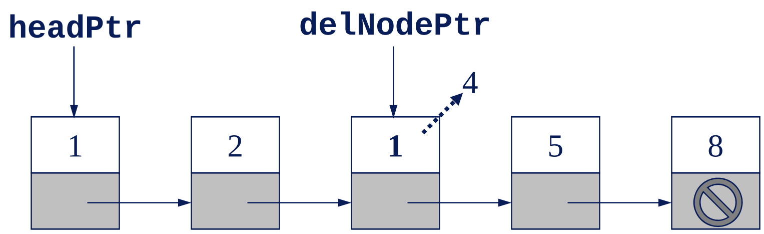
Deleting Other Node (Ordered List)



`delete` the node to be deleted
We're done!

28

Deleting Other Node (Unordered List)

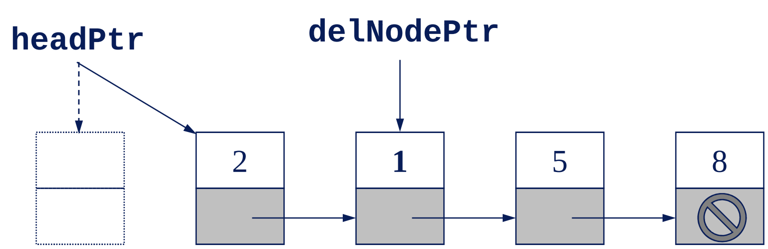


Set the node to be deleted equal to the head node

(all data members except the pointer to next node)

29

Deleting Other Node (Unordered List)



Call the delete first node function to remove the first node in the list

- This approach is more efficient because you don't have to keep track of the previous node

30