Lists

Primitive Data types

Name	Description	Size	Range
char	Character or small integer	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
iong int (long)	Long integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false	1 byte	true or false
float	Floating point number	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character	2 or 4 bytes	1 wide character

Abstract Data Types

- In computing, an abstract data type (ADT) is a mathematical model
  - for a certain class of data structures that have similar behavior; or
  - for certain data types of one or more programming languages that have similar semantics.
- Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types.

Source: http://en.wikipedia.org/wiki/Data_structure

Abstract Data Types

- An *abstract data type* (ADT) is a set of operations
  - The exact implementation of ADT is not mentioned in the definition of ADT
- Basic ADT implementation idea: **write once use again and again**
  - Any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function.
  - Any change in implementation details can be done by merely changing the routines that perform the ADT operations.
  - This change, in a perfect world, would be completely transparent to the rest of the program.

Abstract Data Types

- Abstract data types are purely theoretical entities used
  - to simplify the description of abstract algorithms,
  - to classify and evaluate data structures, and
  - to formally describe the type systems of programming languages.
- However, an ADT may be
  - implemented by specific data types or data structures, in many ways and in many programming languages; or
  - described in a formal specification language.

Abstract Data Types: Example

Example: abstract stack (functional)

- A complete functional-style definition of a stack ADT could use the three operations:
  - **push**: takes a stack state and an arbitrary value, returns a stack state;
  - **top**: takes a stack state, returns a value;
  - **pop**: takes a stack state, returns a stack state;

List and its Operations

- The ***list*** *data type* is a collection type that stores ordered, non-unique elements; that is, it allows duplicate element values.
- Operations
  - traverse through the list
  - search for an element in the list
  - read/update an element in the list (at the beginning, end, anywhere)
  - add a new element in the list
    - add at the beginning of the list
    - add at the end of the list
    - add/insert anywhere in the list
  - Delete an element from the list
    - delete from the beginning of the list
    - delete from the end of the list
    - delete any element in the list

List implemented as an array

- The static declaration of an array requires the size of the array to be known in advance
- Each add/delete operation is of the order O(?)
- Each read/write operation is of the order O(?)

List: Traverse through the list

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

List data	2	4	6	8	10	12	14	16

```
for (int i = 0; i < N; i++)
        cout<<a[i];
```

Complexity : $O(N)$

Assumptions:

1 - **N** → Total number of the elements , currently in the list

2 - The **size** of the array is fixed and is greater than (>) ***N***

In the above example, suppose that

1 - **N = 8**

2 - **int a[10];** The size of the array = 10 > ***N***

List: Search for an element in the list

- Search for a specific element in the list
  - e.g., search and return the index of 10, if found in the list

List data	2	4	6	8	10	12	14	16

$SearchElement = 10;$

for (int $i = 0; i < N; i{+}{+}$){
    if (a[i] = = $SearchElement$){
        $index = i;$
        return $index$;
    }
}

$index = -1;$

cout<< “Element not found”;

return *index*;

Complexity : O(*N*)

List: Read/Update an Element

- Read/update an element in the list
  - at the beginning,
  - at the end
  - anywhere
  - Example
    - Read/print *i*th element (*i*=3)
    - update *j*th element (*j*=4)

				i	j			
Read *i*th element	2	4	6	8	10	12	14	16
Update *i*th element	2	4	6	8	15	12	14	16

- $i = 3;\ cout << a[i];$ → **8**
- $j = 4;\ a[j] = 15;$

Complexity : O(1)

List: Add Element

- Add at the beginning:
  - Step 1 : Move all the elements towards right, creating space for one more element in the beginning
  - Step 2 : Add the new element at the beginning
  - Step 3 : Increment the size of the array by one

Initial data	2	4	6	8	10	12	14	16	
Move elements	2	2	4	6	8	10	12	14	16
Add new element	1	2	4	6	8	10	12	14	16

$N = N + 1;$

for (int $i = N\text{-}1;\ i > 0;\ i\text{--}$)

$\quad a[i] = a[i\text{-}1];$

$a[0] = NewElement;$ // (= 1)

Complexity : O(N)

List: Add Element

- Add at the end:
  - Step 1 : Add the new element at end
  - Step 2 : Increment the size of the array by one

Initial data	2	4	6	8	10	12	14	16	
Add new element	2	4	6	8	10	12	14	16	17

$N = N + 1;$

$a[N\text{-}1] = NewElement$; // (= 17)

Complexity : O(1)

List: Add Element

- Add anywhere in the list (e.g., at the *j*th location)
  - Step 1 : Move all the elements, starting from the index *j*, towards right,
  - Step 2 : Add the new element at the index *j*
  - Step : Increment the size of the array by one

				j					
Initial data	2	4	6	8	10	12	14	16	
Move elements	2	4	6	8	8	10	12	14	16
Add new element	2	4	6	6	8	10	12	14	16

$N = N + 1;$

for (int $i = N\text{-}1;\ i > \mathrm{j};\ i\text{--}$)

$\quad a[i] = a[i\text{-}1];$

$a[\mathrm{j}] = NewElement$; // (= 6)

Complexity : O(N)

List: Delete Element

- Delete the beginning element:
  - Step 1 : Move all the elements towards left, overriding the first element
  - Step 2 : Decrement the size of the array by one

Initial data	2	4	6	8	10	12	14	16
Move elements	4	6	8	10	12	14	16	16
Decrement size	4	6	8	10	12	14	16	

$$
\begin{aligned}
&\text{for (int } i = 0;\ i < N\text{-1};\ i\text{++)} \\
&\qquad a[i] = a[i+1]; \\
&N = N - 1;
\end{aligned}
$$

Complexity : $O(N)$

List: Delete Element

- Add at the end:
  - Step 1 : Delete the element from the end
  - Step 2 : Decrement the size of the array by one

Initial data	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**
Delete element and decrement size	**2**	**4**	**6**	**8**	**10**	**12**	**14**	

$N = N - 1;$

Complexity : O(1)

List: Delete Element

- Delete from anywhere in the list (e.g., at the *j*th location)
  - Step 1 : Move all the elements, after the index *j*, towards left
  - Step 2 : Decrement the size of the array by one

				j				
Initial data	2	4	6	8	10	12	14	16
Move elements	2	4	6	10	12	14	16	16
Decrement size	2	4	6	10	12	14	16	

$$\text{for (int } i = j;\ i < N\text{-}1;\ i\text{++)}$$
$$a[i] = a[i+1];$$
$$N = N - 1;$$

Complexity : $O(N)$

List: A few more operations

- A few specific operations
  - Search and update a specific element in the list (at the beginning, end, anywhere)
  - Add a new element in the list
    - Insert a new element after a specific element (not at a specified location)
      - **Step 1**: Search for the location/index ***j*** of ***AfterElement***
      - **Step 2** : Move all the elements, starting from ***j+1***, towards right
      - **Step 3** : Add the new element at the index ***j+1***
      - **Step 4** : Increment the size of the array by one
    - Insert a new element before a specific element (not a specific location)
      - **Step 1**: Search for the location/index ***j*** of ***BeforeElement***
      - **Step 2** : Move all the elements, starting from ***j***, towards right
      - **Step 3** : Add the new element at the index ***j***
      - **Step 4** : Increment the size of the array by one
    - Complexity ?

List: A few more operations

- A few specific operations
  - Delete an element from the list
    - Delete a specific element (not at a specified location) from the list
      - **Step 1**: Search for the location/index ***j*** of ***ElementToDelete***
      - **Step 2** :Move all the elements, after the index ***j***, towards left, overriding the ***j*th** element
      - **Step 3** : Decrement the size of the array by one
    - Complexity ?

List – Array Implementation Challenges

- The cost of Inserting an element anywhere inside the list is $O(N)$
- The cost of deleting an element anywhere from the list is $O(N)$
- Can we reduce this cost to $O(1)$?
- Further Challenges
  - Static declaration vs dynamic declaration

List implemented as an dynamic array

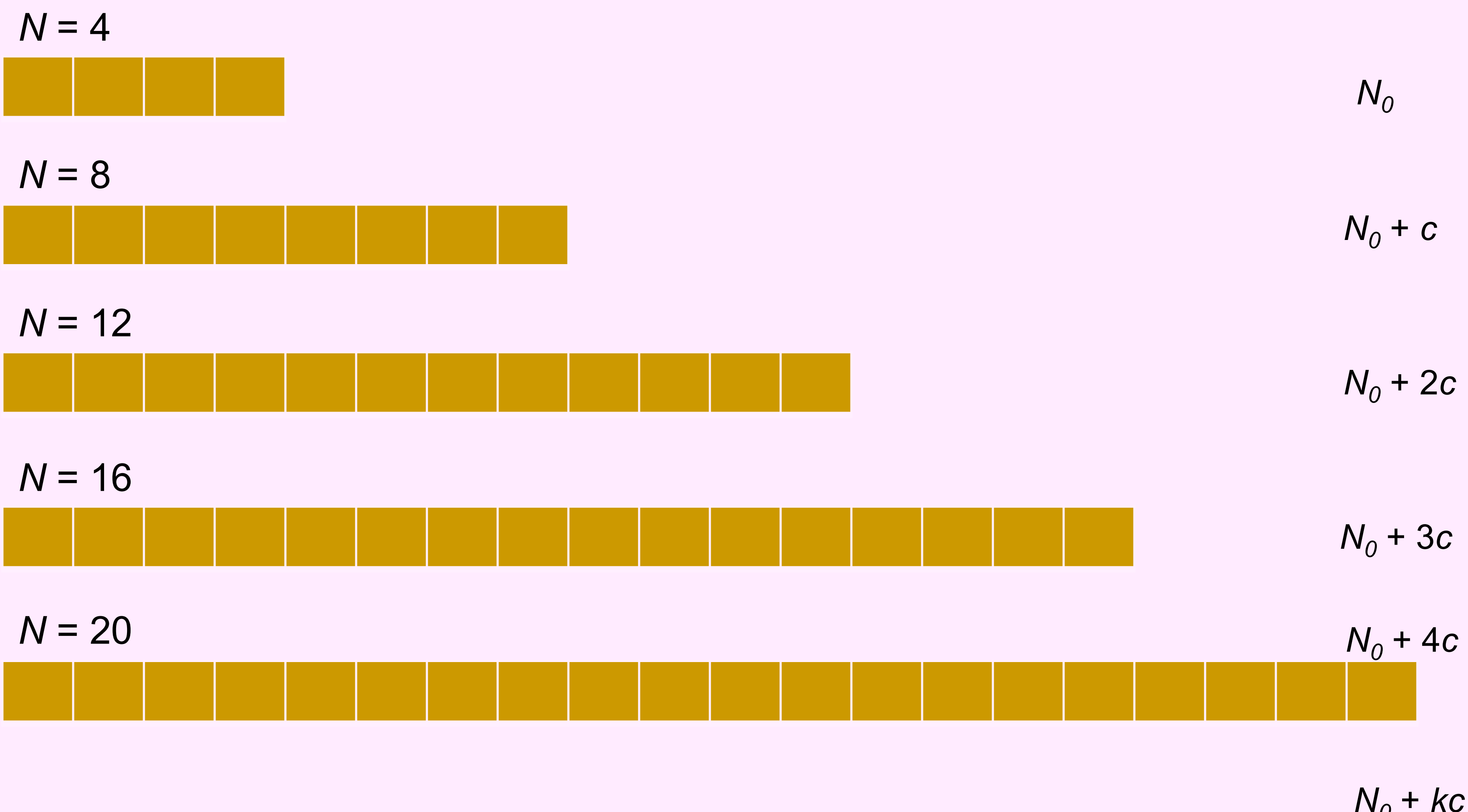
- The static declaration of an array requires the size of the array to be known in advance
- What if the actual size of the list exceeds its expected size?
- Solution?
  - Dynamic array
  - Linked List

List implemented as an dynamic array

- Dynamic Array growth strategy
  - Increase the size by a constant (tight strategy)
    - $N = N + c$
  - Double the size of the array
    - $N = 2*N$

Increase the size by a constant

- **Increment the size by a constant number c, each time the size needs to be re-adjusted**
- **Copy the contents of the previous list into the new list**



Static List: Add Element

- Add at the end:
  - Step 1 : Add the new element at the end
  - Step 2 : Increment the size of the array by one

Initial data	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	
Add new element	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**17**

$$a[N] = NewElement\ ; // (= 17)$$

$$N = N + 1;$$

Complexity : O(1)

Dynamic List: Add Element

Increase the size by a constant

- Add at the end:
  - Step 1 : Increment the size of the array by a constant
  - Step 2 : Copy the old list into the new list
  - Step 3 : Rename the new list as the original list
  - Step 4 : Add the new element at end

```
if index == N  { // if the current index exceeds the size of the list
    N = N + c;        // increase the size of the list by c
    int *tempA =  new int [N]; // allocate memory for the new list
    for (int i = 0; i < N - c; i++)
        tempA[i] = A[i]; // copy the old list into the new list
    delete [] A;
    A = tempA;              // rename the new list as the original list
    }
    A[index]= NewElement;
    Index ++;
```

Increase the size by a constant

Increment the size by a constant number c, each time the size needs to be re-adjusted

- **Copy the contents of the previous list into the new list**

$N = 4$ — N_0

$N = 8$ — $N_0 + c$

$N = 12$ — $N_0 + 2c$

$N = 16$ — $N_0 + 3c$

$N = 20$ — $N_0 + 4c$

.

.

$N_0 + kc$

$N = N_0 + kc$

$k = (N - N_0) / c$

Running Time: $N_0\,\mathrm{k} + c(1 + 2 + \ldots + \mathrm{k})$

$= N_0\,\mathrm{k} + ck\,(\mathrm{k}+1)/2$

$= \mathbf{O}(k^2)$

$= \mathbf{O}(N^2)$

Double The Size of the Array

- **Increment the size twofold each time the size needs to be re-adjusted**
- **Copy the contents of the previous list into the new list**

$N = 4$

N_0

$N = 8$

$2N_0$

$N = 16$

$4N_0$

.

.

$N_0 \times 2^k$

$$N = N_0 \text{ x } 2^k$$

$$k = \log_2(N/N_0)$$

$$\text{Running Time: } = N_0(1 + 2 + 4 + 2^k)$$

$$= N_0 (2^{k+1} - 1)$$

$$= N_0(2^{\log_2(N/N_0)+1} - 1) = 2N - N_0$$

$$= O(N)$$

Dynamic List: Add Element Double The Size of the Array

- Add at the end:
  - Step 1 : Double the size of the array
  - Step 2 : Copy the old list into the new list
  - Step 3 : Rename the new list as the original list
  - Step 4 : Add the new element at end

```
if index == N  {  // if the current index exceeds the size of the list
    N = N * 2;          // double the size of the list
    int *tempA =  new int [N];  // allocate memory for the new list
    for (int i = 0; i < N / 2; i++)
        tempA[i] = A[i];  // copy the old list into the new list
    A = tempA;             // rename the new list as the original list
    }
    A[index]= NewElement;
    Index ++;
```

Linked List

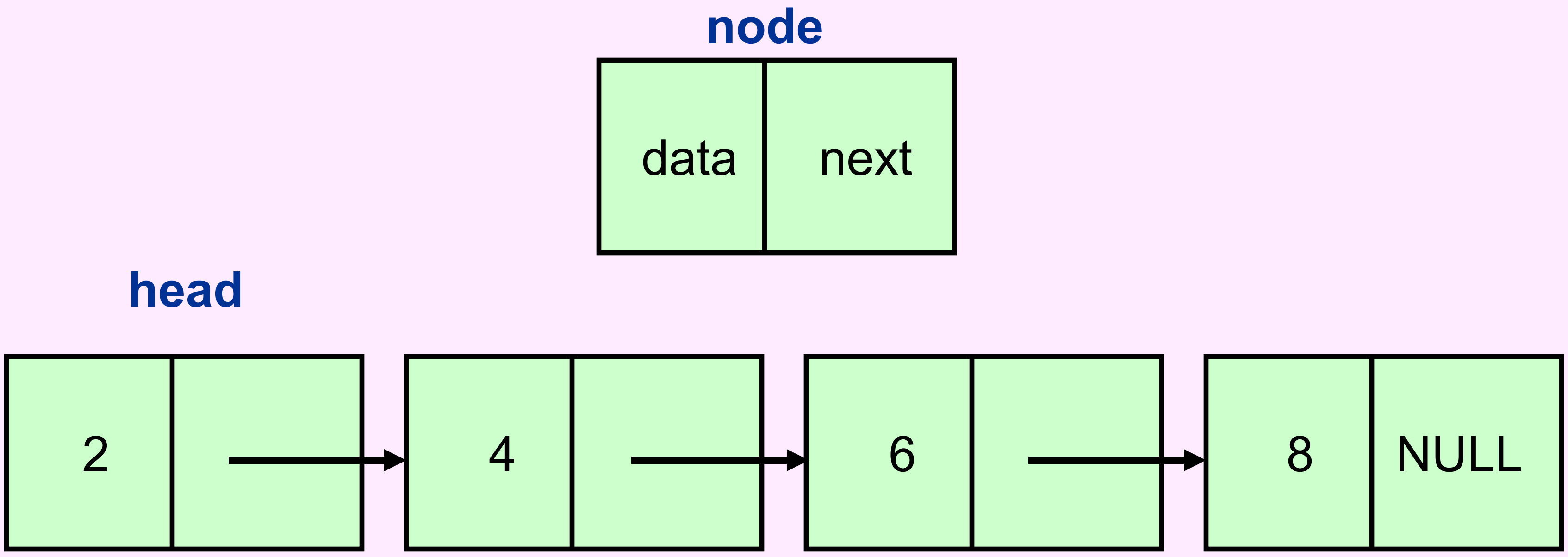
- A continuous list of data stored in a non-contiguous linked-list of nodes

- A node is an abstract data type such as a structure
- Each node consists of at least two types of elements
  - An element to store some data/information
  - A pointer to the next node in the list

- Usually the first node is known as the header node
- The pointer for the last node is usually a null pointer

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

Linked List



Linked List - Operations

- Create the head node
- Traverse through the list
- Search for an element in the list
- Read/update an element in the list (beginning, end, anywhere)
- Add a new element in the list
  - Append at the beginning of the list
  - Append at the end of the list
  - add/insert anywhere in the list
    - Insert before a known element in the list
    - Insert after a known element in the list
- Delete an element from the list
  - delete from the beginning of the list
  - delete from the end of the list
  - delete any element in the list

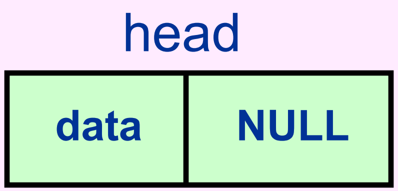
Linked List: Create the head node

```
struct node
    {
    int data;
    node *next;
    };
node *head;
void main()
    {
    head = new node;
    }
```

Linked List: Add an element at the head node

- **Approach 1:**
  - head node may contain data as well as serves as a global pointer to the start of the linked list

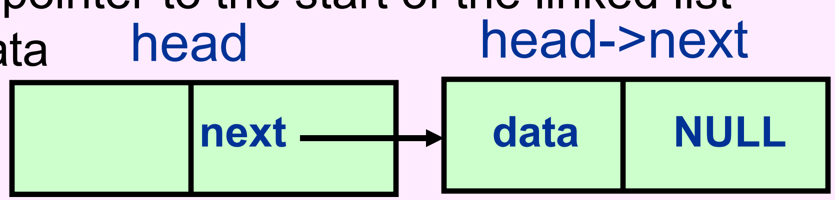
    head = new node;
    cin>>head->data;
    head->next=NULL;

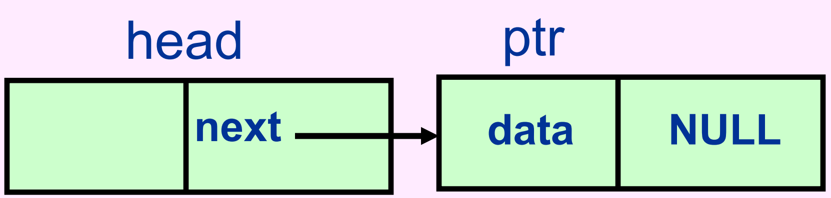


Approach 2:

- head node only serves as a global pointer to the start of the linked list
- head node does not contain any data

    head->next = new node;
    cin>> head->next ->data;
    head->next->next=NULL;
    OR
    node **ptr* =new node;
    cin>> ptr ->data;
    head->next=ptr;
    ptr->next=NULL;





Complexity : O(*1*)

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 2 → 4 → 6 → 8 → 10 → 12 → 14 → 16 → **NULL**

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

Complexity : O(N)

Linked List: Search for an element in the list

- Search for a specific element in the list
  - e.g., search and return the pointer to the element containing data=10, if found in the list

List data 2 → 4 → 6 → 8 → 10 → 12 → 14 → 16 → **NULL**

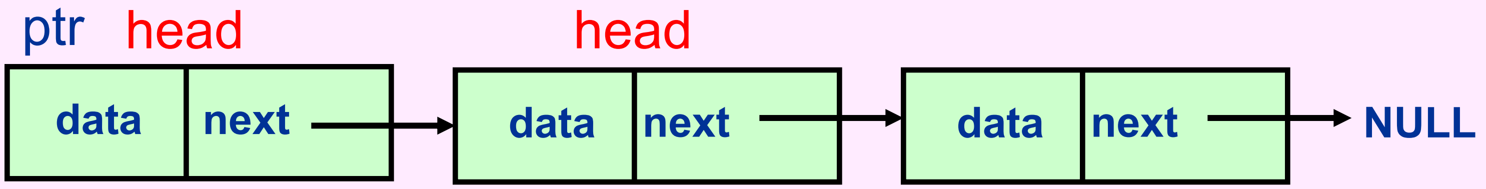
```
SearchElement = 10;
for (node *ptr = head; ptr != NULL; ptr = ptr->next)
        {
        if (ptr->data == SearchElement){
             indexPtr = ptr;
              return indexPtr;
         }
}
indexPtr = NULL;
cout<< "Element not found";
return indexPtr;
```

Complexity : $O(N)$

Linked List: Append a new element before/at the head node

- **Approach 1:**
  - head node contains data as well as a global start pointer

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

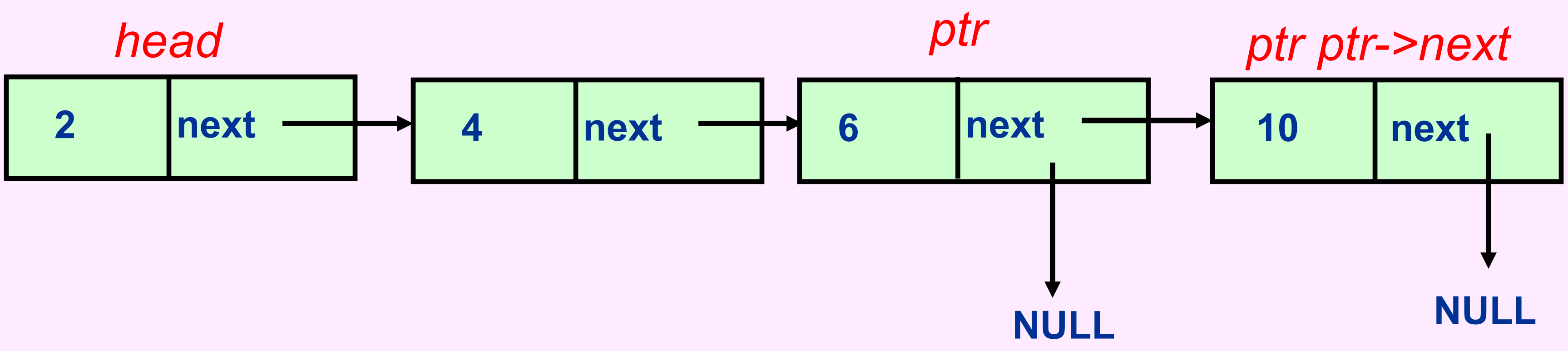


Complexity : O(*1*)

Linked List: Append a new element after the tail node

- head node contains data as well as a global start pointer

```
node *ptr;
for (ptr = head; ptr->next != NULL; ptr = ptr->next){}
    ptr->next = new node;
    ptr = ptr->next;
    ptr->data = NewData; // e.g NewData =10;
    ptr->next = NULL;
```

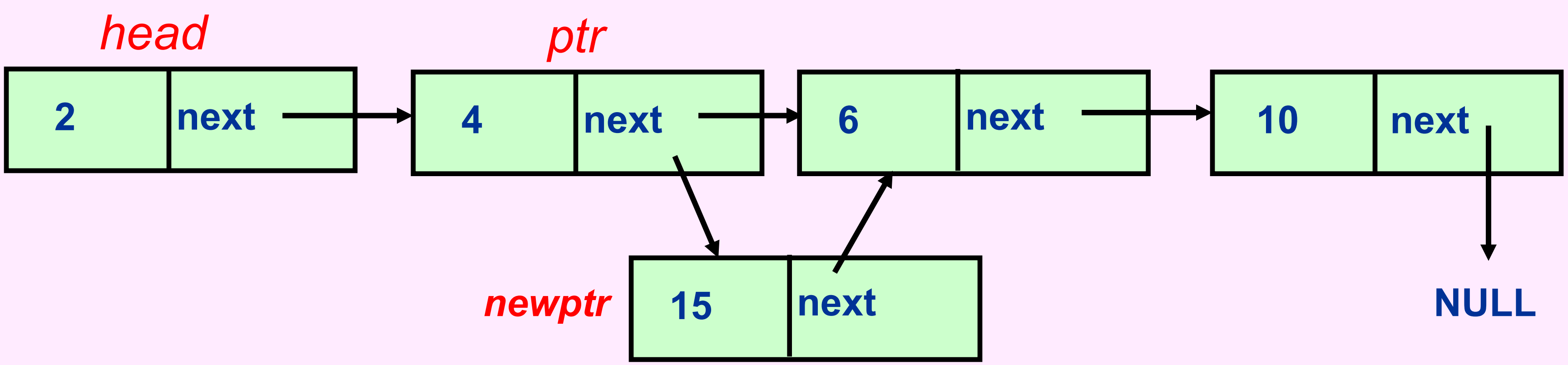


Complexity : O(*N*)

Linked List: Insert a new element after a specific node

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

```
node* newptr;
for(node *ptr=head; 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;
      ptr->next=newptr;
      } // end if
  } // end for
```

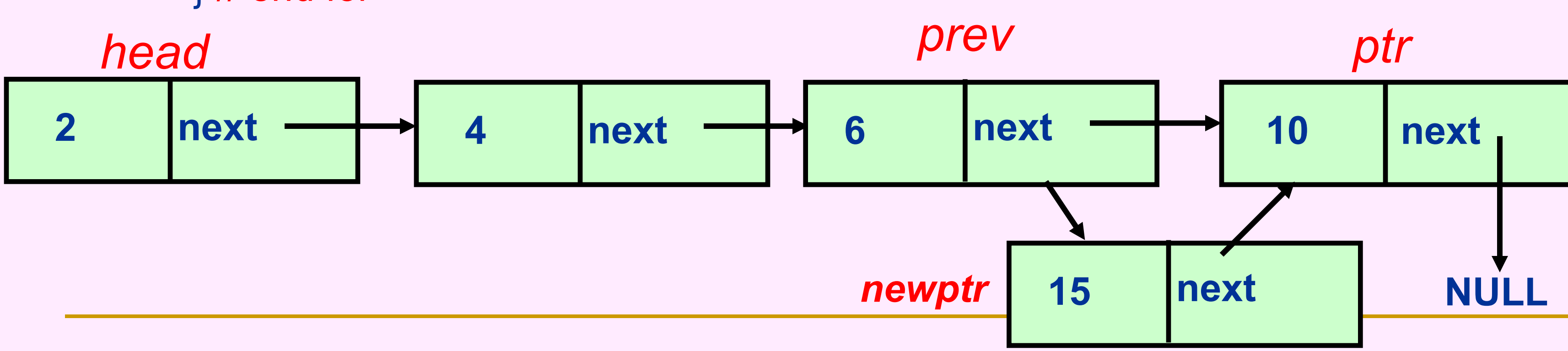


Complexity : O(*N*)

Linked List: Insert a new element before a specific node

- Suppose you want to insert before the node with data = '*beforeValue*'

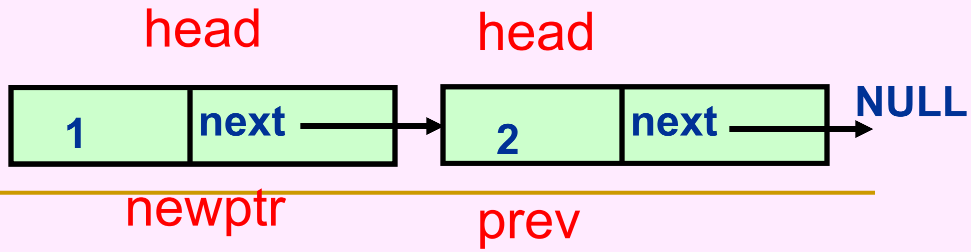
```
node* newptr;
node *prev=head;
for(node *ptr=head->next; ptr!=NULL; ptr=ptr->next){
    if(ptr->data == beforeValue)    //  e.g beforeValue =10;
      {
      newptr=new node;
      newptr->data=NewData;     //  e.g NewData =16;
      prev->next=newptr;
      newptr->next=ptr;
      } // end if
   prev=prev->next;
   } // end for
```



Complexity : O(*N*)

Linked List: Insert a new element before a specific node (Only one node is currently there)

```
node* newptr;
node *prev=head;
if(head->data == beforeValue) {    //  e.g beforeValue =2;
      newptr=new node;
      head=newptr;
      head->next=prev;
      newptr->data=NewData;    //   e.g NewData =1;
      return;
      }
for(node *ptr=head->next; ptr!=NULL; ptr=ptr->next){
    if(ptr->data==beforeValue)
      {
      newptr=new node;
      prev->next=newptr;
      newptr->data=NewData;
      newptr->next=ptr;
      } // end if
   prev=prev->next;
 } // end for`
```

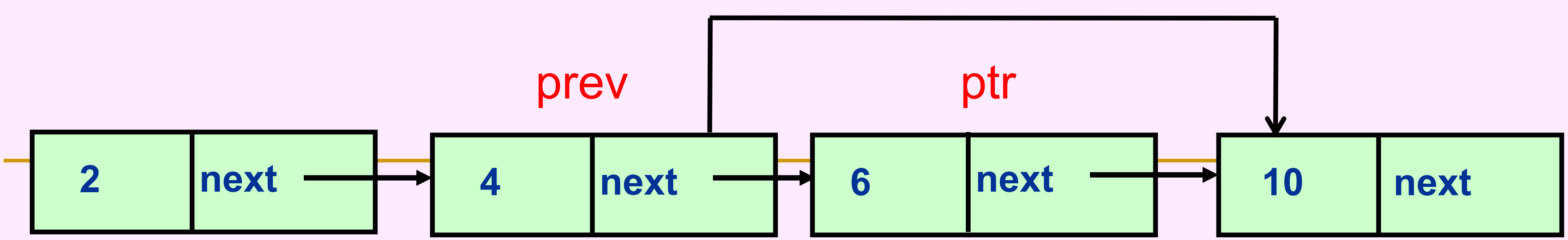


Linked List: Delete an element from the list

- Suppose you want to delete the node with data = '*ValueToDelete*'

```
node *prev=head;
for(node *ptr=prev->next; ptr!=NULL; ptr=ptr->next){
if(ptr->data == ValueToDelete) // e.g ValueToDelete = 6;
  {
  prev->next=ptr->next;
  delete(ptr);
  return;
  }
  prev=prev->next;
}
```

Complexity : O(*N*)

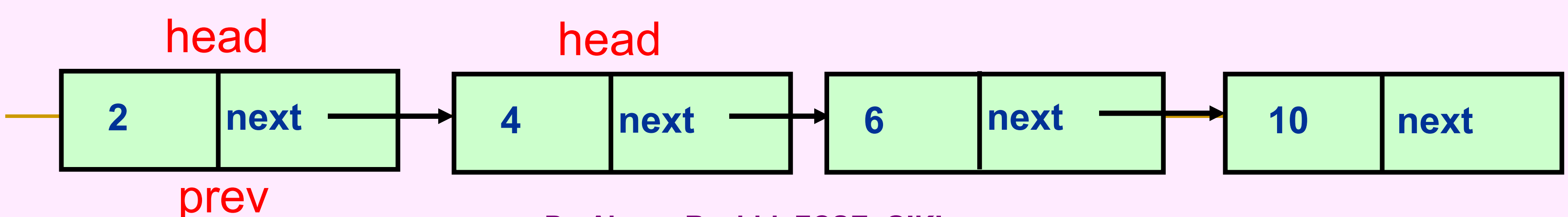


Linked List: Delete an element from the list

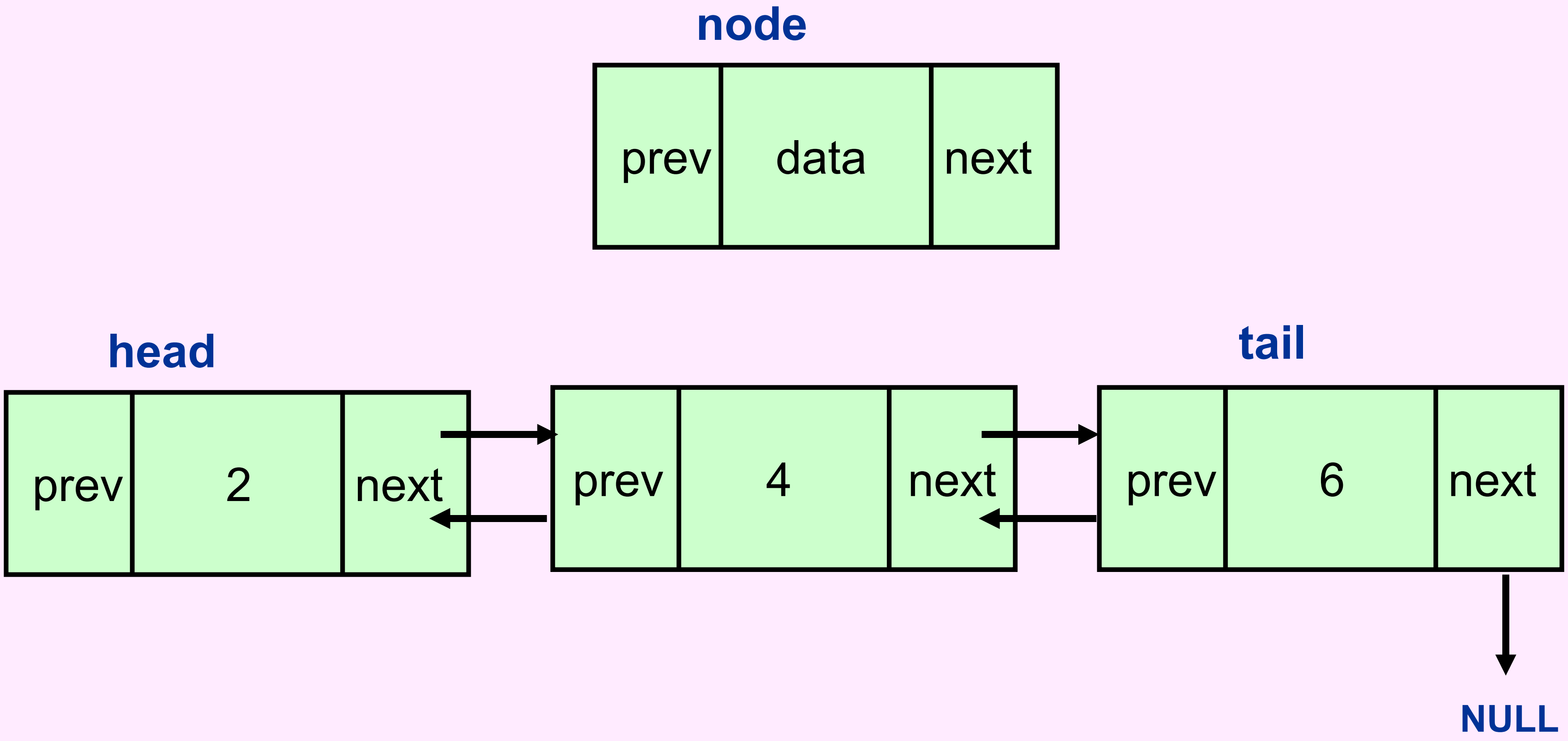
- **Delete the head node from the list**

```
node *prev=head;
if(head->data == ValueToDelete)
  {
  head=head->next;
  delete(prev);
  return;
  }
for(node *ptr=prev->next; ptr!=NULL; ptr=ptr->next){
if(ptr->value==Value)
  {
  prev->next=ptr->next;
  delete(ptr);
  return;
  }
  prev=prev->next;
}
```

Complexity : O(1)



Double Linked List



Double Linked List: Create the head node

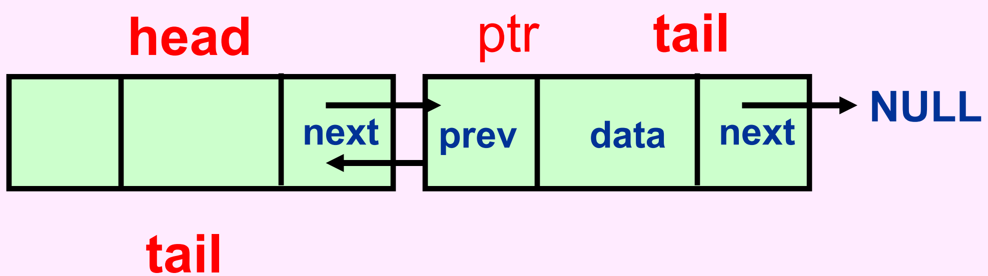
```
struct node
    {
    int data;
    node *next;
    node *prev;
    };
node *head, *tail;
void main()
    {
    head = new node;
    head->next = NULL;
    head->prev = NULL;
    tail = head;        }
```

Double Linked List: Add an element at /after the head node

Approach 2:

- head node only serves as a global pointer to the start of the linked list
- head node does not contain any data

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

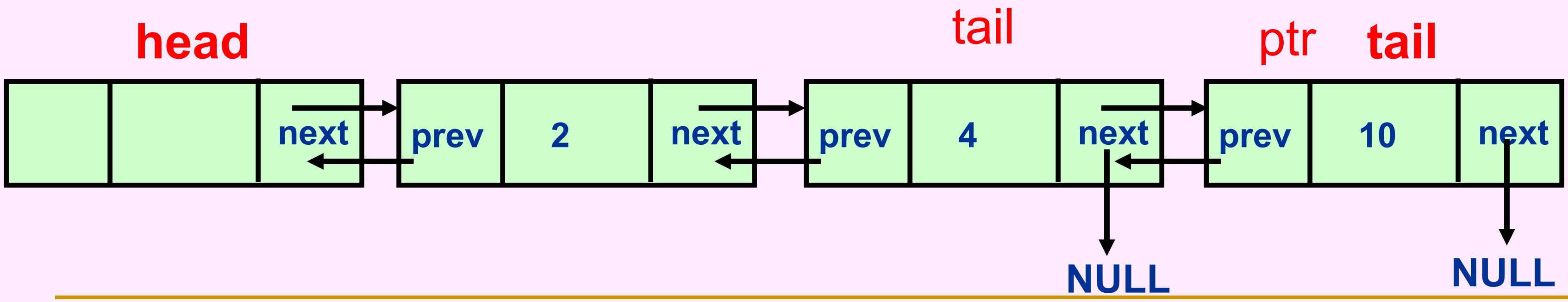


Complexity : O(*1*)

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;
}
```

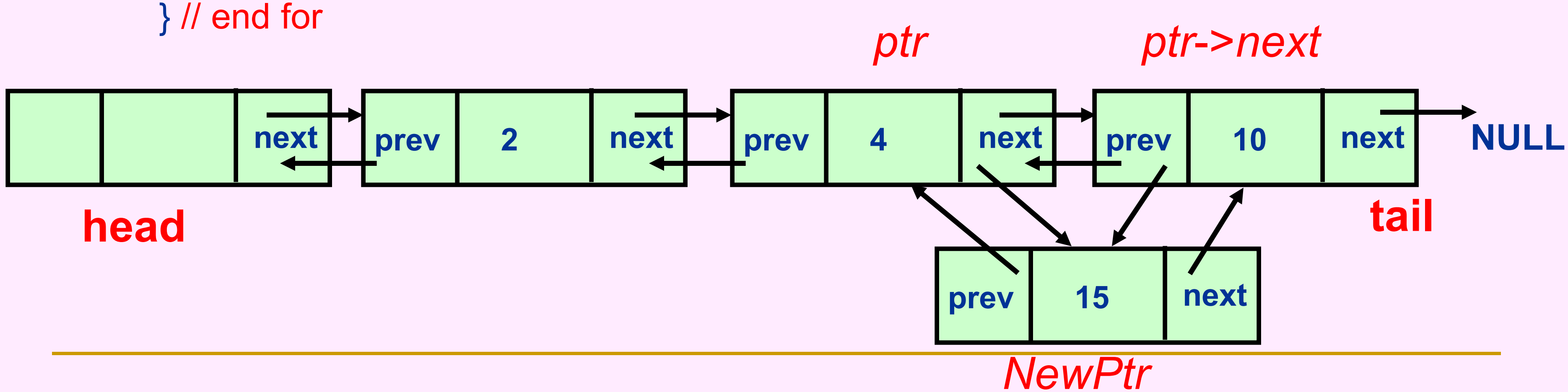


Complexity : O(1)

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
```

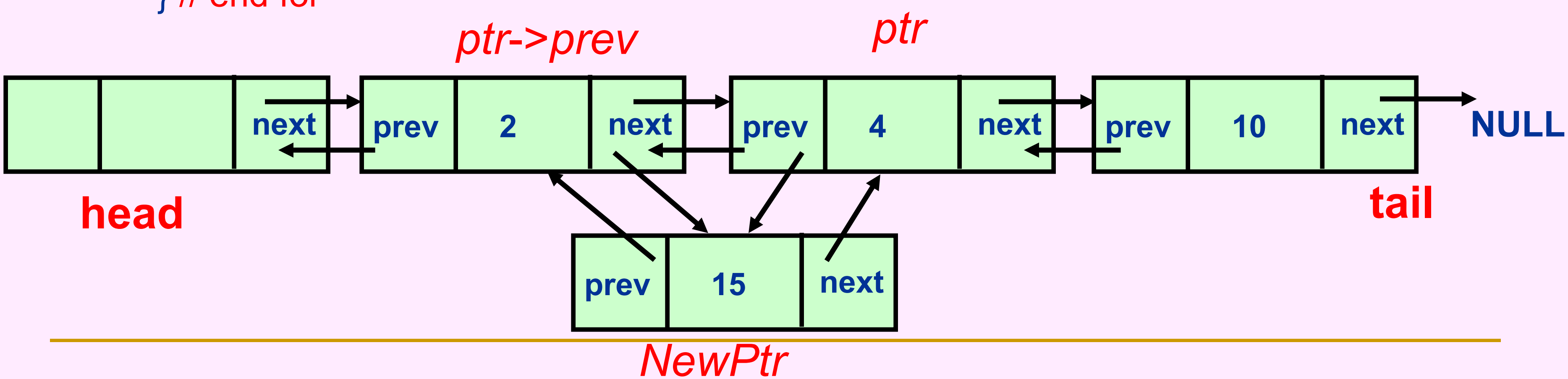


Complexity : *O*(*N*)

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

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

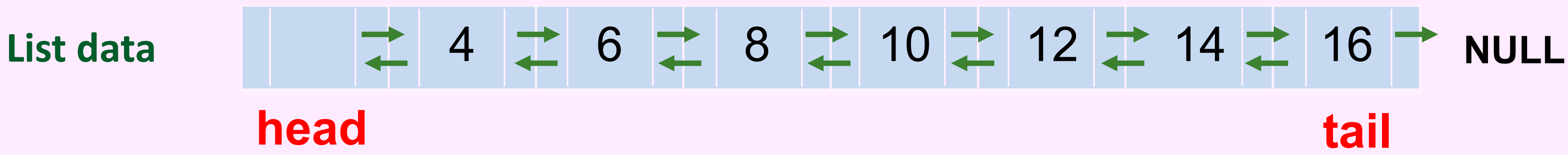
```
node* NewPtr;
 for(node *ptr = head->next; ptr != NULL;ptr = ptr->next){
    if(ptr->data == BeforeValue) {        e.g BeforeValue =4
      NewPtr = new node;
      NewPtr->data = NewData;//      e.g NewData =15;
      NewPtr->next = ptr;
      (ptr->prev)->next = NewPtr;
      NewPtr->prev = ptr->prev;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




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

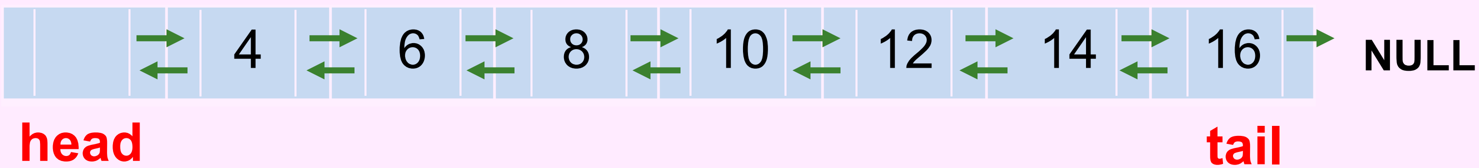
OUTPUT: 4 6 8 10 12 14 16

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<<“ “;
```

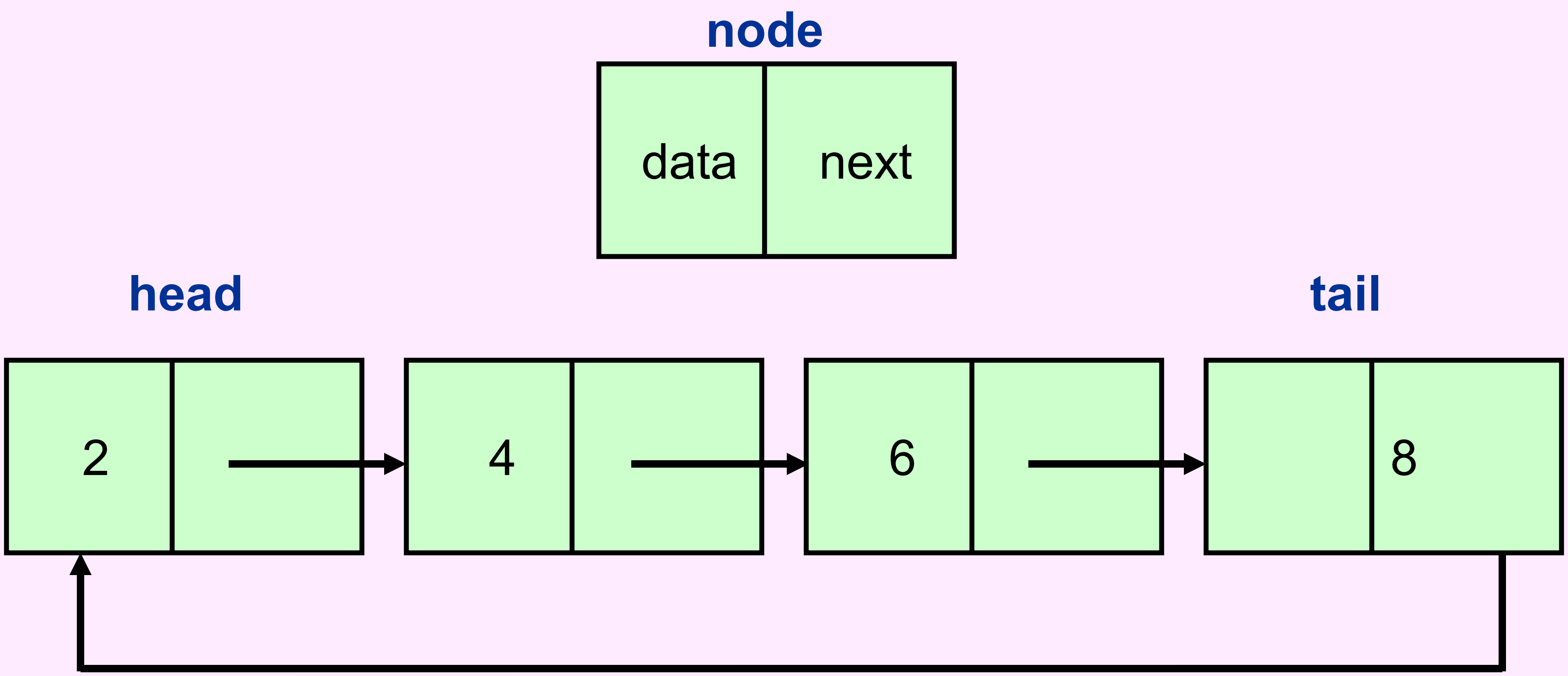
OUTPUT: 16 14 12 10 8 6 4

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



[Sample codes](http://www.go4expert.com/forums/showthread.php?t=1282)
http://www.go4expert.com/forums/showthread.php?t=1282
http://www.indiastudychannel.com/resources/4083-Program-for-implementation-of-CIRCULAR-LINKED-LIST.aspx

Circler Linked List

- Applications
- Round Robin Time Sharing jobs of Operating System, ie simple multi tasking by PC
  Read more:
- http://wiki.answers.com/Q/Application_of_Circular_Linked_List_in_real_life#ixzz1ZoMP6QJL

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