

2/4

Linked List

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

Singly Linked List

Self-referenced structures – review

Exercise: Link a, b and c nodes;

```
struct List {  
    string who;  
    char  data;  
    List *link;  
};
```

List **a, b, c**;

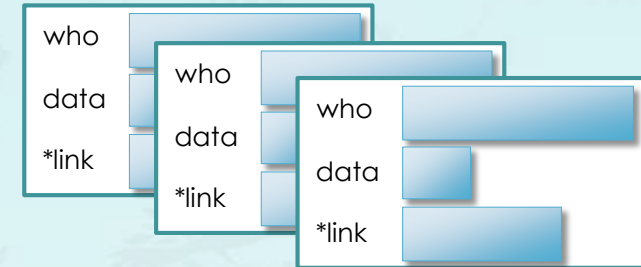
a.data = 'A';

b.data = 'B';

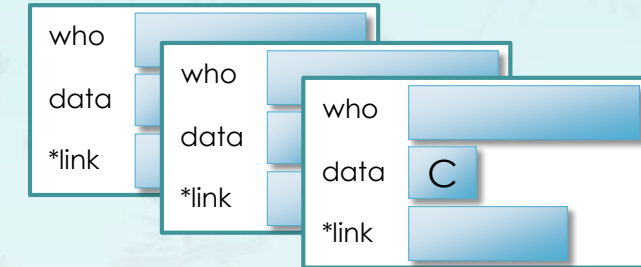
c.data = 'C';

a.link = b.link = c.link = nullptr;

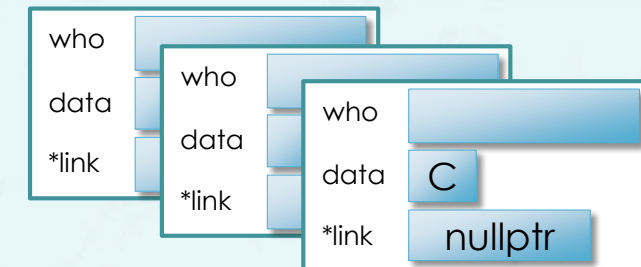
List a, b, c;



a.data = 'A';
b.data = 'B';
c.data = 'C';



a.link = b.link = c.link = nullptr;



Self-referenced structures – review

Exercise: Link a, b and c nodes;

```
struct List {  
    string who;  
    char  data;  
    List *link;  
};
```

List **a, b, c;**

a.data = 'A';

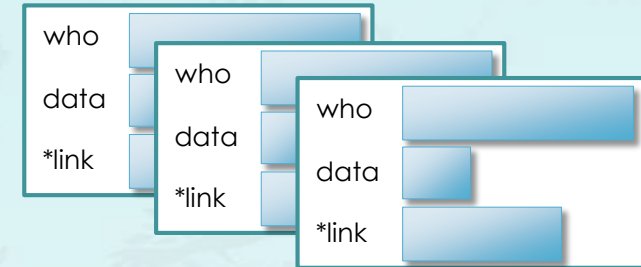
b.data = 'B';

c.data = 'C';

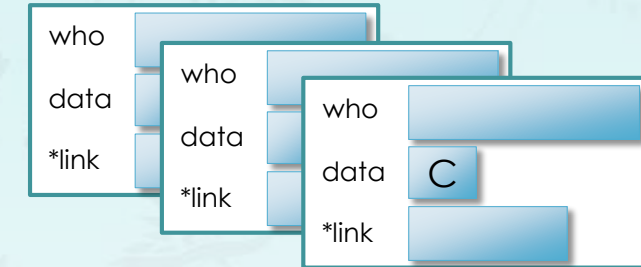
a.link = b.link = c.link = nullptr;

```
_____  
_____  
_____;
```

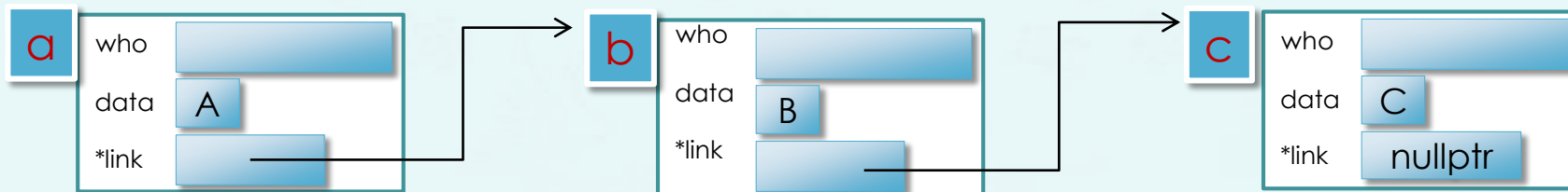
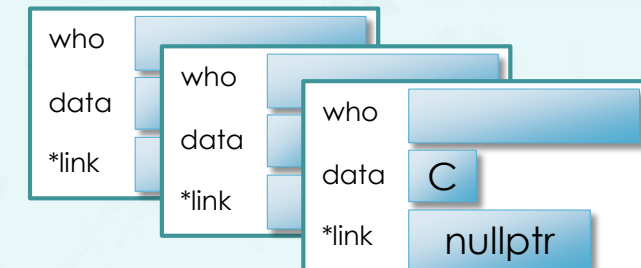
List a, b, c;



a.data = 'A';
b.data = 'B';
c.data = 'C';



a.link = b.link = c.link = nullptr;



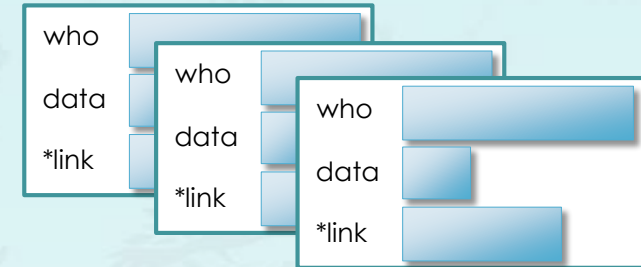
Self-referenced structures – review

Exercise: Link a, b and c nodes;

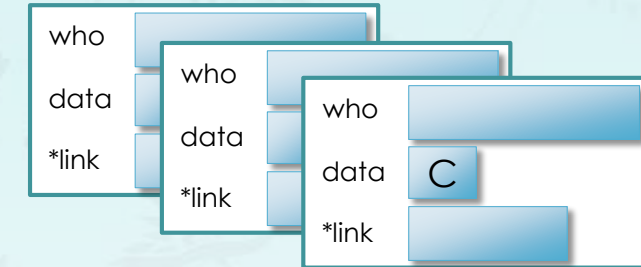
```
struct List {  
    string who;  
    char  data;  
    List *link;  
};
```

```
List a, b, c;  
List *p, *q, *r;
```

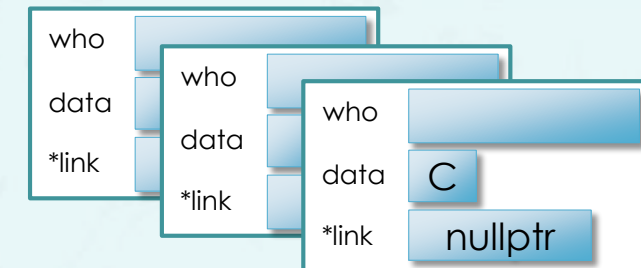
List a, b, c;



a.data = 'A';
b.data = 'B';
c.data = 'C';



a.link = b.link = c.link = nullptr;



Self-referenced structures – review



Exercise: Link a, b and c nodes;

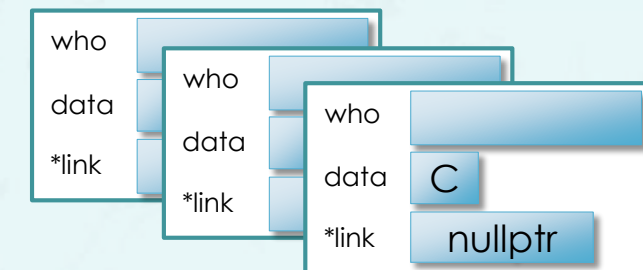
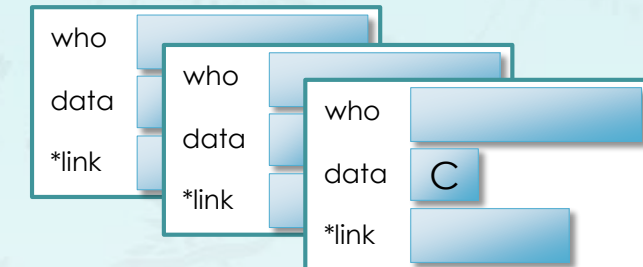
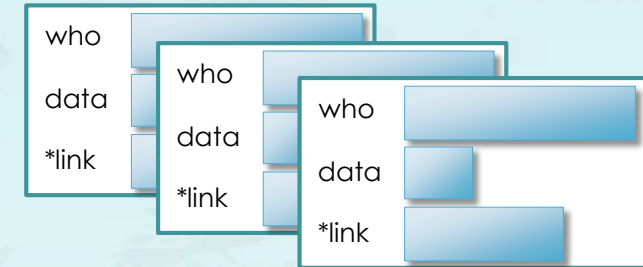
```
struct List {  
    string who;  
    char  data;  
    List *link;  
};
```

```
List a, b, c;  
List *p, *q, *r;
```

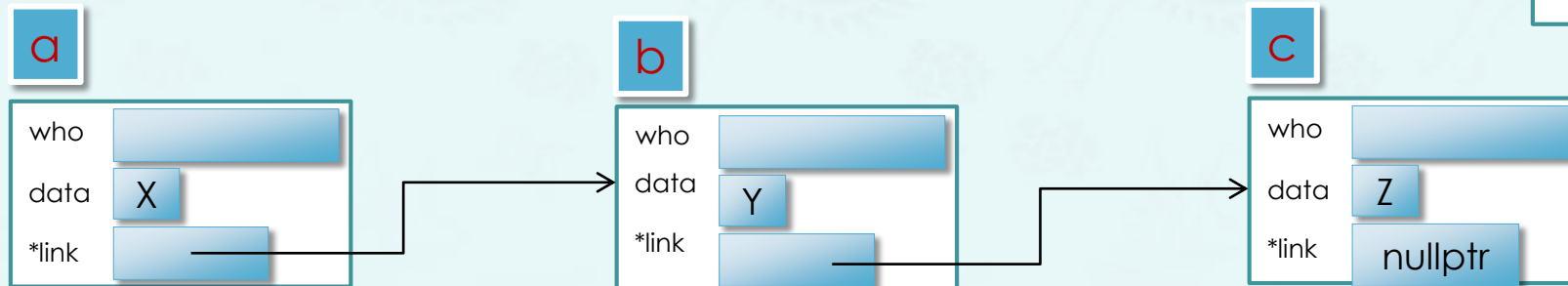
(1)

(2)

(3)



- (1) Let each p, q, and r points to a, b, and c;
- (2) Store each 'X', 'Y', and 'Z' in data
- (3) Connect them using p, q and r as shown below:



Constructor and Destructor

```
class Node {  
public:  
    int    data;  
    Node* next;  
};
```

← constructor, destructor

```
int main( ) {  
    Node* p = new Node;  
    ...  
}
```

constructor →

destructor →

```
struct Node {  
    int    data;  
    Node* next;  
  
    Node(int i=0, Node* n=nullptr){  
        item = i, next = n;  
    }  
    ~Node() {};  
};  
  
int main( ) {  
    Node* p = new Node;  
    ...  
}
```

Dynamic Data Structures

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

← Simplified: No constructor/destructor

...

```
Node* head, *x, *y;
```

```
class Node {  
public:  
    int    data;  
    Node* next;  
};
```

...

```
Node* head, *x, *y;
```

Dynamic Data Structures

```
class Node {  
public:  
    int    data;  
    Node* next;  
};  
  
...  
  
Node* head, *x, *y;
```

```
class Node {  
public:  
    int    data;  
    Node* next;  
  
    Node(int i=0, Node* n=nullptr){  
        item = i, next = n;  
    }  
    ~Node() {};  
};  
  
int main( ) {  
    Node* head, *x, *y;  
    Node* p = new Node;  
    ...  
}
```

```
struct Node {  
    int    data;  
    Node* next;  
};  
  
using pNode Node*;  
  
int main() {  
    pNode head, x, y;  
    pNode p = new Node;  
    ...  
}
```


Dynamic Data Structures

```
class Node {  
public:  
    int    data;  
    Node* next;  
};  
  
...  
  
Node* head, *x, *y;
```

```
class Node {  
public:  
    int    data;  
    Node* next;  
  
    Node(int i=0, Node* n=nullptr){  
        item = i, next = n;  
    }  
    ~Node() {};  
};  
  
int main( ) {  
    Node* head, *x, *y;  
    Node* p = new Node;  
    ...  
}
```

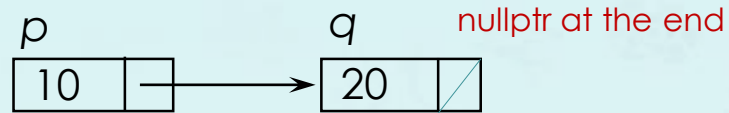
```
struct Node {  
    int    data;  
    Node* next;  
};  
  
using pNode Node*;  
  
int main() {  
    pNode head, x, y;  
    pNode p = new Node;  
    ...  
}
```

Yet another style of constructor: **"initializer"**

```
Node(int i, Node* n): item(i), next(n) {}
```

Linked List

TASK: Code a function that returns the head of following linked list.



```
struct Node {
    int data;
    Node* next;
};
using pNode = Node*;
```

Write a function that creates a node and returns **pNode**.

```
pNode newNode(int val) {
    pNode node = (pNode)malloc(sizeof(Node));
    node->data = val;
    node->next = nullptr;
    return node;
}
```

C

```
pNode newNode(int val) {
    pNode node = new Node;
    node->data = val;
    node->next = nullptr;
    return node;
}
```

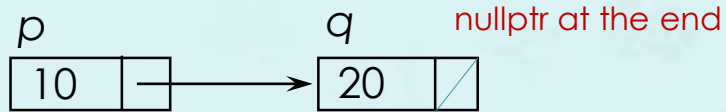
C++

```
pNode node = new Node {0, nullptr};
```

C++

Linked List

TASK: Code a function that returns the head of following linked list.



```
struct Node {  
    int data;  
    Node* next;  
};  
using pNode = Node*;
```

```
pNode newList2(int a, int b) {  
    pNode p = new Node {a, nullptr};  
    pNode q = new Node {b, nullptr};  
    p->next = q;  
    return p;  
}
```

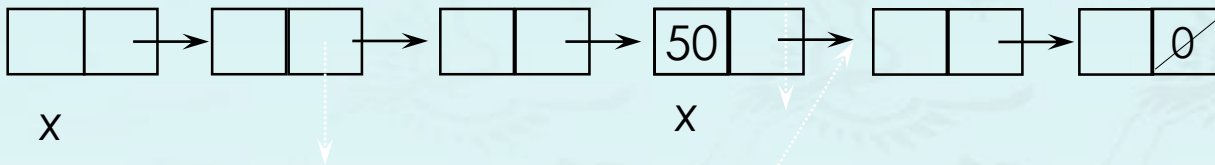
```
pNode newList2(int a, int b) {  
    pNode q = new Node {b, nullptr};  
    pNode p = new Node {a, q};  
    return p;  
}
```

```
pNode newList2(int a, int b) {  
    pNode q = new Node {b, nullptr};  
    return new Node {a, q};  
}
```

Linked List – find()

TASK: Code a function that returns the first node **data = 50** if any, otherwise nullptr.

list



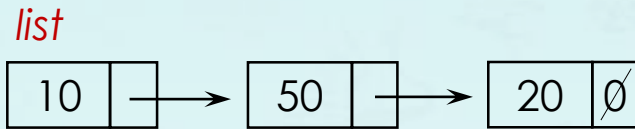
```
pNode find(pNode list, int val)
if (empty(list)) return nullptr;

pNode x = list
while (x != nullptr) {
    if (x->data == val) return x;
    x = x->next;
}
return x;
```

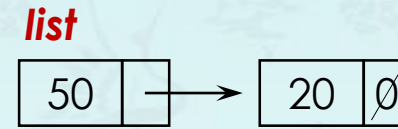
```
bool empty(pNode list)
return list == nullptr;
```

Linked List – `pop_front()`

TASK: Code a function that deletes the first node and returns the new first node.



(before)



(after)

```
pNode pop_front(pNode list)
if (empty(list)) return list;

list = list->next;
return list;
```

What's wrong?

```
pNode pop_front(pNode list)
if (empty(list)) return list;

pNode x = list;
list = list->next;
delete x;
return list;
```

Linked List – **push_front()**

TASK: Code a function that add a node at the beginning of the list.
- If the list is empty, the new node becomes the head node.

list



list



node



list



```
pNode push_front(pNode list, int val)
```

```
if (empty(list))  
    return new Node{val, nullptr};
```

```
pNode node = new Node{val, nullptr};
```

Linked List – **push_front()**

TASK: Code a function that add a node at the beginning of the list.
- If the list is empty, the new node becomes the head node.

list



list



node

list



```
pNode push_front(pNode list, int val)
```

```
if (empty(list))
```

```
    return new Node{val, nullptr};
```

```
pNode node = new Node{val, nullptr};
```

```
node->next = list;
```

```
return node;
```

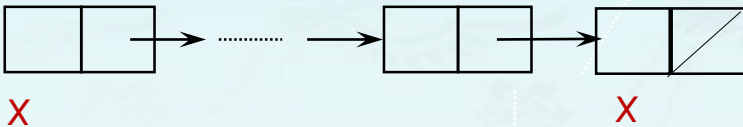
Linked List – **push_back()**

TASK: Code a function that appends a node at the end of the list.
- If the list is empty, the new node becomes the head node.

list



list



```
pNode last(pNode list)
```

```
pNode x = list;  
while (x != nullptr)  
    x = x->next;  
return x'
```

```
pNode push_back(pNode list, int val)
```

```
if (empty(list))  
    return new Node{val, nullptr};
```

```
pNode last(pNode list)
```

```
pNode x = list;  
while (x->next != nullptr)  
    x = x->next;  
return x;
```

Q: Which one is correct?

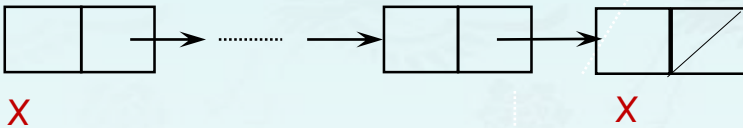
Linked List – **push_back()**

TASK: Code a function that appends a node at the end of the list.
- If the list is empty, the new node becomes the head node.

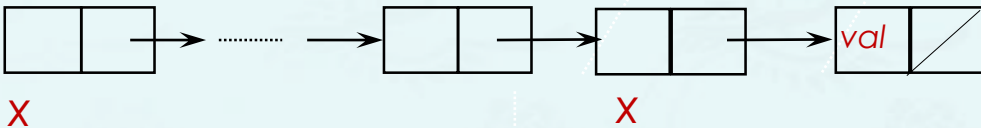
list



list



list



```
pNode push_back(pNode list, int val)
```

```
if (empty(list))  
    return new Node{val, nullptr};
```

```
return list;
```

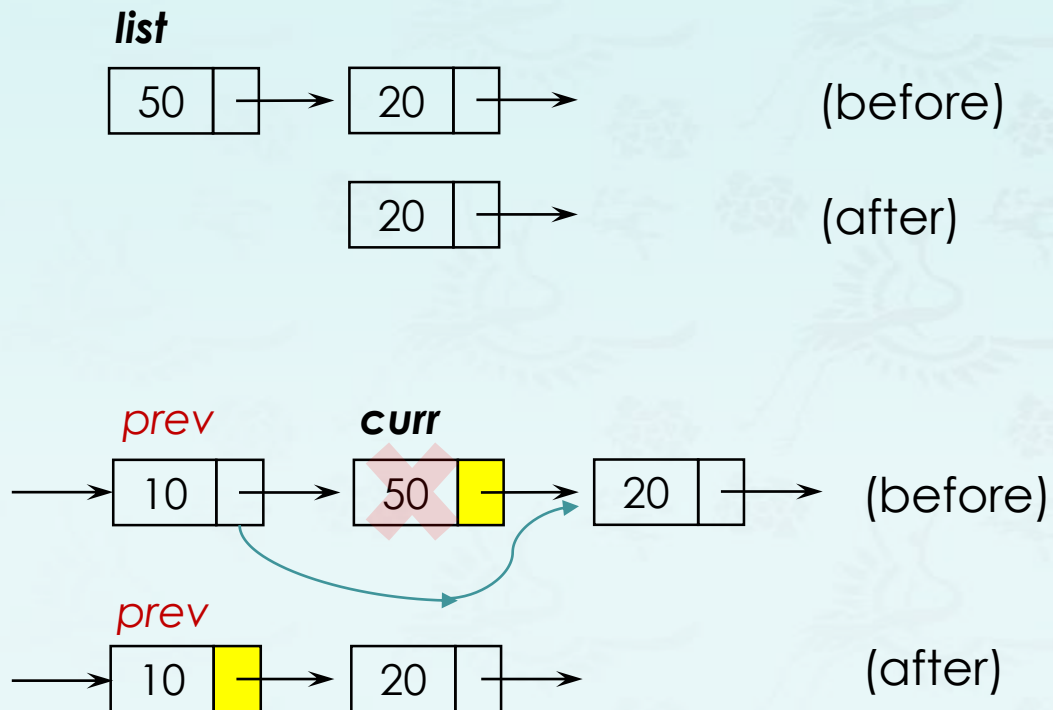
```
pNode last(pNode list)
```

```
pNode x = list;  
while (x->next != nullptr)  
    x = x->next;  
return x;
```

Linked List – pop()

TASK: Code a function that deletes a node with a value specified.

- If the first node(or **head**) is the one to delete, then just invoke **pop_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.



```
pNode pop(pNode list, int val)
```

```
if (list->data == val)
    return pop_front(list);
```

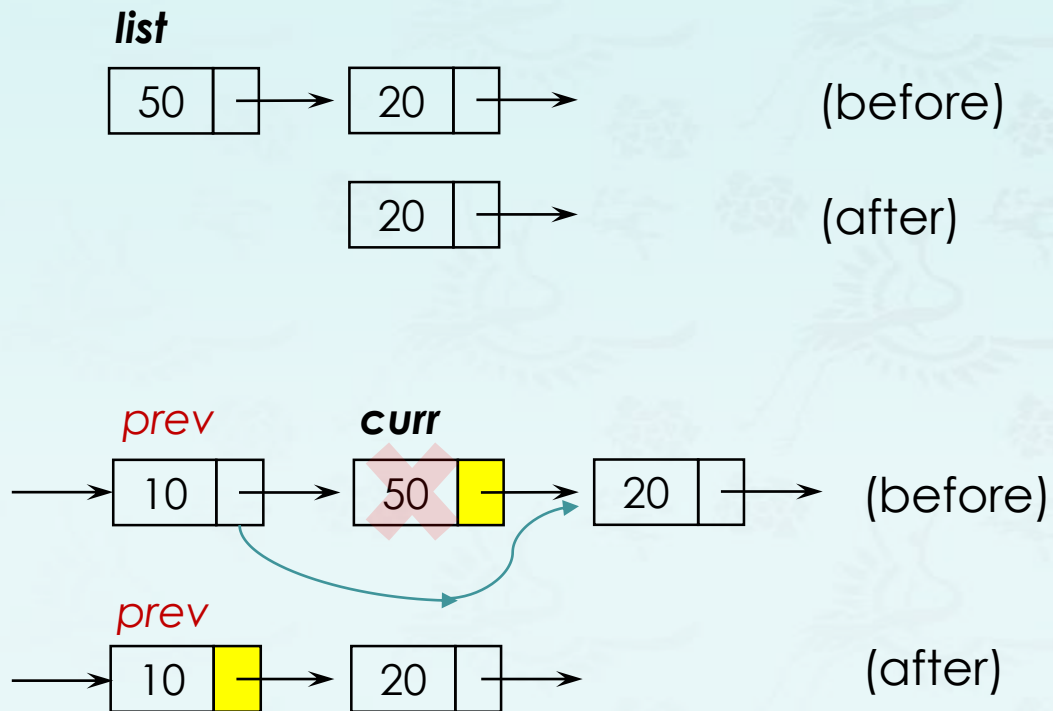
```
pNode curr = list;
pNode prev = nullptr;
while (curr != nullptr) {
```

```
}
return list;
```

Linked List – pop()

TASK: Code a function that deletes a node with a value specified.

- If the first node(or **head**) is the one to delete, then just invoke **pop_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.



```
pNode pop(pNode list, int val)
```

```
if (list->data == val)
    return pop_front(list);
```

```
pNode curr = list;
pNode prev = nullptr;
while (curr != nullptr) {
    if (curr->data == val) {
```

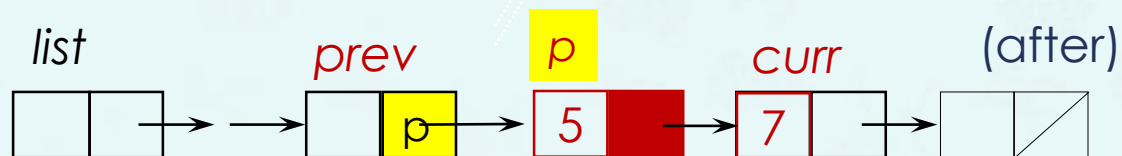
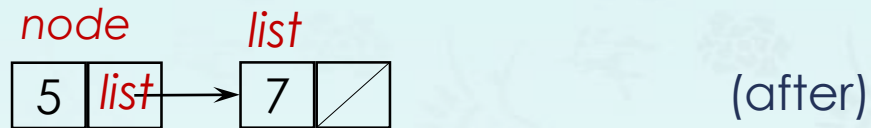
```
}
```

```
}
return list;
```

Linked List – insert()

TASK: Code a function that inserts a node(5) **at a node position x** specified by a value(7).

- If the first node(or **head**) is the position, then just invoke **push_front()**.
- As observed below, we must to know **the pointer x** which is stored in the **previous node** of node x.



```
pNode insert(pNode list, int val, int x)
```

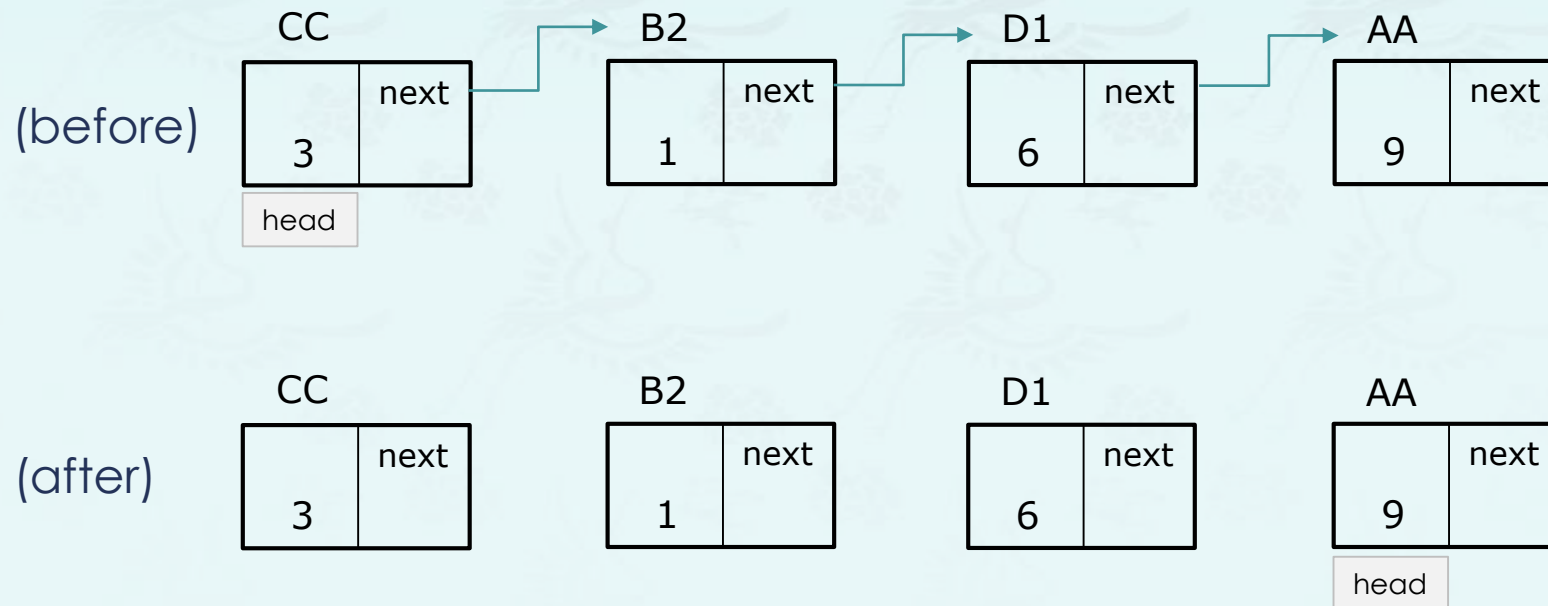
```
if (list->data == x)
```

```
pNode curr = list;  
pNode prev = nullptr;  
while (curr != nullptr) {  
    if (curr->data == x) {
```

```
        prev = curr;  
        curr = curr->next;  
    }  
    return list;
```

Linked List – reverse()

TASK: reverse a singly linked list in $O(n)$ which goes through the list once.



Linked List

❖ resizing array vs. linked list

Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation

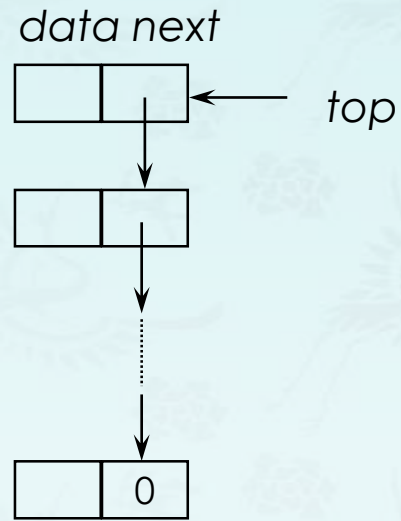
- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation

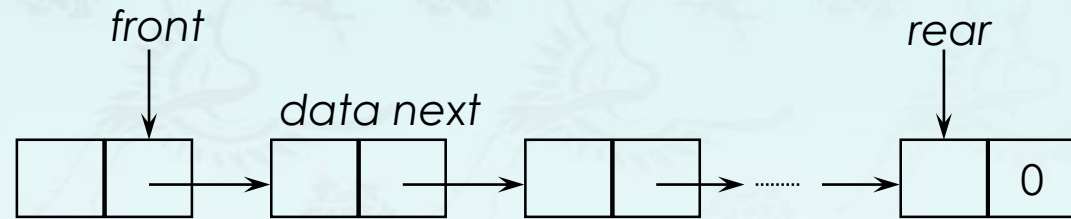
- Every operation takes constant amortized time.
- Less waste space

Linked List

Using linked lists, **stacks** and **queues** facilitate easy insertion and deletion of nodes.



(a) linked stack



(b) linked queue

Polynomials

❖ Polynomials representation

$$A(x) = a_{m-1}x^{e_{m-1}} + \dots + a_0x^{e_0}$$

a_i = nonzero coefficients

e_i = nonnegative integer exponents such as

$$e_{m-1} > e_{m-2} > \dots > a_0 \geq 0$$

❖ We may draw a **poly node** as

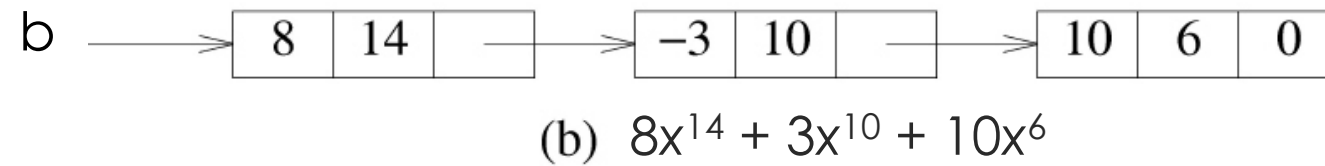
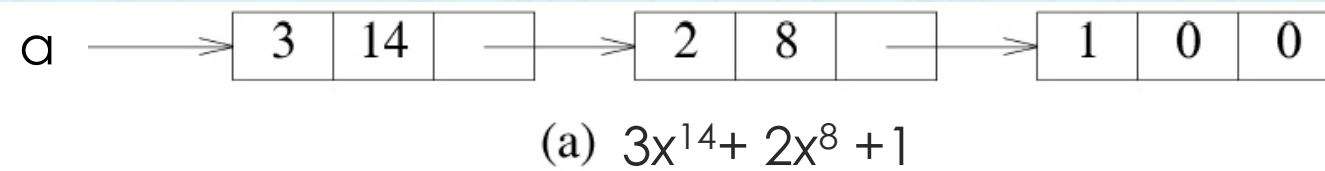
coef	expo	next
------	------	------

❖ Type definition

```
struct Poly {  
    double  coef;  
    double  expo;  
    Poly*   next;  
};  
using pPoly = Poly*;
```


Polynomials

❖ Example:



Q: How to add two polynomials? $c = a + b$

Polynomials

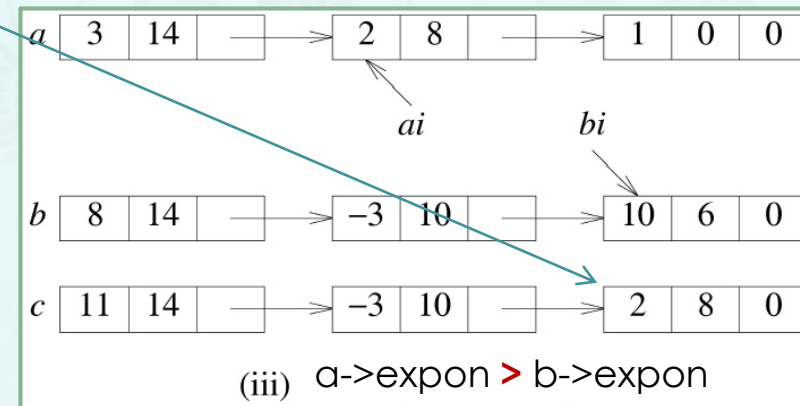
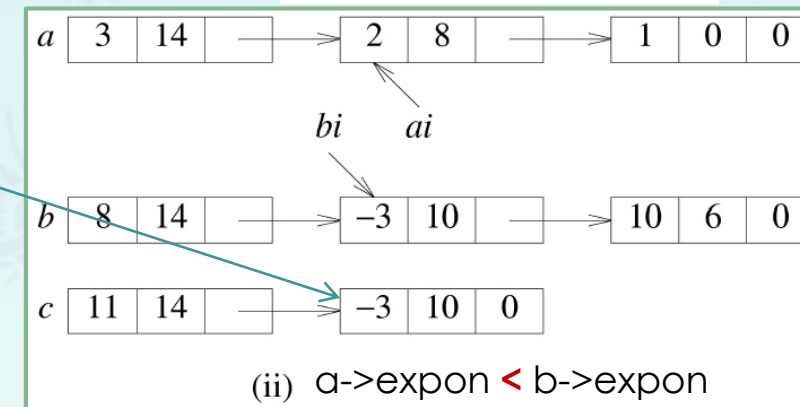
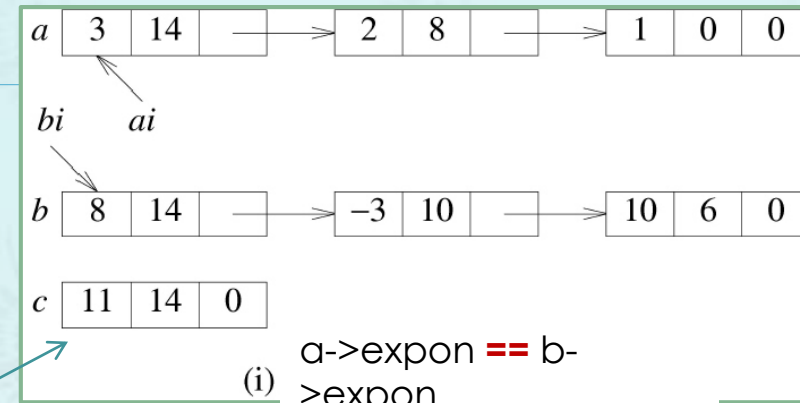
Q: How to add two polynomials?

$$a = 3x^{14} + 2x^8 + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^6$$

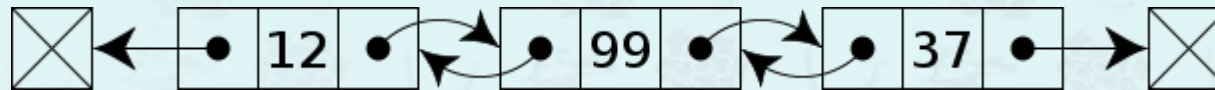
$$c = a + b$$

$$= 11x^{14} - 3x^{10} + 2x^8 + 10x^6 + 1$$



Doubly Linked lists

- ❖ **Doubly linked list:** each node contains, besides the **next**-node link, a second link field pointing to the **previous** node in the sequence. The two links may be called **forward** and **backward**, or **next** and **prev(ious)**.



- ❖ **Type definition**

```
struct Node {
    int    data;
    Node*  prev;
    Node*  next;
};
using pNode = Node*;
```

Q. Array vs. Singly linked list vs. Doubly linked list, Why?

Doubly Linked lists

Q. Array vs. Singly linked list vs. Doubly linked list, **Why?**

Advantages of linked list:

- Dynamic structure (Memory Allocated at run-time)
- Have more than one data type.
- Re-arrange of linked list is easy (Insertion-Deletion).
- **It doesn't waste memory.**

Disadvantages of linked list:

- In linked list, if we want to access any node it is difficult.
- **It uses more memory.**

Advantages of doubly linked list:

- A doubly linked list can be **traversed in both directions** (forward and backward). A singly linked list can only be traversed in one direction.
- Most operations are $O(1)$ instead of $O(n)$.

Pointer Linked – Lab

```
#include <iostream>
using namespace std;
class Node {
public:
    char ch;
    Node* next;
};

int main( ) {
    Node* p = nullptr, *q = nullptr;
    char ch;
    while (cin.get(ch) && ch != '\n') {
        p = new Node;
        p->ch = ch;
        p->next = q;
        q = p;
    }
    while (p != nullptr) {
        cout.put(p->ch);
        p = p->next;
    }
    cout << endl;
}
```

Pointer Linked – Lab

```
#include <iostream>
using namespace std;
class Node {
public:
    char ch;
    Node* next;
};

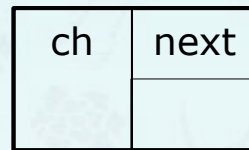
int main( ) {
    Node* p = nullptr, *q = nullptr;
    char ch;
    while (cin.get(ch) && ch != '\n') {
        p = new Node;
        p->ch = ch;
        p->next = q;
        q = p;
    }
    while (p != nullptr) {
        cout.put(p->ch);
        p = p->next;
    }
    cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

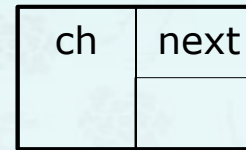
Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



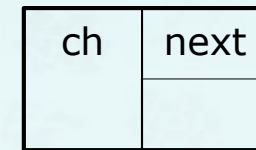
p
D3



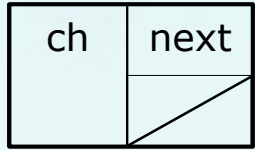
C1



B5



A2



q

What is missing in the figure?

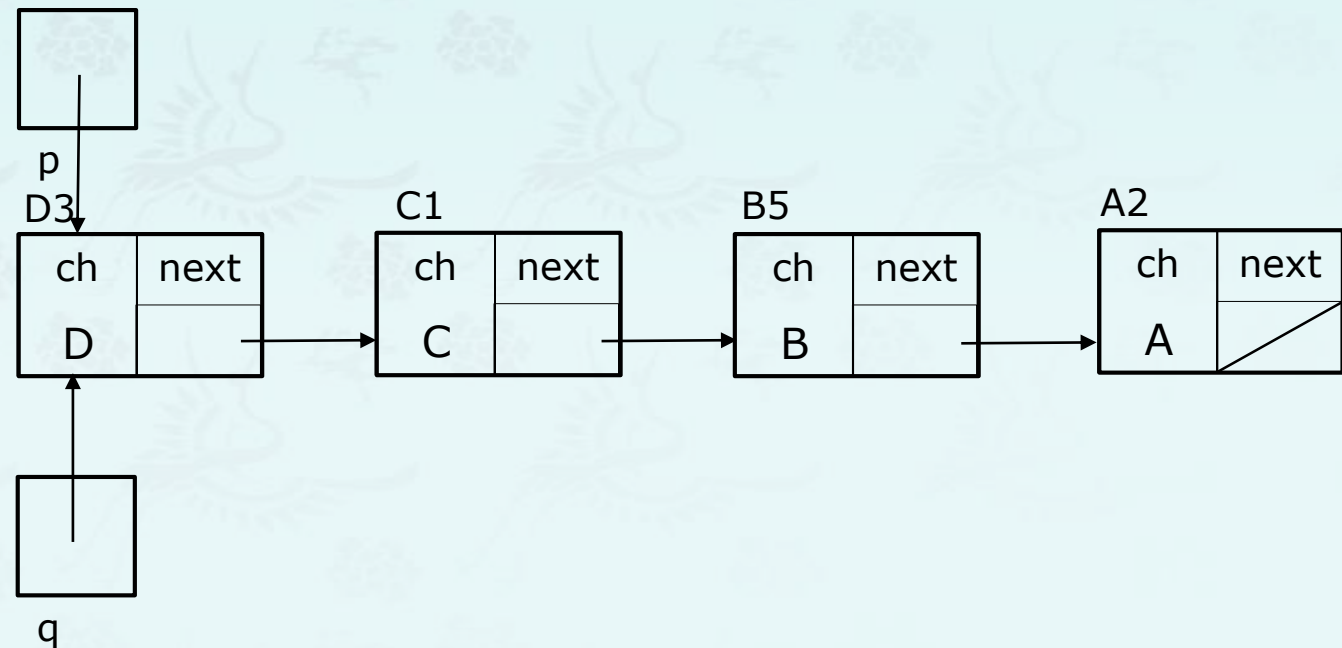
Pointer Linked

```
#include <iostream>
using namespace std;
class Node {
public:
    char ch;
    Node* next;
};

int main( ) {
    Node* p = nullptr, *q = nullptr;
    char ch;
    while (cin.get(ch) && ch != '\n') {
        p = new Node;
        p->ch = ch;
        p->next = q;
        q = p;
    }
    while (p != nullptr) {
        cout.put(p->ch);
        p = p->next;
    }
    cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



What is missing in the figure?

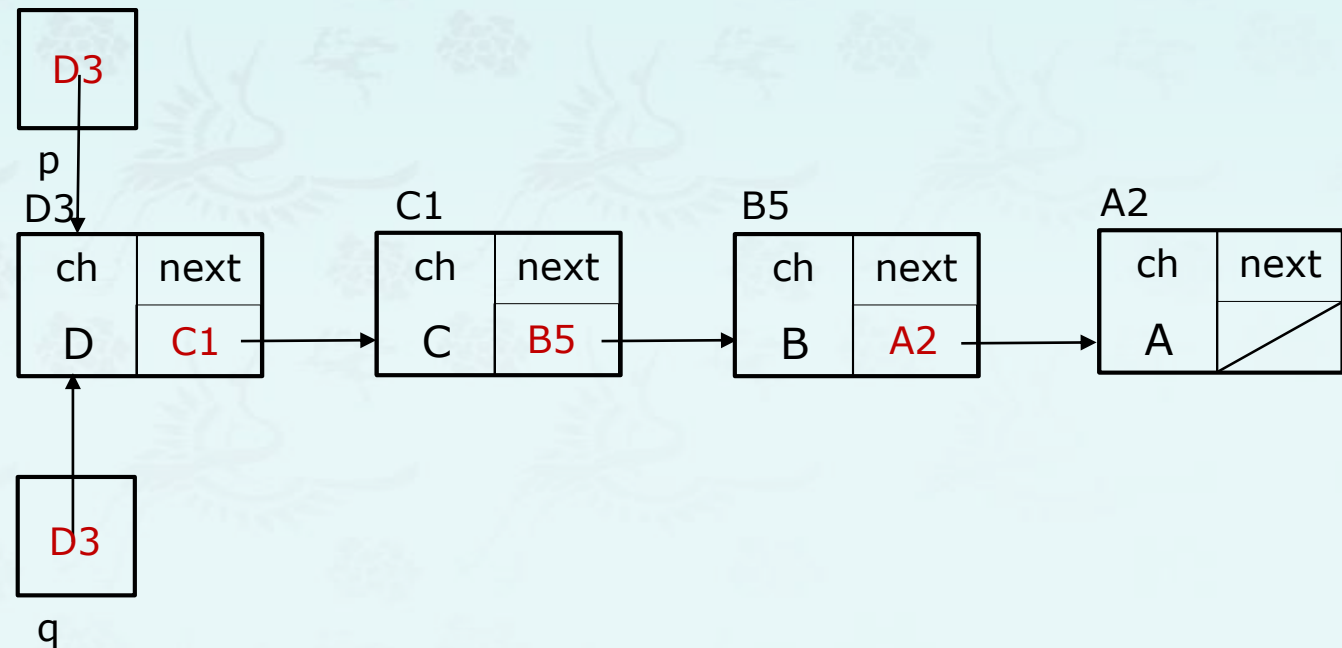
Pointer Linked

```
#include <iostream>
using namespace std;
class Node {
public:
    char ch;
    Node* next;
};

int main( ) {
    Node* p = nullptr, *q = nullptr;
    char ch;
    while (cin.get(ch) && ch != '\n') {
        p = new Node;
        p->ch = ch;
        p->next = q;
        q = p;
    }
    while (p != nullptr) {
        cout.put(p->ch);
        p = p->next;
    }
    cout << endl;
}
```

Assuming the input A, B, C, D to this program, what would be the data structure after the input?

Draw a figure to represent the data structure in memory. Use a mnemonic memory address to represent each node such as A2, B5, C1, ..., etc.



Pointer Linked

```
#include <iostream>
using namespace std;
class Node {
public:
    char ch;
    Node* next;
};

int main( ) {
    Node* p = nullptr, *q = nullptr;
    char ch;
    while (cin.get(ch) && ch != '\n') {
        p = new Node;
        p->ch = ch;
        p->next = q;
        q = p;
    }
    while (p != nullptr) {
        cout.put(p->ch);
        p = p->next;
    }
    cout << endl;
}
```

After executing the while loop,
What is the output?
What is the values of p and q?

