Data Structures
Chapter 4

1. Singly Linked List
   - Pointer & Linking
   - Singly Linked List (SLL)
   - **SLL Basic Operations**
   - SLL Advanced Operations
2. Doubly Linked List

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

내 아들들을 먼 곳에서 이끌며 내 딸들을 땅 끝에서 오게 하며 내 이름으로 불려지는 모든 자 곧 내가 내 영광을 위하여 창조한 자를 오게 하라 그를 내가 지었고 그를 내가 만들었노라 (사43:6-7)

그런즉 너희가 먹든지 마시든지 무엇을 하든지 다 하나님의 영광을 위하여 하라 (고전10:31)

Self-Referenced Data Structures

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

constructor, destructor

Self-Referenced Data Structures

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

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

  Node(int i=0, Node* n=nullptr){
    data = i, next = n;
  }
  ~Node() {};
};

int main( ) {
  Node* head, *x, *y;
  Node* p = new Node;
  ...

}
```

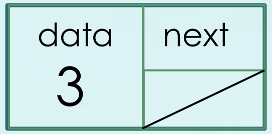
```
struct Node {
  int   data;
  Node* next;
};
using pNode Node*;

int main() {
  pNode head, x, y;
  pNode p = new Node;
  …
}
```

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

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

a new node instantiation

A0



(1)

```
pNode n = new Node;
```

```
Node* n = new Node;
```

← no values set

(2)

```
Node* n = new Node();
```

← set to 0 or nullptr

(3)

```
Node* n = new Node{};
```

← set to 0 or nullptr

(4)

```
Node* n = new Node(4);
```

← Compiler error

(5)

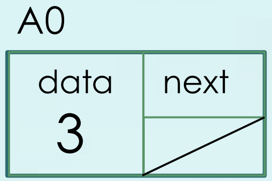
```
Node* n = new Node{5};
```

← set to 5 or nullptr

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

struct List {
  Node*   head;
  Node*   tail;
  int     size; //optional
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

a new node instantiation



(1)

```
pNode n = new Node(3);
```

```
Node* n = new Node(3);
```

(2)

```
Node* n = new Node{3};
```

(3)

```
Node* n = new Node{3, nullptr};
```

(4)

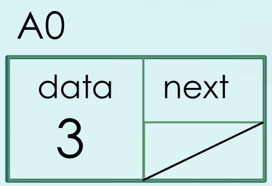
```
Node* n = new Node{3, nullptr, nullptr};
```

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

struct List {
  Node*   head;
  Node*   tail;
  int     size; //optional
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

Any invalid initialization code?

a new node instantiation




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

struct List {
  Node*   head;
  Node*   tail;
  int     size; //optional
};
using pNode = Node*;
using pList = List*;
```

unused in singly linked

```
pNode n = new Node{3};
```

```
Node* n = new Node{3};
```

```
pNode n = new Node{3, nullptr, nullptr};
```

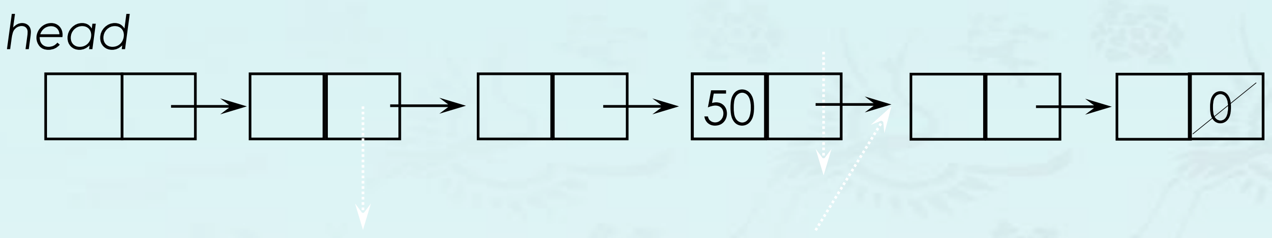
```
Node* n = new Node{3, nullptr, nullptr};
```

```
struct Node{
  int data;
  Node* prev;
  Node* next;
  // constructor
  Node(int d=0, Node* p=nullptr, Node* x=nullptr) {
    data = d;     prev = p; next = x;
  }
  // destructor
  ~Node() {}
};
```

can be omitted

Linked List – find()

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

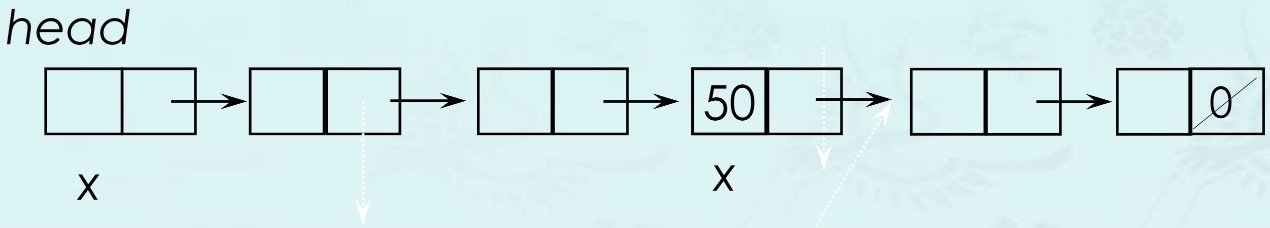



```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;
```

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

Linked List – find()

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



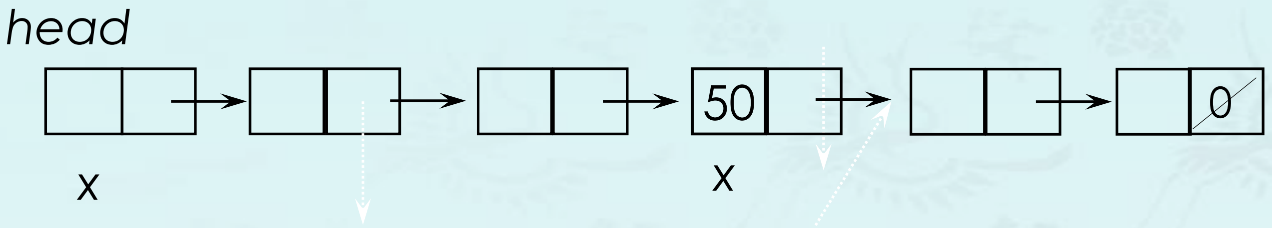

```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;

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

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

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

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

Linked List – find()

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




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

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

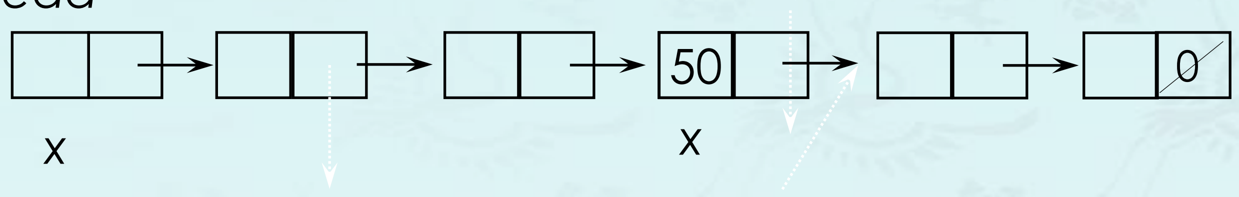
```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;

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

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

Linked List – find()

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

head




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

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

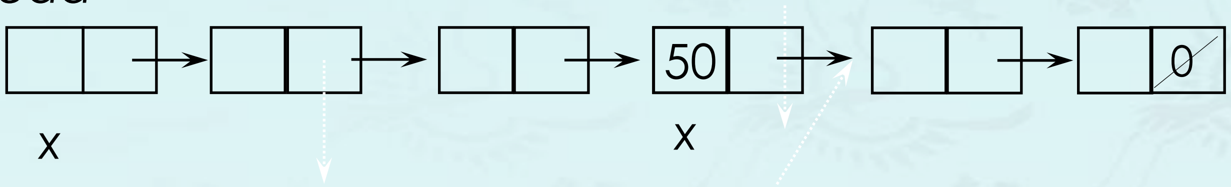
```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;


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

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

Linked List – find()

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

head



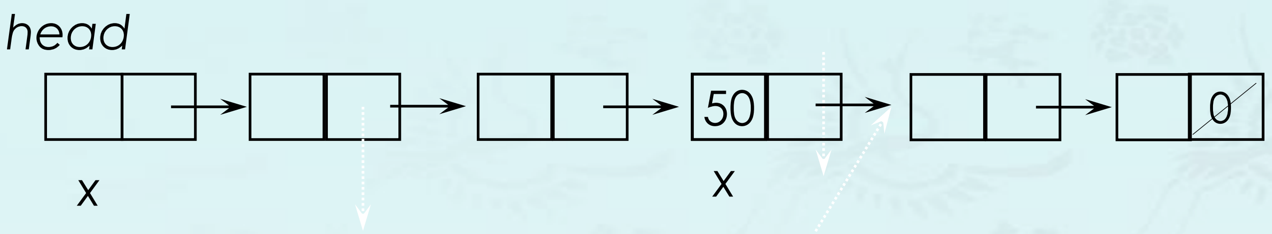

```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;

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

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


for (pNode x=head; x!=nullptr; x=x->next;){
  if (x->data == val) return x;

}
return x;
```

What is wrong?

Linked List – find()

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

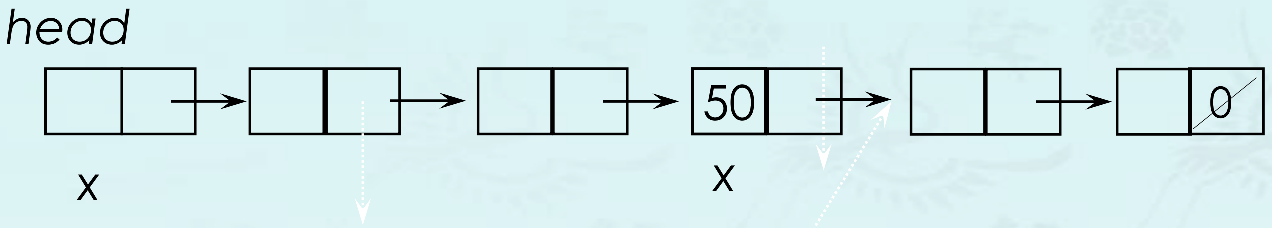



```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;

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

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

pNode x = head;
for ( ; x != nullptr; )
  if (x->data == val) return x;
  x = x->next;
}
return x;
```

Linked List – find()

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

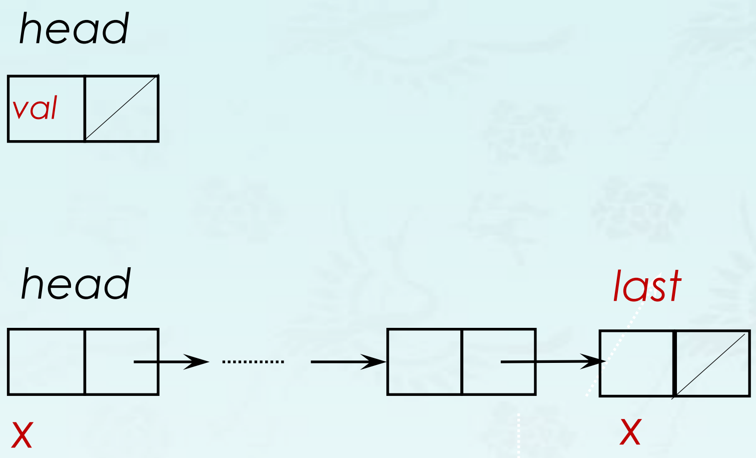



```
pNode find(pNode head, int val)
if (empty(head)) return nullptr;

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

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

pNode x = head;
for ( ; x != nullptr; x = x->next;){
  if (x->data == val) return x;

}
return x;
```

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.




```
pNode push_back(pNode head, int val)
```
```
if (empty(head))
  return new Node{val, nullptr};
```

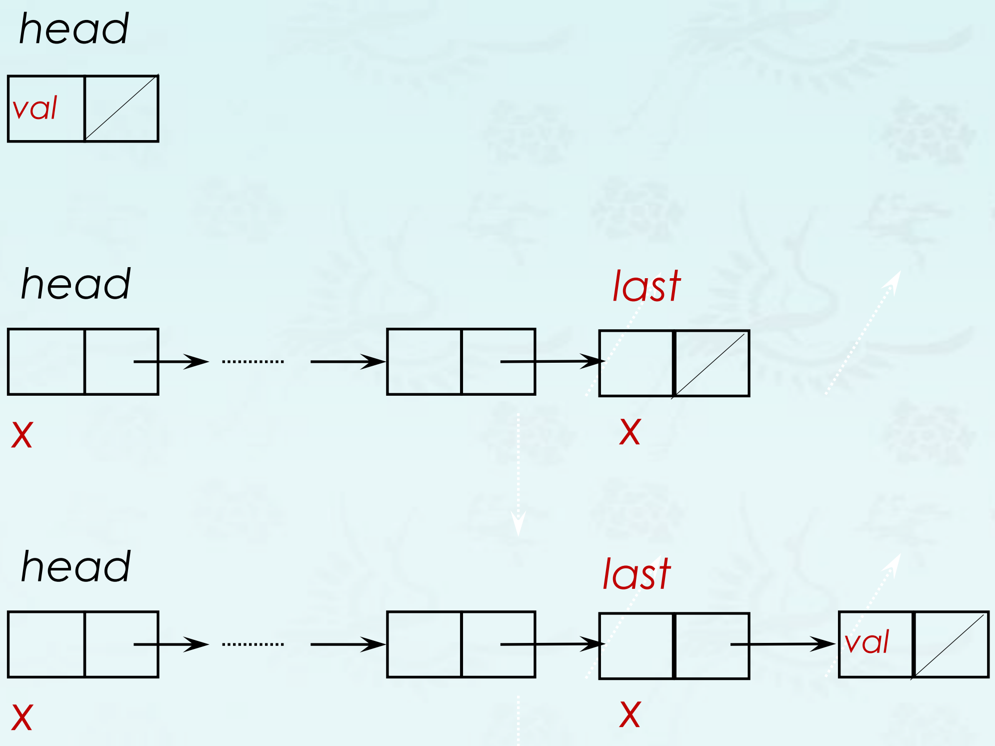
```
pNode last(pNode head)
```
```
  pNode x = head;
  while (x != nullptr)
    x = x->next;
  return x
```

```
pNode last(pNode head)
```
```
  pNode x = head;
  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.

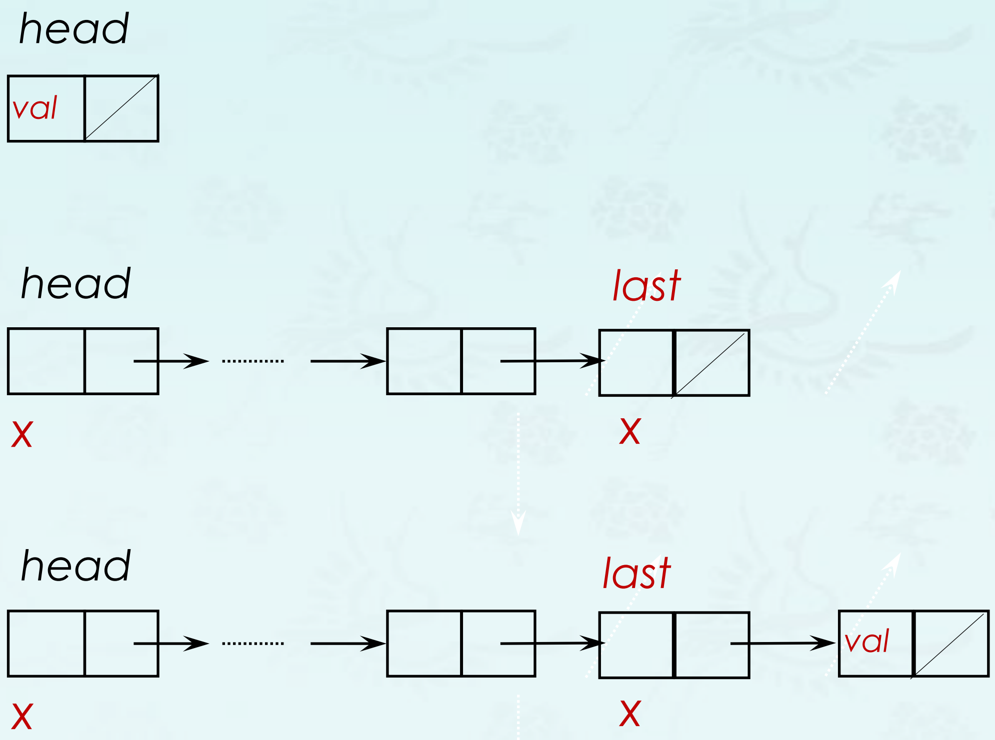



```
pNode push_back(pNode head, int val)
if (empty(head))
  return new Node{val, nullptr};
```

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

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.




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

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

pNode x = last(head);
x->next = new Node{val, nullptr};
return head;
```

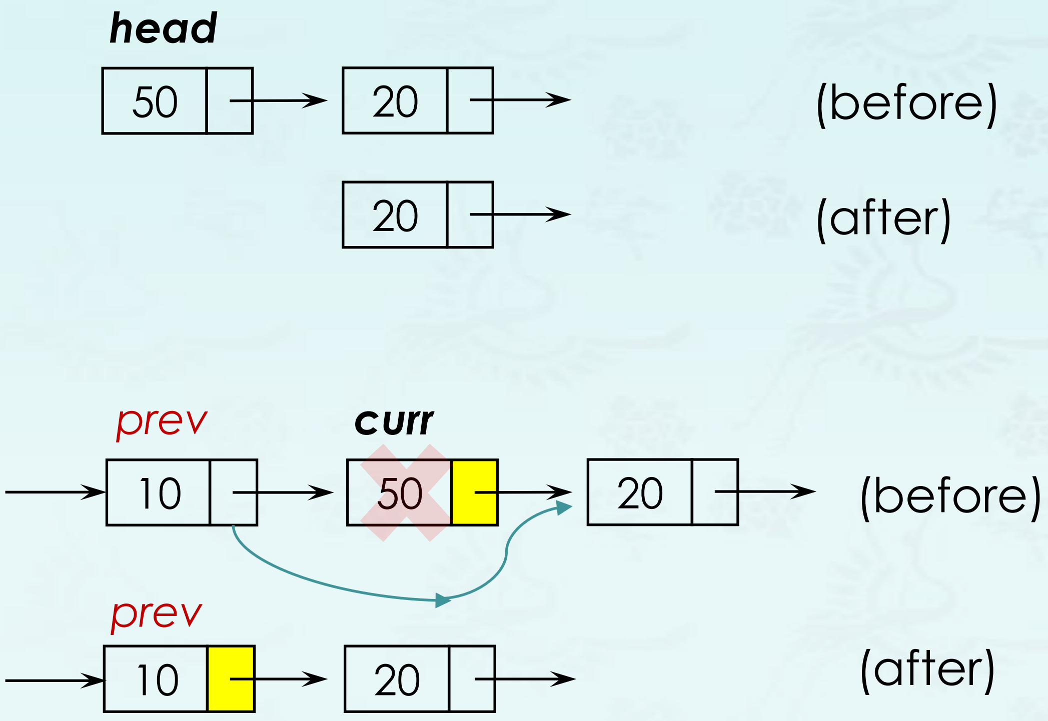
```
pNode last(pNode head)
```

```
  pNode x = head;
  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 know **the pointer x** which is stored in the **previous node** of node x.



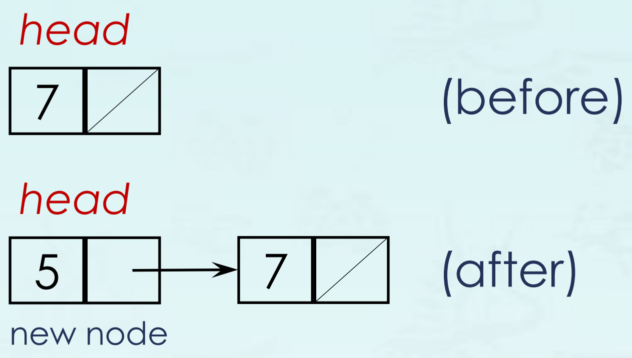

```
pNode pop(pNode head, int val)

if (head->data == val)
  return pop_front(head);

pNode curr = head;
pNode prev = nullptr;
while (curr != nullptr) {
  if (curr->data == val) {
    prev->next = curr->next;
    delete curr;
    return head;
  }
  prev = curr;
  curr = curr->next;
}
return head;
```

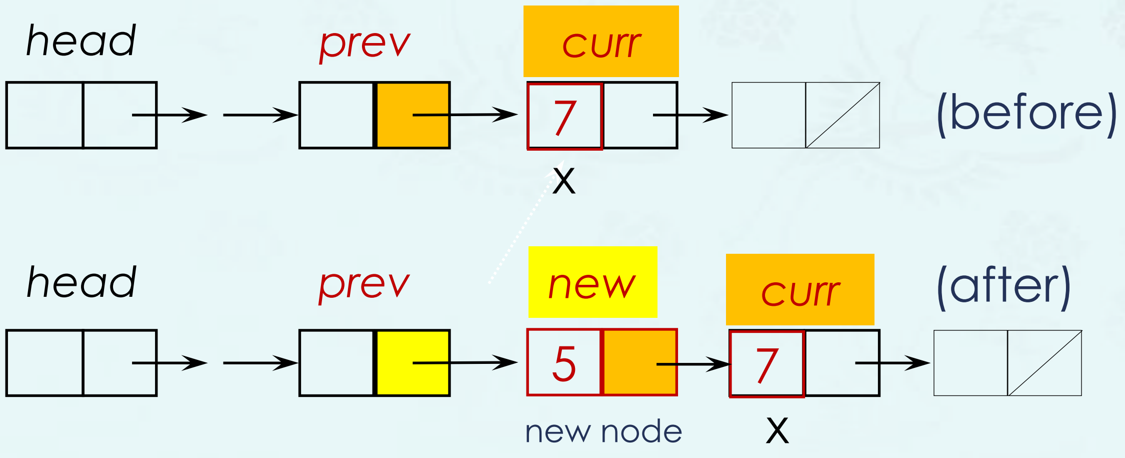
Simplifying this while() loop is left as a part of Problem Set.

Linked List – insert() or push()

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 know **the pointer x** which is stored in the **previous node** of node x.





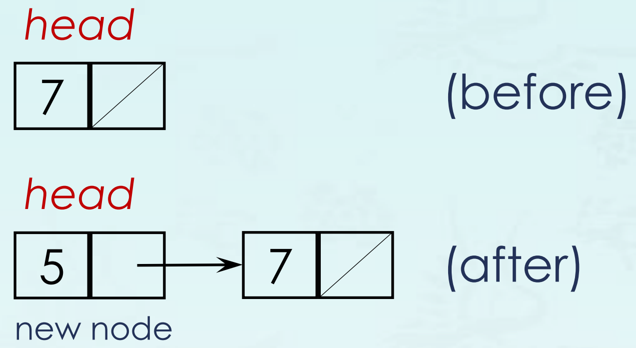

```
pNode insert(pNode head, int val, int x)
```

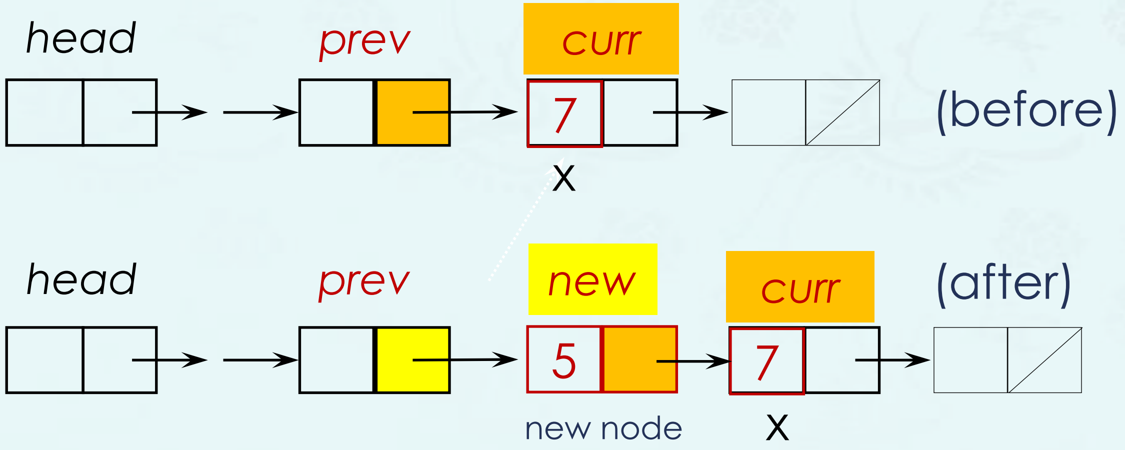
```
if (head->data == x)
  return push_front(val, head);

pNode curr = head;
pNode prev = nullptr;
while (curr != nullptr) {
  if (curr->data == x) {
                = new Node{                };
    return head;
  }
  prev = curr;
  curr = curr->next;
}
return head;
```

Linked List – insert() or push()

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 know **the pointer x** which is stored in the **previous node** of node x.



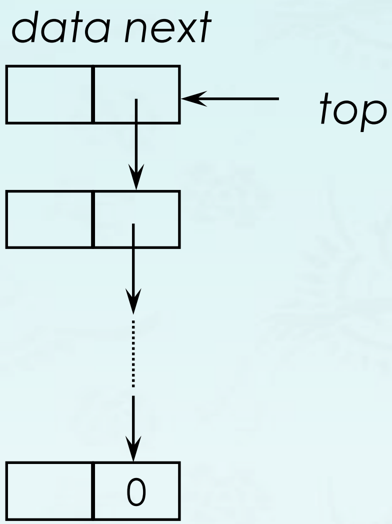



```
pNode insert(pNode head, int val, int x)

if (head->data == x)
  return push_front(val, head);

pNode curr = head;
pNode prev = nullptr;
while (curr != nullptr) {
  if (curr->data == x) {
    prev->next = new Node{val, prev->next};
    return head;
  }
  prev = curr;
  curr = curr->next;
}
return head;
```

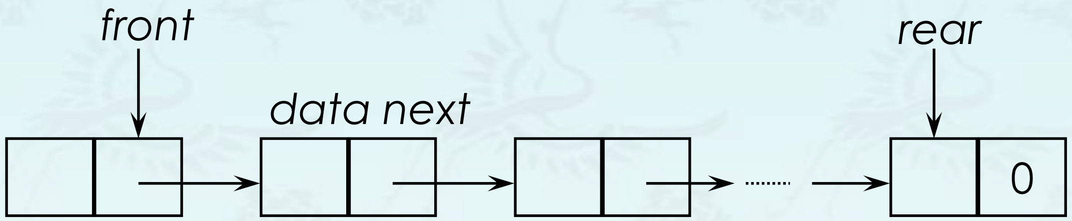
Simplifying this while() loop is left as a part of Problem Set.

Linked List

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



(a) linked stack



(b) linked queue

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

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).

Data Structures
Chapter 4

1. Singly Linked List
    - Pointer & Linking
    - Singly Linked List (SLL)
    - **SLL Basic Operations**
    - SLL advanced Operations
2. Doubly Linked List

Summary &
quaestio quaestio qo ? ? ? ?