Data Structures Chapter 4

- 1. Singly Linked List
- 2. Doubly Linked List
 - Revisit Singly Linked List
 - Sentinel Nodes & Basic Operations
 - Two Key Operations: erase, insert
 - Advanced Operations



우리가 알거니와 하나님을 사랑하는 자 곧 그의 뜻대로 부르심을 입은 자들에겐 모든 것이 합력하여 선을 이루느니라 (롬8:28)

And we know that in all things God works for the good of those who love him, who have been called according to his purpose. (Rom8:28)

하나님이 우리를 구원하사 거룩하신 소명으로 부르심은 우리의 행위대로 하심이 아니요 오직자기의 뜻과 영원 전부터 그리스도 예수 안에서 우리에게 주신 은혜대로 하심이라 (딤후1:9)

With an error Linked List – reverse using stack Algorithm: next next next next Step 1. Push all nodes onto the stack. 6 Step 2. Top/pop all nodes and relink. head next next next 6 next next next next next next next stack stack stack stack

Corrected: Linked List – reverse using stack Algorithm: next next next next Step 1. Push all nodes onto the stack. 6 Step 2. Top/pop all nodes and relink. head (3) this operation is done after while() loop. next next next 6 next next next next next next next

stack

stack

stack

stack

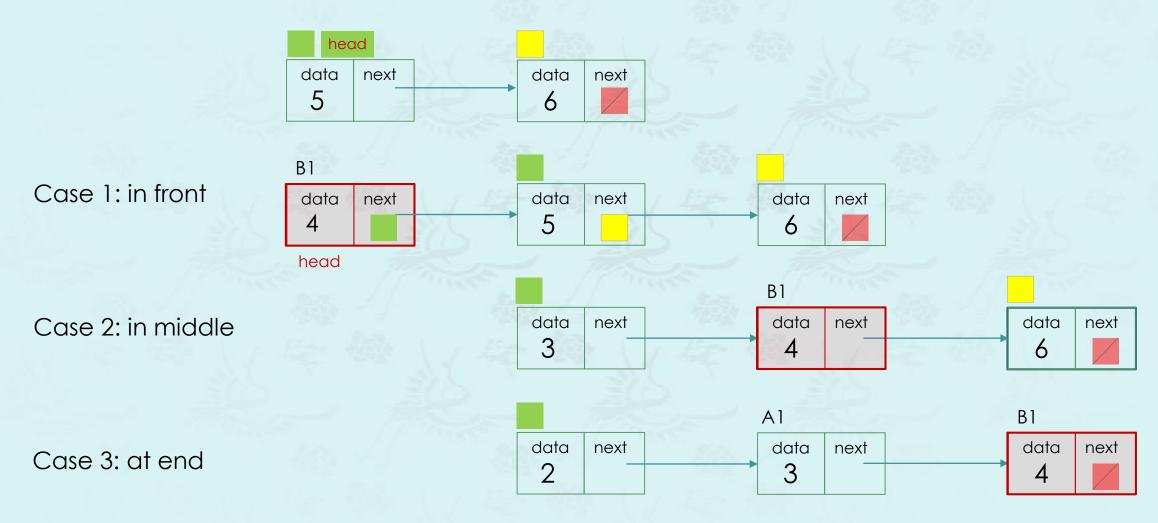
A Node and List Data Structure with Constructor and Destructor

```
struct Node {
  int
         data;
                           unused in
                           singly linked
 Node*
         prev;
 Node* next;
};
struct List {
  Node* head;
 Node* tail;
  int
         size; //optional
using pNode = Node*;
using pList = List*;
```

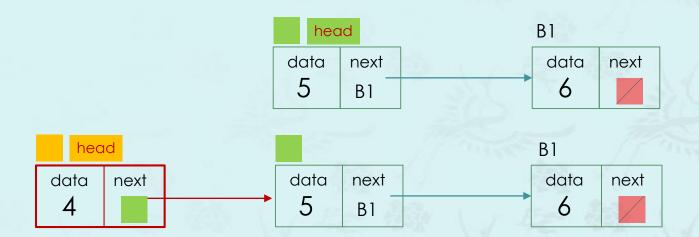
```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
  Node(int d = 0, Node* p = nullptr,
                  Node* x = nullptr)
      { data = d; prev = p; next = x; }
  ~Node() {}
struct List {
  Node* head;
  Node* tail:
 int size; // optional
  List() { head = new Node{}; tail = new Node{};
           head->next = tail; tail->prev = head;
           size = 0;
  ~List() {}
};
using pNode = Node*;
using pList = List*;
```

push a node - three different cases

Given: an data(4) to insert as sorted – What was the most difficult part of this coding?



push a node - Case 1: insert in front, head given

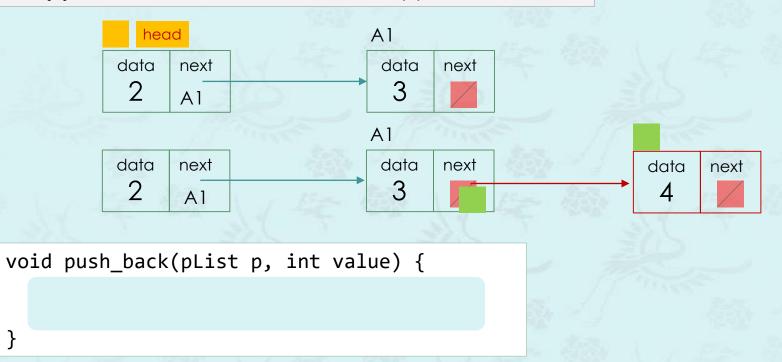


```
void push_front(pList p, int value) {
  p->head = new Node{value, p->head};
}
```

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

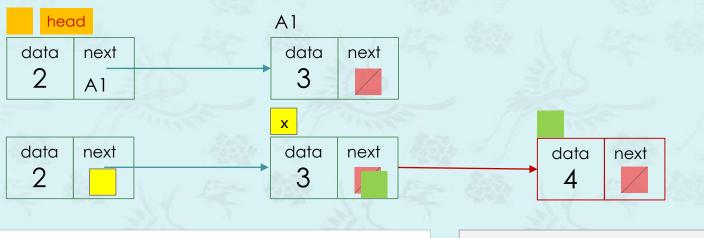
push a node - Case 3; insert at end, head given

Append a new Node{4}; then find the last node first to append.



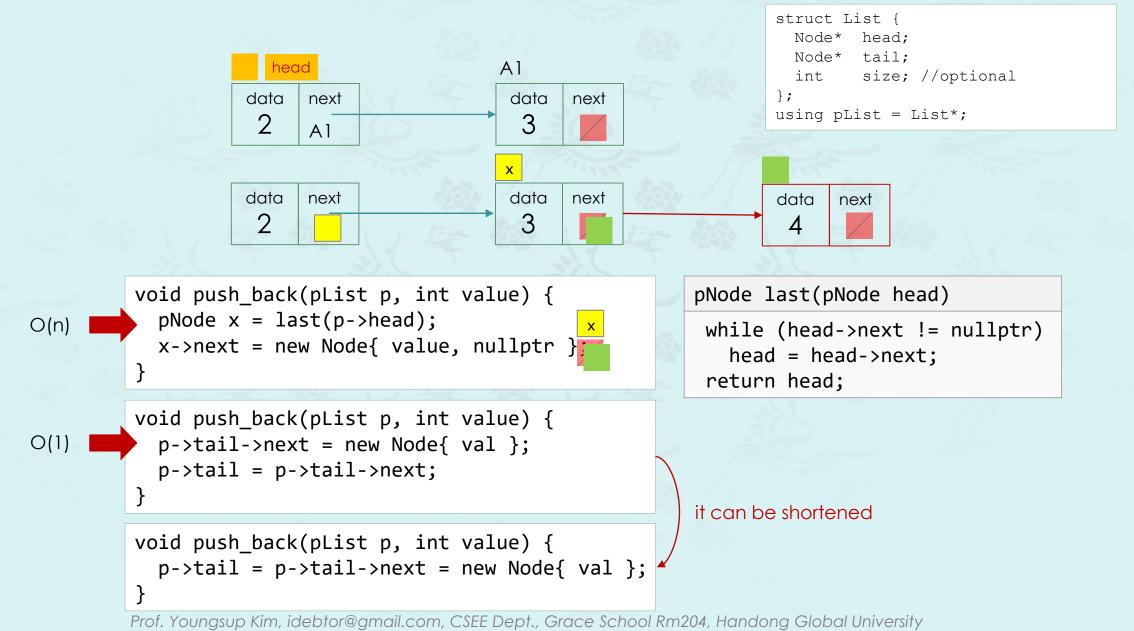
push a node - Case 3; insert at end, head given

Append a new Node{4}; then find the last node first to append.

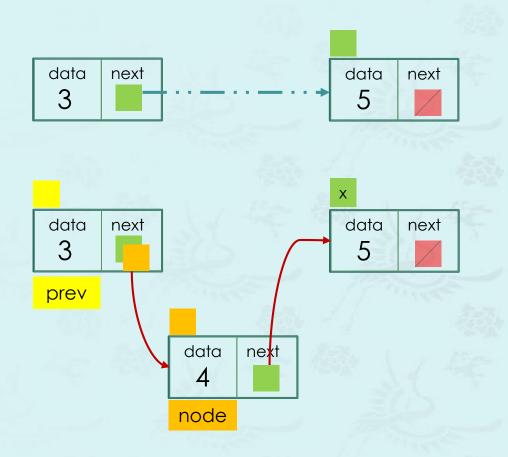


```
void push_back(pList p, int value) {
    pNode x = last(p->head);
    x->next = new Node{ value, nullptr };
}
while (head->next != nullptr)
head = head->next;
return head;
```

push a node - Case 3; insert at end, head given

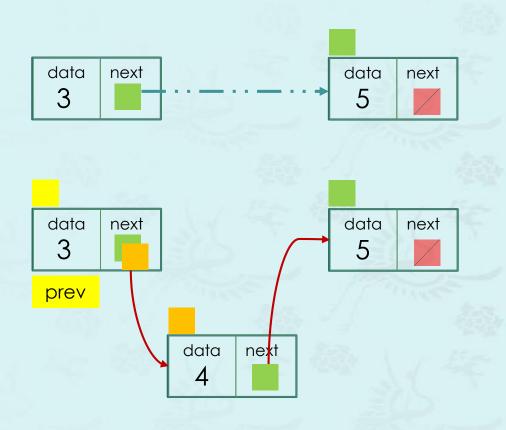


push a node - Case 2; insert in middle, head given



```
// inserts a node val at node z
void push_at(pList p, int value, int z) {
  if (empty(p) | | (p->head->data == z) }
    return push front(p, value);
  pNode x = p->head;
  pNode prev = nullptr;
  while (x != nullptr) {
    prev = x;
    x = x-\text{next};
```

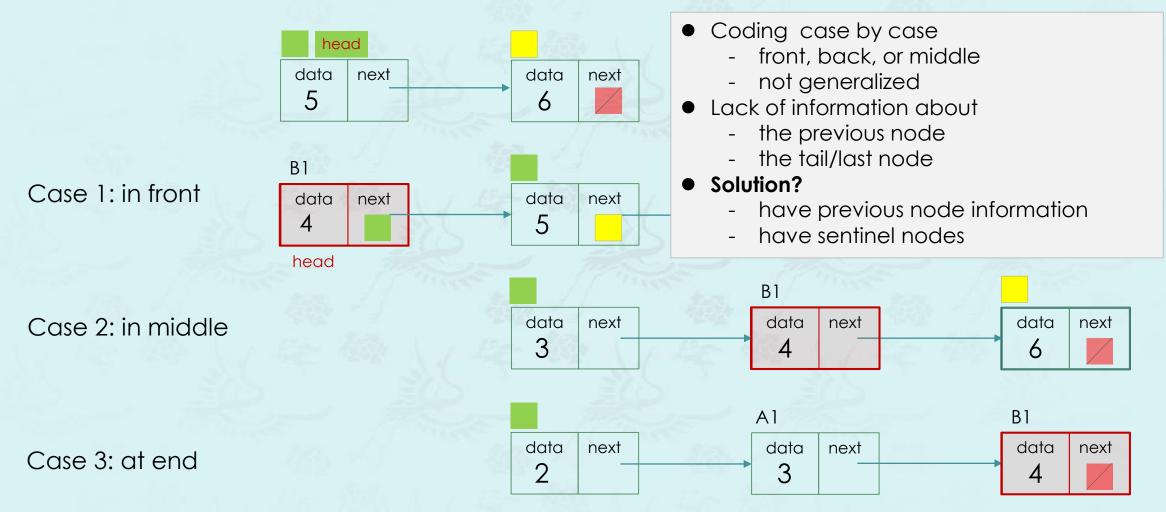
push a node - Case 2; insert in middle, head given



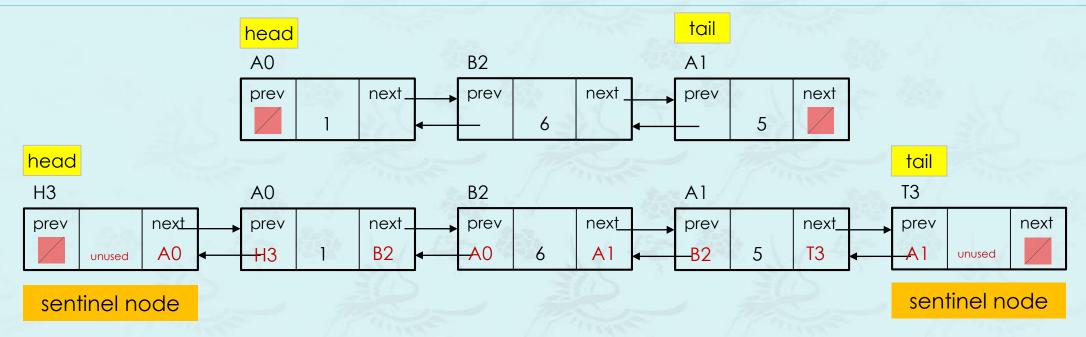
```
// inserts a node val at node z
void push_at(pList p, int value, int z) {
  if (empty(p) || (p->head->data == z) }
    return push front(p, value);
  pNode x = p->head;
  pNode prev = nullptr;
  while (x != nullptr) {
    if (x->data == z) {
      prev->next = new Node{value, prev->next};
      return;
    prev = x;
    x = x \rightarrow next;
```

push a node – Three different cases

Given: an data(4) to insert – What was the most difficult part of this coding?



doubly linked list with sentinel nodes



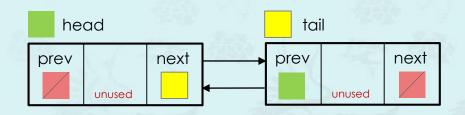
- Solution
 - doubly linked list with two sentinel nodes
 - Each node carries the pointer to the previous node.
 - There is only one case (middle) with two sentinel nodes.

doubly linked list with sentinel nodes



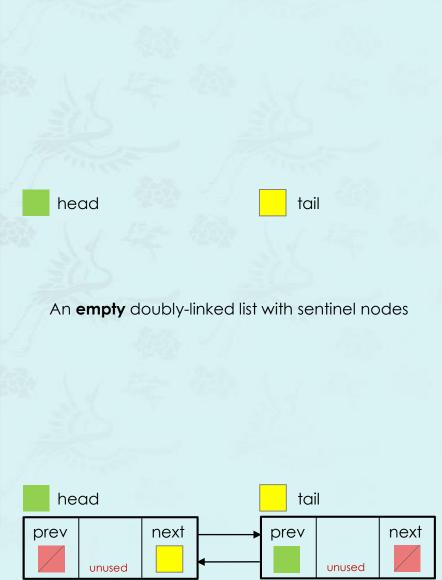
- These extra nodes are known as sentinel nodes. The node at the front is known as head node, and the node at the end as a tail node. The head and tail nodes are created when the doubly linked list is initialized. The purpose of these nodes is to simply the insert, push/pop front and back, remove methods by eliminating all need for special-case code when the list empty, or when we insert at the head or tail of the list. This would greatly simplify the coding unbelievably.
- For instance, if we do not use a head node, then removing the first node becomes a special case, because we must reset the list's link to the first node during the remove and because the remove algorithm in general needs to access the node prior to the node being removed (and without a head node, the first node does not have a node prior to it).

doubly linked list with sentinel nodes

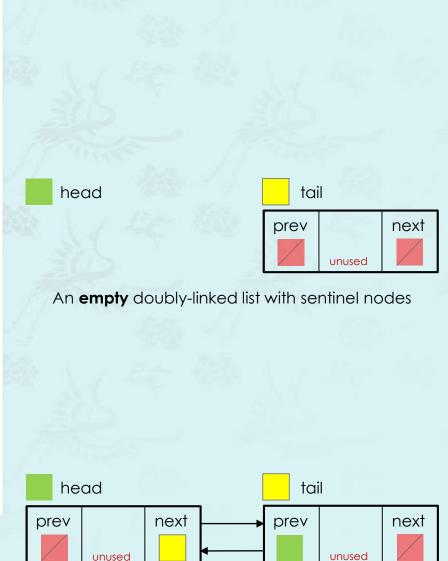


An **empty** doubly linked list with sentinel nodes

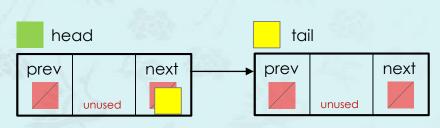
```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
};
struct List {
 Node*
                //sentinel
         head;
         tail; //sentinel
 Node*
          size; //size of list, optional
  int
 List() {
  ~List() {}
using pNode = Node*;
using pList = List*;
```



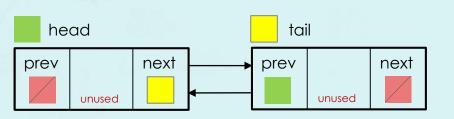
```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
};
struct List {
 Node*
          head;
                //sentinel
         tail; //sentinel
 Node*
          size; //size of list, optional
  int
  List() {
       tail = new Node{};
  ~List() {}
using pNode = Node*;
using pList = List*;
```



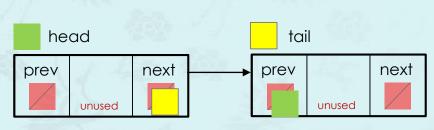
```
struct Node {
  int
         data;
 Node*
         prev;
 Node*
         next;
};
struct List {
 Node*
         head;
               //sentinel
 Node*
        tail; //sentinel
 int
         size; //size of list, optional
 List() {
      tail = new Node{};
      head = new Node{0, nullptr, tail};
 ~List() {}
using pNode = Node*;
using pList = List*;
```



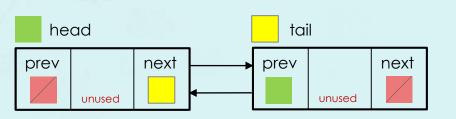
An **empty** doubly-linked list with sentinel nodes



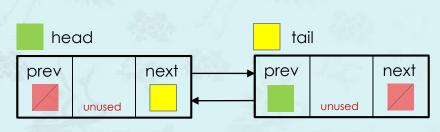
```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
};
struct List {
 Node*
          head;
                //sentinel
 Node*
         tail; //sentinel
  int
         size; //size of list, optional
  List() {
       tail = new Node{};
       head = new Node{0, nullptr, tail};
  ~List() {}
using pNode = Node*;
using pList = List*;
```



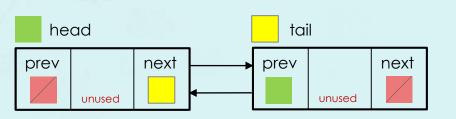
An **empty** doubly-linked list with sentinel nodes



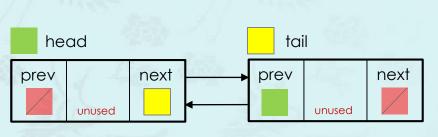
```
struct Node {
  int
         data;
 Node*
         prev;
 Node*
         next;
};
struct List {
 Node*
         head; //sentinel
 Node*
        tail; //sentinel
 int size; //size of list, optional
 List() {
      tail = new Node{};
      head = new Node{0, nullptr, tail};
      tail->prev = head;
 ~List() {}
using pNode = Node*;
using pList = List*;
```



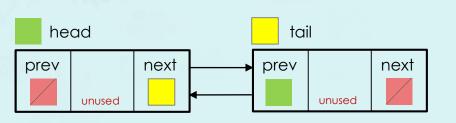
An **empty** doubly-linked list with sentinel nodes



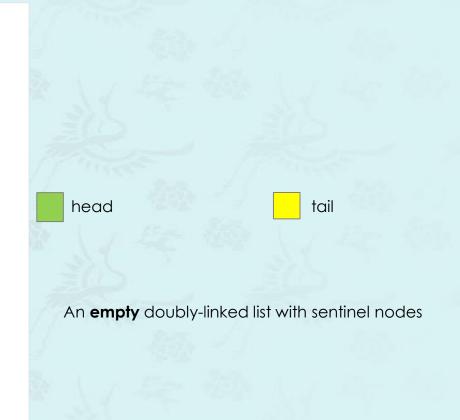
```
struct Node {
         data;
  int
 Node*
         prev;
 Node*
         next;
};
struct List {
 Node*
         head; //sentinel
 Node*
        tail; //sentinel
 int size; //size of list, optional
 List() {
      tail = new Node{};
      head = new Node{0, nullptr, tail};
      tail->prev = head;
      size = 0;
 ~List() {}
using pNode = Node*;
using pList = List*;
```



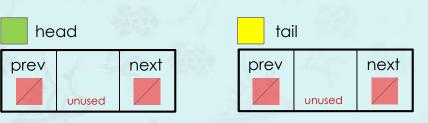
An **empty** doubly-linked list with sentinel nodes



```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
    data = d; prev = p; next = x;
  ~Node() {}
struct List {
                       another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int size; //size of list, optional
 List() { /
  ~List() {}
};
using pNode = Node*;
using pList = List*;
```

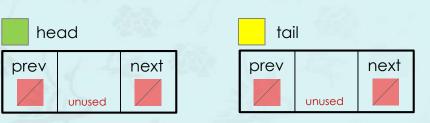


```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
    data = d; prev = p; next = x;
  ~Node() {}
struct List {
                       another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int size; //size of list, optional
 List() {
  ~List() {}
};
using pNode = Node*;
using pList = List*;
```



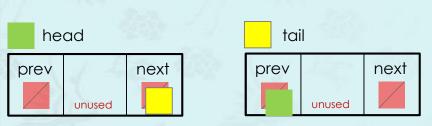
An **empty** doubly-linked list with sentinel nodes

```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
   data = d; prev = p; next = x;
  ~Node() {}
struct List {
                      another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int size; //size of list, optional
 List() { head = new Node{}; tail = new Node{};
 ~List() {}
};
using pNode = Node*;
using pList = List*;
```



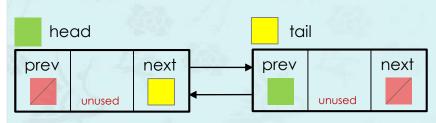
An **empty** doubly-linked list with sentinel nodes

```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
          next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
    data = d; prev = p; next = x;
  ~Node() {}
struct List {
                      another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int
      size; //size of list, optional
 List() { head = new Node{}; tail = new Node{};
 ~List() {}
};
using pNode = Node*;
using pList = List*;
```



An **empty** doubly-linked list with sentinel nodes

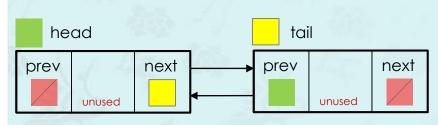
```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
         next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
    data = d; prev = p; next = x;
  ~Node() {}
struct List {
                      another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int
      size; //size of list, optional
  List() { head = new Node{}; tail = new Node{};
           head->next = tail; tail->prev = head;
 ~List() {}
};
using pNode = Node*;
using pList = List*;
```



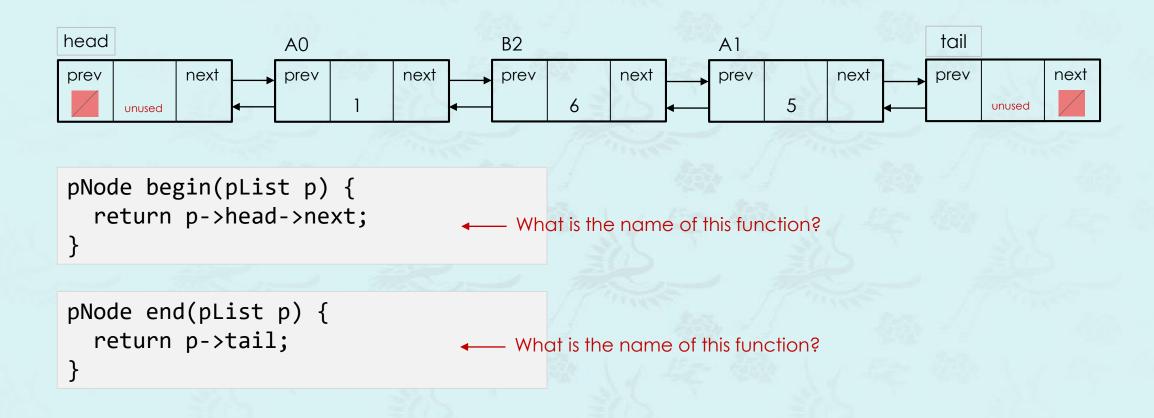
An **empty** doubly-linked list with sentinel nodes

27

```
struct Node {
  int
          data;
 Node*
          prev;
 Node*
         next;
 Node(const int d = 0, Node* p = nullptr, Node* x = nullptr) {
    data = d; prev = p; next = x;
  ~Node() {}
struct List {
                      another way of doubly linked list initialization
 Node* head;
 Node* tail;
  int
      size; //size of list, optional
  List() { head = new Node{}; tail = new Node{};
           head->next = tail; tail->prev = head;
           size = 0;
  ~List() {}
};
using pNode = Node*;
using pList = List*;
```

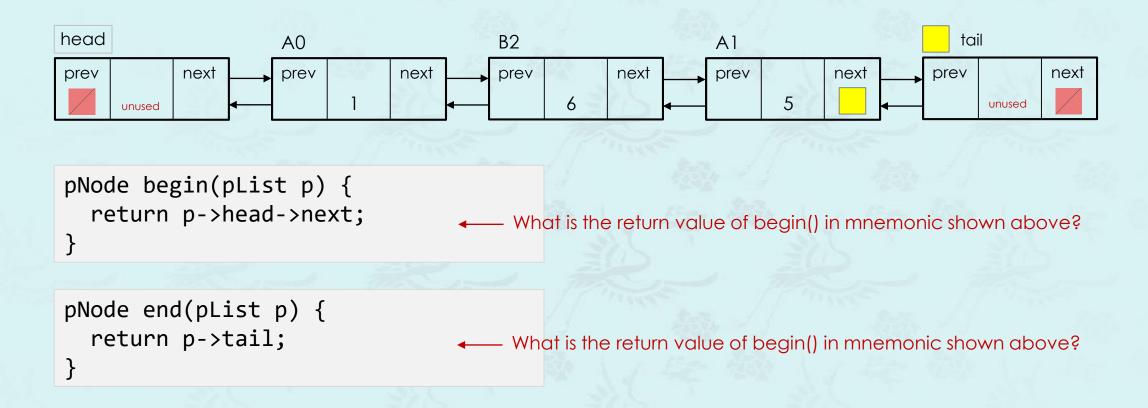


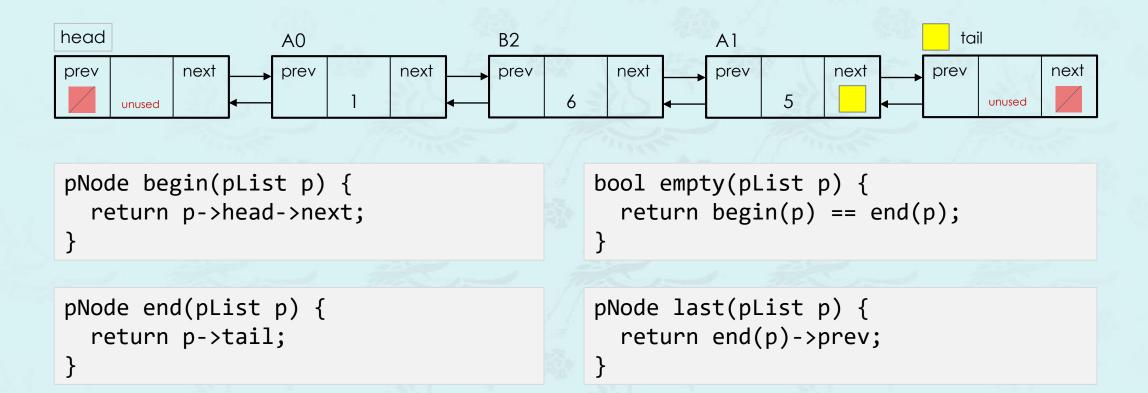
An **empty** doubly-linked list with sentinel nodes

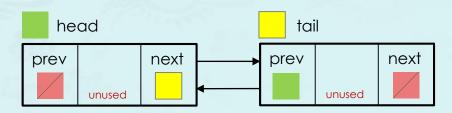


Basic Operations: begin() and end()

```
// returns the first node which list::head points to in the container.
pNode begin(pList p) {
  return p->head->next;
// returns the tail node referring to the past -the last- node in the list.
// The past -the last- node is the sentinel node which is used only as a sentinel that would
// follow the last node. It does not point to any node next, and thus shall not be dereferenced.
  Because the way we are going use during the iteration, we don't want to include the node
  pointed by this. This function is often used in combination with List::begin to specify a
  range including all the nodes in the list. This is a kind of simulated used in STL. If the
// container is empty, this function returns the same as List::begin.
pNode end(pList p) {
                         With the container given below,
  return p->tail;
                         what does begin(p) and end(p) return
                         in mnemonic code, respectively?
      head
                                                                                   tail
                         A0
                                                               A1
                next_
                         prev
                                                                          next
                                                                                  prev
                                                                                             next
      prev
                                                                     5
                                                                          T3
                 Α0
```







An **empty** doubly-linked list with sentinel nodes



```
int size(pList p) {
  int count = 0;

return count;
}
```

```
int size(pList p) {
  int count = 0;
  pNode x = begin(p);
  while(x != end(p)) {
    count++;
    x = x->next;
  }
  return count;
}
```



```
int size(pList p) {
  int count = 0;
  for (pNode x = begin(p); x != end(p); x = x->next)
     count++;
  return count;
}
```

```
int size(pList p) {
  int count = 0;
  pNode x = begin(p);
  while(x != end(p)) {
    count++;
    x = x->next;
  }
  return count;
}
```



```
int find(pList p, int value) {
  for (pNode x = begin(p); x != end(p); x = x->next)
    if (x->data == value) return x;
  return x;
} // there is a bug.
```

```
int find(pList p, int value) {
  pNode x = begin(p);
  for (; x != end(p); x = x->next)
    if (x->data == value) return x;
  return x;
}
```

```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```



What does it return if not found? ———
 (1) A1, (2) tail, (3) nullptr.

```
pNode find(pList p, int value){
  pNode x = begin(p);
  while(x != end(p)) {
    if (x->data == value) return x;
    x = x->next;
  }
  return x;
}
```



```
pNode find(pList p, int value){
  pNode x = begin(p);

return x;
}
```

```
pNode x = begin(p);
while(x != end(p)) {
   if (x->data == value) return x;
   x = x->next;
}
return x;
```

pNode find(pList p, int value){

Can we reduce the number lines by two?

Data Structures Chapter 4

- 1. Singly Linked List
- 2. Doubly Linked List
 - Revisit Singly Linked List
 - Sentinel Nodes & Basic Operations
 - Two Key Operations: erase, insert
 - Advanced Operations



