Linked Lists

Linked Lists

- Linked lists and arrays are similar since they both store collections of data.
  - The *array's* features all follow from its strategy of allocating the memory for all its elements in one block of memory (contiguously).
  - *Linked lists* use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

Linked Lists

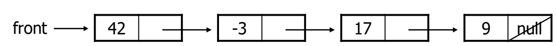
- Linked lists are used to store a collection of data (like arrays)
  - A linked list is made of nodes that are pointing to each other
  - We only know the address of the first node
  - Other nodes are reached by following the "**next**" pointers
  - The last node's "next" is NULL

Linked List vs. Array

- In a linked list, nodes are not necessarily contiguous in memory (each node is allocated with a separate "new" call)
- Arrays are contiguous in memory:

0	1	2	3
42	-3	17	9

- Linked lists:

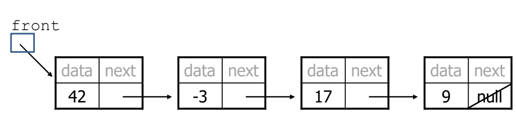


A list node class

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

- Each list node stores:
  - one piece of integer data
  - a reference to another list node
- `ListNode`s can be "linked" into chains to store a list of values:

Linked Lists

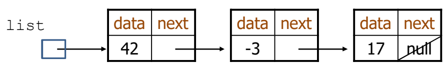


- Empty linked list is a single pointer having the value `nullptr`.

```
ListNode * front;
front = nullptr;
```

Construct a simple list

```
int main(){
    ListNode * list = new ListNode();
    list->data = 42;
    list->next = new ListNode();
    list->next->data = -3;
    list->next->next = new ListNode();
    list->next->next->data = 17;
    list->next->next->next = nullptr;
    cout << list->data << " " << list->next->data <<
                              " " << list->next->next->data;
        // 42 -3 17
}
```



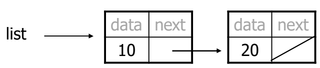
Node class with constructor

```
class ListNode {
  public:
    int data;
    ListNode *next;

    ListNode(int x) {
        data = x;
        next = nullptr;
    }

    ListNode(int x, ListNode *p) {
        data = x;
        next = p;
    }
}
```

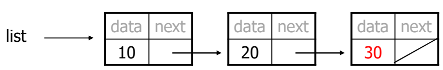
– Exercise: Modify the previous slide to use these constructors.

Linked node problem 1

- What set of statements turns this picture:

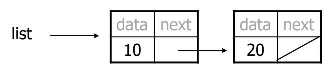


- Into this?

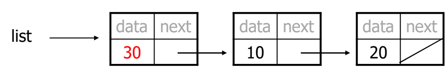


Linked node problem 2

- What set of statements turns this picture:

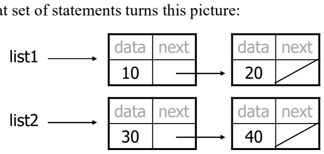


- Into this?

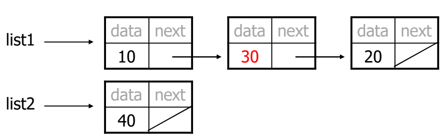


Linked node problem 3

- What set of statements turns this picture:



- Into this?

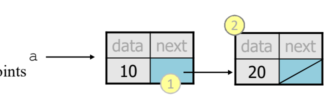


Pointer references vs. objects

variable = value;

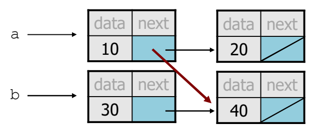
a *variable* (left side of =) is an arrow (the base of an arrow)
a *value* (right side of =) is an object (a box; what an arrow points at)

- For the list at right:
  - `a->next = ` **value;**
    means to adjust where 1 points
  - **variable** `= a->next;`
    means to make **variable** point at 2



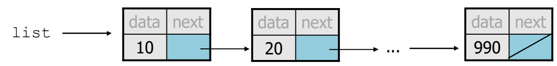
Reassigning pointers

- when you say:
  - `a->next = b->next;`
- you are saying:
  - "Make the *variable* **`a->next`** refer to the same *value* as **`b->next`**."
  - Or, "Make `a->next` point to the same place that `b->next` points."



Printing a Linked list

- Suppose we have a long chain of list nodes:

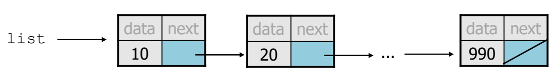


  - We don't know exactly how long the chain is.
- How would we print the data values in all the nodes?

Algorithm pseudocode

- Start at the **front** of the list.
- While (there are more nodes to print):
  - Print the current node's **data**.
  - Go to the **next** node.
- How do we walk through the nodes of the list?

```
list = list->next;   // is this a good idea?
```

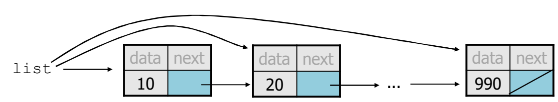


Traversing a list?

- One (bad) way to print every value in the list:

```
while (list != nullptr) {
    cout << list->data << endl;
    list = list->next;    // move to next node
}
```

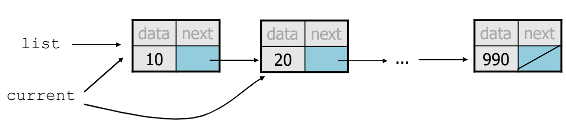
  - What's wrong with this approach?
    - (It loses the linked list as it prints it!)



A `current` pointer

- Don't change `list`. Make another variable, and change that.

```
ListNode *current = list;
```



- What happens to the picture above when we write:

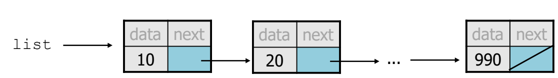
```
current = current->next;
```

Traversing a list correctly

- The correct way to print every value in the list:

```
ListNode *current = list;
while (current != nullptr) {
    cout << current-> data << endl;
    current = current->next;  // move to next node
}
```

  - Changing `current` does not damage the list.



Constructing a long list

```
int main(){
   ListNode * list = new ListNode(1);
   ListNode * p = list;

   for (int i = 2; i <=100; i++){
      p->next = new ListNode(i);
      p = p->next;
   }

   p = list;
   while (p!=nullptr){
      cout << p->data << " " ;
      p = p->next;
   }
   cout << endl;
}
```

Linked List vs Arrays

- Algorithm to print list:

```
ListNode *front = ...;

ListNode *current = front;
while (current != nullptr) {
    cout << current->data << endl;
    current = current->next;
}
```

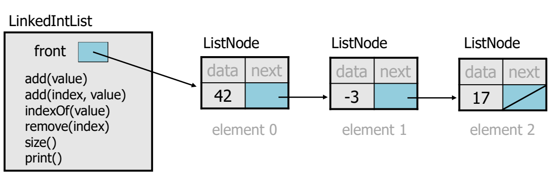
- Similar to array code:

```
int  a[] = ...;

int i = 0;
while (i < aSize) {
    cout << a[i] << endl;
    i++;
}
```

20

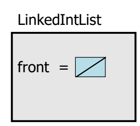
A `LinkedIntList` class

- Let's write a class named `LinkedIntList`.
  - Has the methods :
    - `add`, `get`, `indexOf`, `remove`, `size`, `print`
  - The list is internally implemented as a chain of linked nodes
    - The `LinkedIntList` keeps a pointer to its `front` as a private field
    - `null` is the end of the list; a `null` front signifies an empty list



`LinkedIntList` class

```
class LinkedIntList {
public:
  LinkedIntList() {
        front = nullptr;
   }

        methods go here

private:
  ListNode * front;
}
```



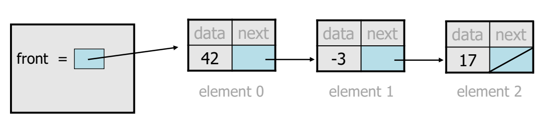
`LinkedIntList.h`

```
class LinkedIntList{
public:
    LinkedIntList() {
        front = nullptr;
    }
    ~LinkedIntList();
    LinkedIntList(const LinkedIntList & rhs);
    LinkedIntList & operator=(const LinkedIntList rhs);
    void add (int value);
    void add (int index, int value);
    int get (int index);
    int remove(); // throws NoSuchElementException;
    void remove(int index);
    void print();

private:
    ListNode *front;
};
```

Implementing `add`

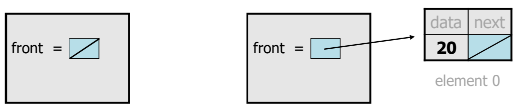
```
// Adds the given value to the end of the list.
void add(int value) {
    ...
}
```

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?



Adding to an empty list

- Before adding 20: After:



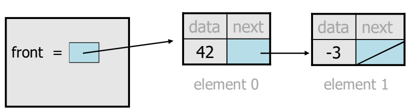
  - We must create a new node and attach it to the list.

The add method, 1st try

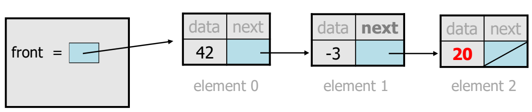
```
// Adds the given value to the end of the list.
void add(int value) {
    if (front == nullptr) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list

        ...

    }
}
```

Adding to non-empty list

- Before adding value 20 to end of list:

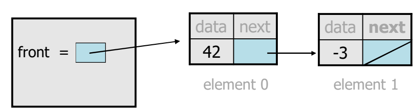


- After:



Don't fall off the edge!

- To add/remove from a list, you must modify the `next` of the node *before* the place you want to change.



  - Where should `current` be pointing, to add 20 at the end?
  - What loop test will stop us at this place in the list?

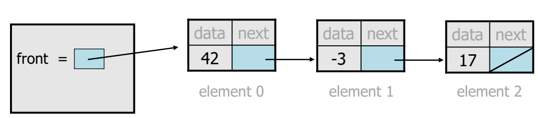
The add method

```
// Adds the given value to the end of the list.
void add(int value){
    if (front == nullptr) {
        // adding to an empty list
        front = new ListNode(value);
    }
    else {
        // adding to the end of an existing list
        ListNode *current = front;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = new ListNode(value);
    }
}
```

Implementing get

```
// Returns value in list at given index.
int get(int index) {
    ...
}
```

  - Exercise: Implement the `get` method.



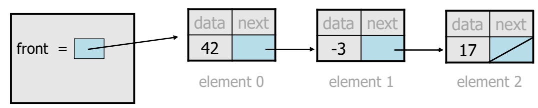
The `get` method

```
// Returns value in list at given index.
// Precondition: 0 <= index < size()
int get(int index) {
    ListNode *current = front;
    for (int i = 0; i < index; i++) {
        current = current->next;
    }
    return current->data;
}
```

Implementing `add` (2)

```
// Inserts the given value at the given index.
void add(int index, int value) {
    ...
}
```

– Exercise: Implement the two-parameter `add` method.



The `add` method (2)

```
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
void add(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    }
    else {
        // inserting into an existing list
        ListNode *current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        current->next = new ListNode(value,
                                     current->next);
    }
}
```

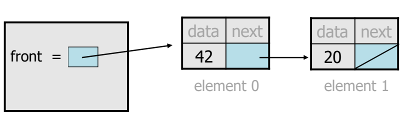
Implementing `remove`

```
// Removes and returns the list's first value.
int remove() {
    ...
}
```

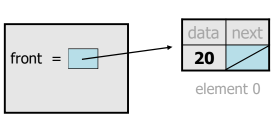
– How do we remove the front node from a list?
– Does it matter what the list's contents are before the remove?

Removing front element

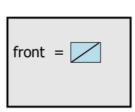
- Before removing front element:



- After first removal:



After second removal:



`remove` solution

```
// Removes and returns the first value.
// Throws a NoSuchElementException on empty list.
int remove() {
    if (front == nullptr) {
        throw NoSuchElementException();
    }
    else {
        int result = front->data;
        ListNode *tmp = front;
        front = front->next;
        delete tmp;
        return result;
    }
}
```

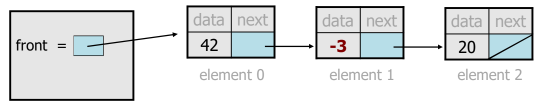
Implementing `remove` (2)

```
// Removes value at given index from list.
// Precondition: 0 <= index < size
void remove(int index) {
    ...
}
```

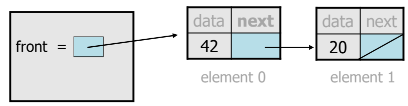
- How do we remove any node in general from a list?
- Does it matter what the list's contents are before the remove?

Removing from a list

- Before removing element at index 1:

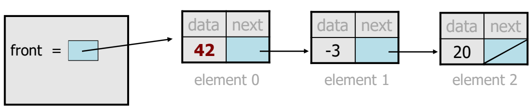


- After:

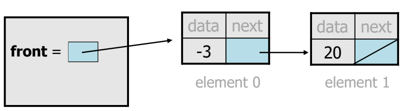


Removing from the front

- Before removing element at index 0:

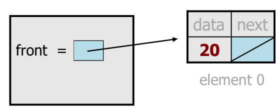


- After:

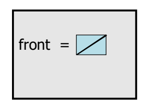


Removing the only element

- Before:



After:



- We must change the front field to store `null` instead of a node.
- Do we need a special case to handle this?

`remove` (2) solution

```
// Removes value at given index from list.
// Precondition: 0 <= index < size()
void remove(int index) {
    if (index == 0) {
        // special case: removing first element
        ListNode* tmp = front;
        front = front->next;
        delete tmp;
    }
    else {
        // removing from elsewhere in the list
        ListNode *current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current->next;
        }
        ListNode *tmp = current->next;
        current->next = current->next->next;
        delete tmp;
    }
}
```

Implementing `print`

```
// Prints the data values in the list in one line.
void print(){
    ListNode * current = front;
    while (current != nullptr){
        cout << current->data << " " ;
        current = current->next;
    }
    cout << endl;
}
```

Rule of three

```
// Destructor
~LinkedIntList();

// Copy constructor
LinkedIntList(const LinkedIntList & rhs);

// Assignment operator
LinkedIntList & operator=(const LinkedIntList rhs);
```

See the given C++ code for their implementation.

Using `LinkedIntList` class

```
int main(){

   LinkedIntList list;

   list.add(5);
   list.add(10);
   list.add(15);
   list.print();
   cout <<"second element is " << list.get(1) << endl;
   try{
     list.remove(2);
     list.remove();
     list.remove();
     list.remove();
     list.print();
   }
   catch (NoSuchElementException e){
     cout << "List is empty!!"<< endl;
   }
}
```

Conceptual questions

- What is the difference between a `LinkedIntList` and a `ListNode`?
- What is the difference between an empty list and a `null` list?
  - How do you create each one?
- Why are the fields of `ListNode` public? Is this bad style?
- What effect does this code have on a `LinkedIntList`?

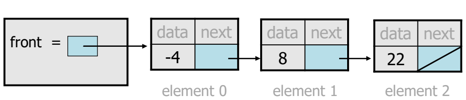
```
ListNode *current = front;
current = nullptr;
```

Conceptual answers

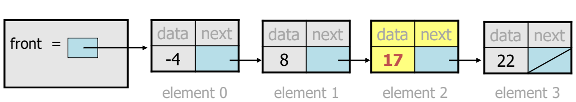
- A list consists of 0 to many node objects.
  - Each node holds a single data element value.
- null list: `LinkedIntList *list = nullptr;`
  empty list: `LinkedIntList *list = new LinkedIntList();`
- It's okay that the node fields are public, because client code never directly interacts with `ListNode` objects.
- The code doesn't change the list.
  You can change a list only in one of the following two ways:
  - Modify its `front` field value.
  - Modify the `next` reference of a node in the list.

Exercise

- Write a method `addSorted` that accepts an integer value as a parameter and adds that value to a sorted list in sorted order.
  - Before `addSorted(17)`:

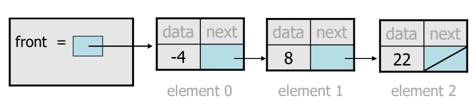


  - After `addSorted(17)`:



The common case

- Adding to the middle of a list:
  `addSorted(17)`

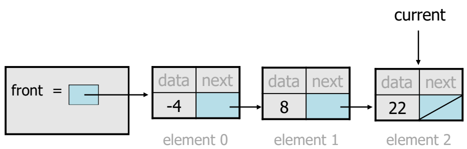


  - Which pointers must be changed?
  - What sort of loop do we need?
  - When should the loop stop?

First attempt

- An incorrect loop:

```
ListNode *current = front;
while (current->data < value) {
    current = current->next;
}
```

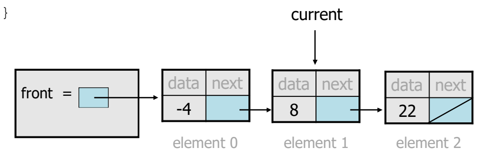


- What is wrong with this code?
  - The loop stops too late to affect the list in the right way.

Key idea: peeking ahead

- Corrected version of the loop:

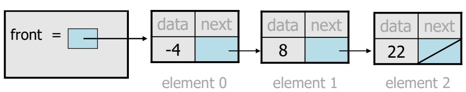
```
ListNode *current = front;
while (current->next->data < value) {
    current = current->next;
}
```



  - This time the loop stops in the right place.

Another case to handle

- Adding to the end of a list:
  `addSorted(42)`



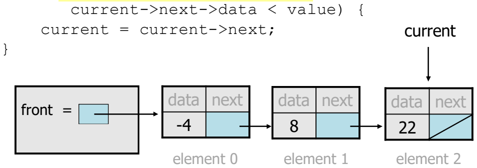
RUN TIME ERROR!!

  - Why does our code crash?
  - What can we change to fix this case?

Multiple loop tests

- A correction to our loop:

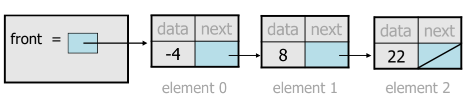
```
ListNode current = front;
while (current->next != nullptr &&
        current->next->data < value) {
    current = current->next;
}
```



  - We must check for a `next` of `null` *before* we check its `data`.

Third case to handle

- Adding to the front of a list:
  `addSorted(-10)`



  - What will our code do in this case?
  - What can we change to fix it?

Handling the front

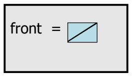
- Another correction to our code:

```
if (value <= front->data) {
    // insert at front of list
    front = new ListNode(value, front);
}
else {
    // insert in middle of list
    ListNode *current = front;
    while (current->next != nullptr &&
           current->next->data < value) {
        current = current->next;
    }
    current->next =
           new ListNode(value,current->next);
}
```

  - Does our code now handle every possible case?

Fourth case to handle

- Adding to (the front of) an empty list:
  `addSorted(42)`



  - What will our code do in this case?
  - What can we change to fix it?

Final version of code

```
// Adds given value to list in sorted order.
// Precondition: Existing elements are sorted
void addSorted(int value) {
    if (front == null || value <= front->data) {
        // insert at front of list
        front = new ListNode(value, front);
    } else {
        // insert in middle of list
        ListNode *current = front;
        while (current->next != null &&
               current->next->data < value) {
            current = current->next;
        }
        current->next =
           new ListNode(value,current->next);
    }
}
```

Other list features

- Add the following methods to the `LinkedIntList`:
  - `size`
  - `isEmpty`
  - `clear`
  - `indexOf`
  - `contains`

- Add a `size` field to the list to return its size more efficiently.

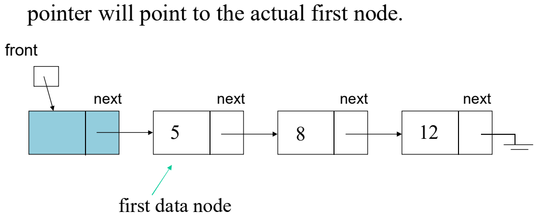
- Add preconditions and exception tests to appropriate methods.

Variations of Linked Lists

- The linked list that we studied so far is called **singly linked list.**
- Other types of linked lists exist, namely:
  - Doubly linked list
  - Circular linked linked list
  - Circular doubly linked list
- Each type of linked list may be suitable for a different kind of application.
- We may also use a dummy node for simplifying insertions and deletions.

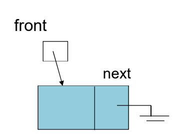
Dummy head node

- To avoid checking if list front is null at every insert and delete operation, we can add a **dummy head node** to the beginning of the list.
- This dummy node will be the zeroth node and its next pointer will point to the actual first node.

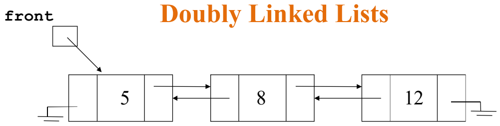


Dummy head node

- An empty list will look like this (the contents of the node is irrelevant):



Doubly Linked Lists




```
class DLL_Node {
  public:
      int data;
      DLL_Node *prev;
      DLL_Node *next;
}
```

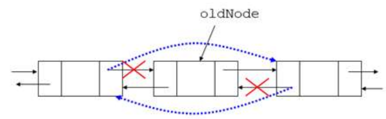
Advantages:

- Convenient to traverse the list backwards too.
- **e.g.** printing the contents of the list in reverse order.

Disadvantage:

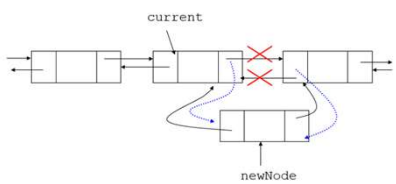
- Increase in space requirements due to storing two pointers instead of one

Deletion




```
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
delete oldNode;
```

Insertion




```
newNode = new Node(x,NULL,NULL);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
```

Doubly Linked List – Exercise

```
class DLL_Node {
  public:
      int data;
      DLL_Node* next;
      DLL_Node* prev;
}
// create a doubly linked list with a single node
DLL_Node * head = new DLL_Node();
head->data = 3;
head->next = nullptr;  head->prev = nullptr;

// create another node
DLL_Node * nd = new DLL_Node();
nd->data = 5;
nd->next = nullptr;
nd->prev = nullptr;

//make node pointed by nd the first node in this doubly linked
list
```

Doubly Linked List – Exercise (cont.)

```
class DLL_Node {
  public:
      int data;
      DLL_Node* next;
      DLL_Node* prev;
}
// create another node
DLL_Node * nd = new DLL_Node();
nd->data = 7;
nd->next = nullptr;
nd->prev = nullptr;

//make node pointed by nd the second node in this doubly linked
list
```

Doubly Linked List – Solution

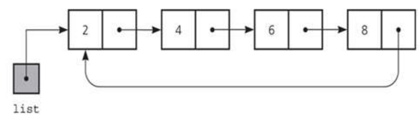
```
class DLL_Node {
  public:
      int data;
      DLL_Node* next;
      DLL_Node* prev;
}
// create a doubly linked list with a single node
DLL_Node * head = new DLL_Node();
head->data = 3;
head->next = nullptr;  head->prev = nullptr;

// create another node
DLL_Node * nd = new DLL_Node();
nd->data = 5;
nd->next = nullptr;
nd->prev = nullptr;


//make node pointed by nd the first node in this doubly linked list

nd->next = head;
head->prev = nd;
head = nd;
```

Doubly Linked List – Solution

```
class DLL_Node {
  public:
      int data;
      DLL_Node* next;
      DLL_Node* prev;
}
// create another node
DLL_Node * nd = new DLL_Node();
nd->data = 7;
nd->next = nullptr;
nd->prev = nullptr;

//make node pointed by nd the second node in this doubly linked
list
 nd->prev = head;
 nd->next = head->next;
 head->next = nd;
 nd->next->prev = nd;
```

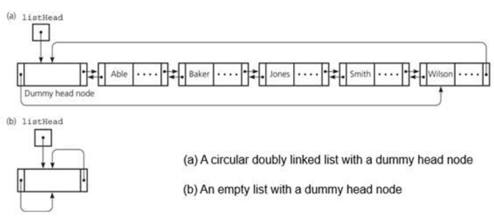
Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*



Circular Doubly Linked Lists

- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node