Linked Lists

Linked List Basics

- 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.
- *Linked lists* use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

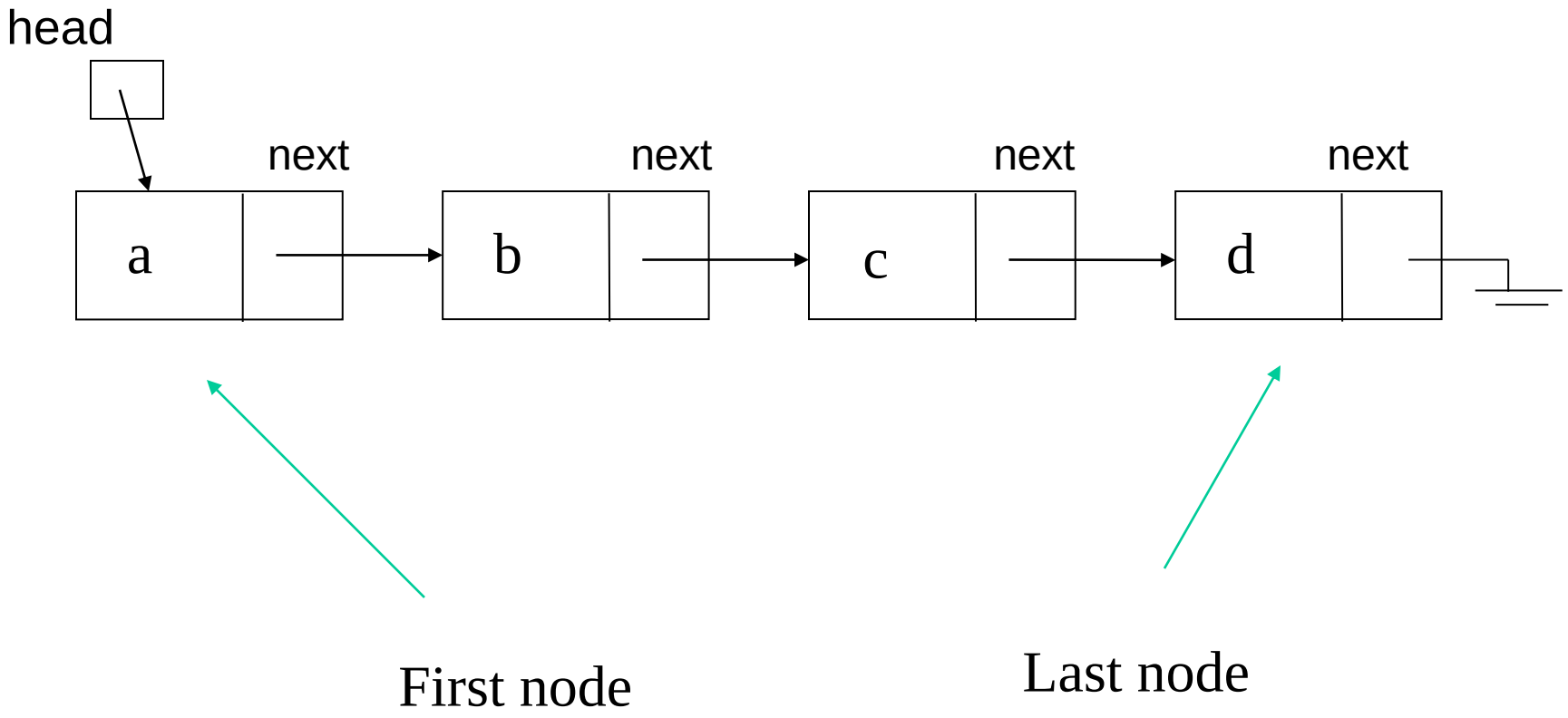
Disadvantages of Arrays

1. **The size of the array is fixed**.
   - In case of **dynamically resizing** the array from size S to 2S, we need 3S units of available memory.
   - Programmers allocate arrays which seem **"large enough**" This strategy has two disadvantages: (a) most of the time there are just 20% or 30% elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than the declared size, the code breaks.
2. **Inserting (and deleting)** elements into the middle of the array is potentially expensive because existing elements need to be shifted over to make room

Linked lists

- Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.

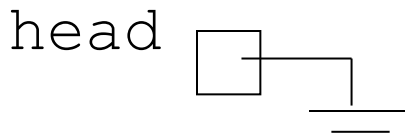
Singly Linked Lists



Empty List

- Empty Linked list is a single pointer having the value of NULL.

```
head = NULL;
```



Basic Ideas

- Let's assume that the node is given by the following type declaration:

```
struct Node {
   Object element;
   Node *next;
};
```

Basic Linked List Operations

- List Traversal
- Searching a node
- Insert a node
- Delete a node

Traversing a linked list

```
Node *pWalker;
int count = 0;

cout <<"List contains:\n";

for (pWalker=pHead; pWalker!=NULL;
     pWalker = pWalker->next)
{
   count ++;
   cout << pWalker->element << endl;
}
```

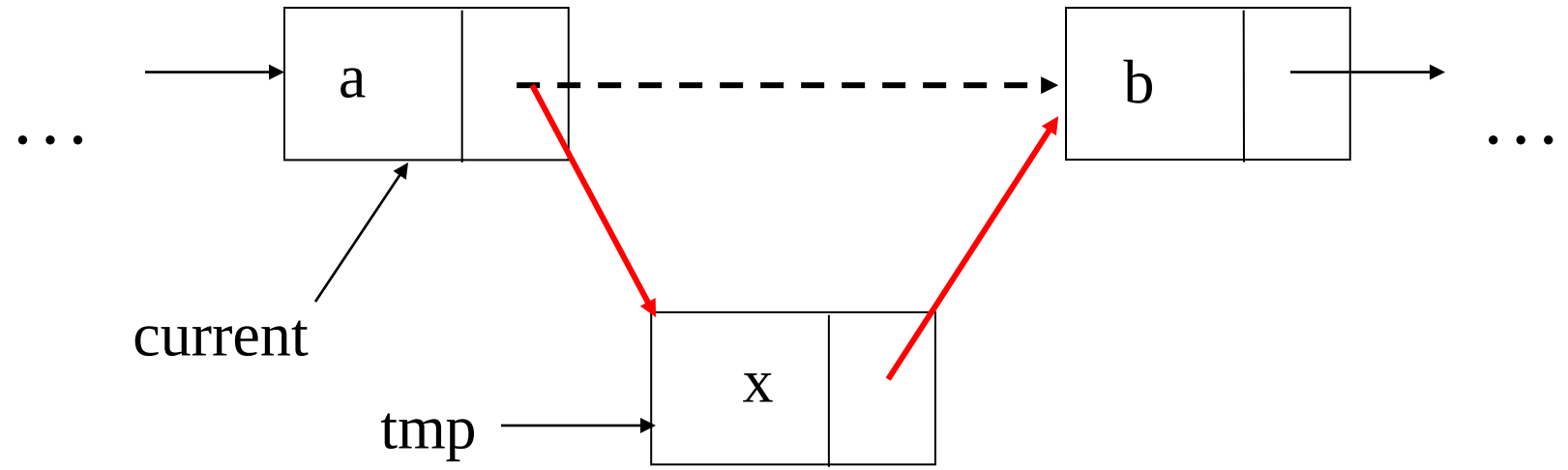
Searching a node in a linked list

```
pCur = pHead;

// Search until target is found or we reach
// the end of list
while (pCur != NULL &&
       pCur->element != target)
{
   pCur = pCur->next;
}

//Determine if target is found
if (pCur) found = 1;
else found = 0;
```

Insertion in a linked list

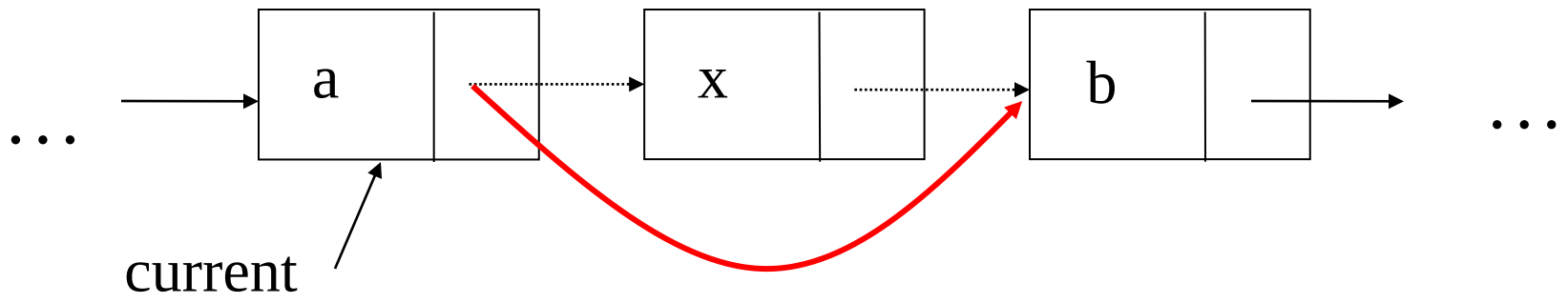



```
tmp = new Node;
tmp->element = x;
tmp->next = current->next;
current->next = tmp;
```

Or simply (if Node has a constructor initializing its members):

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

Deletion from a linked list




```
Node *deletedNode = current->next;
current->next = current->next->next;
delete deletedNode;
```

Special Cases (1)

- Inserting before the first node (or to an empty list):

```
tmp = new Node;
tmp->element = x;
if (current == NULL){
   tmp->next = head;
   head = tmp;
}
else {  // Adding in middle or at end
   tmp->next = curent->next;
   current->next = tmp;
}
```

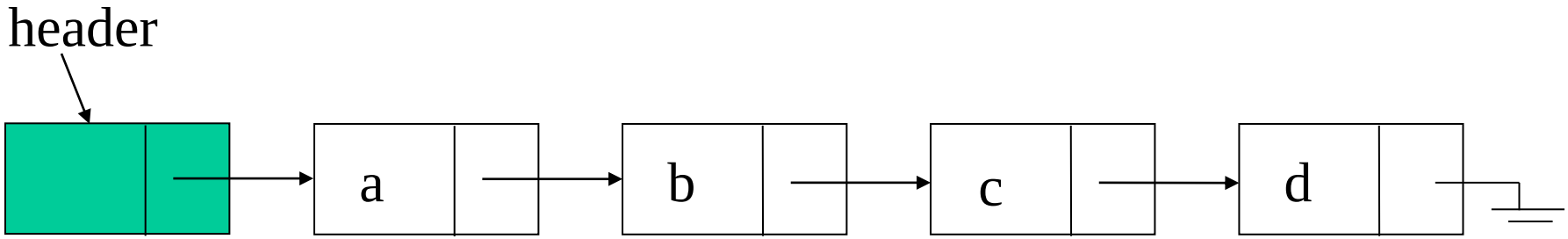
Special Cases (2)

```
Node *deletedNode;
if (current == NULL){
   // Deleting first node
   deletedNode = head;
   head = head ->next;
}
else{
   // Deleting other nodes
   deletedNode = current->next;
   current->next = deletedNode ->next;
}
delete deletedNode;
```

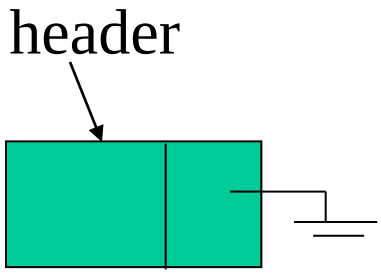
Header Nodes

- One problem with the basic description: it assumes that whenever an item x is removed (or inserted) some previous item is always present.
- Consequently removal of the first item and inserting an item as a new first node become special cases to consider.
- In order to avoid dealing with special cases: introduce a **header node (dummy node)**.
- A header node is an extra node in the list that holds no data but serves to satisfy the requirement that every node has a previous node.

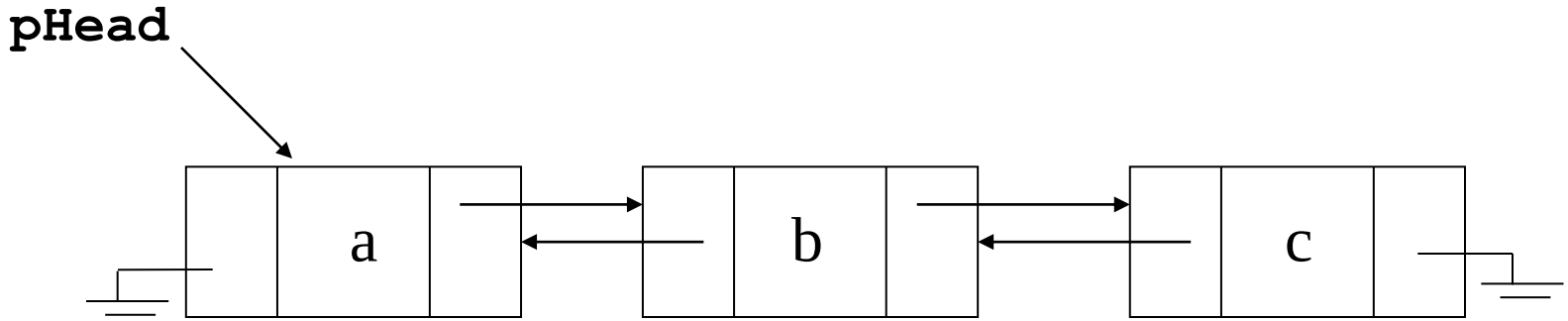
List with a header node



Empty List



Doubly Linked Lists



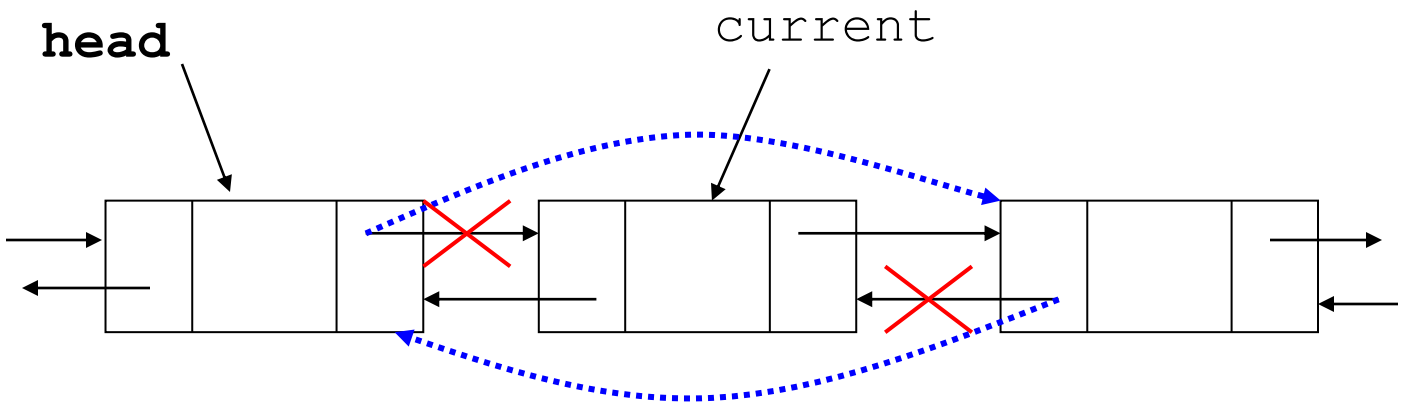
Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

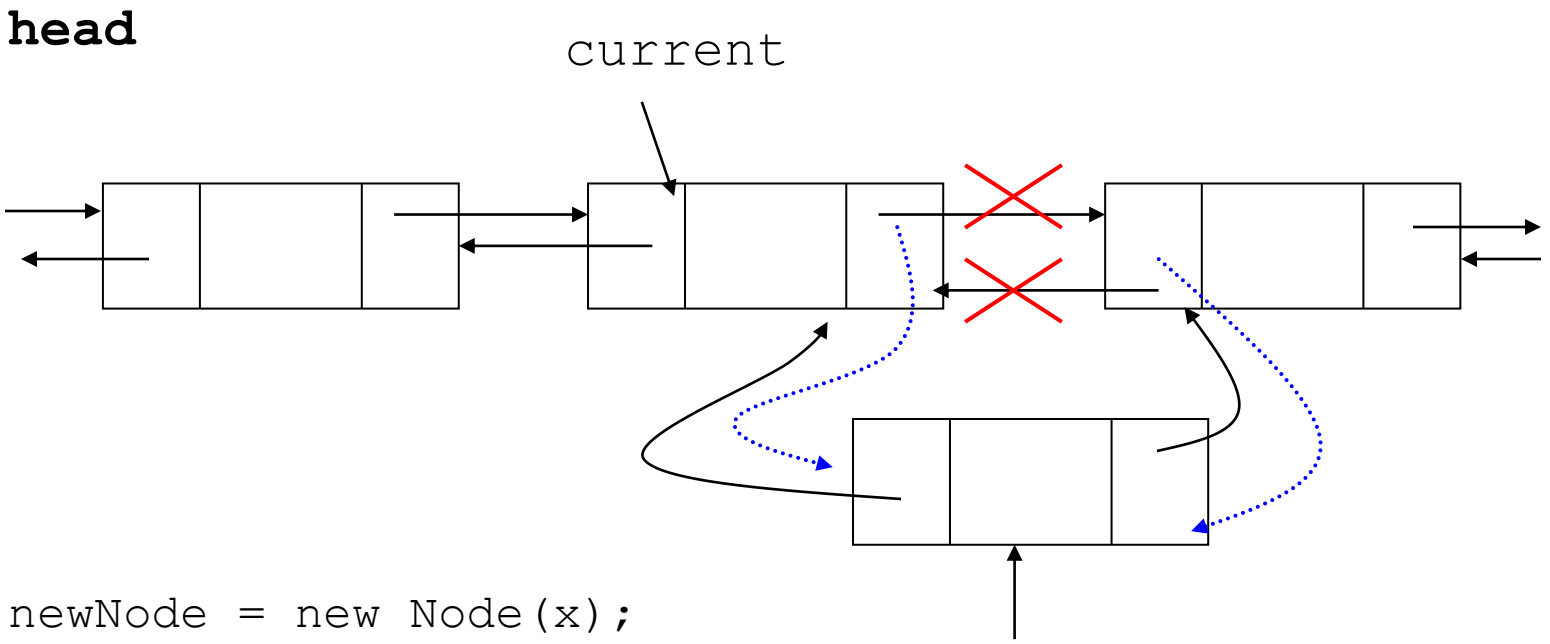
- Increase in space requirements.

Deletion




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

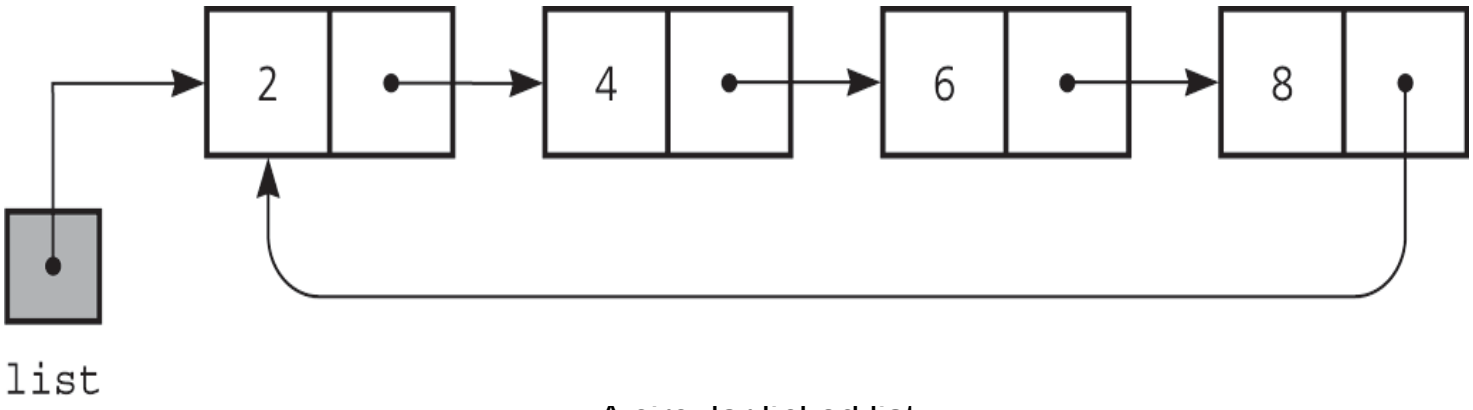
Insertion




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

Circular Linked Lists

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

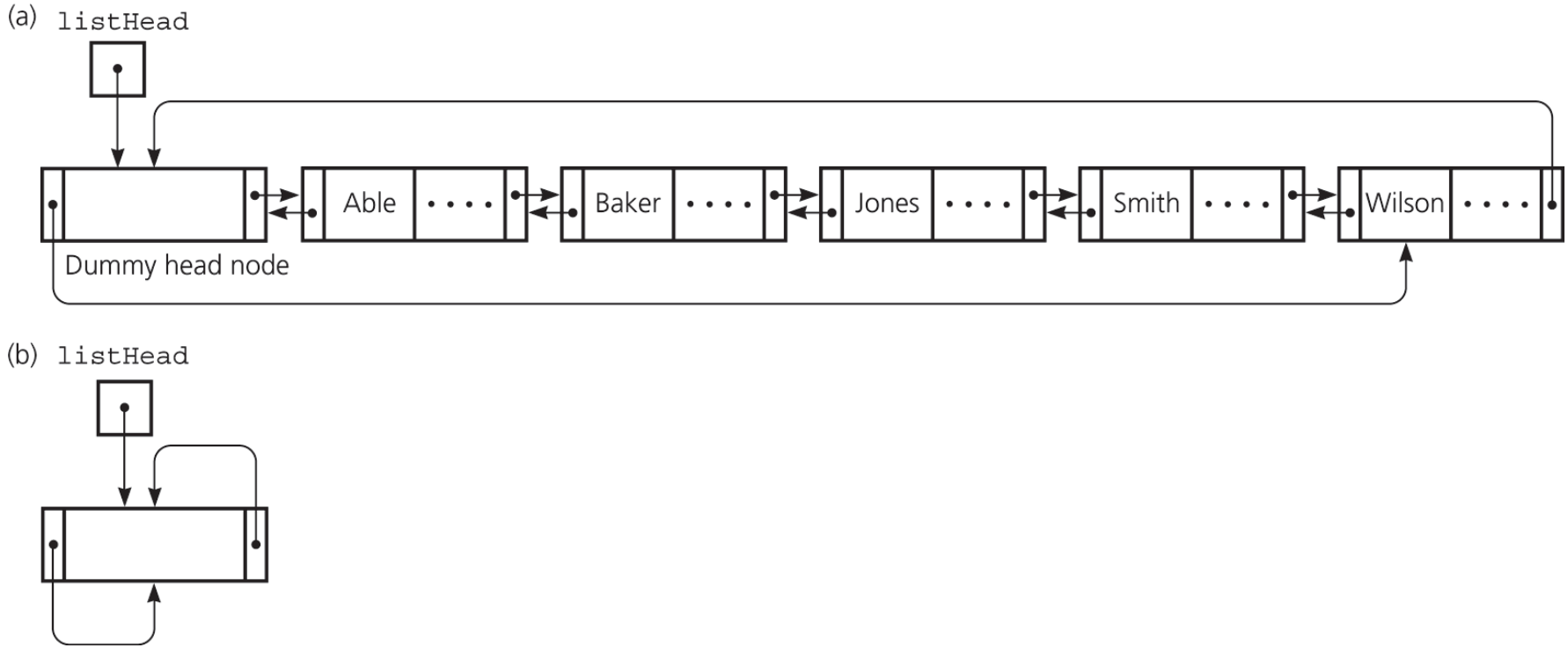


A circular linked list

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

Circular Doubly Linked Lists



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

(b) An empty list with a dummy head node