Maciej Sobieraj

Lecture 12

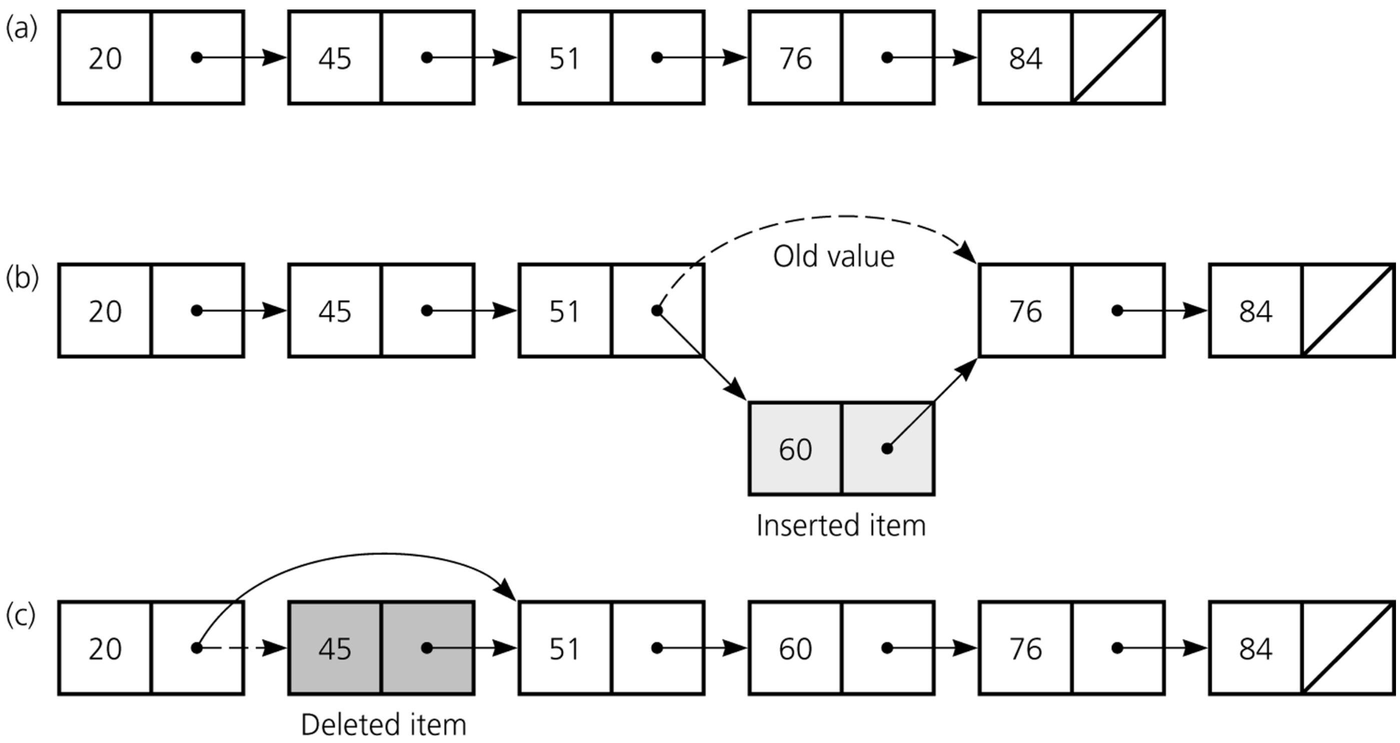
Outline

1. Linked Lists

Preliminaries

- Options for implementing an ADT List
  - Array has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

Preliminaries



a) A linked list of integers; b) insertion; c) deletion

Pointers

- A pointer contains the location, or address in memory, of a memory cell
  - Initially undefined, but not *NULL*
  - A statically allocated pointer declaration

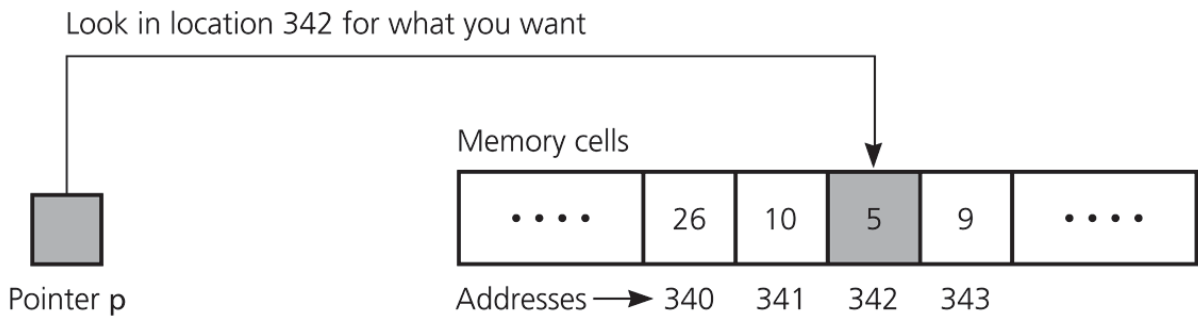
    ```
    int *p;
    ```

  - A dynamically allocated pointer variable

    ```
    p =  new int;
    ```

Pointers

- The expression, `*p`, denotes the memory cell to which `p` points
- The `&` address-of operator places the address of a variable into a pointer variable



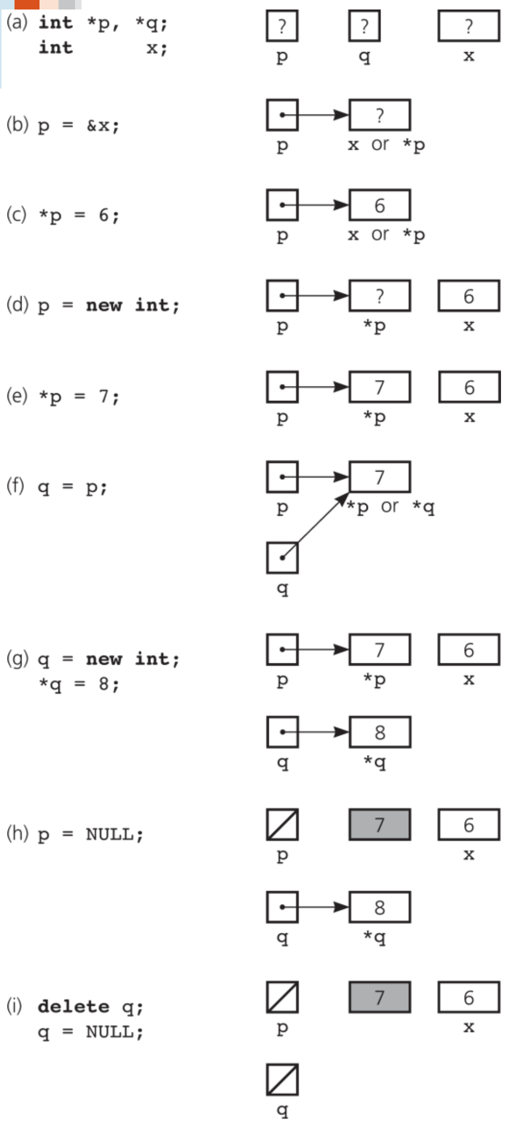
A pointer to an integer

Pointers

- The `delete` operator returns dynamically allocated memory to the system for reuse, and leaves the variable undefined
  - **`delete`** `p`;
  - A pointer to a deallocated memory cell is possible and dangerous
- Assign the pointer `q` the value in `p`

```
q = p;
```

Pointers



(a) declaring pointer variables;

(b) pointing to statically allocating memory;

(c) assigning a value;

(d) allocating memory dynamically;

(e) assigning a value

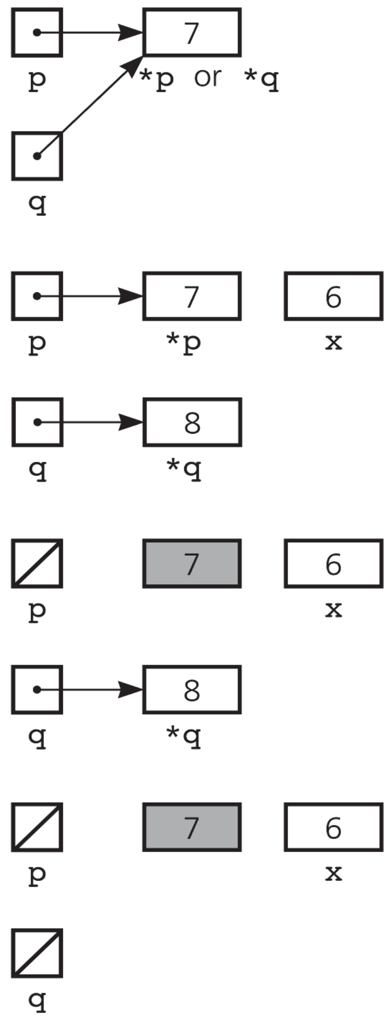
Pointers

(f) `q = p;`

(g) `q = new int;`
`*q = 8;`

(h) `p = NULL;`

(i) `delete q;`
`q = NULL;`



(f) copying a pointer;

(g) allocating memory dynamically

and assigning a value;

(h) assigning NULL to a pointer variable;

(i) deallocating memory

Dynamic Allocation of Arrays

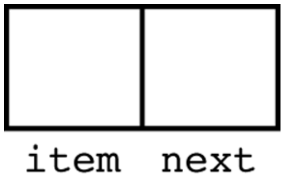
- Use the `new` operator to allocate an array dynamically
- An array name is a pointer to the array's first element
- The size of a dynamically allocated array can be increased

```
double* oldArray = anArray;
anArray = new double[2*arraySize];
```

Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

```
struct Node
{ int item
  Node *next;
}; //end struct
```



A node

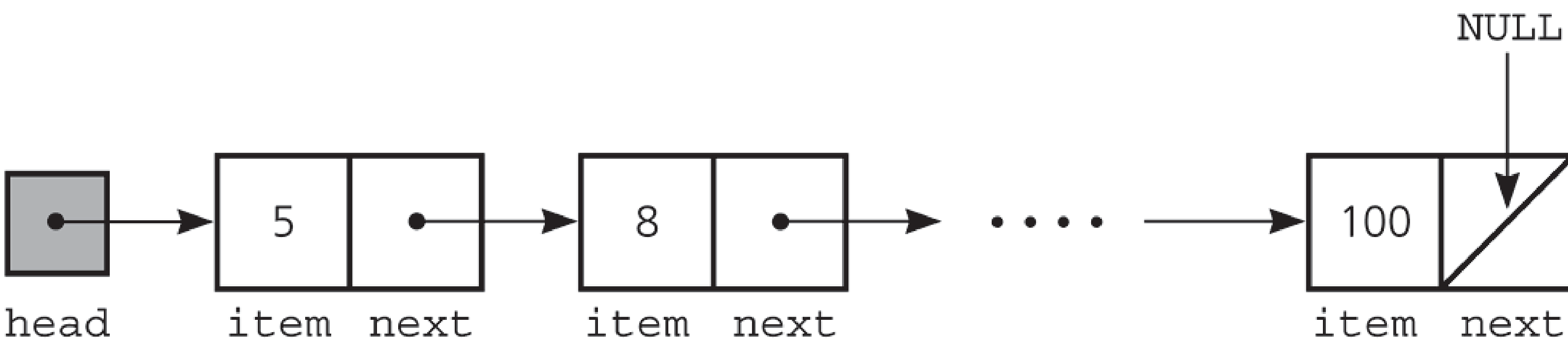
- A node is dynamically allocated

```
Node *p;
p = new Node;
```

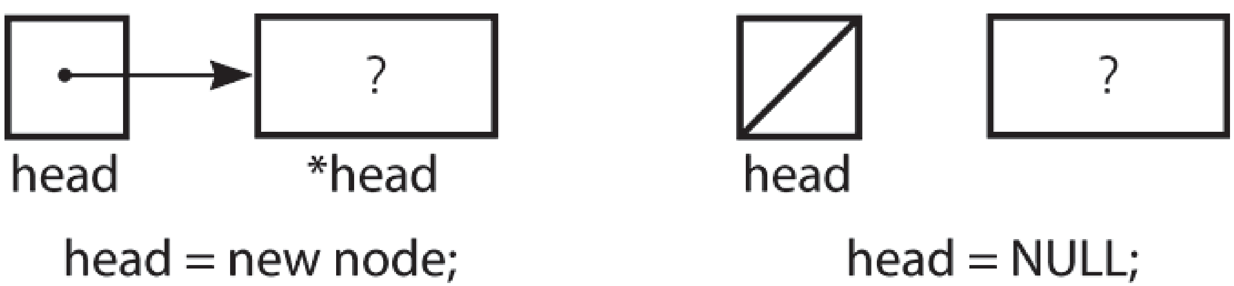
Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is *NULL*, the linked list is empty
- Executing the statement `head=`**`new`**` Node` before `head=NULL` will result in a lost cell

Pointer-Based Linked Lists



A head pointer to a list



A lost cell

Displaying the Contents of a Linked List

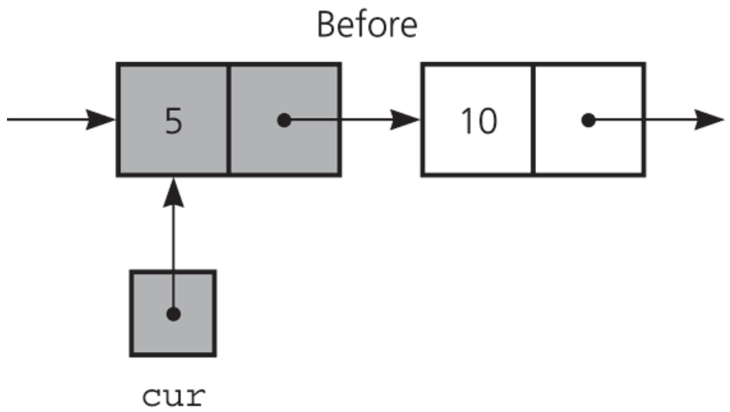
- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable `cur` keeps track of the current node

```
for (Node *cur = head;
     cur != NULL; cur = cur->next)
   cout << cur->item << endl;
```

Displaying the Contents of a Linked List



The effect of the assignment ***cur = cur->next***

Deleting a Specified Node from a Linked List

- Deleting an interior node

```
prev->next=cur->next;
```

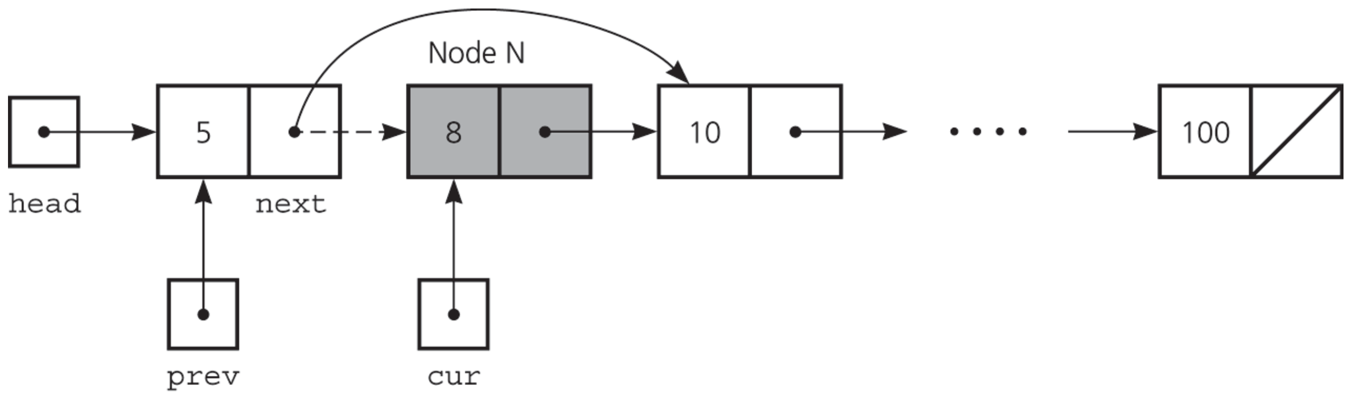
- Deleting the first node

```
head=head->next;
```

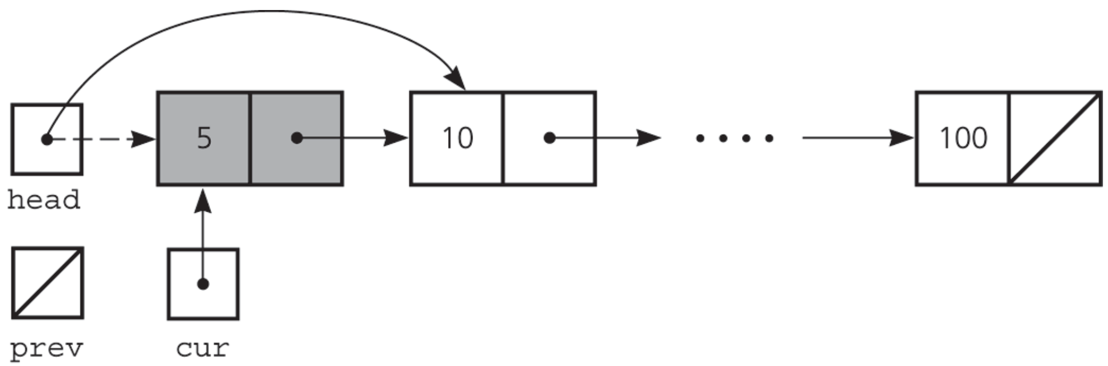
- Return deleted node to system

```
cur->next = NULL;
delete cur;
cur=NULL;
```

Deleting a Specified Node from a Linked List



Deleting a node from a linked list

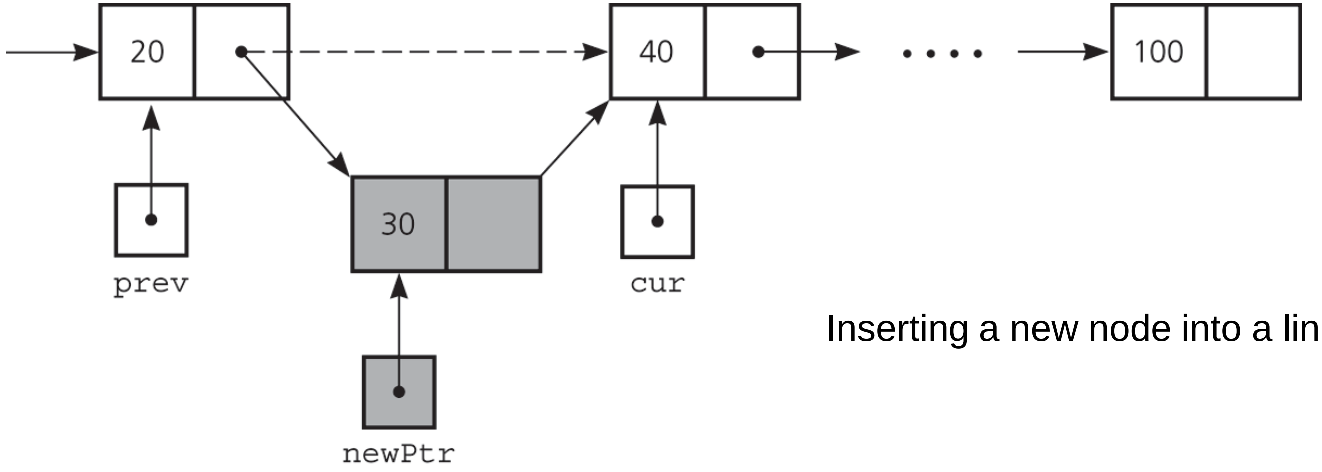


Deleting the first node

Inserting a Node into a Specified Position of a Linked List

- To insert a node between two nodes

```
newPtr->next = cur;
prev->next = newPtr;
```

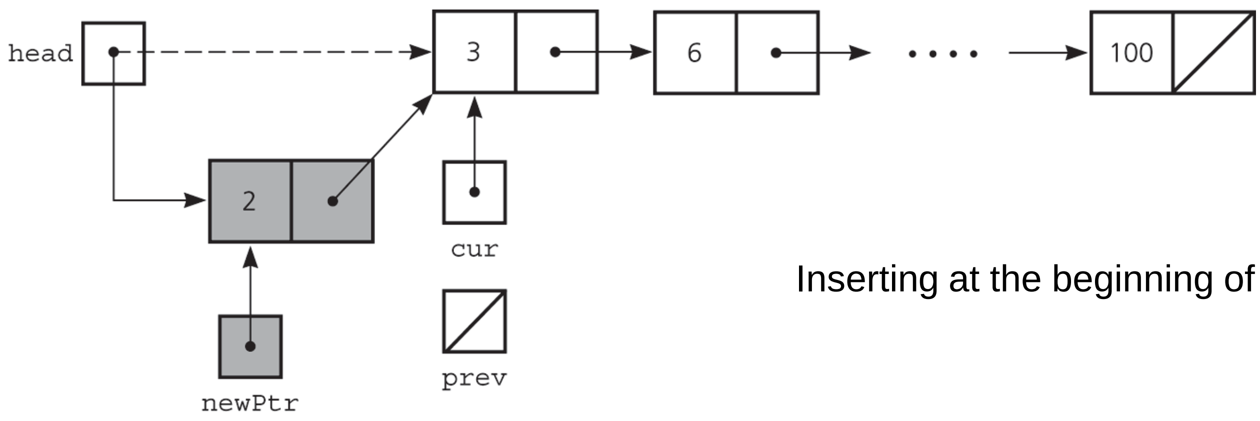


Inserting a new node into a linked list

Inserting a Node into a Specified Position of a Linked List

- To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```

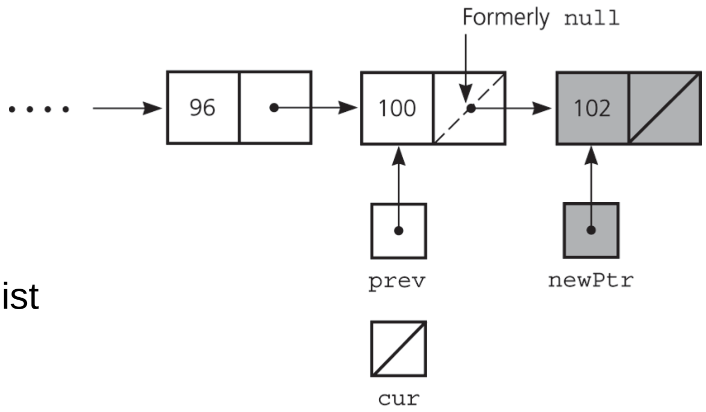


Inserting at the beginning of a linked list

Inserting a Node into a Specified Position of a Linked List

- Inserting at the end of a linked list is not a special case if `cur` is *NULL*

```
newPtr->next = cur;
prev->next = newPtr;
```



Inserting at the end of a linked list

Inserting a Node into a Specified Position of a Linked List

- Determining the point of insertion or deletion for a sorted linked list of objects

```
for(prev = NULL, cur= head;
     (cur != null)&&
    (newValue > cur->item);
    prev = cur, cur = cur->next;
```

A Pointer-Based Implementation of the ADT List

- Public methods
  - `isEmpty`
  - `getLength`
  - `insert`
  - `remove`
  - `retrieve`
- Private method
  - `find`
- Private Data

  Members
  - `head`
  - `Size`
- Local variables to member functions
  - `cur`
  - `prev`

Constructors and Destructors

- Default constructor initializes size and head
- Copy constructor allows a deep copy
  - Copies the array of list items and the number of items
- A destructor is required for dynamically allocated memory

Comparing Array-Based and Pointer-Based Implementations

- Size
  - Increasing the size of a resizable array can waste storage and time
- Storage requirements
  - Array-based implementations require less memory than a pointer-based ones

Comparing Array-Based and Pointer-Based Implementations

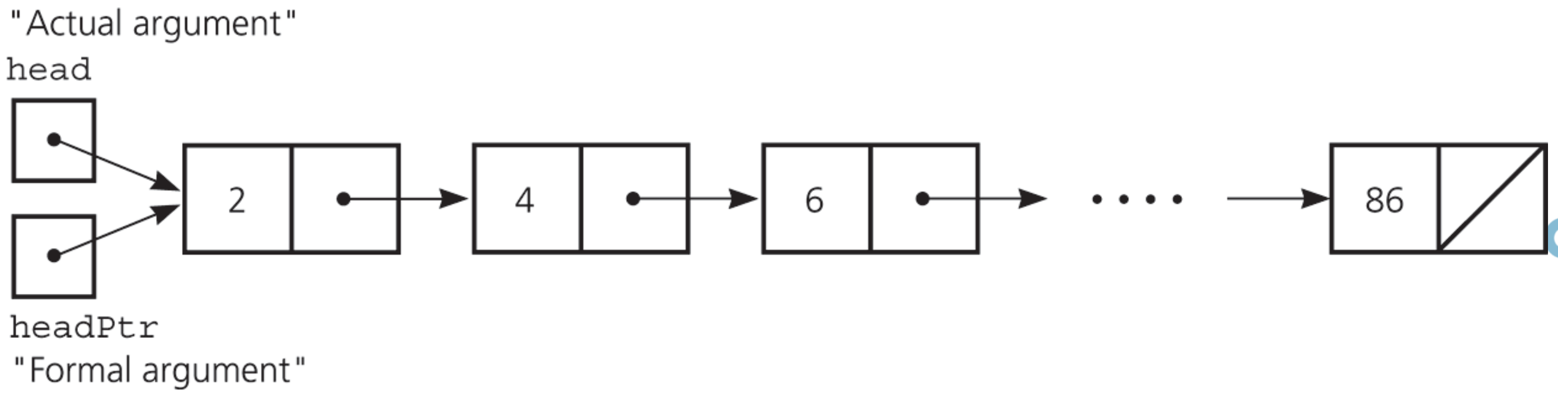
- Access time
  - Array-based: constant access time
  - Pointer-based: the time to access the i^{th} node depends on *i*
- Insertion and deletions
  - Array-based: require shifting of data
  - Pointer-based: require a list traversal

Saving and Restoring a Linked List by Using a File

- Use an external file to preserve the list between runs
- Do not write pointers to a file, only data
- Recreate the list from the file by placing each item at the end of the list
  - Use a tail pointer to facilitate adding nodes to the end of the list
  - Treat the first insertion as a special case by setting the tail to head

Passing a Linked List to a Function

- A function with access to a linked list's `head` pointer has access to the entire list
- Pass the head pointer to a function as a reference argument



A head pointer as a value argument

Processing Linked Lists Recursively

- Recursive strategy to display a list
  - Write the first node of the list
  - Write the list minus its first node
- Recursive strategies to display a list backward
  - `writeListBackward` strategy
    - Write the last node of the list
    - Write the list minus its last node backward

Processing Linked Lists Recursively

- `writeListBackward2` strategy
  - Write the list minus its first node backward
  - Write the first node of the list
- Recursive view of a sorted linked list
  - The linked list to which `head` points is a sorted list if
    - `head` is *`NULL`* *or*
    - `head->next` is *`NULL`* *or*
    - `head->item < head->next->item`, and `head->next` points to a sorted linked list

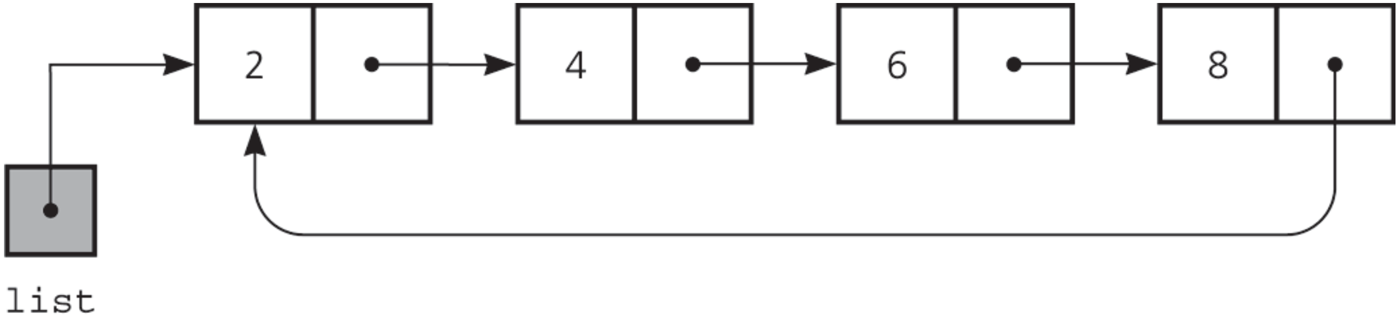
Objects as Linked List Data

- Data in a linked list node can be an instance of a class

```
typedef ClassName ItemType;
struct Node
{ ItemType item;
   Node *next;
}; //end struct
Node *head;
```

Circular Linked Lists

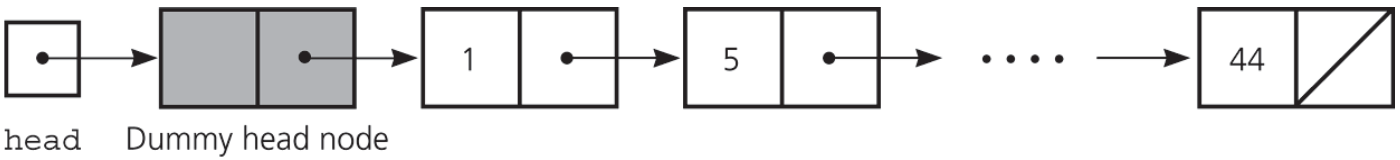
- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*



A circular linked list

Dummy Head Nodes

- Dummy head node
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than *NULL*

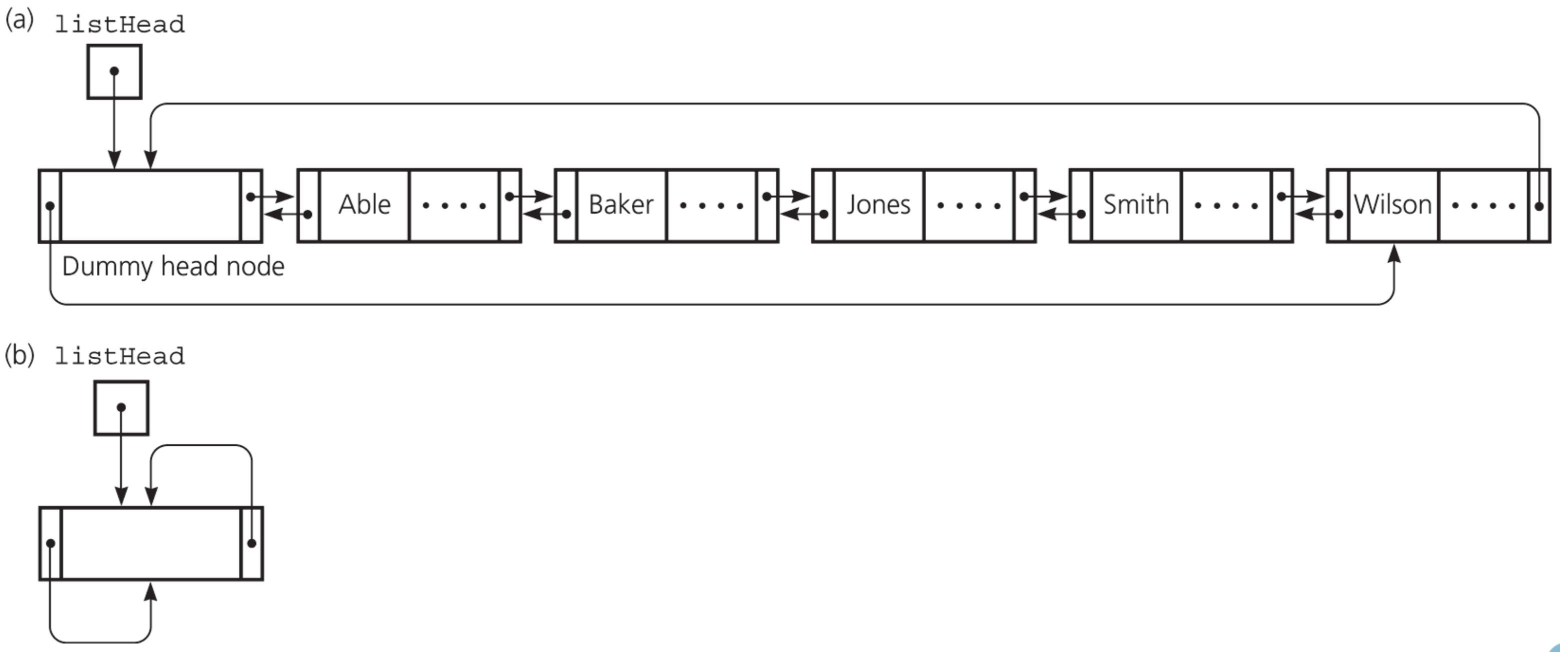


A dummy head node

Doubly Linked Lists

- Each node points to both its predecessor and its successor
- Circular doubly linked list
  - `precede` 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

Doubly Linked Lists



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

(b) An empty list with a dummy head node

Doubly Linked Lists

- To delete the node to which `cur` points

```
(cur->precede)->next = cur->next;
(cur->next)->precede = cur->precede;
```

- To insert a new node pointed to by `newPtr` before the node pointed to by `cur`

```
newPtr->next = cur;
newPtr->precede = cur->precede;
cur->precede = newPtr;
newPtr->precede->next = newPtr;
```

Application: Maintaining an Inventory

- Operations on the inventory
  - List the inventory in alphabetical order by title (L command)
  - Find the inventory item associated with title (I, M, D, O, and S commands)
  - Replace the inventory item associated with a title (M, D, R, and S commands)
  - Insert new inventory items (A and D commands)

The C++ Standard Template Library

- The STL contains class templates for some common ADTs, including the `list` class
- The STL provides support for predefined ADTs through three basic items
  - Containers are objects that hold other objects
  - Algorithms act on containers
  - Iterators provide a way to cycle through the contents of a container

Summary

- The C++ `new` and `delete` operators enable memory to be dynamically allocated and recycled
- Each pointer in a linked list is a pointer to the next node in the list
- Array-based lists use an implicit ordering scheme; pointer-based lists use an explicit ordering scheme

Summary

- Algorithms for insertions and deletions in a linked list involve traversing the list and performing pointer changes
  - Inserting a node at the beginning of a list and deleting the first node of a list are special cases
- A class that allocates memory dynamically needs an explicit copy constructor and destructor

Summary

- Recursion can be used to perform operations on a linked list
- In a circular linked list, the last node points to the first node
- Dummy head nodes eliminate the special cases for insertion into and deletion from the beginning of a linked list