Searching a Linked List

- A loop that visits the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node

```
for (p = first; p != NULL; p = p->next)
  …
```


Searching a Linked List

- A loop of this form can be used in a function that searches a list for an integer `n`
- If it finds `n`, the function will return a pointer to the node containing `n`; otherwise, it will return a *null pointer*
- An initial version of the function

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Searching a Linked List

- There are many other ways to write `search_list`
- One alternative is to eliminate the `p` variable, instead using `list` itself to keep track of the current node

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

- Since `list` is *a copy of* the original list *pointer*, there is no harm in changing it within the function

Searching a Linked List

- Another alternative

```
struct node *search_list(struct node *list, int n)
{
 for (; list != NULL && list->value != n; list = list->next)
    ;
 return list;
}
```

- Since `list` is equal `NULL` when we reach the end of the list, returning `list` is correct even if we do not find `n`

  Thanks to the *short-circuit* nature of the `&&`

Searching a Linked List

- This version of `search_list` might be a bit clearer if we used a `while` statement

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```


Deleting a Node from a Linked List

- A big advantage of storing data in a linked list is that we can easily delete nodes
- Deleting a node involves three steps
  1. Locate the node to be deleted
  2. Alter the previous node so that it "bypasses" the deleted node
  3. Call `free` to reclaim the space occupied by the deleted node
- Step 1 is harder than it looks, because step 2 requires changing the *previous* node
- There are various solutions to this problem

Deleting a Node from a Linked List

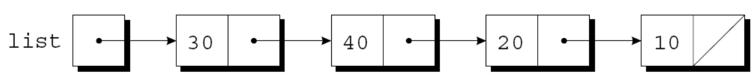
- The "trailing pointer" technique involves keeping a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`)
- Assume that `list` points to the beginning of the list to be searched and `n` is the integer to be deleted
- A loop that implements step 1

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

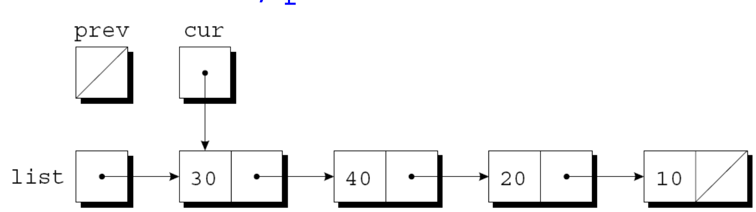
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node

Deleting a Node from a Linked List

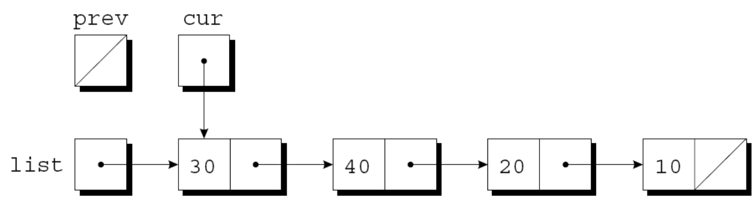
- Assume that `list` has the following appearance and `n` is `20`



- After `cur = list, prev = NULL` has been executed

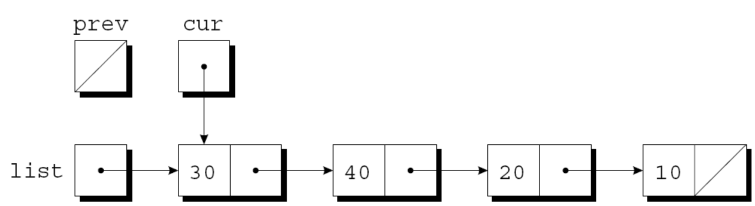


Deleting a Node from a Linked List

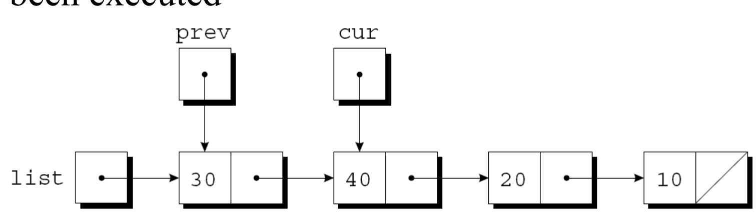


- The test `cur != NULL && cur->value != n` is `true`, since `cur` is pointing to a node and the node does not contain `20`

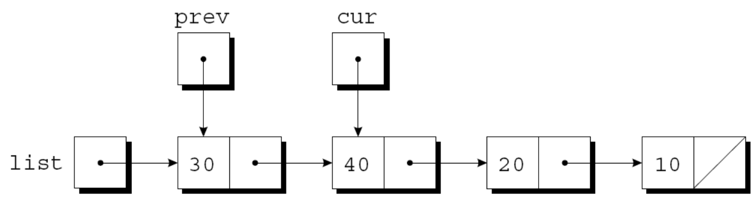
Deleting a Node from a Linked List



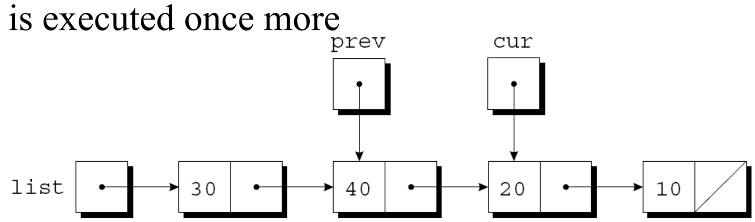
- After `prev = cur, cur = cur->next` has been executed



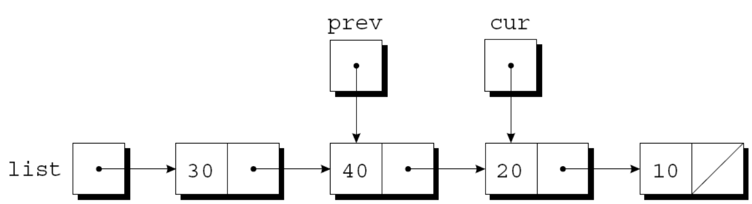
Deleting a Node from a Linked List



- The test `cur != NULL && cur->value != n` is again `true`, so `prev = cur, cur = cur->next` is executed once more



Deleting a Node from a Linked List



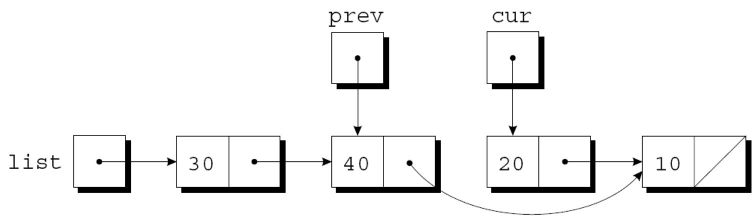
- Since `cur` now points to the node containing `20`, the condition `cur->value != n` is `false` and the loop terminates

Deleting a Node from a Linked List

- Next, we will perform the bypass required by step 2
- The statement

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

  makes the pointer in the previous node point to the node *after* the current node



Deleting a Node from a Linked List

- Step 3 is to release the memory occupied by the current node

```
free(cur);
```

Deleting a Node from a Linked List

- When given a list and an integer `n`, the function deletes *the first node containing* `n`
- Special cases:
  - If *no node* contains `n`, `delete_from_list` *should do nothing*
  - *Deleting the first node* in the list is a special case that requires a different bypass step
- In either case, the function returns a pointer to the list

Deleting a Node from a Linked List

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
  if (cur == NULL)
    return list;              /* n was not found */

  if (prev == NULL)
    list = cur->next;        /* n is in the first node */
  else
    prev->next = cur->next;  /* n is in some other node */

  free(cur);
  return list;
}
```

Ordered Lists

- When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ***ordered***
- Inserting a node into an ordered list is more difficult, because the node will not always be put at the beginning of the list
- However, searching the list is faster, as we can stop looking after reaching the point at which the desired node would have been located

Program: Maintaining a Parts Database (Revisited)

- The `inventory2.c` program is a modification of the parts database program of Chapter 16, with the database stored in a linked list this time
- Advantages of using a linked list
  - No need to put a limit on the size of the database
  - Database can easily be kept sorted by part number
- In the original program, the database was not sorted

Program: Maintaining a Parts Database (Revisited)

- The `part` structure will contain an additional member (a pointer to the next node)

```
struct part
{ int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};
```

- `inventory` will point to the first node in the list

```
struct part *inventory = NULL; /* empty list */
```


Program: Maintaining a Parts Database (Revisited)

- Most of the functions in the new program will closely resemble their counterparts in the original program
- Yet, `find_part` and `insert` will be more complex, since we want to keep the nodes in the inventory list sorted by part number

Program: Maintaining a Parts Database (Revisited)

- In the original program,
  - `find_part` returns an *index of the array element* that contains the desired part number
- In the new program,
  - `find_part` will return a *pointer to the node* that contains the desired part number
  - If it does not find the part number, `find_part` will return a *null pointer*

Program: Maintaining a Parts Database (Revisited)

- Since the list of parts is sorted, `find_part` can stop when it finds a node containing a part number that is greater than or equal to the desired part number
- `find_part`'s search loop

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
  ;
```

- When the loop terminates, we will need to test whether the part was found or not

```
if (p != NULL && number == p->number)
  return p;
else
  return NULL;
```

Program: Maintaining a Parts Database (Revisited)

- The original version of `insert` stores a new part in the next available array element
- The new version must determine where the new part belongs in the list and insert it there
- It will also check whether the part number is already present in the list
- A loop that accomplishes both tasks

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next)
  ;
```


Program: Maintaining a Parts Database (Revisited)

- Once the loop terminates, `insert` will check whether `cur` is not `NULL` and whether `new_node->number` equals `cur->number`
  - If both are `true`, the part number is already in the list
  - Otherwise, `insert` will insert a new node between the nodes pointed to by `prev` and `cur`
- This strategy works even if the new part number is larger than any in the list
- Like the original program, this version requires the `read_line` function of Chapter 16

inventory2.c

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"
#define NAME_LEN 25

struct part
{ int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```



```
/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;

  for (;;)
  { printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```

```
    switch (code)
    { case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```



```
/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            list. Returns a pointer to the node         *
 *            containing the part number; if the part     *
 *            number is not found, returns NULL.          *
 **********************************************************/
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  else
    return NULL;
}
```

```
/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         inventory list; the list remains sorted by     *
 *         part number. Prints an error message and       *
 *         returns prematurely if the part already exists *
 *         or space could not be allocated for the part.  *
 **********************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL)
  { printf("Database is full; can not add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);
```



```
 for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number)
  { printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 *                                                        *
 **********************************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL)
  { printf("Part name: %s\n", p->name);

    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}
```



```
/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part does not    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL)
  { printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Part numbers will appear in   *
 *        ascending order.                                *
 **********************************************************/
void print(void)
{
  struct part *p;
  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
```


Pointers to Pointers

- Chapter 13 introduced the idea of a *pointer to a pointer*
- When an argument to a function *is a pointer variable*, and we want the function to be able to *modify this pointer variable*, we need to use a ***pointer to a pointer***

Pointers to Pointers

- The `add_to_list` function (*inserting at the beginning of a list*)
  - is passed *a pointer to the first node* in a list
  - it returns *a pointer to the first node* in the updated list

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
  { printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```


Pointers to Pointers

- Modifying `add_to_list` so that it *attempts* to assign `new_node` to `list` instead of returning `new_node` *does not* work
- Example

```
add_to_list(first, 10);
```

- At the point of the call, `first` is copied into `list`
- If the function changes the value of `list`, making it point to the new node, `first` is not affected

Pointers to Pointers

- Letting `add_to_list` to modify `first` requires passing `add_to_list` a *pointer* to `first`

```
void add_to_list(struct node **list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
  { printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = *list;
  *list = new_node;
}
```


Pointers to Pointers

- When the new version of `add_to_list` is called, the first argument will be the address of `first`

```
add_to_list(&first, 10);
```

- Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`
- In particular, assigning `new_node` to `*list` will modify `first`