Linked Lists

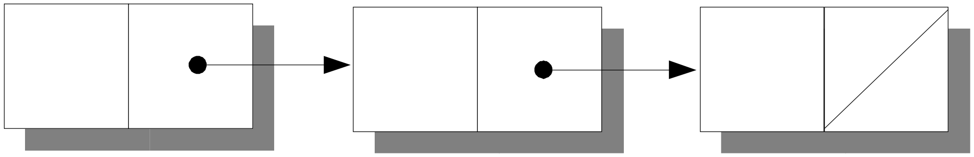
CS2023 Winter 2004

Outcomes: Linked Lists

- *C for Java Programmers*, Chapter 11 (11.4.9) and *C Programming - a Modern Approach*, Chapter 17 (17.5)
- After the conclusion of this section you should be able to
  - Write modules using linked lists
  - Begin creating other similar data structures, such as trees

Linked Lists

- Chain of structures (nodes) each containing pointer to next node in chain



- More flexible than array
  - easily insert/delete nodes
- but lose random access to elements
  - accessing node fast if node is close to the beginning, but slow if node is near the end of the list

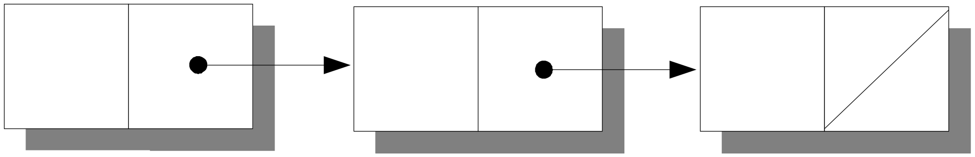
Declaring a Node Type

- Define **DataType** to improve maintainability of code:

```
typedef int DataType;
```

must use structure tag

```
typedef struct node {
   DataType value;
   struct node *next;
} NodeT, *NodeTP;
```



- The value of **next** will be **NULL** if there is no next element, otherwise it will be a structure representing the next element.

Declaring a Node Type

- Need variable that always points to first node in list (*C for Java Programmers* uses another structure to do this):

  ```
  NodeTP first = NULL;
  ```

- `first` initialized to `NULL` to indicate that list is initially empty

Creating Nodes

1. Allocate memory for the node

2. Store data in the node

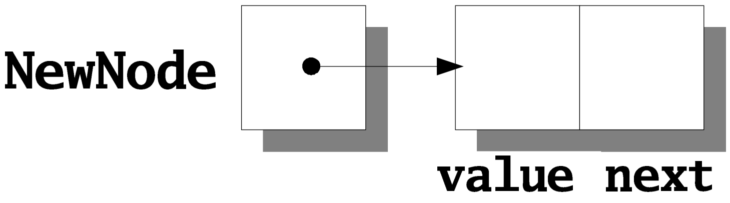
3. Insert the node into the list

1. Need variable to point to the node temporarily:

```
NodeTP newNode;
```

Allocate memory for the new node:

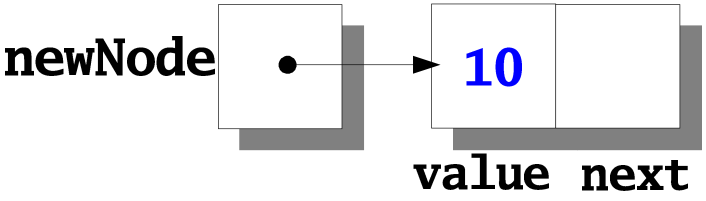
```
if ((newNode = malloc(sizeof(NodeT)))
      == NULL) error;
```



Creating Nodes

2. Store data in the **`value`** member of the new node:

```
newNode->value = 10;
```



3. Insert node into list

- inserting in at beginning of list is easiest

Inserting Node at Beginning of List

1. Modify the new node's **`next`** member to point to the node that was previously at beginning of list:

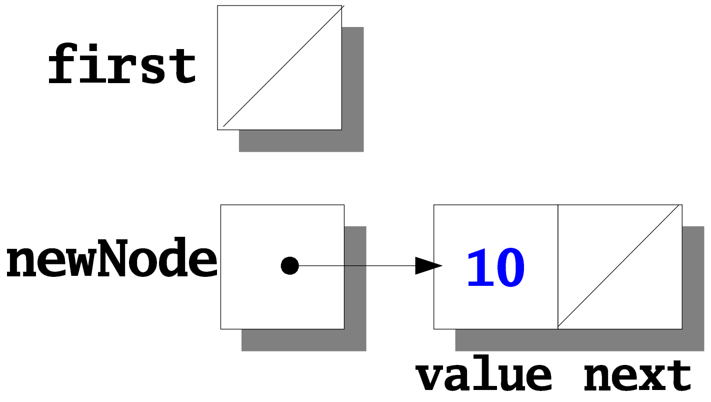
```
newNode->next = first;
```

2. Make **`first`** point to the new node:

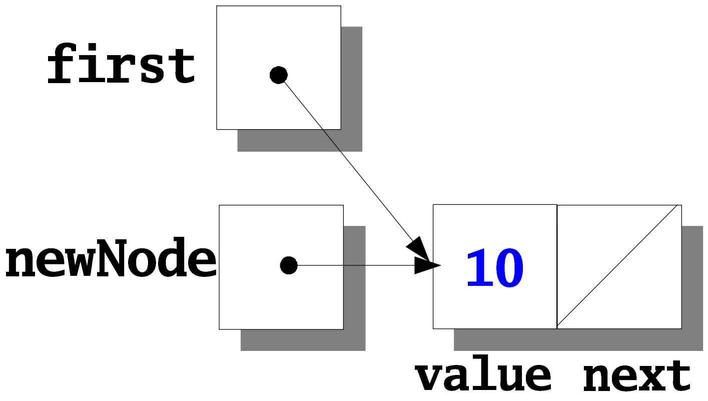
```
first = newNode;
```

Inserting Node at Beginning of List

1. `newNode->next = first;`



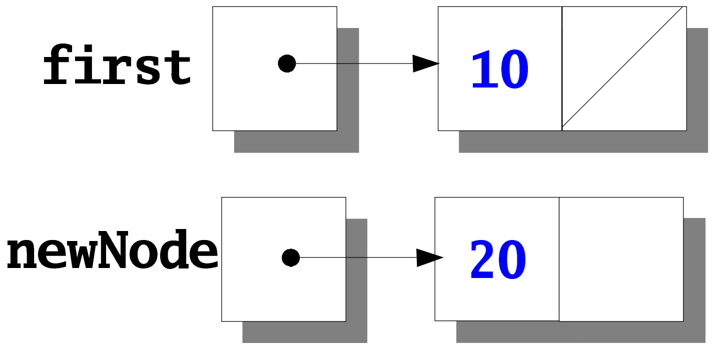
2. `first = newNode;`



Inserting Node at Beginning of List

Add another node:

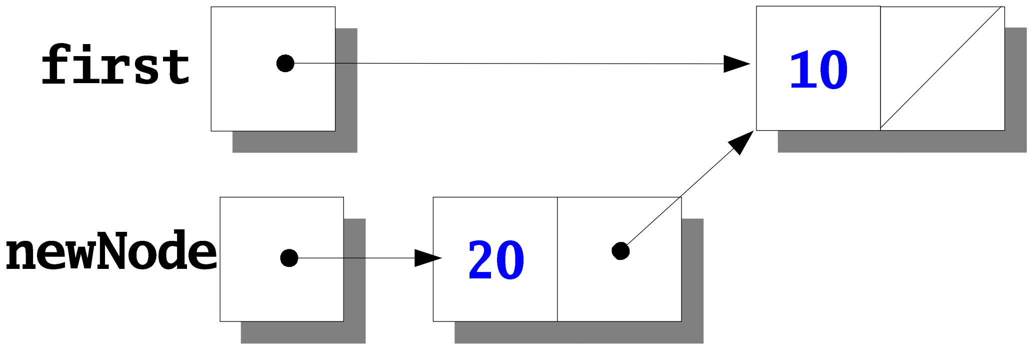
```
newNode = malloc(sizeof(NodeT));
newNode->value = 20;
```



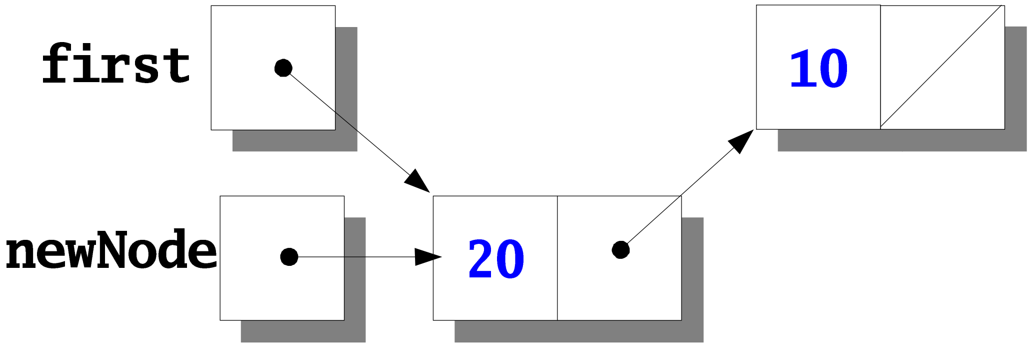
Inserting Node at Beginning of List

Insert new node at beginning of list

```
newNode->next = first;
```



```
first = newNode;
```



Inserting Node at Beginning of List

```
NodeTP insertFront(NodeTP first, DataType d)
  {
   NodeTP aux;

   if((aux = malloc(sizeof(NodeT))) == NULL)
      exit(EXIT_FAILURE);

   aux->value = d;
   aux->next = first;
   return aux;
}
```

Inserting Node at Beginning of List

```
first = insertFront(first, 10);
first = insertFront(first, 20);
```

How to get **insertFront** to update **first** directly, rather than return a new value of **first**?

```
void insertFront(NodeTP first, DataType d) {
   NodeTP aux;
   if((aux = malloc(sizeof(NodeT))) == NULL)
      exit(EXIT_FAILURE);
   aux->value = d;
   aux->next = first;
   first = aux;
}
```

Inserting Node at Beginning of List

- Recall that pointers, like other arguments, are passed by value.
- Need to pass a *pointer* to **first**

```
void insertFront(NodeTP *firstp, DataType d) {
   NodeTP aux;
   if((aux = malloc(sizeof(NodeT))) == NULL)
      exit(EXIT_FAILURE);
   aux->value = d;
   aux->next = *firstp;
   *firstp = aux;
}
```

Call: `insertFront(&first, 10);`

Searching a Linked List

Idiom for traversal of a list:

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

- Search a list for data n (of type **DataType**).
  - return pointer to node containing n,
  - otherwise return null pointer:

```
NodeTP searchList(NodeTP first, DataType n) {
  NodeTP p;
  for (p = first; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```

Searching a Linked List

Since `first` passed by value, can modify it:

```
NodeTP searchList(NodeTP first, DataType n) {
  for (; first != NULL; first = first->next)
    if (first->value == n)
      return first;
  return NULL;
}
```

Deleting a Node from a Linked List

1. Locate the node to be deleted

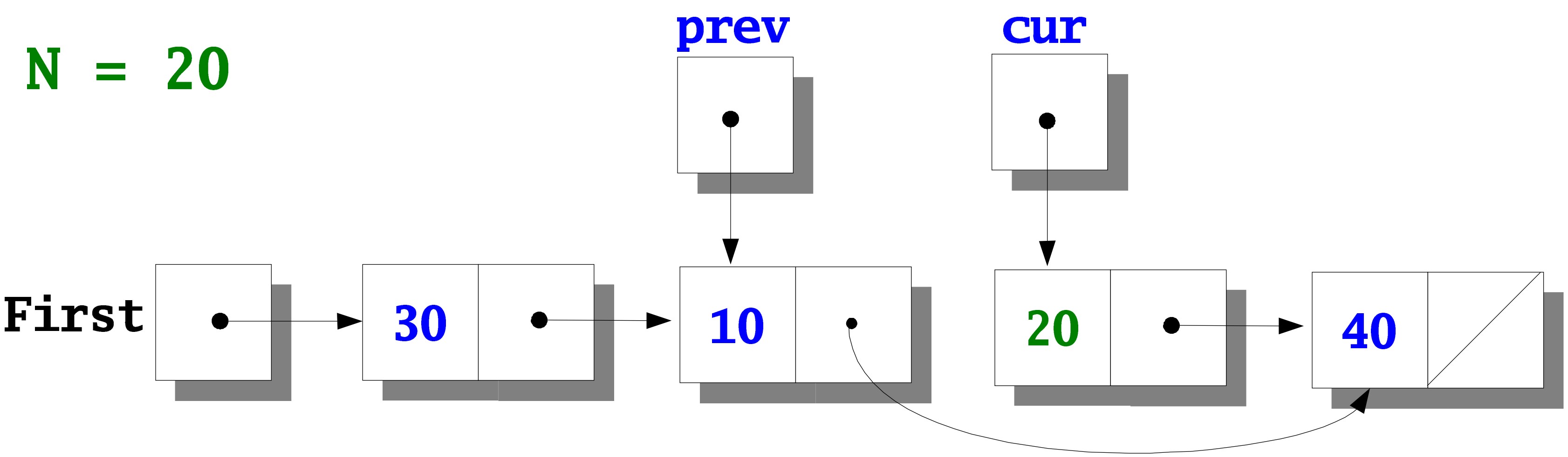
2. Alter previous node so that it bypasses deleted node

3. Call `free` to reclaim space occupied by deleted node

1. Need pointer to previous node as well as current node:

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

Deleting a Node from a Linked List



```
2.    prev->next = cur->next;
3.    free(cur);
```

```
NodeTP deleteFromList(NodeTP first, DataType n) {
  NodeTP cur, prev;

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

     if (cur == NULL)
       return first;    /* n was not found */
     if (prev == NULL) /* n in first node */
       first = first->next;
     else              /* n in some other node */
       prev->next = cur-> next;
     free(cur);
     return first;
   }
```

Deleting an Entire Linked List

1. Remove all nodes one at a time from the beginning of the list

2. Set pointer to first element to null.

```
void destructList(NodeTP *firstp)
{
   while(deleteFirst(firstp))
   ;
   *firstp = NULL;
}
```

Call: `destructList(&first);`

Deleting the First Node

```
int deleteFirst(NodeTP *firstp)
{
   NodeTP aux = *firstp;
   if(aux == NULL)    /* empty list */
     return 0;
   *firstp = aux->next;
   free(aux);

   return 1;
}
```

Ordered Lists

- Searching is faster than unordered linked list
    - can stop after reaching point where desired node would have been located
- Inserting a node is more difficult
- Illustrate using a parts database
    - see *C Programming: a Modern Approach*, p. 379

Parts Database

```
typedef struct part {
   int number;
   char name[NAME_LEN+1];
   int onHand;
   struct part *next;
} PartT, *PartTP;

PartTP inventory = NULL;
```

(here **`inventory`** is a global variable)

Find Part in Inventory

Look up a part number in inventory and return pointer to node containing part number. If not found, return **NULL**

```
PartTP findPart(int number);
```

Find Part in Inventory

Look up a part number in inventory and return pointer to node containing part number. If not found, return **NULL**

```
PartTP findPart(int number)
{
  PartTP p;
  for (p = inventory;
       p != NULL && number > p->number ;
       p = p-> next)
  ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}
```

```
void search(void);
```

- Prompt user for part number then look it up in inventory. If part exists, print name and quantity; if not, print an error message

Prompt user for part number then look it up in inventory. If part exists, print name and quantity; if not, print an error message

```
void search(void)
{
  int number;
  PartTP p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = findPart(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity: %d\n", p->onHand);
  } else
    printf("Part not found\n");
}
```

```
void update(void);
```

- Prompt user for part number. Print error message if part doesn't exist; otherwise prompts user to enter change in quantity and updates the inventory

Prompt user for part number. Print error message if part doesn't exist; otherwise prompts user to enter change in quantity and updates the inventory

```
void update(void)
{
  int number, change;
  PartTP p;

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

Insert Part in Inventory

Prompts user for information about a new part and then inserts it into the inventory list. The list remains sorted by part number

```
void insert(void);
```

Insert Part in Inventory

Prompts user for information about a new part and then inserts it into the inventory list. The list remains sorted by part number

```
void insert(void)
{
  PartTP cur, prev, newNode;

  newNode = malloc(sizeof(PartT));
  if(newNode == NULL) {
    printf("Database if full\n");
    return;
  }
```

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

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

```
printf("Enter part name: ");
readLine(newNode->name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &newNode->onHand);

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