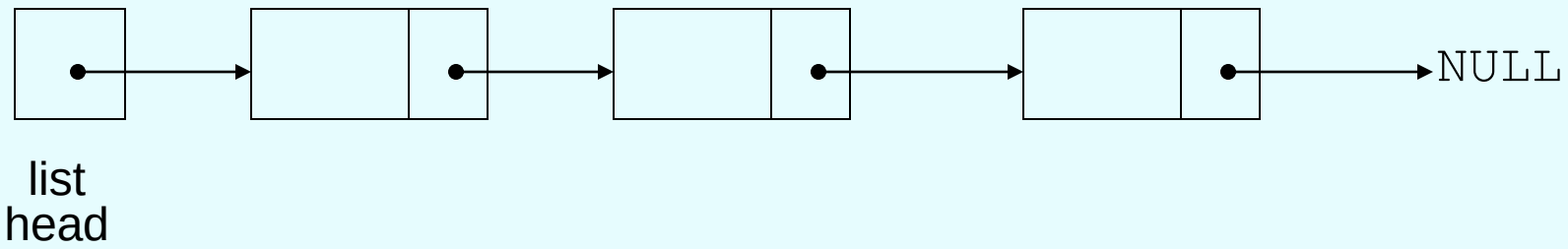
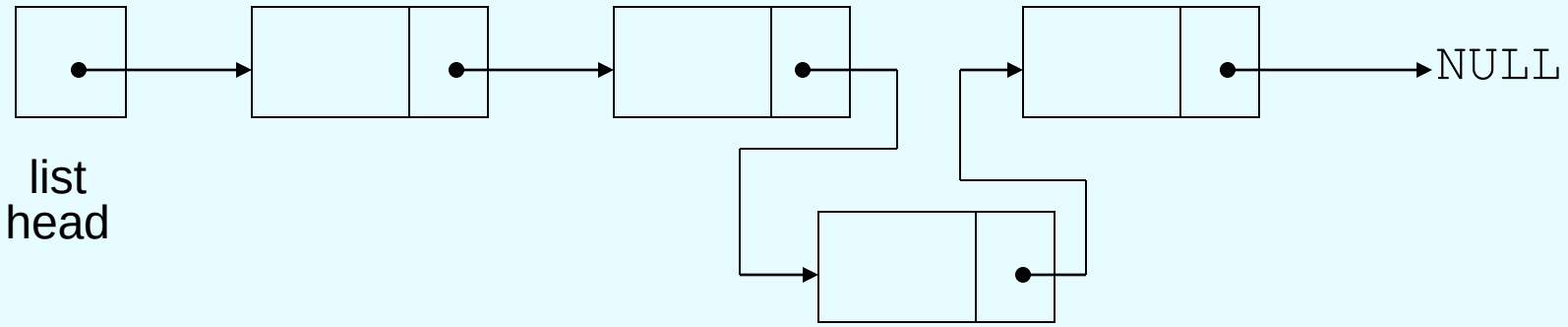
Introduction to the Linked List ADT

- Linked list: set of data structures (nodes) that contain references to other data structures



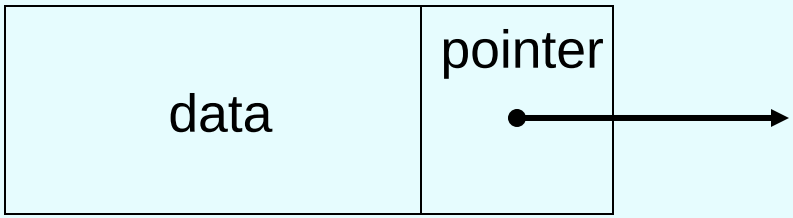
Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



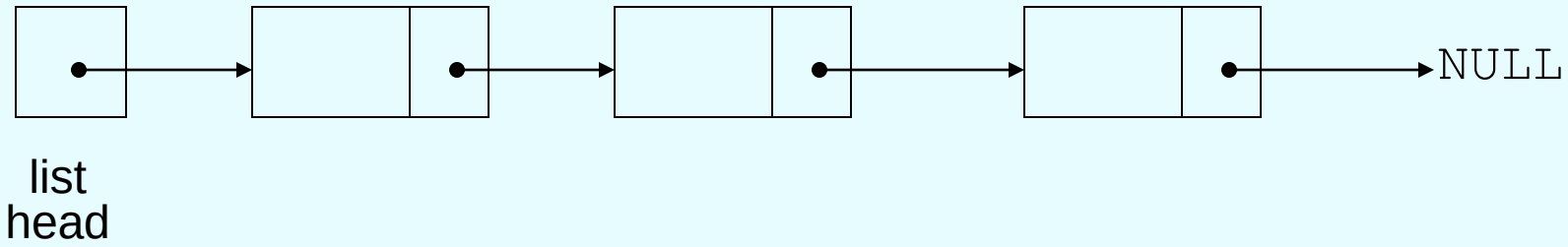
Node Organization

- A node contains:
  - data: one or more data fields – may be organized as structure, object, etc.
  - a pointer that can point to another node



Linked List Organization

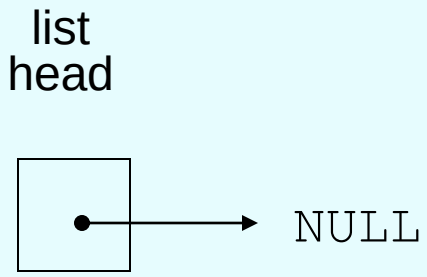
- Linked list contains 0 or more nodes:



- Has a list head to point to first node
- Last node points to `NULL`

Empty List

- If a list currently contains 0 nodes, it is the empty list
- In this case the list head points to `NULL`



Declaring a Node

- Declare a node:

```
struct ListNode
{
      int data;
      ListNode *next;
};
```

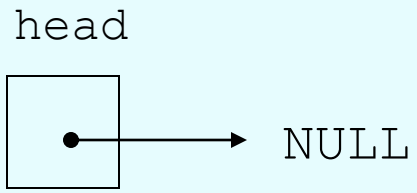
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
ListNode *head = NULL;
```

- Head pointer initialized to `NULL` to indicate an empty list



NULL Pointer

- Is used to indicate end-of-list
- NULL is just the 0 address, prefer using 0
- Should always be tested for before using a pointer:

```
ListNode *p;
while (p != 0) ...
```

- Can also test the pointer itself:

```
while (!p) ... // same meaning
               // as above
```

Linked List Operations

- Basic operations:
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

Contents of `NumberList.h`

```
 1  // Specification file for the NumberList class
 2  #ifndef NUMBERLIST_H
 3  #define NUMBERLIST_H
 4
 5  class NumberList
 6  {
 7  private:
 8     // Declare a structure for the list
 9     struct ListNode
10     {
11        double value;           // The value in this node
12        struct ListNode *next;  // To point to the next node
13     };
14
15     ListNode *head;            // List head pointer
16
```

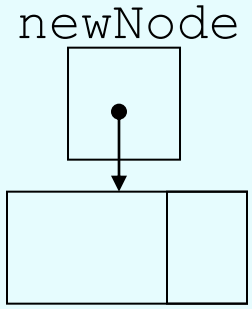
Contents of **NumberList.h** (Continued)

```
17  public:
18     // Constructor
19     NumberList()
20        { head = 0; }
21
22     // Destructor
23     ~NumberList();
24
25     // Linked list operations
26     void appendNode(double);
27     void insertNode(double);
28     void deleteNode(double);
29     void displayList() const;
30  };
31  #endif
```

Create a New Node

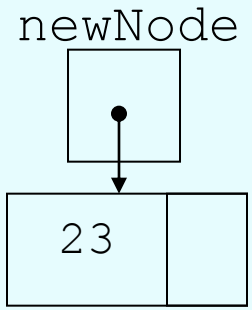
- Allocate memory for the new node:

```
newNode = new ListNode;
```



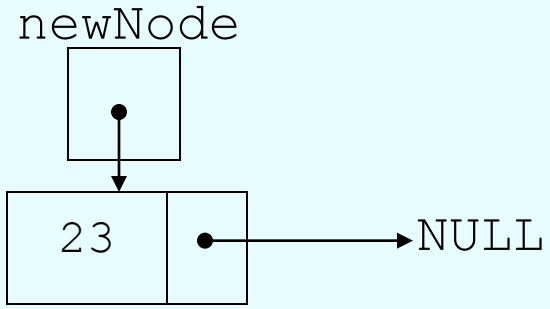
- Initialize the contents of the node:

```
newNode->value = num;
```



- Set the pointer field to `NULL`:

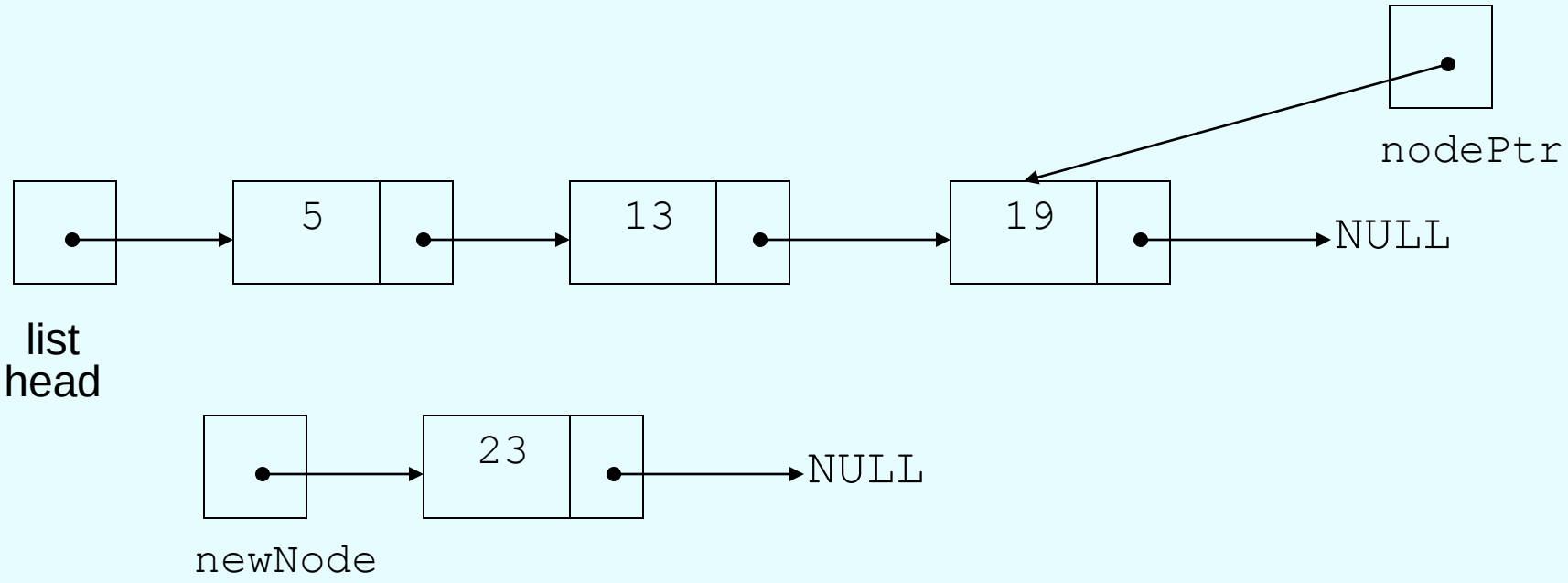
```
newNode->next = 0;
```



Appending a Node

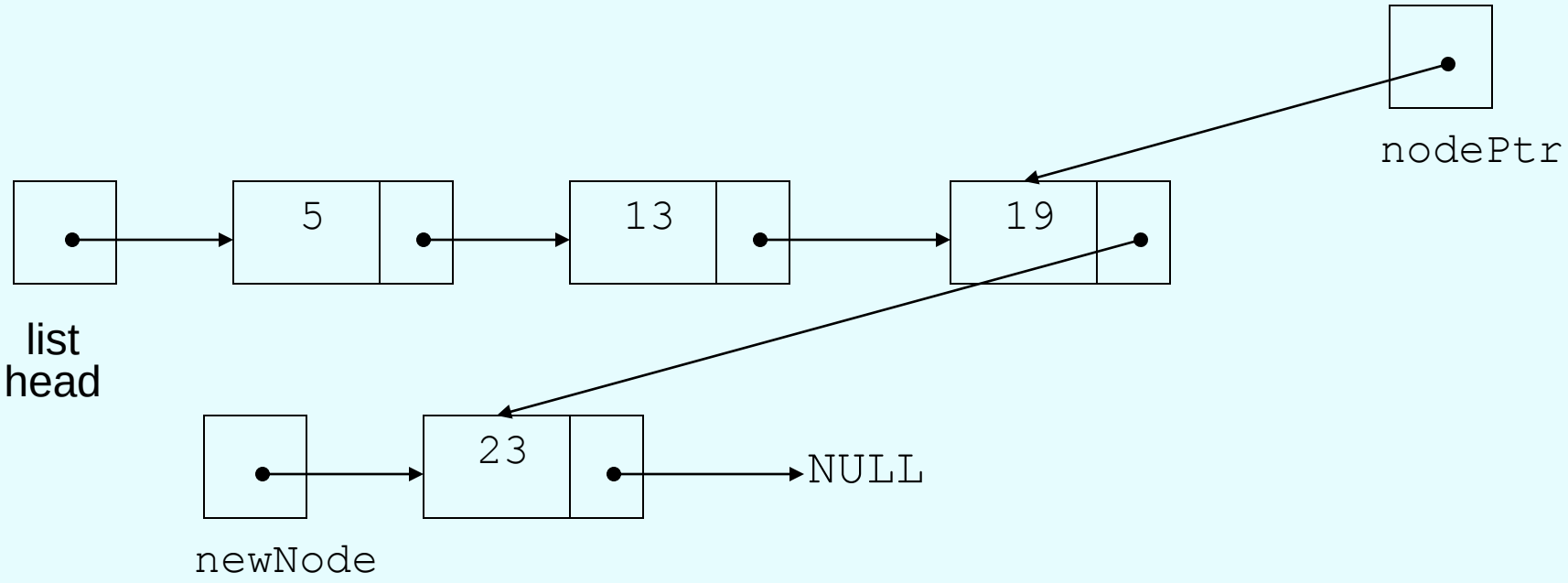
- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



New node created, end of list located

Appending a Node



New node added to end of list

C++ code for Appending a Node

```
11  void NumberList::appendNode(double num)
12  {
13     ListNode *newNode;  // To point to a new node
14     ListNode *nodePtr;  // To move through the list
15
16     // Allocate a new node and store num there.
17     newNode = new ListNode;
18     newNode->value = num;
19     newNode->next = NULL;
20
21     // If there are no nodes in the list
22     // make newNode the first node.
23     if (!head)
```

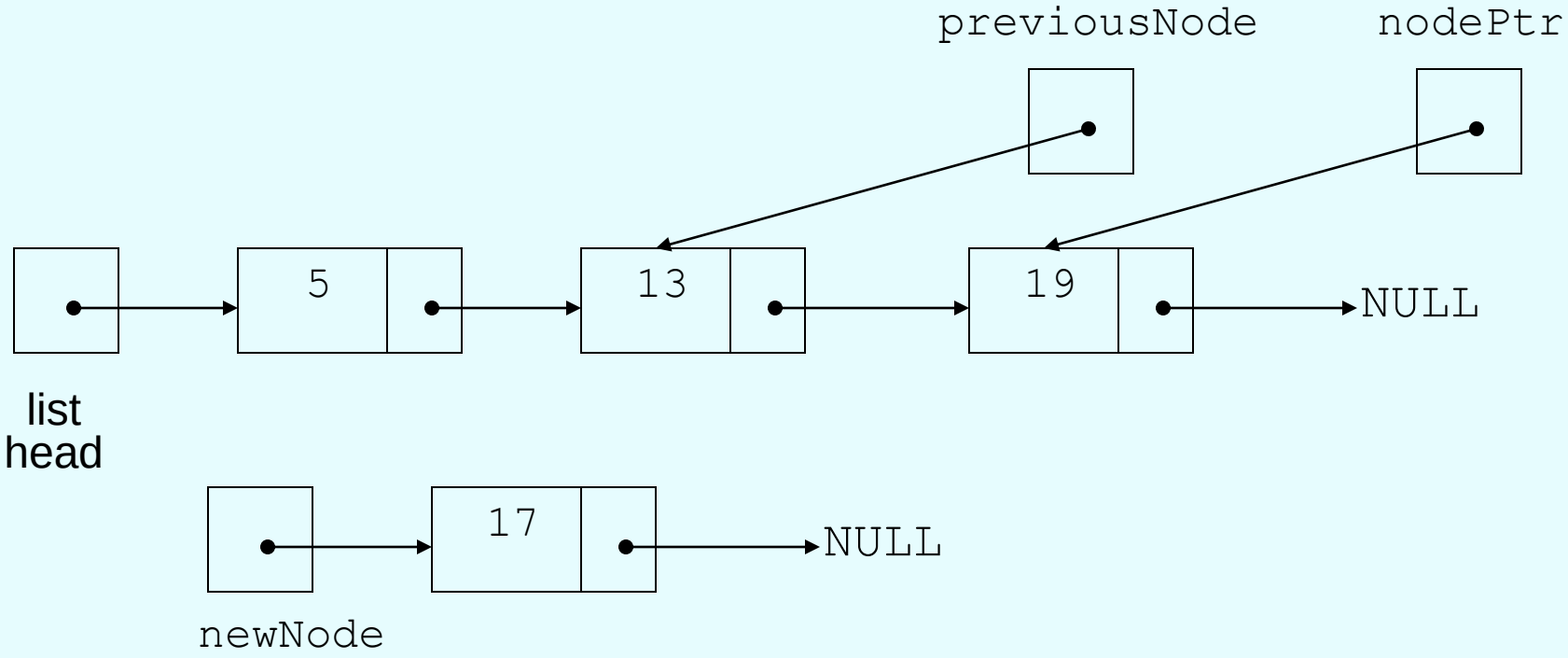
C++ code for Appending a Node (Continued)

```
24          head = newNode;
25       else  // Otherwise, insert newNode at end.
26       {
27          // Initialize nodePtr to head of list.
28          nodePtr = head;
29
30          // Find the last node in the list.
31          while (nodePtr->next)
32             nodePtr = nodePtr->next;
33
34          // Insert newNode as the last node.
35          nodePtr->next = newNode;
36       }
37    }
```

Inserting a Node into a Linked List

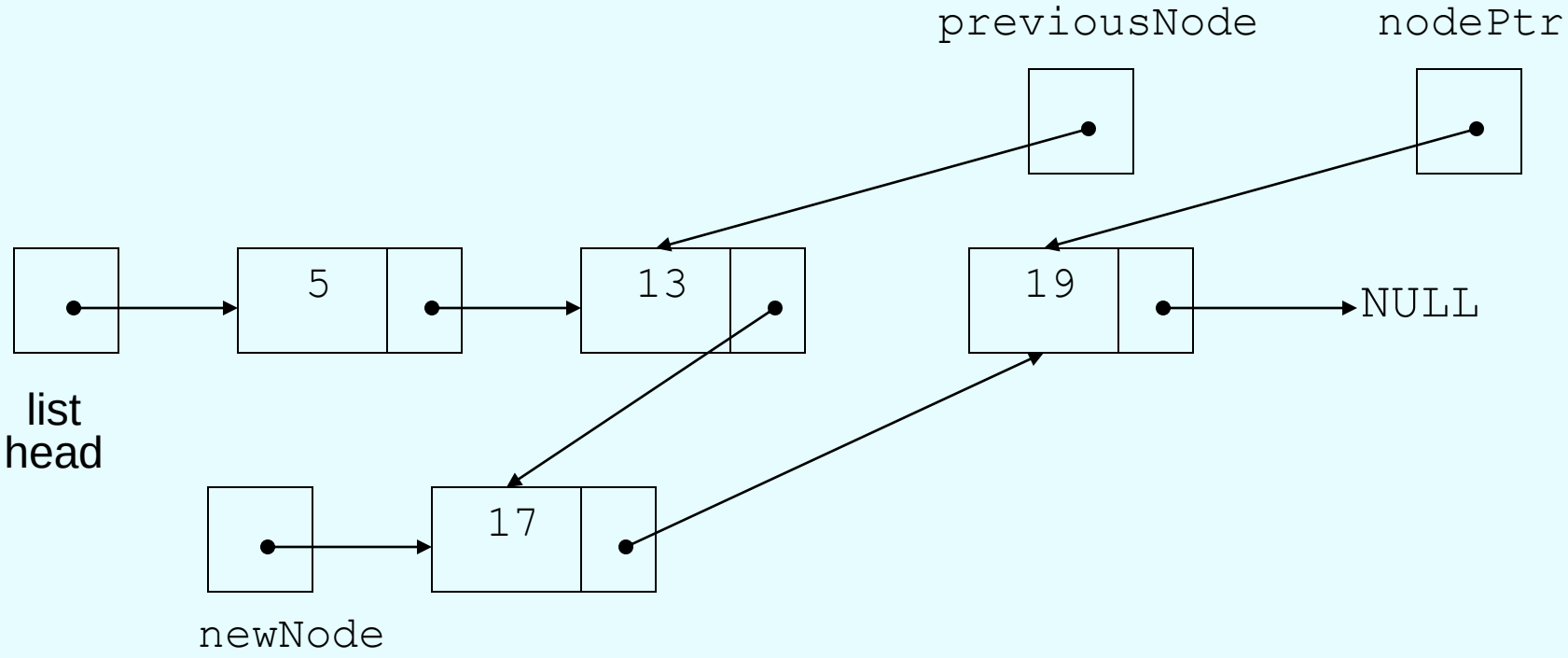
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

Inserting a Node into a Linked List



New node created, correct position located

Inserting a Node into a Linked List

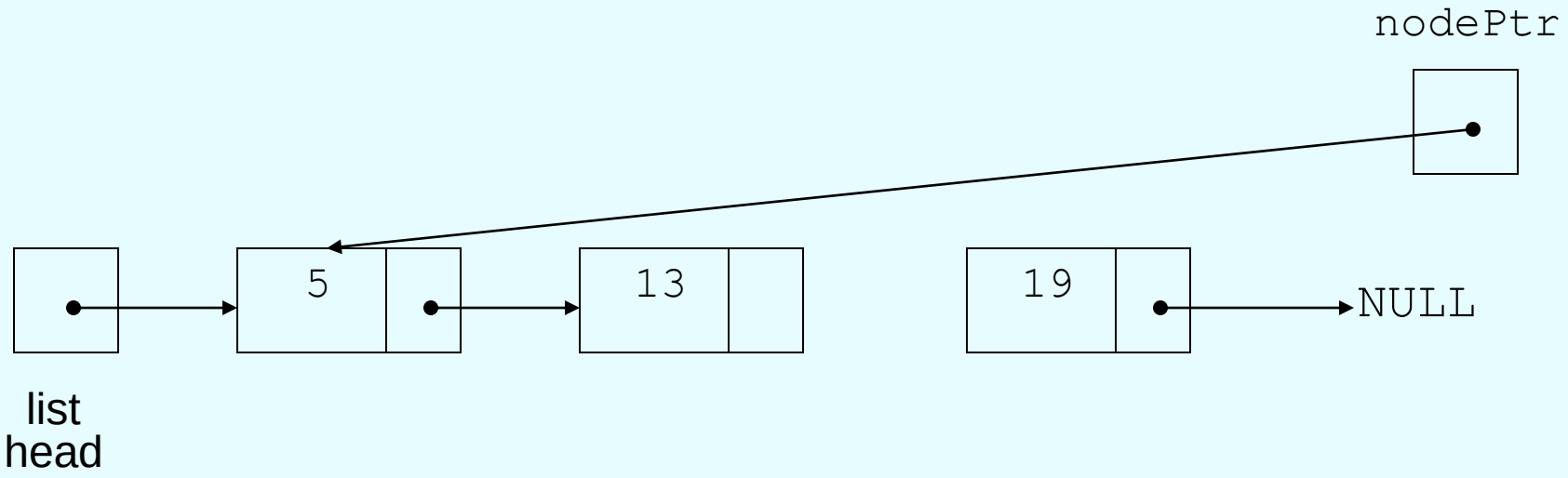


New node inserted in order in the linked list

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not `NULL`
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

Traversing a Linked List

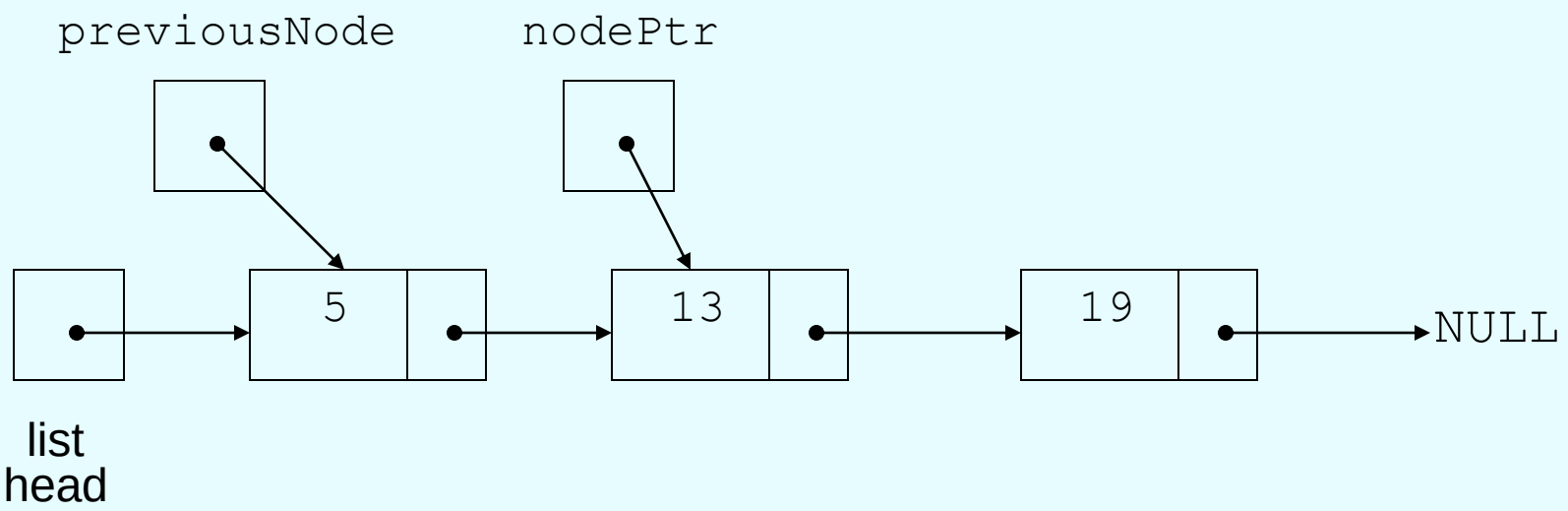


`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to `NULL`, and the list traversal stops

Deleting a Node

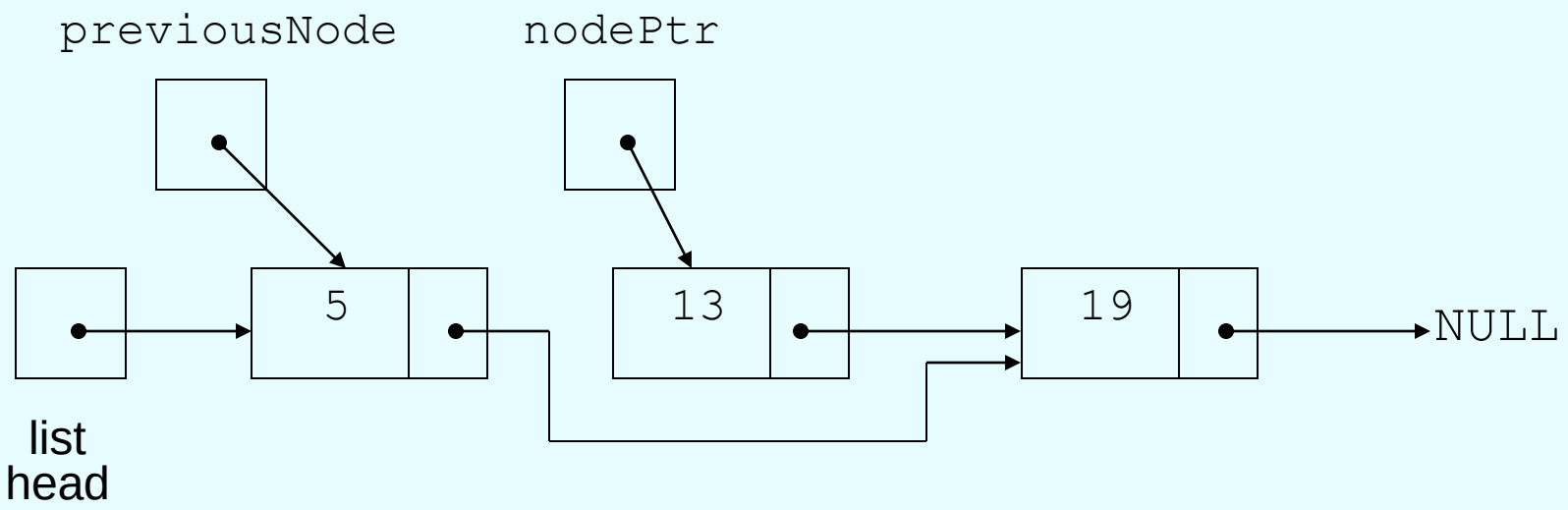
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node



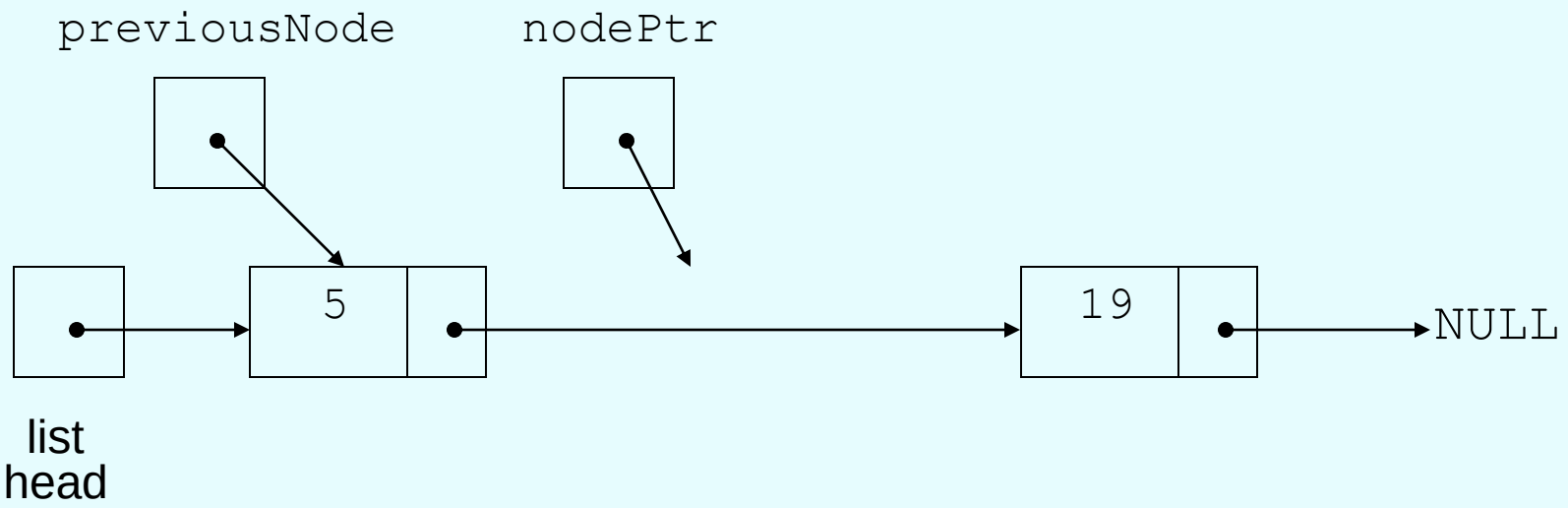
Locating the node containing `13`

Deleting a Node



Adjusting pointer around the node to be deleted

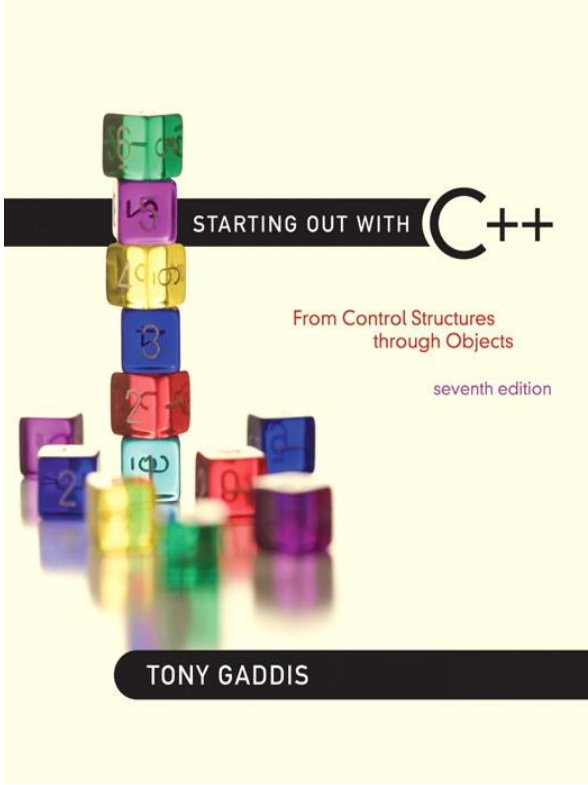
Deleting a Node



Linked list after deleting the node containing `13`

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `NULL`



17.5

The STL `list` Container

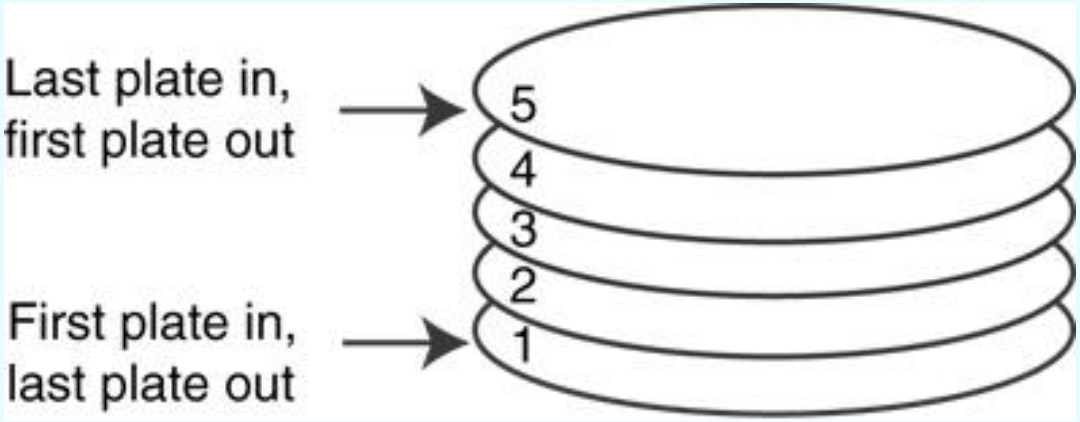
The STL `list` Container

- Template for a doubly linked list
- Member functions for
  - locating beginning, end of list: `front, back, end`
  - adding elements to the list: `insert, merge, push_back, push_front`
  - removing elements from the list: `erase, pop_back, pop_front, unique`
- See Table 17-1 for a list of member functions

Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

A LIFO Structure

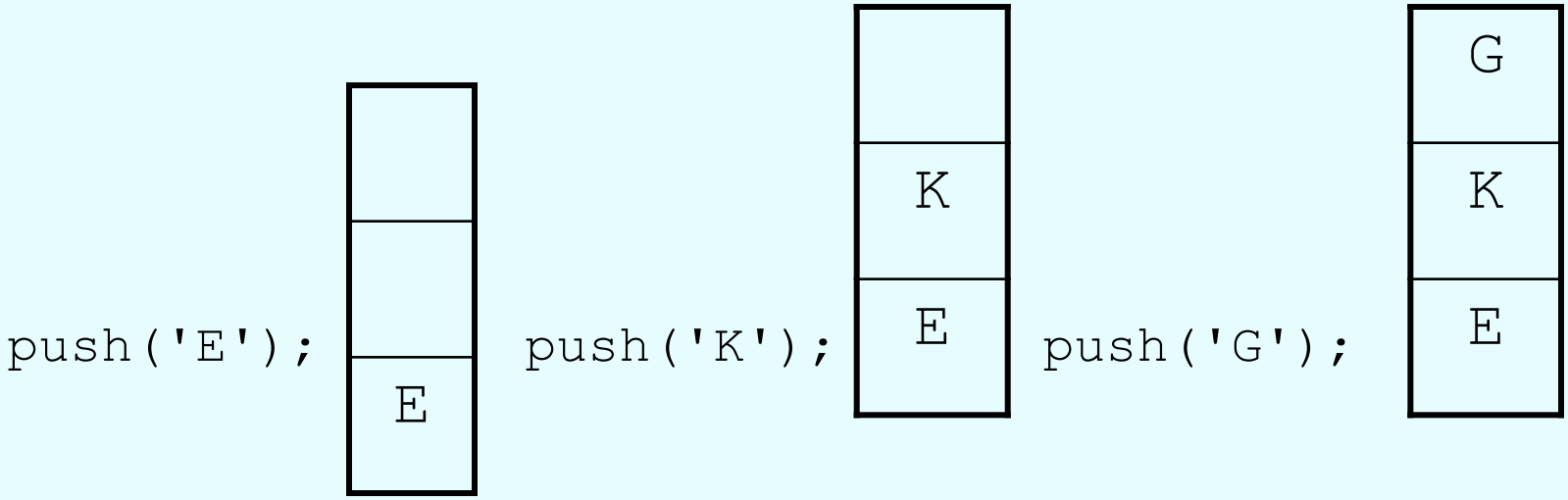


Stack Operations and Functions

- Operations:
  - push: add a value onto the top of the stack
  - pop: remove a value from the top of the stack

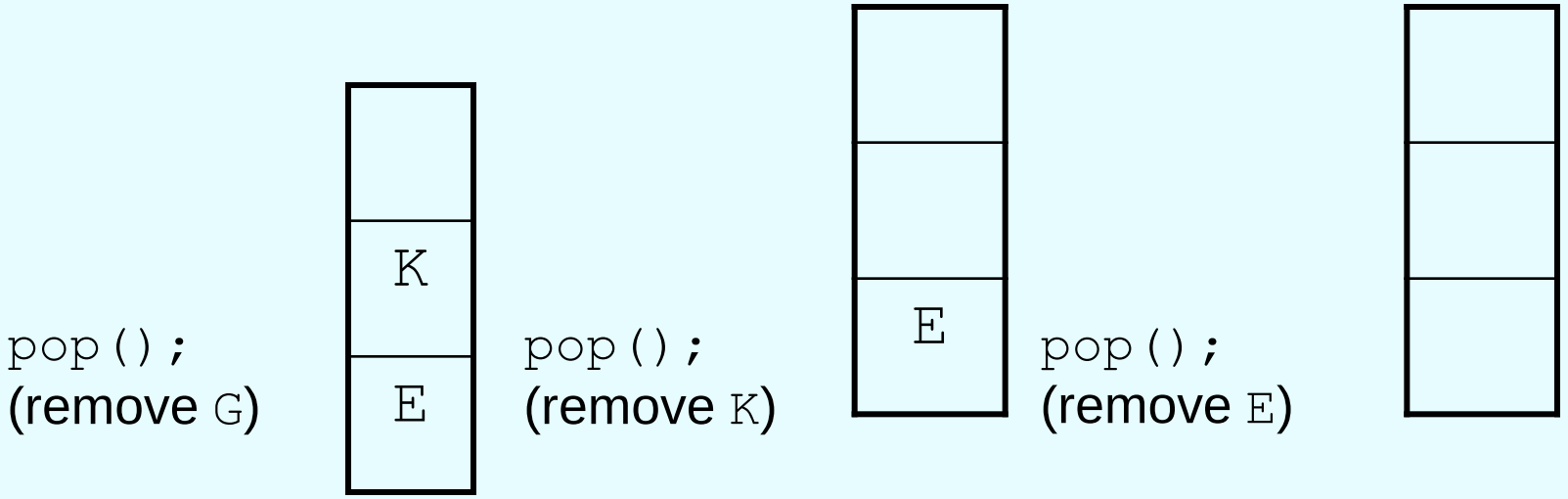
Stack Operations - Example

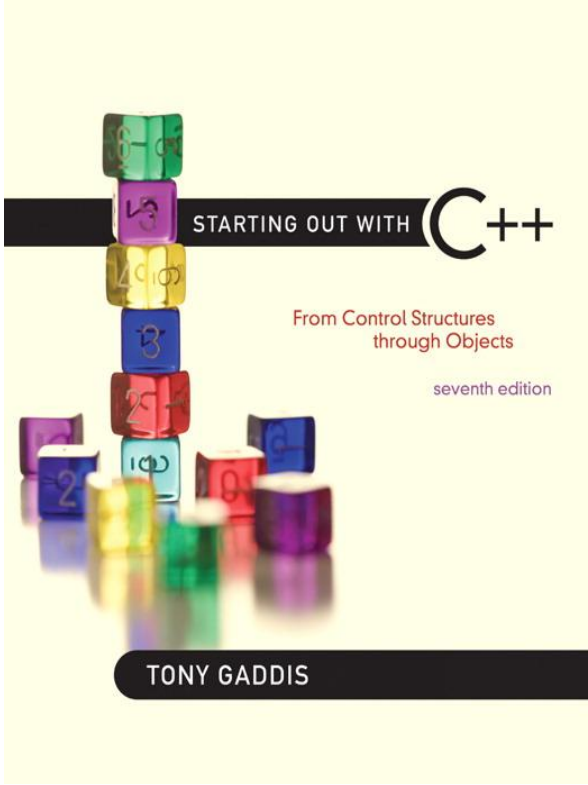
- A stack that can hold `char` values:



Stack Operations - Example

- A stack that can hold `char` values:





18.2

Dynamic Stacks

Dynamic Stacks

- Grow and shrink as necessary
- Can't ever be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can program their own routines to implement stack functions

- Can also use the implementation of stack available in the STL

The STL `stack` container

- Stack template can be implemented as a `vector`, a linked list, or a `deque`
- Implements `push`, `pop`, and `empty` member functions
- Implements other member functions:
  - `size`: number of elements on the stack
  - `top`: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `char`s, named `cstack`, implemented using a `vector`:

```
stack< char, vector<char> > cstack;
```

- implemented using a list:

```
stack< char, list<char> > cstack;
```

- implemented using a `deque`:

```
stack< char > cstack;
```

- Spaces are required between consecutive `>>`, `<<` symbols

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

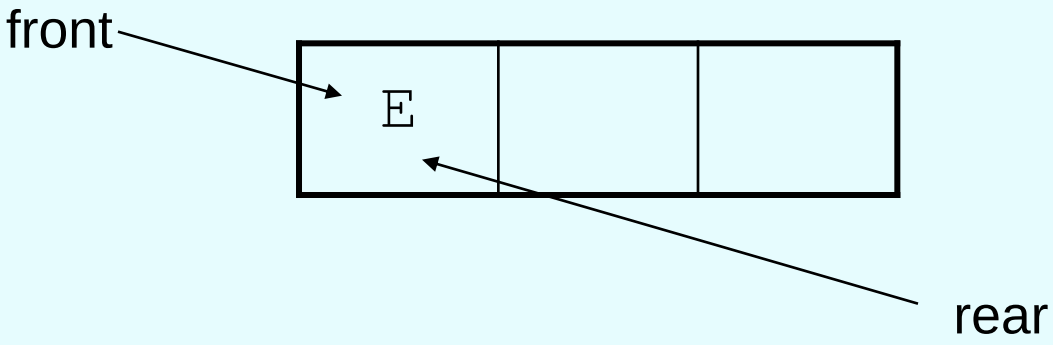
Queue Locations and Operations

- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Queue Operations - Example

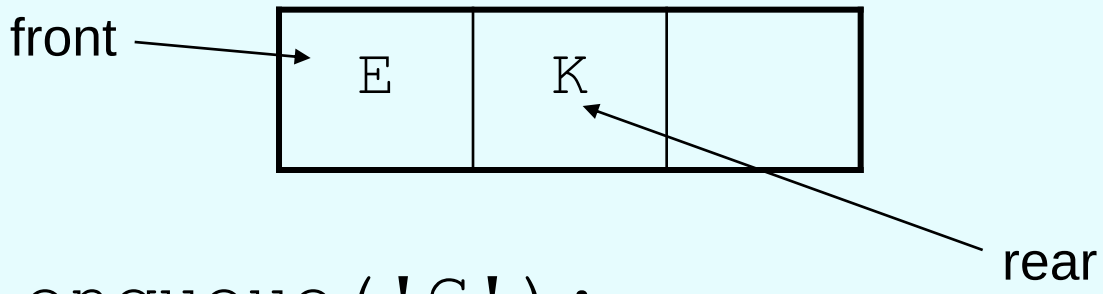
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

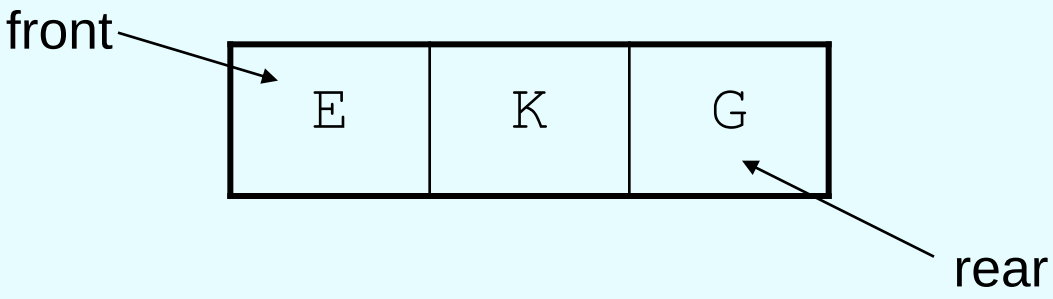


Queue Operations - Example

- `enqueue('K');`

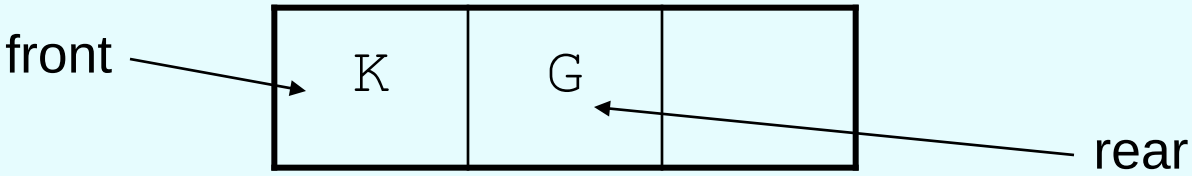


- `enqueue('G');`

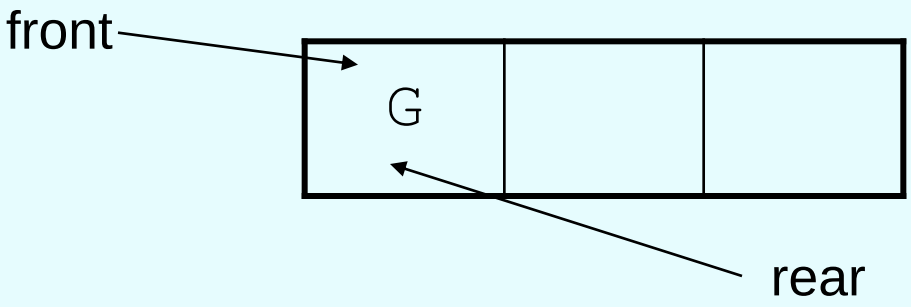


Queue Operations - Example

- `dequeue(); // remove E`

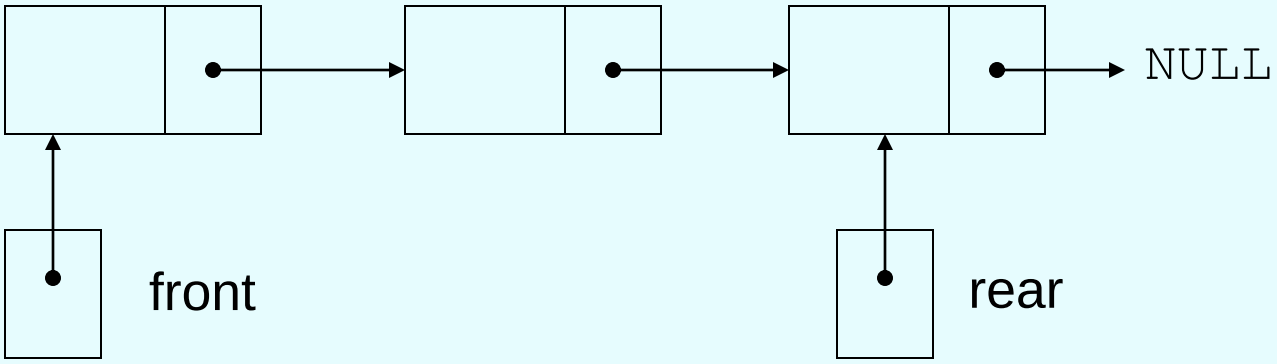


- `dequeue(); // remove K`



Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can program their own routines to implement queue operations
- Can also use the implementation of queue and dequeue available in the STL

The STL `deque` and `queue` Containers

- `deque`: a double-ended queue. Has member functions to enqueue (`push_back`) and dequeue (`pop_front`)
- `queue`: container ADT that can be used to provide queue as a `vector`, list, or `deque`. Has member functions to enque (`push`) and dequeue (`pop`)

Defining a `queue`

- Defining a queue of `char`s, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a `list`:

```
queue< char, list<char> > cQueue;
```

- Spaces are required between consecutive `>>`, `<<` symbols