# Chapter 4 Linked Stacks and Queues

信息科学与技术学院　　　黄方军


data_structures@163.com

东校区实验中心B502

# 4.1.1 Introduction and Survey

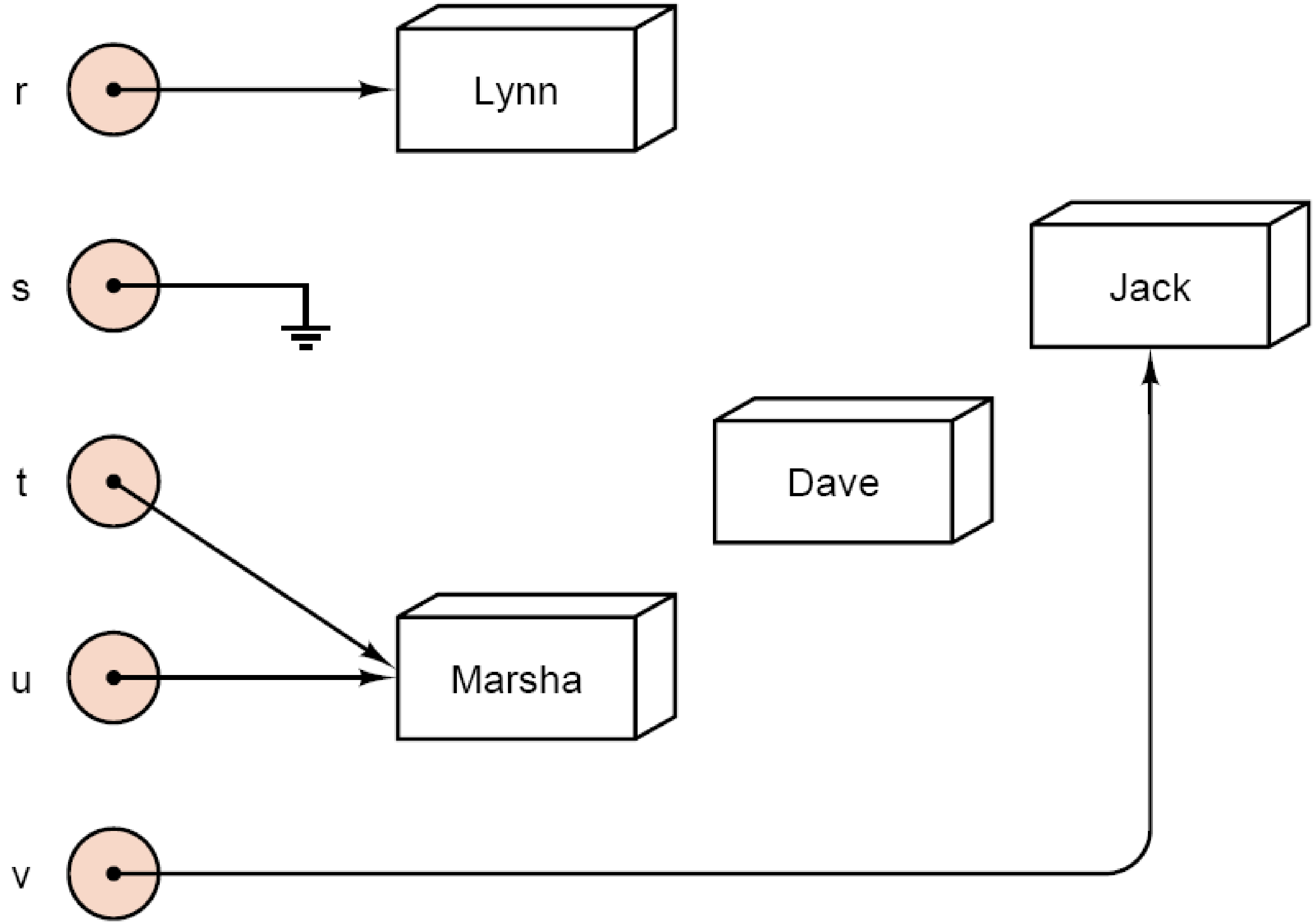



Figure 4.1. Pointers to objects

# 4.1.1 Introduction and Survey

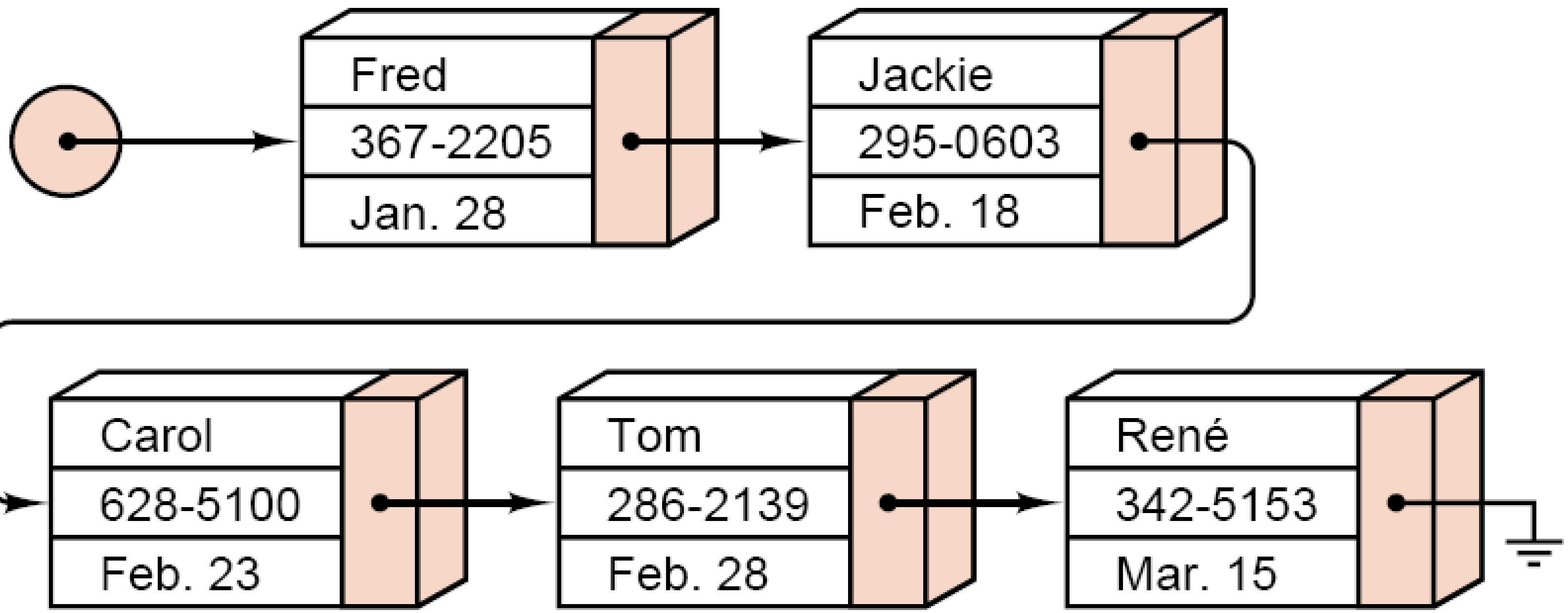



Figure 4.2. A linked list

# 4.1.2 Pointers and Dynamic Memory in C++

1. Automatic and Dynamic Objects;
2. C++ Notation;

   Item *item_ptr;

3. Creating and Destroying Dynamic Objects;

   p = new Item;

   p = new (nothrow) Item;

   delete p;

# 4.1.2 Pointers and Dynamic Memory in C++

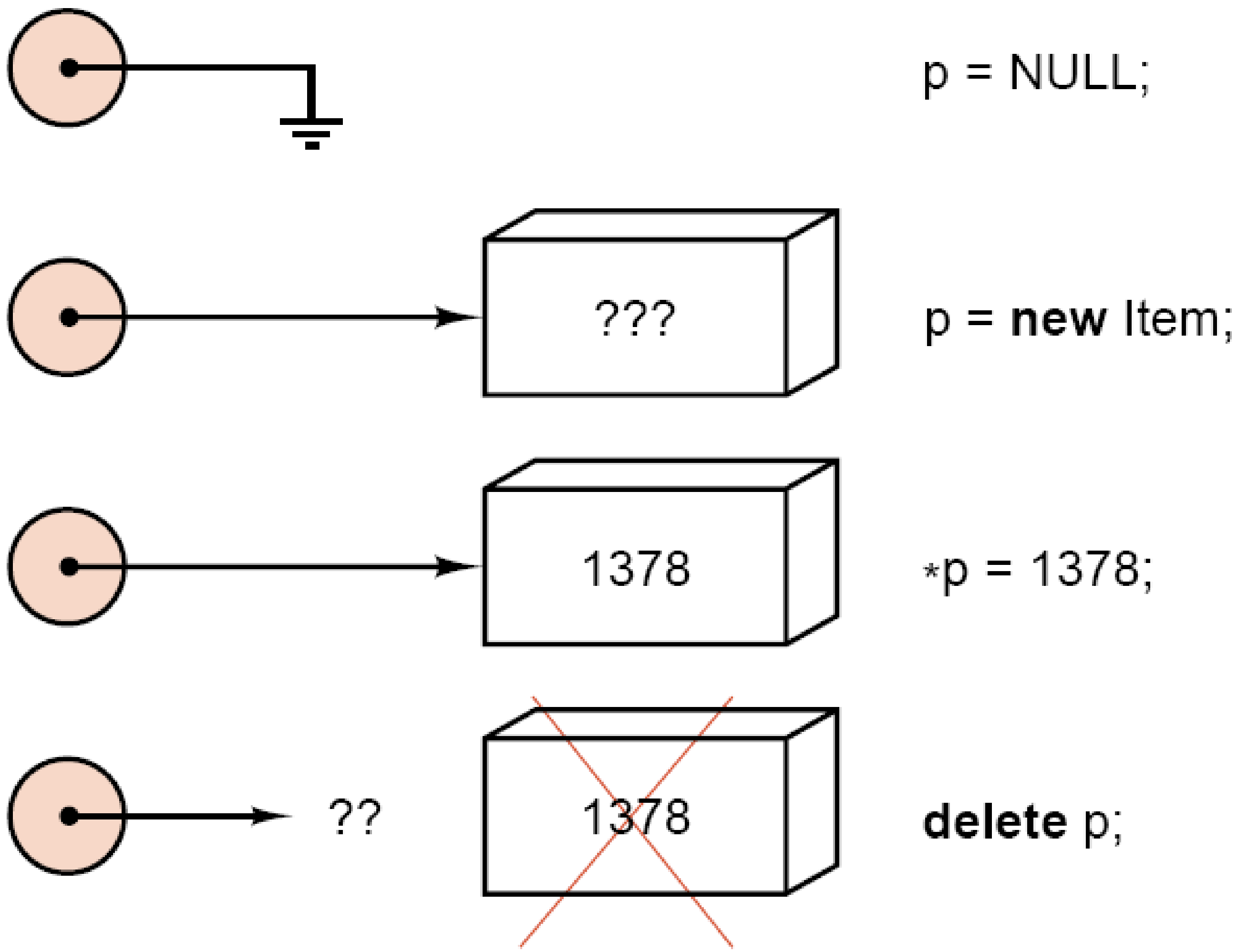



Figure 4.3. Creating and disposing of dynamic objects

# 4.1.2 Pointers and Dynamic Memory in C++

## 4. Following the Pointers

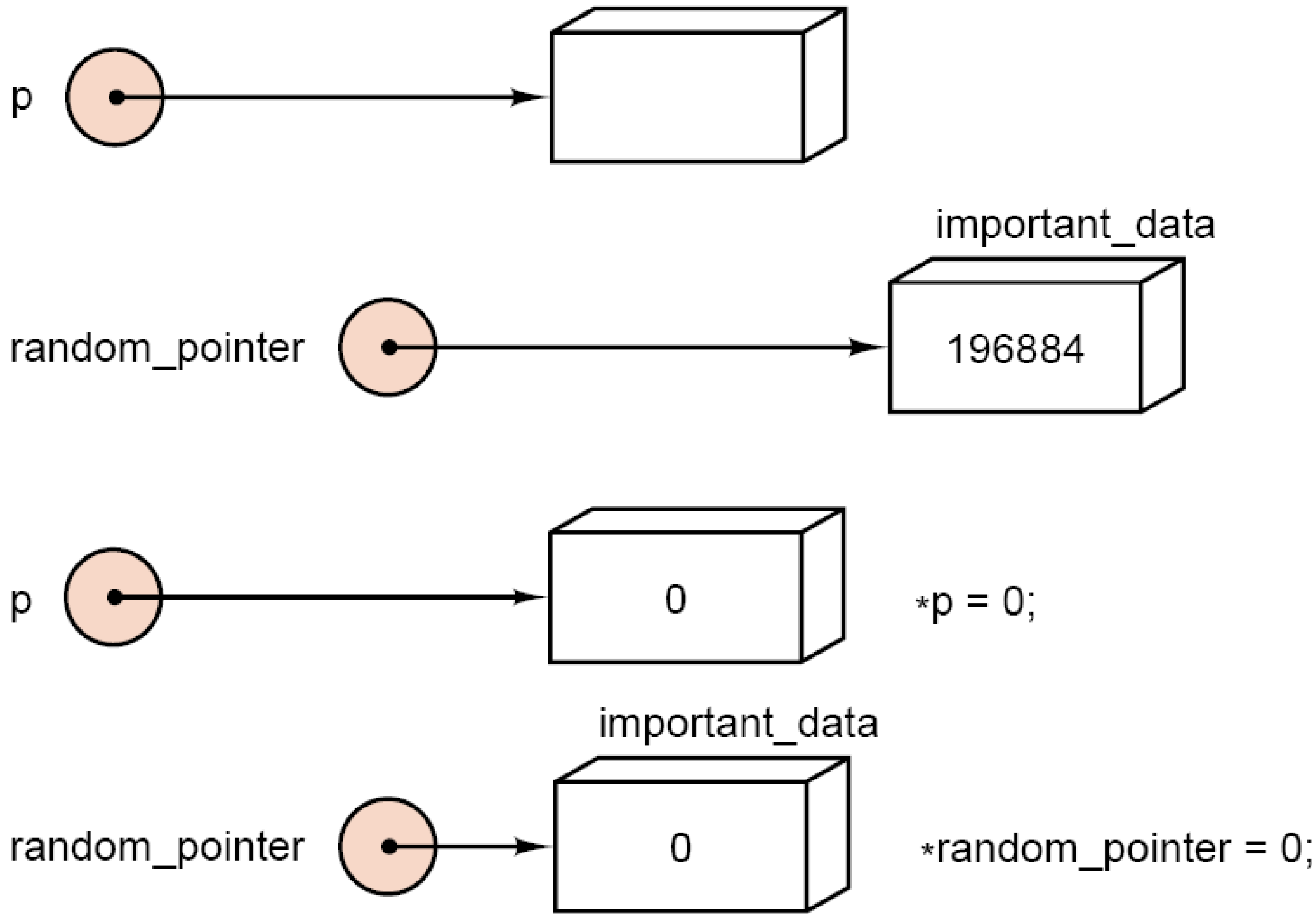



Figure 4.4. Modifying dereferenced pointers

## 5. NULL Pointers

This situation can be established by the assignment

```
p = NULL;
```

and subsequently checked by a condition such as

```
if (p != NULL) ....
```

In diagrams we reserve the electrical ground symbol

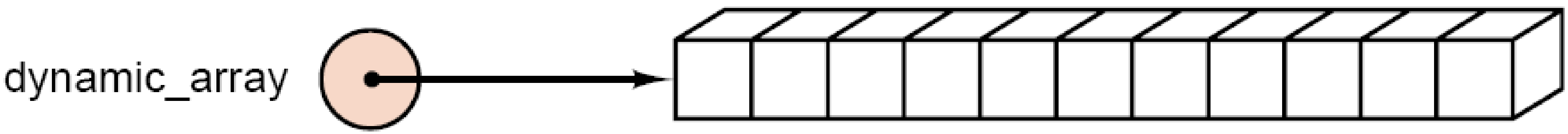
### 6. Dynamically allocated arrays

```
int size, *dynamic_array, i;
cout << "Enter an array size: " << flush;
cin >> size;
dynamic_array = new int[size];
for (i = 0; i < size; i++) dynamic_array[i] = i;
```

# 4.1.2 Pointers and Dynamic Memory in C++

## 6. Dynamically Allocated Arrays

dynamic_array = **new int** [size];




**for** (i=0; i<size; i++) dynamic_array[i] = i;

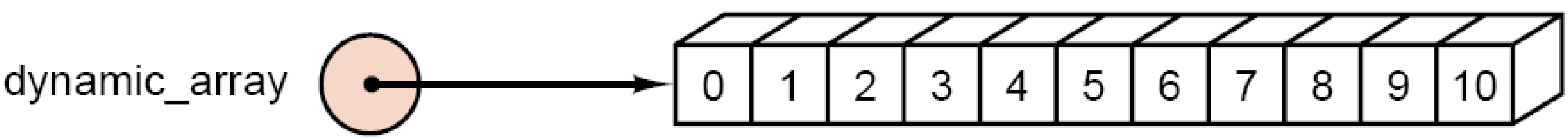



Figure 4.5. Dynamic arrays and pointers

**delete** [ ] dynamic_array; ***

## 7. Pointer Arithmetic

$p + i;$

actually yields the address $p + n \times i;$

## 8. Pointer Assignment

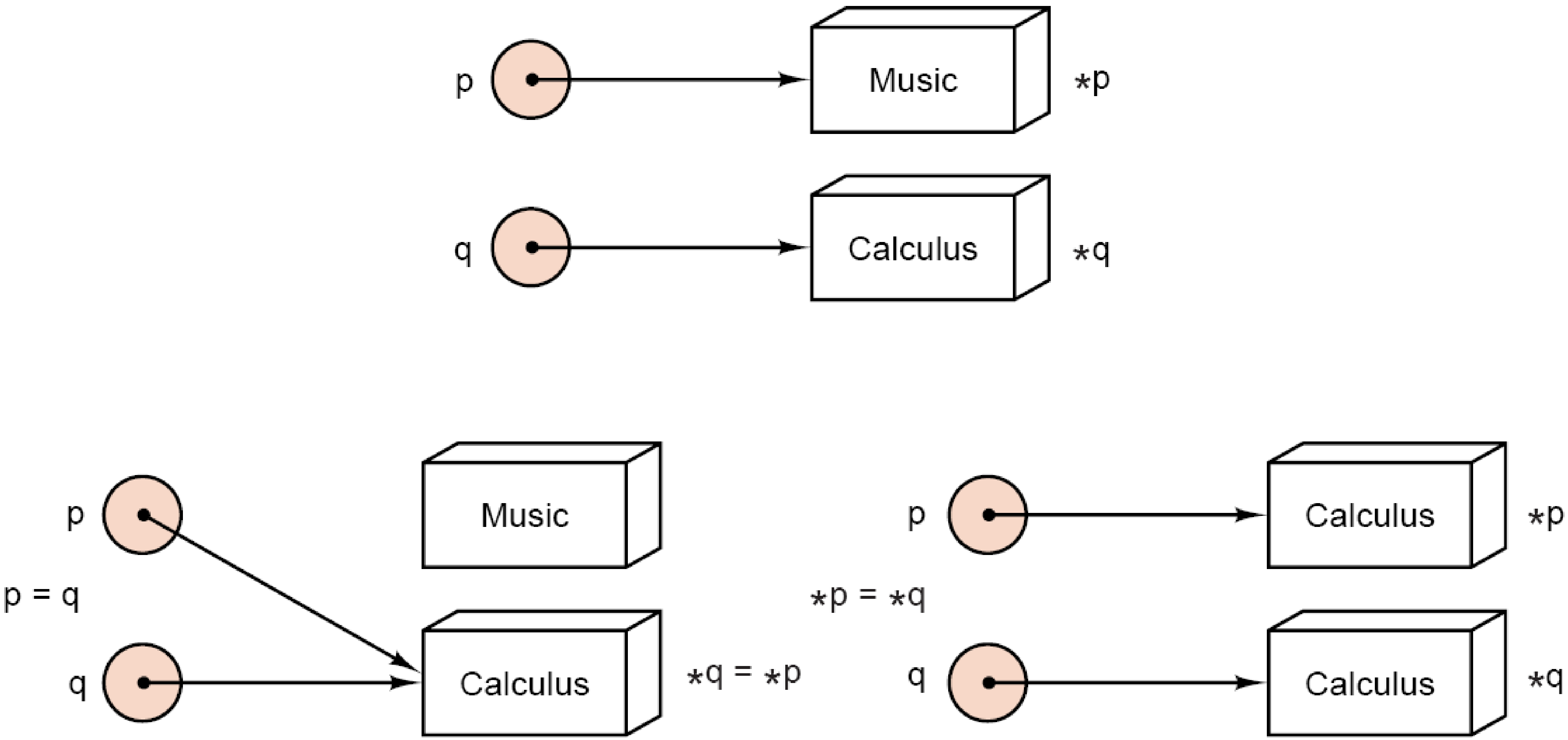



Figure 4.6. Assignment of pointer variables

## 9. Pointer Assignment

```
Item x[20];
Item *ptr = x;
ptr = &(x[0]);
```

## 10. Pointers to Structures

```
class Fraction{
public:
    int numerator;
    int denominator;
};
Fraction *p;
```

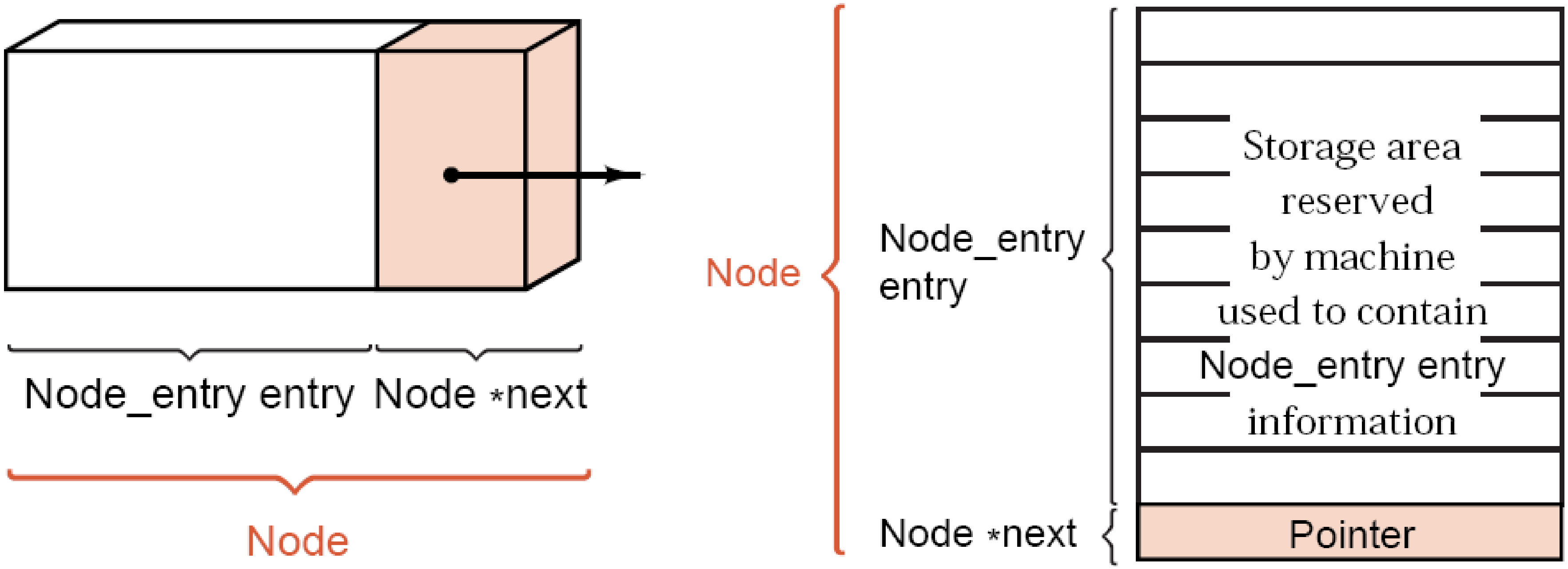
```
p->numerator = 0;
(*p).numerator = 0;
```

# 4.1.3 The Basics of Linked Structures

## 1. Nodes and Type Declarations

```
struct Node {
//    data members
   Node_entry entry;
   Node *next;
//    constructors
   Node( );
   Node(Node_entry item, Node *add_on = NULL);
};
```

# 4.1.3 The Basics of Linked Structures




(a) Structure of a Node

(b) Machine storage representation of a Node

**Figure 4.7. Structures containing pointers**

## 2. Node Constructors

```
Node::Node()
{
   next = NULL;
}

Node::Node(Node_entry item, Node *add_on)
{
   entry = item;
   next = add_on;
}
```

# 4.1.3 The Basics of Linked Structures

## 2. Node Constructors

```
Node first_node('a');          //  Node first_node stores data 'a'.
Node *p0 =  &first_node;       //  p0 points to first_Node.
Node *p1 = new Node('b');      //  A second node storing 'b' is created.
p0->next = p1;                 //  The second Node is linked after first_node.
Node *p2 = new Node('c', p0);  //  A third Node storing 'c' is created.
    //  The third Node links back to the first node, *p0.
p1->next = p2;                 //  The third Node is linked after the second Node.
```

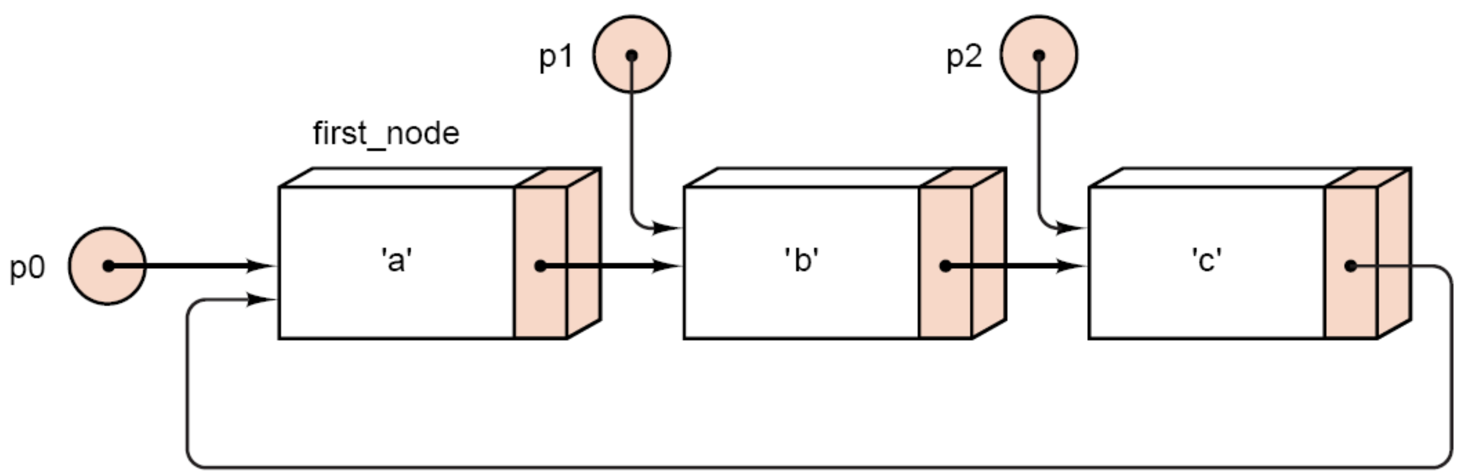



Figure 4.8. Linking nodes

# 4.2 Linked Stacks

```
class Stack {
public:
   Stack();
   bool empty() const;
   Error_code push(const Stack_entry &item);
   Error_code pop();
   Error_code top(Stack_entry &item) const;
protected:
   Node *top_node;
};
```

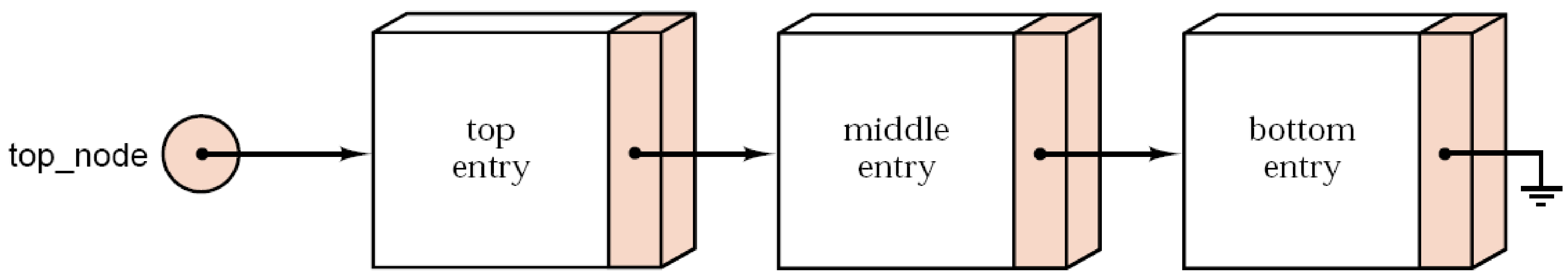
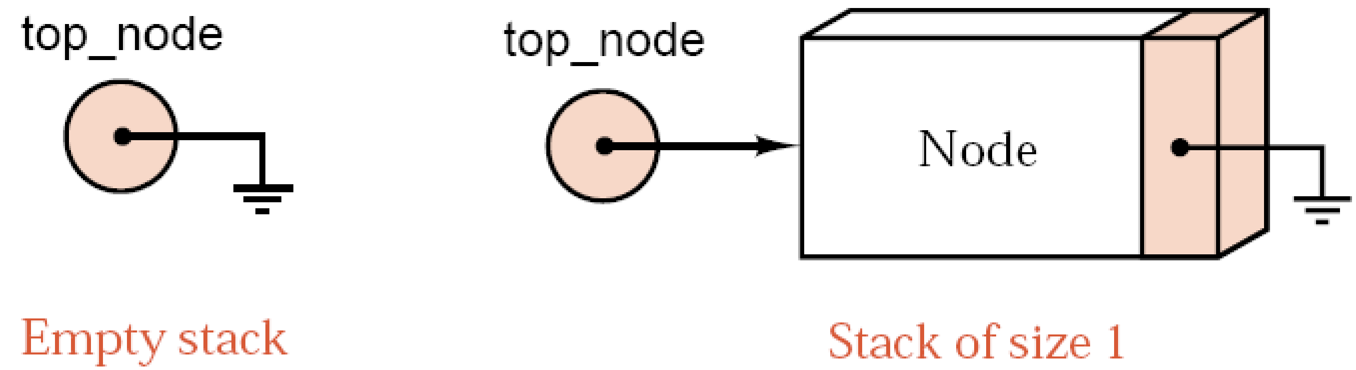



Figure 4.9. The linked form of a stack

# 4.2 Linked Stacks



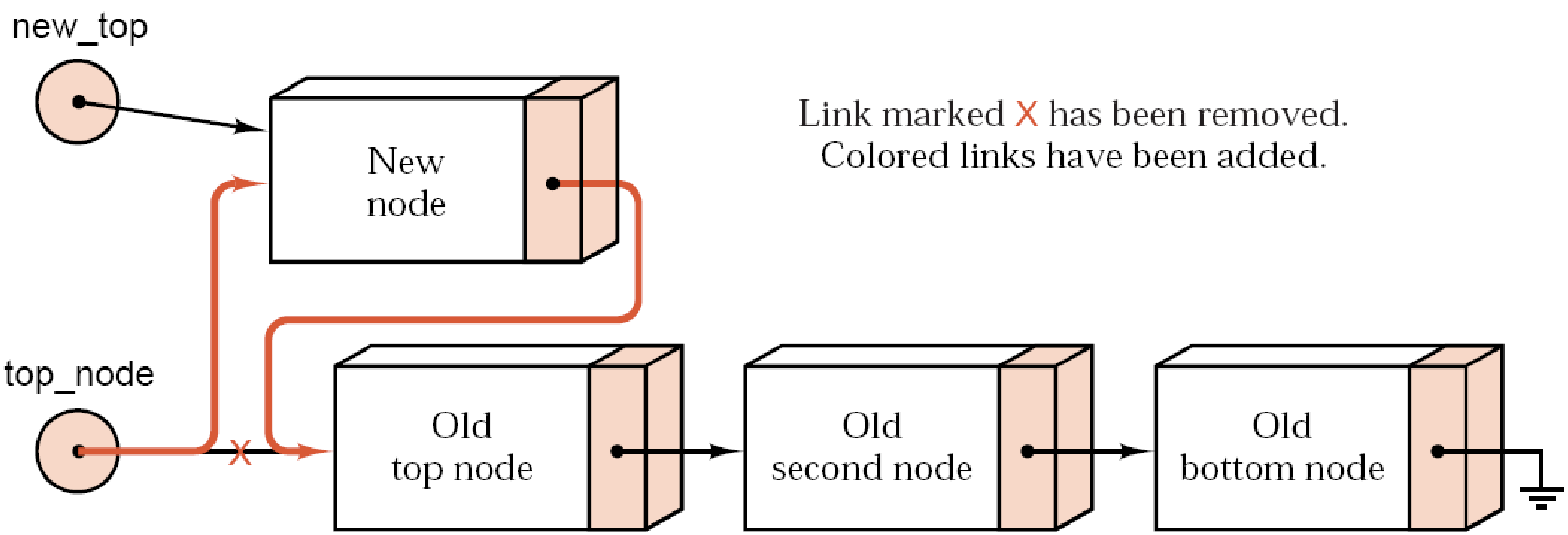
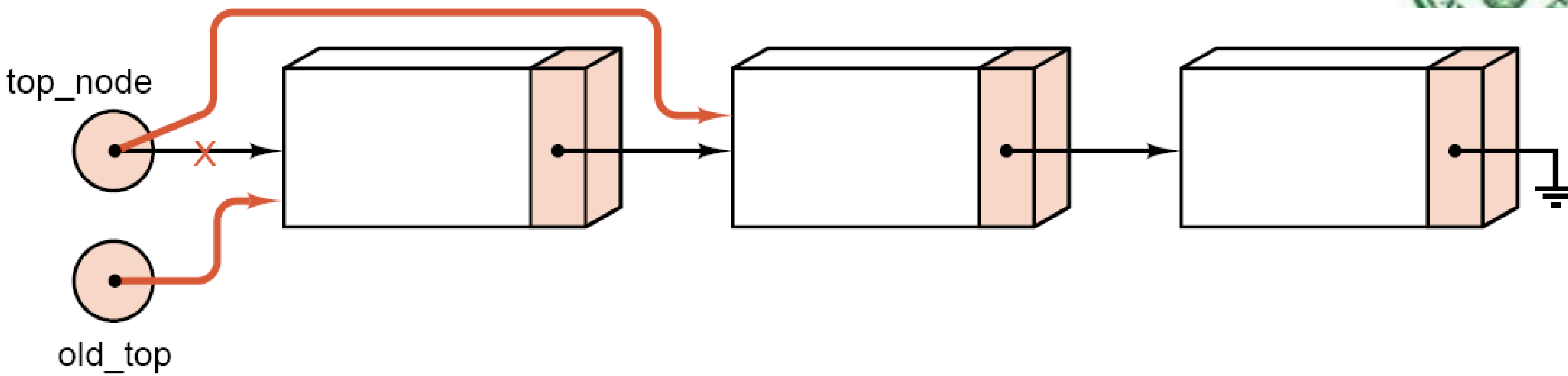




Figure 4.10. Pushing a node onto a linked stack

# 4.2 Linked Stacks

```
Error_code Stack :: push(const Stack_entry &item)
/* Post:  Stack_entry item is added to the top of the Stack; returns success or returns
          a code of overflow if dynamic memory is exhausted. */
{
   Node *new_top = new Node(item, top_node);
   if (new_top == NULL) return overflow;
   top_node = new_top;
   return success;
}
```

# 4.2 Linked Stacks




```
Error_code Stack :: pop( )
/* Post: The top of the Stack is removed. If the Stack is empty the method returns
         underflow; otherwise it returns success. */
{
   Node *old_top = top_node;
   if (top_node ==  NULL) return underflow;
   top_node = old_top->next;
   delete old_top;
   return success;
}
```

# 4.3 Linked Stacks with Safeguards

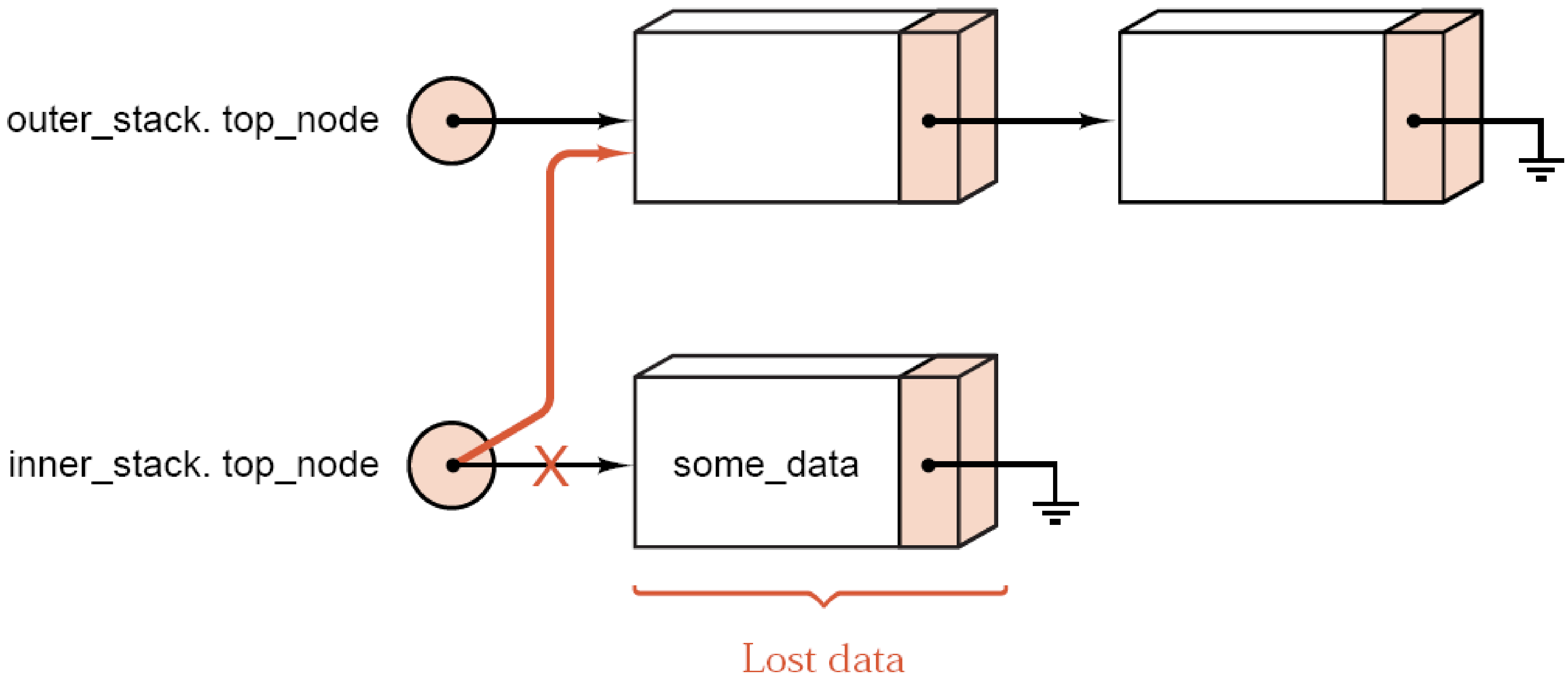
- Destructors
- Copy Constructors
- Overloaded Assignment Operators

# 4.3.1 The Destructor

```
Stack :: ~Stack()                    //    Destructor
/* Post: The Stack is cleared. */
{
   while (!empty())
      pop();
}
```

# 4.3.2 Overloading the Assignment Operator

```
Stack outer_stack;
for (int i = 0; i < 1000000; i++) {
    Stack inner_stack;
    inner_stack.push(some_data);
    inner_stack = outer_stack;
}
```



# 4.3.2 Overloading the Assignment Operator

```
void Stack::operator = (const Stack &original)  //   Overload assignment
/* Post:  The Stack is reset as a copy of Stack original. */
{
   Node *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                                  //   Duplicate the linked nodes
      new_copy = new_top = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())                        //   Clean out old Stack entries
      pop();
   top_node = new_top;                     //   and replace them with new entries.
}
```

# 4.3.3 The Copy Constructor

```
Stack :: Stack(const Stack &original)  //   copy constructor
/* Post:  The Stack is initialized as a copy of Stack original. */
{
   Node *new_copy, *original_node = original.top_node;
   if (original_node ==  NULL) top_node = NULL;
   else {                                   //    Duplicate the linked nodes.
      top_node = new_copy = new Node(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node(original_node->entry);
         new_copy = new_copy->next;
      }
   }
}
```

# 4.3.4 The Modified Linked-Stack Specification

```
class Stack {
public:
//    Standard Stack methods
   Stack();
   bool empty() const;
   Error_code push(const Stack_entry &item);
   Error_code pop();
   Error_code top(Stack_entry &item) const;
//    Safety features for linked structures
   ~Stack();
   Stack(const Stack &original);
   void operator = (const Stack &original);
protected:
   Node *top_node;
};
```

# Derive From ArrayLinearList

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |

0 1 2 3 4 5 6

- **stack top is either left end or right end of linear list**
  - **empty() //判断是否为空**

    **O(1) time**
  - **top()**

    **O(1) time**

# Derive From ArrayLinearList

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |

0 1 2 3 4 5 6

- **when top is left end of linear list**
  - **push(theObject)**

    **O(size) time**
  - **pop()**

    **O(size) time**

# Derive From ArrayLinearList

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |

0 1 2 3 4 5 6

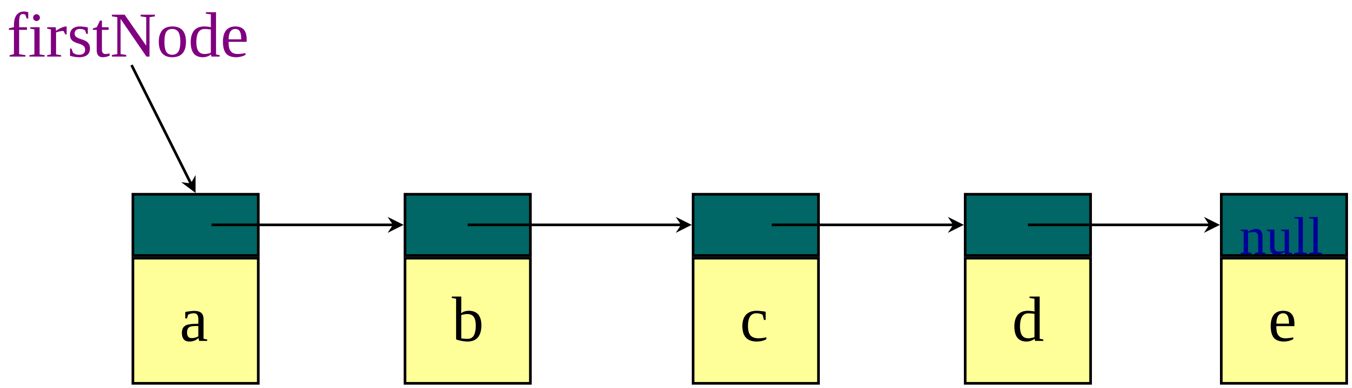
- **when top is right end of linear list**
  - **push(theObject)**

    **O(1) time**
  - **pop()**

    **O(1) time**

**use right end of list as top of stack**

# Derive From Chain

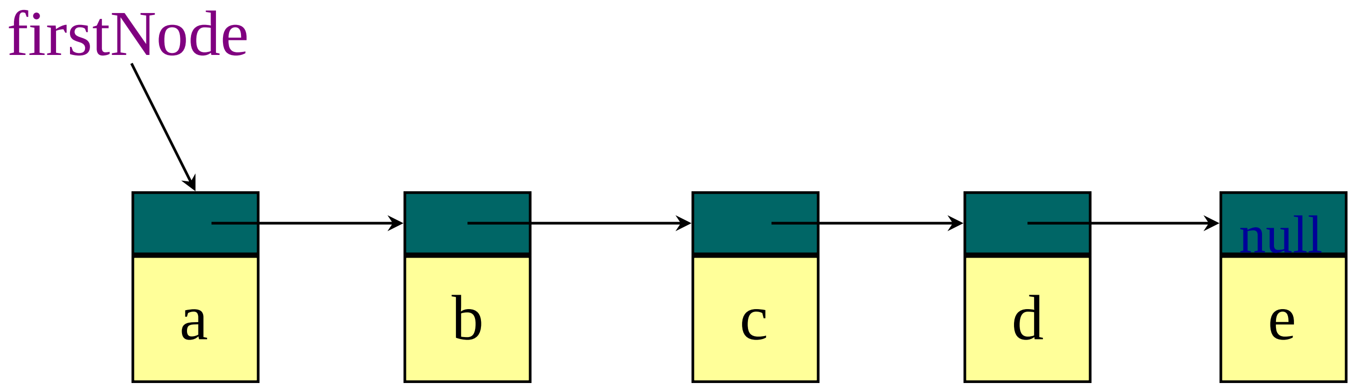



- stack top is either left end or right end of linear list

- empty ()

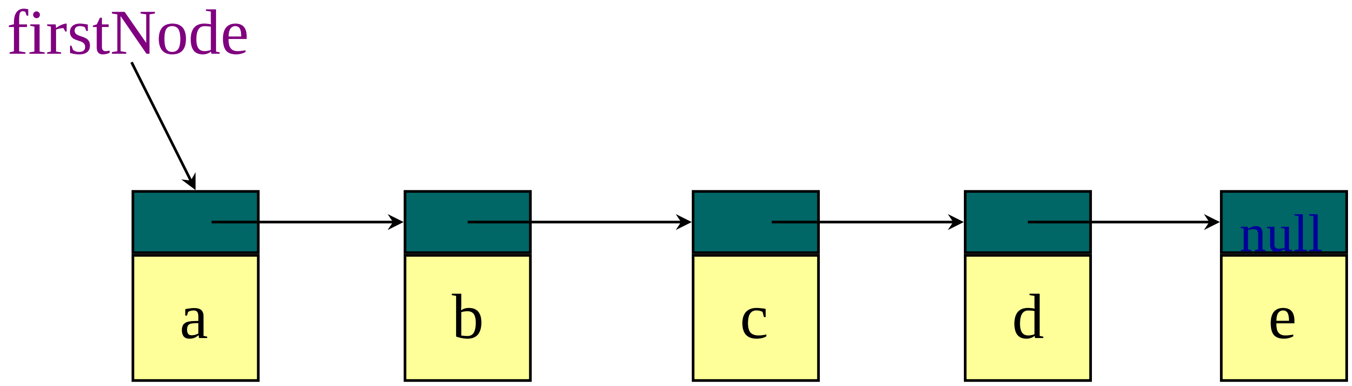
  O(1) time

# Derive From Chain




- when top is left end of linear list
  - top()
    O(1) time
  - push(theObject)
    O(1) time
  - pop()
    O(1) time

# Derive From Chain




- when top is right end of linear list
  - top()
  - O(size) time
  - push(theObject)
  - O(size) time
  - pop()
  - O(size) time

# 4.4 Linked Queues

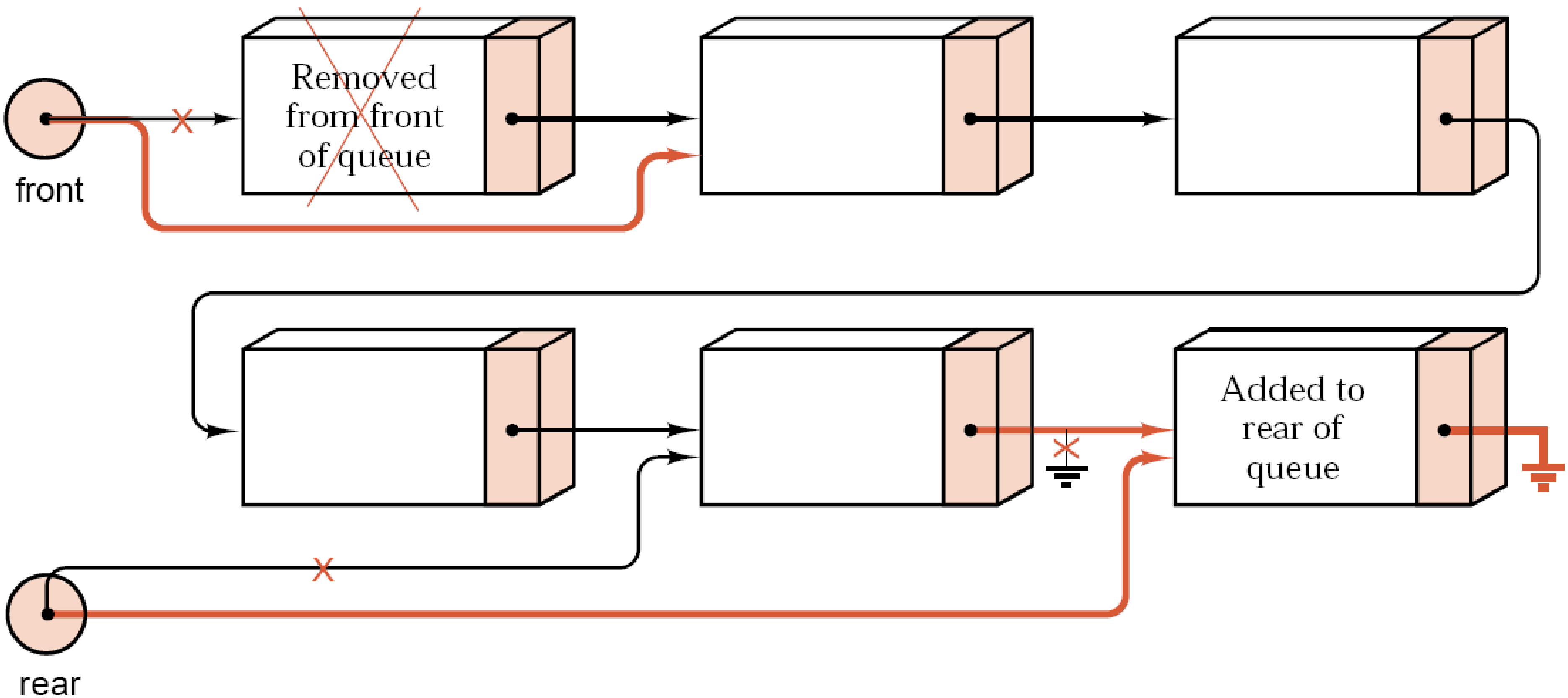



Figure 4.13. Operations on a linked queue

# 4.4.1 Basic Declarations

```
class Queue {
public:
//    standard Queue methods
   Queue( );
   bool empty( ) const;
   Error_code append(const Queue_entry &item);
   Error_code serve( );
   Error_code retrieve(Queue_entry &item) const;
//    safety features for linked structures
   ~Queue( );
   Queue(const Queue &original);
   void operator = (const Queue &original);
protected:
   Node *front, *rear;
};
```

# 4.4.1 Basic Declarations

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   front = rear = NULL;
}

Error_code Queue :: append(const Queue_entry &item)
/* Post: Add item to the rear of the Queue and return a code of success or return
         a code of overflow if dynamic memory is exhausted. */
{
   Node *new_rear = new Node(item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}
```

# 4.4.1 Basic Declarations

```
Error_code Queue::serve()
/* Post:  The front of the Queue is removed.
          If the Queue is empty, return an
          Error_code of underflow. */
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}
```

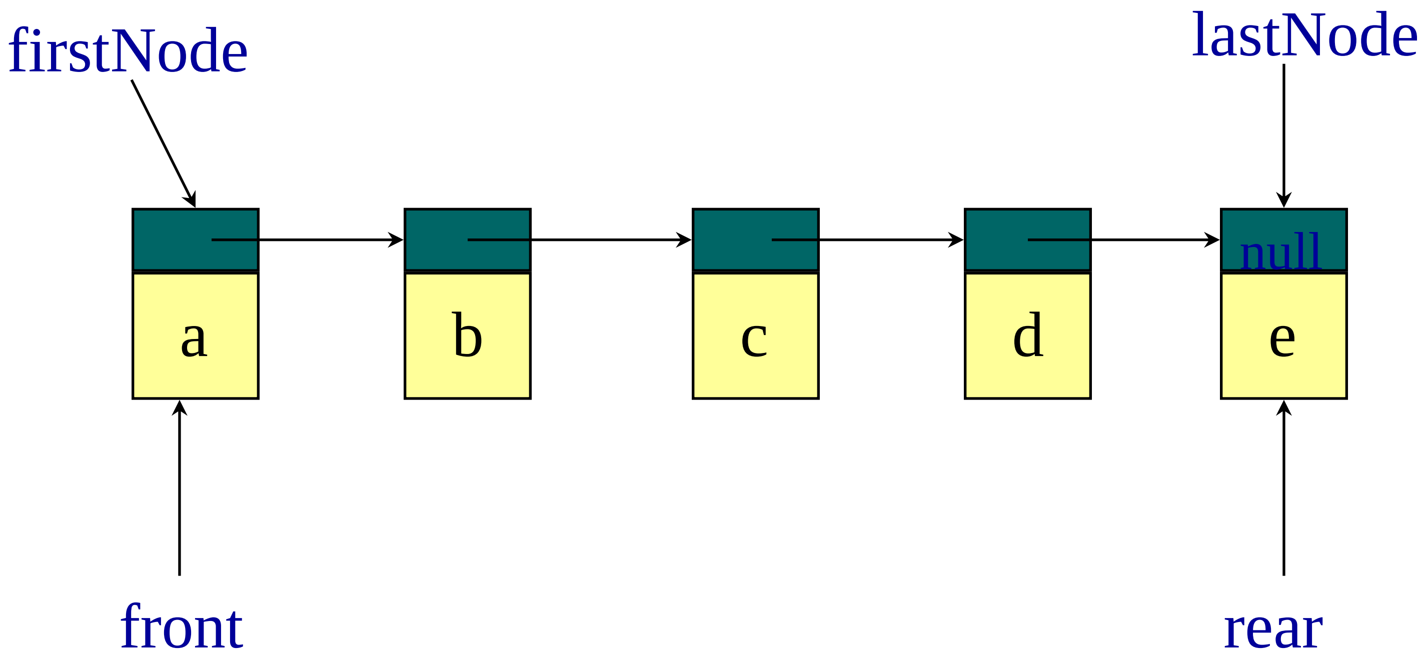
# 4.4.2 Extended Linked Queues

```
class Extended_queue:  public Queue {
public:
   bool full( ) const;
   int size( ) const;
   void clear( );
   Error_code serve_and_retrieve(Queue_entry &item);
};
```

# 4.4.2 Extended Linked Queues

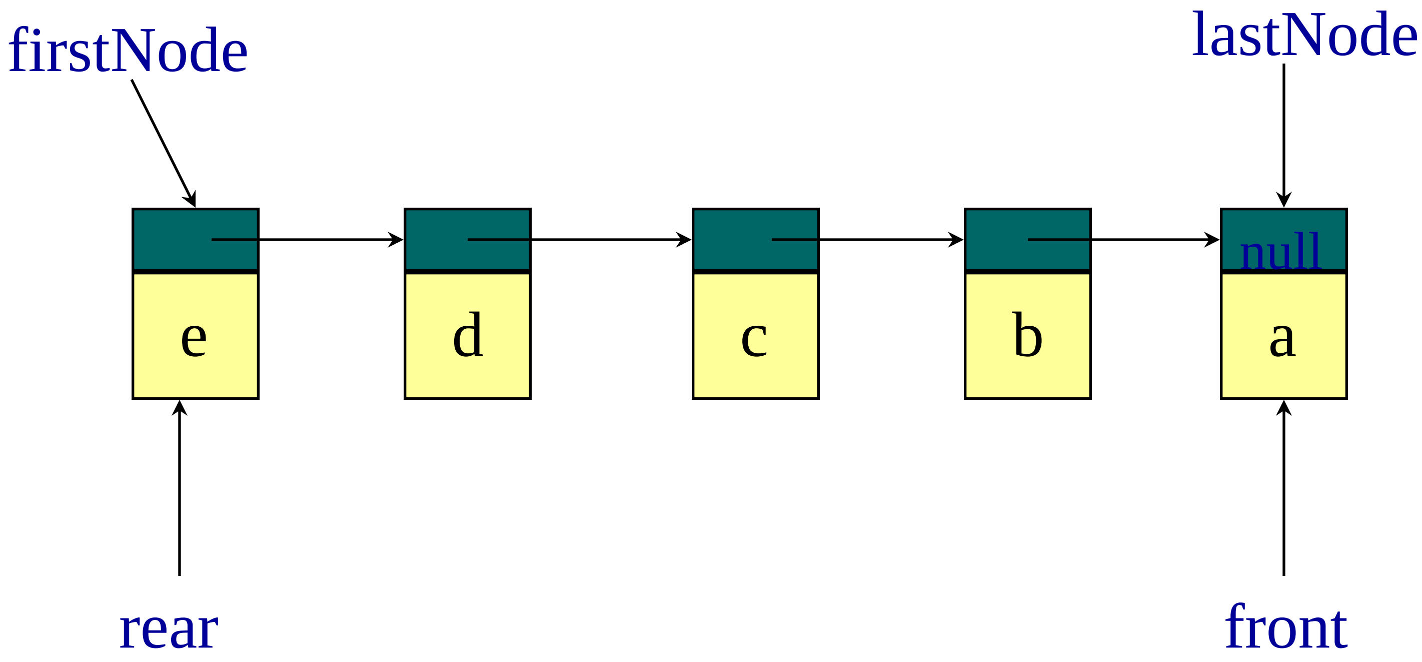
```
int Extended_queue :: size( ) const
/* Post: Return the number of entries in the Extended_queue. */
{
   Node *window = front;
   int count = 0;
   while (window != NULL) {
      window = window->next;
      count++;
   }
   return count;
}
```

# Linked Representation




- serve(the Object)
  ---O(1) time
- append()
  ---O(1) time

# Linked Representation




- append(theObject)
  ---O(1) time
- serve()
  ---O(size) time

# 课后阅读

# 4.6 Abstract Data Types and Implementations

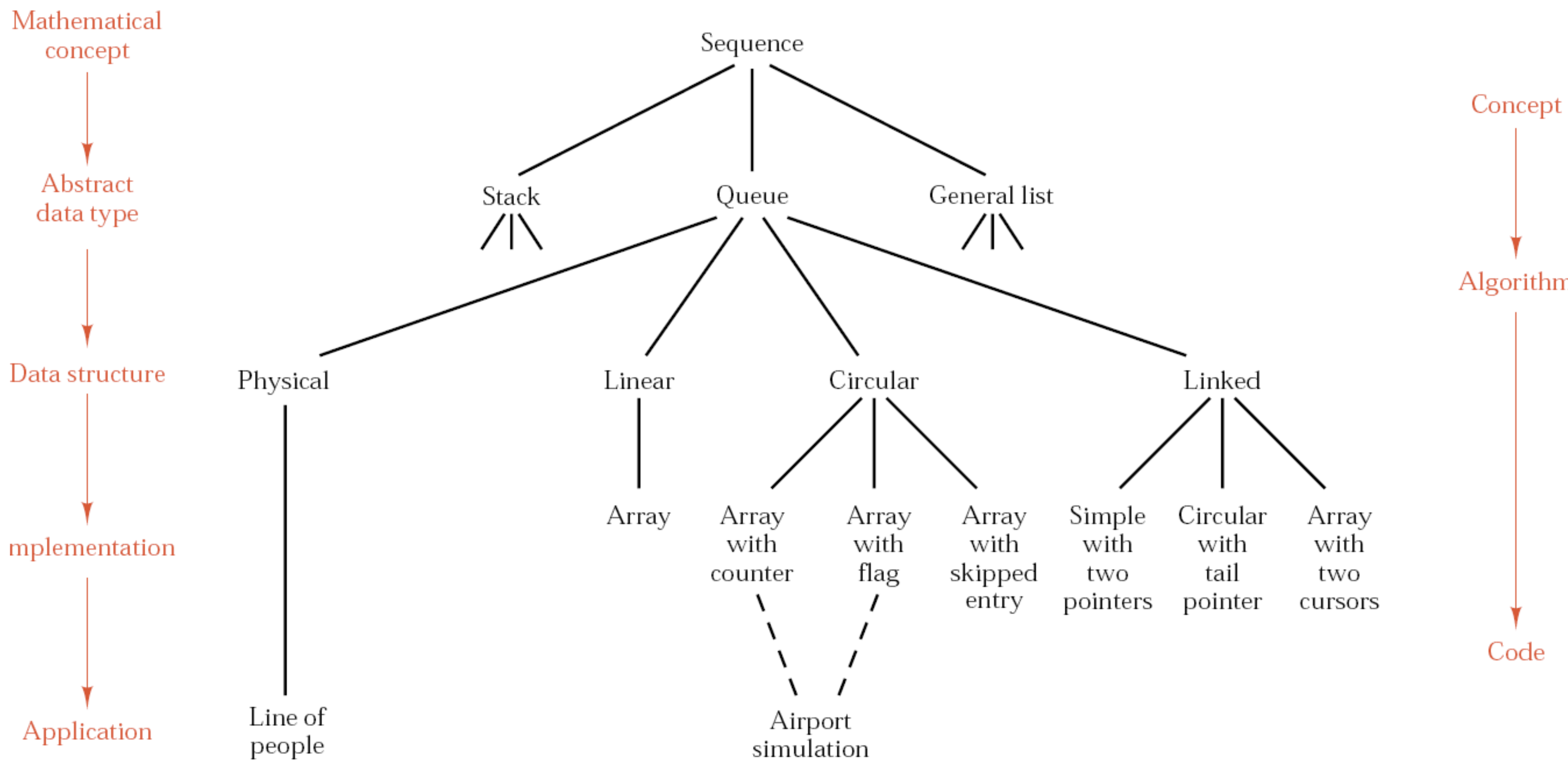



Figure 4.16. Refinement of a queue