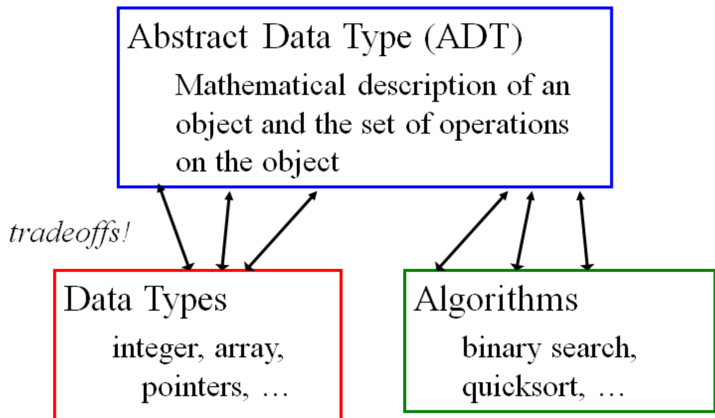
Lists, Stacks, and Queues

Abstract Data Types (ADT)

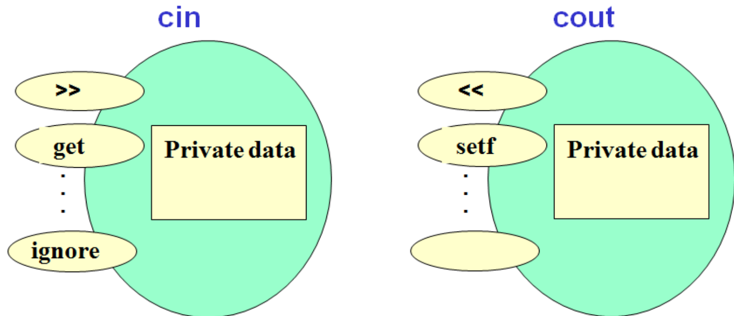
- *Data type*
  - a set of objects + a set of operations
  - Example: integer
    - set of integer numbers
    - operations: +, -, x, /
- Can this be generalized? e.g. procedures generalize the notion of an operator
- *Abstract data type*
  - what can be stored in the ADT
  - what operations can be done on/by the ADT
- Encapsulation
  - Operations on ADT can only be done by calling appropriate functions
  - no mention of how the set of operations is implemented
  - ADT $\rightarrow$ C++: class
  - method $\rightarrow$ C++: member function

Abstract Data Types (Cont'd)



Object-Oriented Design

A technique for developing a program in which the solution is expressed in terms of objects – self- contained entities composed of data and operations on that data.



More about OOD

- Languages supporting OOD include: C++, Java, Smalltalk, Eiffel, and Object-Pascal.
- A class is a programmer-defined data type and objects are variables of that type.
- In C++, cin is an object of a data type (class) named istream, and cout is an object of a class ostream. Header files iostream.h and fstream.h contain definitions of stream classes.

Client Code Using DateType

```
#include  "datetype.h"                // includes specification of the class
#include  "bool.h"

int  main ( void )
{
     DateType   startDate ;           // declares 2 objects of DateType
     DateType   endDate ;
     bool       retired  =  false ;

     startDate.Initialize ( 6, 30, 1998 ) ;
     endDate.Initialize ( 10, 31, 2002 ) ;
     cout <<  startDate.MonthIs( ) << "/" << startDate.DayIs( )
          << "/" <<  startDate.YearIs( ) << endl;
     while  ( ! retired )
     {       finishSomeTask( ) ;
             . . .
     }
}
```

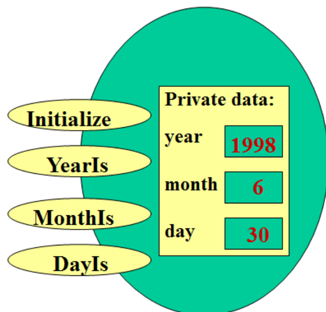
12

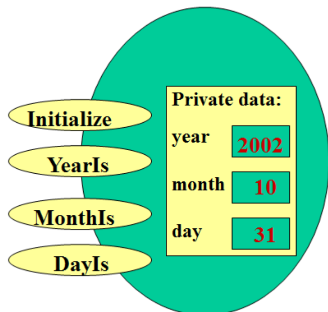
2 Separate Files Generally Used for Class Type

```
//  SPECIFICATION FILE              ( datetype .h )
//  Specifies the data and function members.
class DateType
 {
   public:
         . . .

   private:
         . . .
 } ;
```

```
//  IMPLEMENTATION FILE          ( datetype.cpp )
//  Implements the DateType member functions.

      . . .
```

DateType Class Instance Diagrams





Implementation of DateType member functions

```
// IMPLEMENTATION FILE                    (datetype.cpp)
#include "datetype.h"       // also must appear in client code

void DateType :: Initialize ( int  newMonth, int  newDay,
                              int  newYear )
// Post:  year is set to newYear.
//          month is set to newMonth.
//          day is set to newDay.
{
     year    =  newYear ;
    month =  newMonth ;
    day      =  newDay ;
}
```

Implementation of DateType member functions (Cont'd)

```
int DateType :: MonthIs ( )  const
// Accessor function for data member month
{
     return  month ;
}

int DateType :: YearIs ( )  const
// Accessor function for data member year
{
     return  year ;
}

int DateType :: DayIs ( )  const
// Accessor function for data member day
{
     return  day ;
}
```

The List ADT

- A sequence of zero or more elements:

$$A_1, A_2, A_3, \cdots, A_N$$

- N: length of the list
- A_1: first element
- A_N: last element
- If $N = 0$, then empty list
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}

List Operations

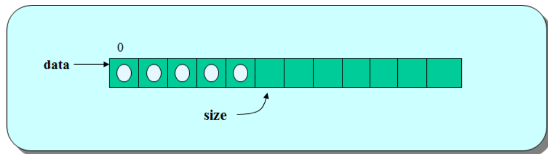
- *printList*: print the list
- *makeEmpty*: create an empty list
- *find*: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) $\rightarrow$ 3
- *insert*: insert an object to a list
  - insert(x,3) $\rightarrow$ 34, 12, 52, x, 16, 12
- *remove*: delete an element from the list
  - remove(52) $\rightarrow$ 34, 12, x, 16, 12
- *findKth*: retrieve the element at a certain position

Implementation of an ADT

- Choose a data structure to represent the ADT. E.g. arrays, records, etc.
- Each operation associated with the ADT is implemented by one or more subroutines
- Two standard implementations for the list ADT
  - *Array-based*
  - *Linked list*

Array Implementation

- Requires an estimate of the maximum size of the list – waste space
- printList and find: linear
- findKth: constant
- insert and delete: slow
  - e.g. insert at position 0 (making a new element) requires first pushing the entire array down one spot to make room
  - e.g. delete at position 0 requires shifting all the elements in the list up one
  - On average, half of the lists needs to be moved for either operation



Pointer Implementation (Linked List)

- The list is not stored contiguously
- A series of structures that are not necessarily adjacent
- Each node contains the element and a pointer to a structure containing its successor; the last cells next link points to NULL
- Compared to the array implementation,
  - $\surd$ the pointer implementation uses only as much space as is needed for the elements currently on the list
  - $\times$ but requires space for the pointers in each cell

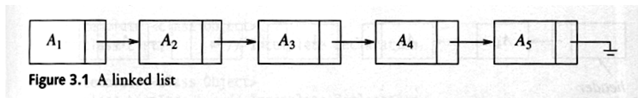
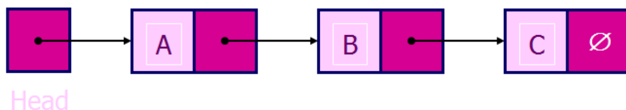


Figure 3.1 A linked list

Linked Lists



- A linked list is a series of connected nodes
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- Head: pointer to the first node
- The last node points to NULL

A Simple Linked List Class

- Declare Node class for the nodes

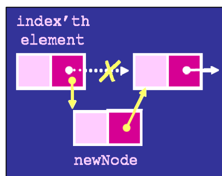
```
class Node {
public:
      double      data;       // data
      Node*       next;       // pointer to next
};
```

- Declare List, which contains
  - head: a pointer to the first node in the list.
  - Since the list is empty initially, head is set to NULL Operations on List

```
class List {
public:
      List(void) { head = NULL; }        // constructor
      ~List(void);                       // destructor

      bool IsEmpty() { return head == NULL; }
      Node* InsertNode(int index, double x);
      int FindNode(double x);
      int DeleteNode(double x);
      void DisplayList(void);
private:
      Node* head;
};
```

Inserting a New Node

- **Node* InsertNode(int index, double x)**
  - Insert a node with data equal to x after the index'th elements. (i.e., when index = 0, insert the node as the first element; when index = 1, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node. Otherwise, return NULL.
    (If index is < 0 or $>$ length of the list, the insertion will fail.)
- Steps
  - Locate index'th element
  - Allocate memory for the new node
  - Point the new node to its successor
  - Point the new nodes predecessor to the new node



Inserting a New Node (Cont'd)

```
Node* List::InsertNode(int index, double x) {
      if (index < 0) return NULL;

      int currIndex   =      1;
      Node* currNode =      head;
      while (currNode && index > currIndex) {
            currNode        =      currNode->next;
            currIndex++;
      }
      if (index > 0 && currNode == NULL) return NULL;

      Node* newNode   =      new    Node;
      newNode->data   =      x;
      if (index == 0) {
            newNode->next   =      head;
            head            =      newNode;
      }
      else {
            newNode->next   =      currNode->next;
            currNode->next =       newNode;
      }
      return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

Inserting a New Node (Cont'd)

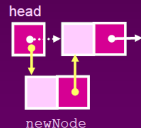
```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex   =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode        =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode   =       new     Node;
        newNode->data   =       x;
        if (index == 0) {
                newNode->next   =       head;
                head            =       newNode;
        }
        else {
                newNode->next   =       currNode->next;
                currNode->next  =       newNode;
        }
        return newNode;
}
```

Create a new node

Inserting a New Node (Cont'd)

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode        =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next   =       head;
                head            =       newNode;
        }
        else {
                newNode->next   =       currNode->next;
                currNode->next  =       newNode;
        }
        return newNode;
}
```

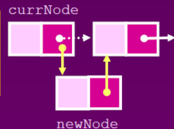
Insert as first element



Inserting a New Node (Cont'd)

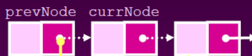
```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Insert after `currNode`



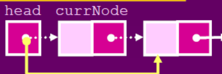
Deleting a Node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Deleting a Node (Cont'd)

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



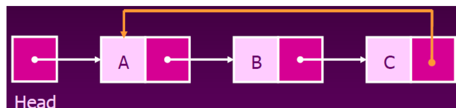
Destroying the list

- **~List(void)**
  - Use the destructor to release all the memory used by the list.
  - Step through the list and delete each node one by one.

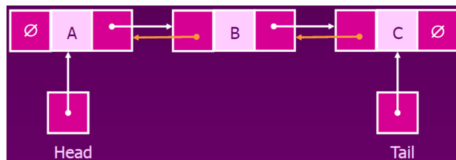
```
List::~List(void) {
    Node* currNode = head, *nextNode = NULL;
    while (currNode != NULL)
    {
        nextNode     =     currNode->next;
        // destroy the current node
        delete currNode;
        currNode     =     nextNode;
    }
}
```

Variations of Linked Lists

- **Circular linked lists**
  - The last node points to the first node of the list
  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)



- **Doubly linked lists**
  - Each node points to not only successor but the predecessor
  - There are two NULL: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Doubly-Linked List

- *insert(X, A)*: insert node A before X

```
newA = new Node(A);
newA->prev = X->prev;
newA->next = X;
X->prev->next = newA;
X->prev = newA;
```

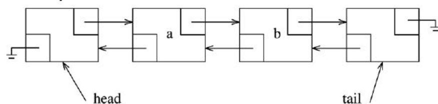
- *remove(X)*

```
X->prev->next = X->next;
X->next->prev = X->prev;
```

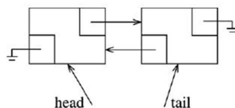
- Problems with operations at ends of list

Sentinel Nodes

- Dummy head and tail nodes to avoid special cases at ends of list
- Doubly-linked list with sentinel nodes



- Empty doubly-linked list with sentinel nodes



Link Inversion

- Forward:

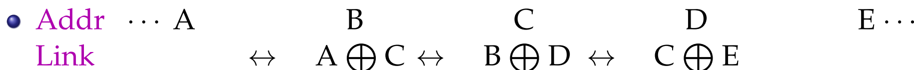
$$\begin{array}{ccccccccc} \cdots & & A & \rightarrow & B & \rightarrow & C & \rightarrow & D\cdots \\ & \Uparrow & \uparrow & & & & & & \\ \text{prev} & & \text{pres} & & & & & & \end{array}$$

$$\begin{array}{ccccccccc} \cdots & \leftarrow & A & & B & \rightarrow & C & \rightarrow & D\cdots \\ & & \Uparrow & & \uparrow & & & & \end{array}$$

$$\begin{array}{ccccccccc} \cdots & \leftarrow & A & \leftarrow & B & & C & \rightarrow & D\cdots \\ & & & & \Uparrow & & \uparrow & & \end{array}$$

- Backward ...

Exclusive-Or Doubly Linked List

- An *XOR linked list* compresses the same information into one address field by storing the bitwise XOR (here denoted by $\bigoplus$) of the address for previous and the address for next in one field
- 

- More formally:
  link(B) = addr(A)$\bigoplus$ addr(C), link(C) = addr(B)$\bigoplus$ addr(D), ...
- Note: $X \bigoplus X = 0 \quad X \bigoplus 0 = X \quad X \bigoplus Y = Y \bigoplus X$
  $(X \bigoplus Y) \bigoplus Z = X \bigoplus (Y \bigoplus Z)$
-
- Move forward:
  at C, take addr(B), XOR it with C's link value (i.e., B$\bigoplus$D), yields addr(D), and you can continue traversing the list.
- Move backward: Similar

C++ Standard Template Library (STL)

- Implementation of common data structures
  - Available in C++ library, known as Standard Template Library (STL)
  - List, stack, queue,
  - Generally these data structures are called containers or collections
- WWW resources
  - www.sgi.com/tech/stl
  - www.cppreference.com/cppstl.html

Lists Using STL

- Two popular implementation of the List ADT
  - The vector provides a growable array implementation of the List ADT
    - Advantage: it is indexable in constant time
    - Disadvantage: insertion and deletion are computationally expensive
  - The list provides a doubly linked list implementation of the List ADT
    - Advantage: insertion and deletion are cheap provided that the position of the changes are known
    - Disadvantage: list is not easily indexable
- Vector and list are class templates
  - Can be instantiated with different type of items

Iterators

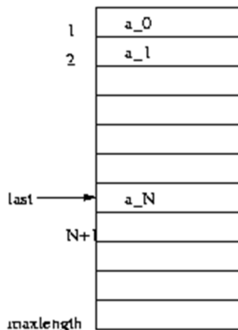
- Insert and remove from the middle of the list
  - Require the notion of a position
  - In STL a position is represented by a nested type, iterator
  - Example: list <string>:: iterator; vector<int>:: iterator
- *Iterator*: Represents position in the container
- Three Issues:
  - How one gets an Iterator?
  - What operations the iterators themselves can perform?
  - Which LIST ADT methods require iterators as parameters?

Example: The Polynomial ADT

- An ADT for single-variable polynomials

$$f(x) = \sum_{i=0}^{N} a_i x^i$$

- Array implementation



The Polynomial ADT

- Multiplying two polynomials

$$P_1(x) = 10x^{1000} + 5x^{14} + 1$$

$$P_2(x) = 3x^{1990} - 2x^{1492} + 11x + 5$$

- Implementation using a singly linked list

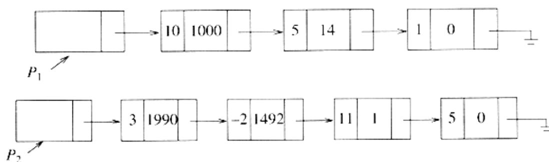
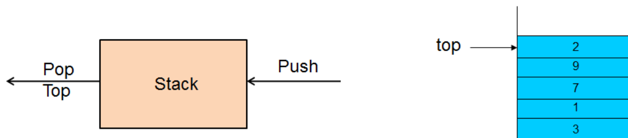


Figure 3.23 Linked list representations of two polynomials

Stack ADT

- Stack is a list where insert and remove take place only at the top
- Operations: Constant time
  - *Push* - inserts element on top of stack
  - *Pop* - removes element from top of stack
  - *Top* - returns element at top of stack
  - *Empty, size*
- LIFO (Last In First Out)



Stack Implementation

- **Singly Linked List**
  - Push - inserts the element at the front of the list
  - Pop - deletes the element at the front of the list
  - Top - returns the element at the front of the list
- **Array**
  - Back, push_back, pop_back implementation from vector
  - theArray and topOfStack which is -1 for an empty stack
  - Push - increment topOfStack and then set theArray[topOfStack] = x
  - Pop - return the value to theArray[topOfStack] and then decrement topOfStack

Stack Implementation (cont'd)

Linked List

```
template <typename Object>
class stack {
  public:
    stack() { }
    void push(Object & x) {
      s.push_front(x);
    }
    void pop() {
      s.pop_front();
    }
    Object & top() {
      s.front();
    }
  private:
    list<Object> s;
};
```

Vector

```
template <typename Object>
class stack {
  public:
    stack() { }
    void push(Object & x) {
      s.push_back(x);
    }
    void pop() {
      s.pop_back();
    }
    Object & top() {
      s.back();
    }
  private:
    vector<Object> s;
};
```

Applications of Stack

- Balancing symbols
  - Compiler checks for program syntax errors
  - Every right brace, bracket, and parenthesis must correspond to its left counterpart
  - The sequence [()] is legal, but [(]) is wrong
- Postfix Expressions
- Infix to Postfix Conversion
- Function calls

Balancing Symbols

- Balancing symbols: ((()())(()))

```
stack<char> s;
while not end of file or input {
   read character c
   if (c == '(') then
      s.push(c)
   if (c == ')') then
      if (s.empty()) then
         error
      else
         s.pop();
}
if (!s.empty()) then
   error
else
   okay
```

Evaluation of Postfix Expressions

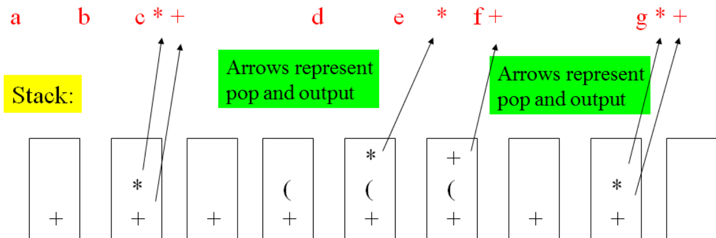
- *Infix expression*: ((1 * 2) + 3) + (4 * 5)
- *Postfix expression*: 1 2 * 3 + 4 5 * +
  - Unambiguous (no need for parenthesis)
  - Infix needs parenthesis or else implicit precedence specification to avoid ambiguity
    E.g. ″a + b * c″ can be ″(a + b) * c″ or ″a + (b * c)″
  - Postfix expression evaluation uses stack
  - E.g. Evaluate 1 2 * 3 + 4 5 * +
- Rule of postfix expression evaluation
  - When a number/operand is seen push it onto the stack
  - When a operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and
  - Result is pushed onto the stack

Infix to Postfix Conversion

- When an operand is read, place it onto the output
- Operators are not immediately output, so save them somewhere else which is stack
  If a left parenthesis is encountered, stack it
- Start with an initially empty stack
- If we see a right parenthesis, pop the stack, writing symbols until we encounter a left parenthesis which is popped but not output
- If we see any other symbols, then we pop the entries from the stack until we find an entry of lower priority. One exception is we never remove a "(" from the stack except when processing a ")"
- Finally, if we read the end of input, pop the stack until it is empty, writing symbols onto the output

Infix to Postfix Conversion (Cont'd)

- Infix: a + b * c + (d * e + f) * g
- Postfix: a b c * + d e * f + g * +

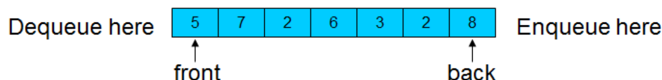


Applications of Stack: Function Calls

- Programming languages use stack to keep track of function calls
- When a function call occurs
  - Push CPU registers and program counter on to stack (activation record or stack frame)
  - Upon return, restore registers and program counter from top stack frame and pop

Queue ADT

- Queue is a list where insert takes place at the back, but remove takes place at the front
- Operations
  - *Enqueue* - inserts element at the back of queue
  - *Dequeue* - removes and returns element from the front of the queue
- FIFO (First In First Out)



Queue Implementation of Array

- When an item is enqueued, make the rear index move forward.
- When an item is dequeued, the front index moves by one element towards the back of the queue (thus removing the front item, so no copying to neighboring elements is needed).

  (front) XXXXOOOOO (rear)
  OXXXXOOOO (after 1 dequeue, and 1 enqueue)
  OOXXXXXOO after another dequeue, and 2 enqueues)
  OOOOXXXXX (after 2 more dequeues, and 2 enqueues)

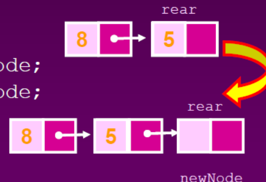
  The problem here is that the rear index cannot move beyond the last element in the array.
- Using a circular array
  - OOOOO7963 $\rightarrow$ 4OOOO7963 (after Enqueue(4))

Empty or Full?

- Empty queue:
  - back = front - 1
- Full queue?
  - the same!
  - Reason: n values to represent $n + 1$ states
- Solutions
  - Use a boolean variable to say explicitly whether the queue is empty or not
  - Make the array of size n+1 and only allow n elements to be stored
  - Use a counter of the number of elements in the queue

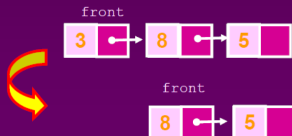
Queue Implementation based on Linked List

```
void Queue::Enqueue(double x) {
      Node* newNode       =     new Node;
      newNode->data       =     x;
      newNode->next       =     NULL;
      if (IsEmpty()) {
            front         =     newNode;
            rear          =     newNode;
      }
      else {
            rear->next    =     newNode;
            rear          =     newNode;
      }
      counter++;
}
```



Queue Implementation based on Linked List (Cont'd)

```
bool Queue::Dequeue(double & x) {
      if (IsEmpty()) {
            cout << "Error: the queue is empty." << endl;
            return false;
      }
      else {
            x                  =     front->data;
            Node* nextNode     =     front->next;
            delete front;
            front              =     nextNode;
            counter--;
      }
}
```



Applications of Queue

- Job scheduling, e.g., Jobs submitted to a line printer
- Customer service calls are generally placed on a queue when all operators are busy
- Priority queues
- Graph traversals
- Queuing theory