CS 2351 Data Structures

Linked Lists

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University

Outline

- Singly linked lists and chains (Sec. 4.1)
- C++ representation of chains (Sec. 4.2)
- The template class chain (Sec. 4.3)
  - C++ Iterator
- Circular lists and doubly linked lists (Sec. 4.4, 4.10)

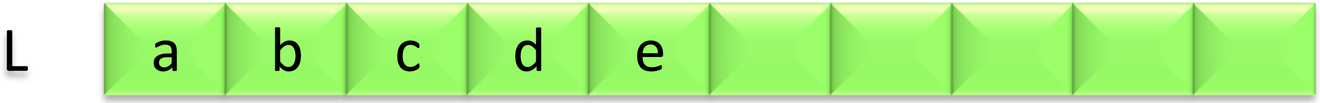
Review of C-type Arrays

- When you declare an array in C or C++

  ```
  int L[100];
  ```

  you conceptually envision a contiguous space of 100 integers, with each element stored next to another

  - Ex.: layout of L = {a,b,c,d,e} in an array representation



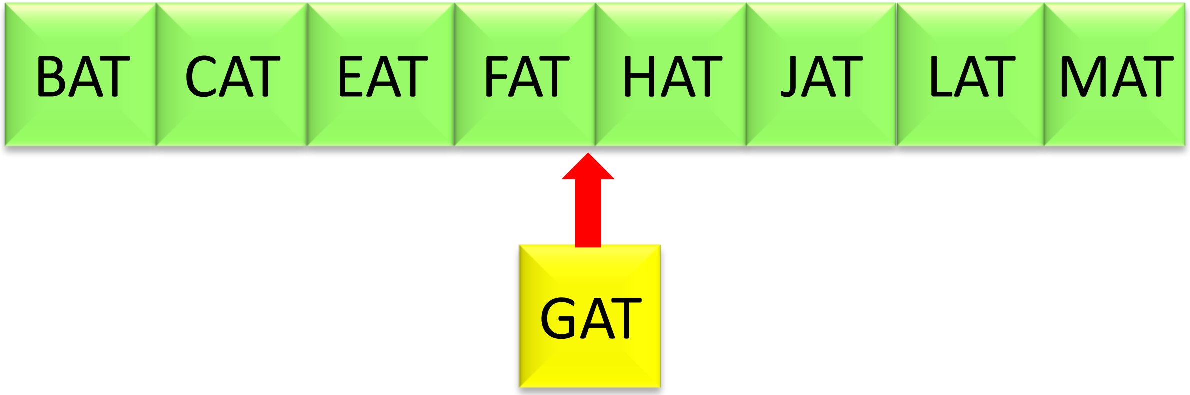
  - Actually, this is how the array is usually stored in the computer memory (each block above is a memory location)

Contiguous Space for Storing Arrays

- Pros:
  - Adequate for special data structures like stack and queue
  - Efficient to insert/delete from the ends
  - Suitable for random accesses
  - Good for the types of data structures discussed in the previous two chapters, e.g. polynomial addition, sparse matrix transpose, stack, queue, etc.
- Cons:
  - Difficult to insert/delete elements at arbitrary locations

Insertion/Deletion in an Array

- Suppose we have an array that stores 3-letter words in their alphabetic order



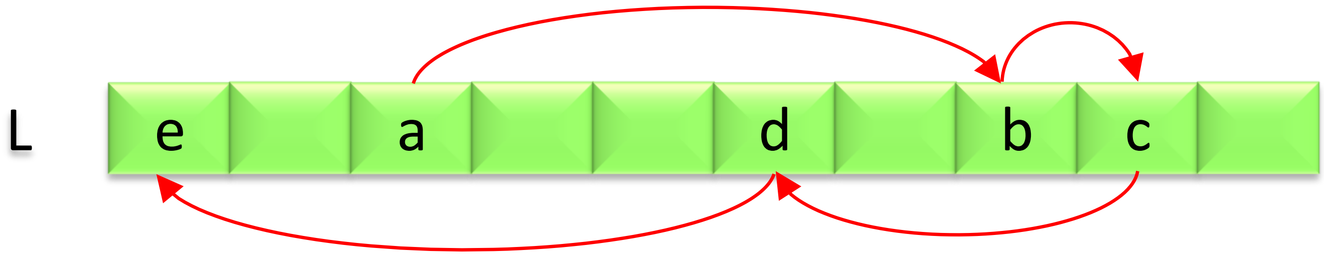
BAT CAT EAT FAT HAT JAT LAT MAT

GAT

- Given a new word “GAT”, we would certainly like it to be inserted between “FAT” and “HAT”
  - This would require shifting either “BAT” … “FAT” left or “HAT” … “MAT” right; both are expensive operations

Any Alternative?

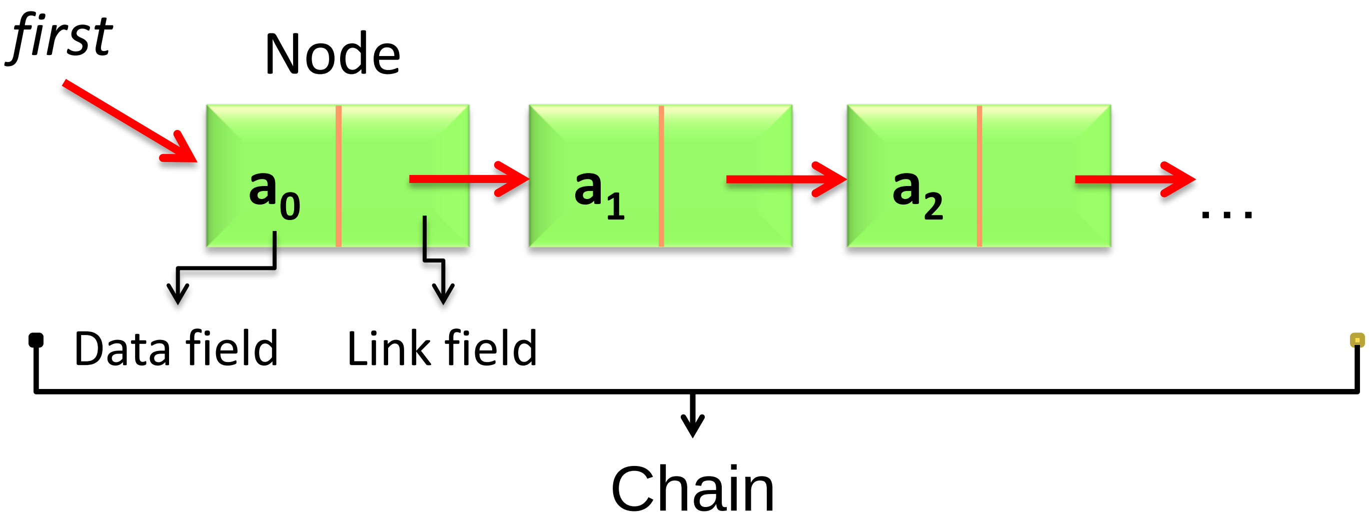
- Linked list representation



  - List elements are stored in memory in an arbitrary order
  - Explicit information (called a link) is used to go from one element to the next

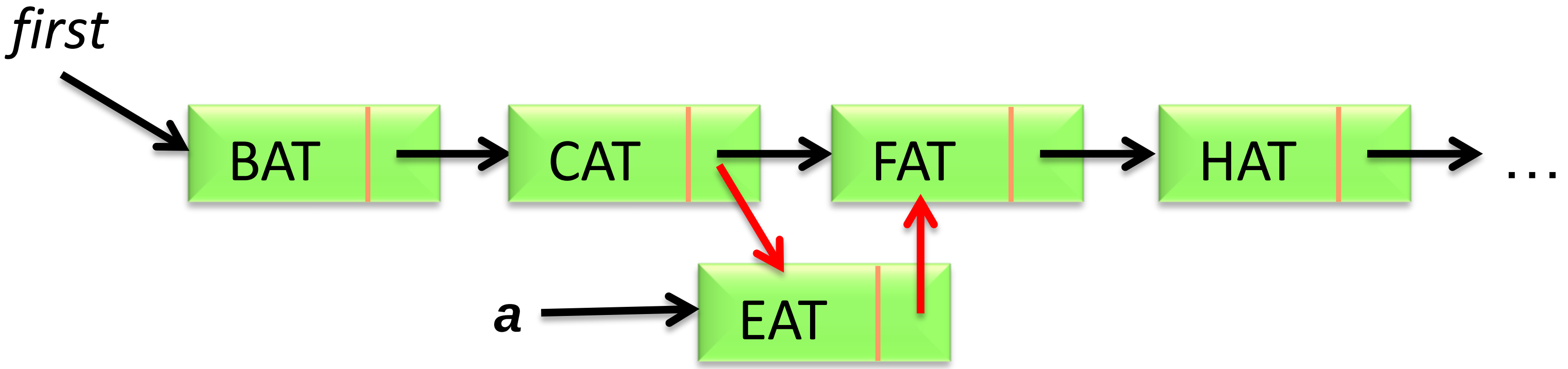
Linked List Representation

- Nodes are **no longer contiguous** in the memory
- Each node stores **address** or **location** of the next one
- Singly Linked List (SLL)
  - Each node has exactly one pointer (link) field



SLL Operation: Insertion

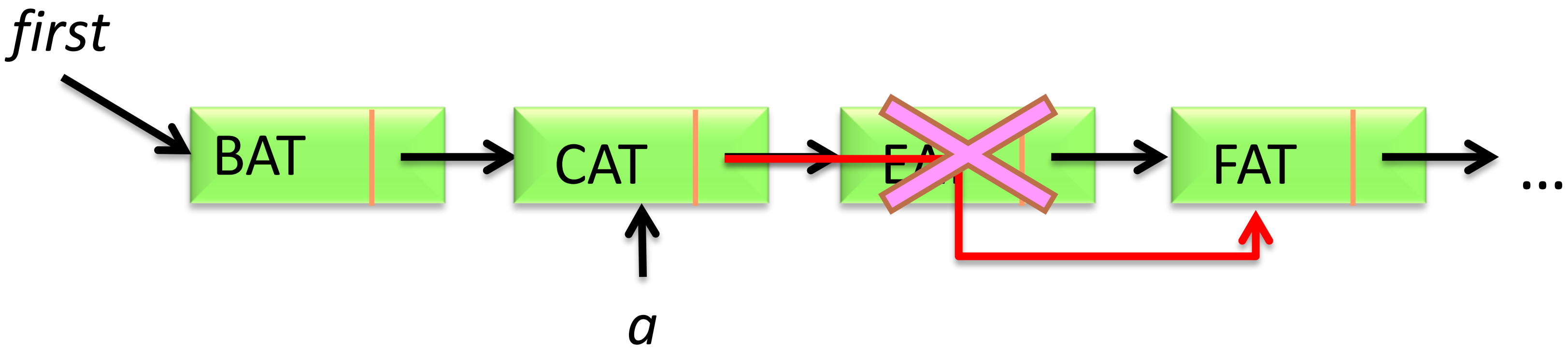
- Steps to do when we want to insert ”EAT” in between “CAT” and “FAT”
  - Create a new node “a” and set data field to “EAT”
  - Set the link field of “a” to “FAT” node
  - Set the link field of “CAT” node to “a”



You do not need to move or shift any node!

SLL Operation: Deletion

- Steps to do when we want to delete ”EAT” from the list
  - Locate the node “a” precedes the “EAT” node
  - Set the link field of “a” to the node next to “EAT” node
  - Delete the “EAT” node



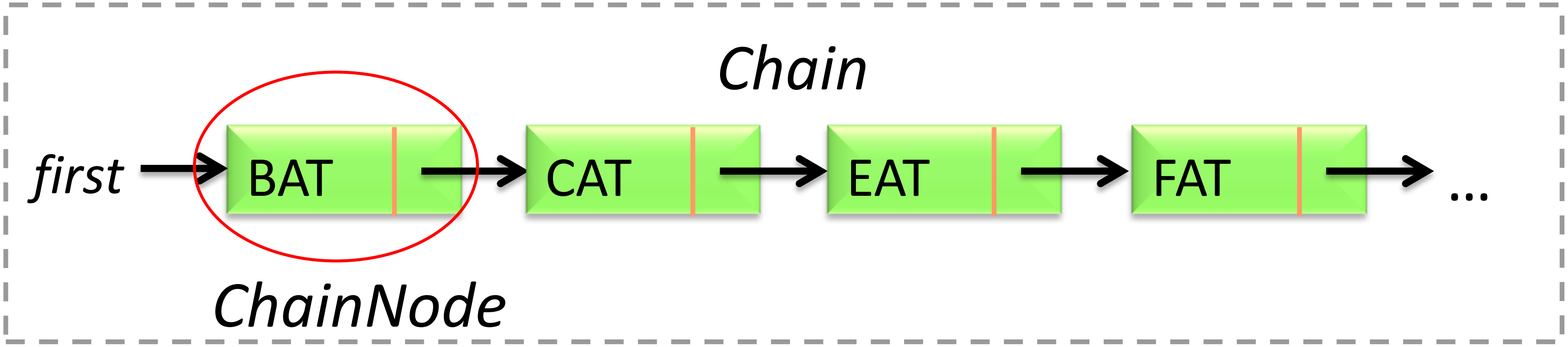
You do not need to move or shift any node!

Outline

- Singly linked lists and chains (Sec. 4.1)
- C++ representation of chains (Sec. 4.2)
- The template class chain (Sec. 4.3)
  - C++ Iterator
- Circular lists and doubly linked lists (Sec. 4.4, 4.10)

Conceptual Design

- Defining a "ChainNode" class
  - Data field
  - Link field
- Designing a "Chain" class
  - A *container class* of ChainNodes
  - Support various operations on ChainNodes



ChainNode and Chain Classes

```
class ChainNode {
friend class Chain;
public:
 // Constructor
 ChainNode(int
  value=0, ChainNode*
  next=NULL)
 {
   data = value;
   link = next;
 }
private:
 int data;
 ChainNode *link;
};
```

```
class Chain
{
public:
 // Create a chain with two nodes
 void Create2();

 // Insert a node with data=50
 void Insert50(ChainNode *x);

 // Delete a node
 void Delete(ChainNode *x,
                   ChainNode *y);

private:
 ChainNode *first;
};
```

Nested ChainNode and Chain Classes

- Alternative specification

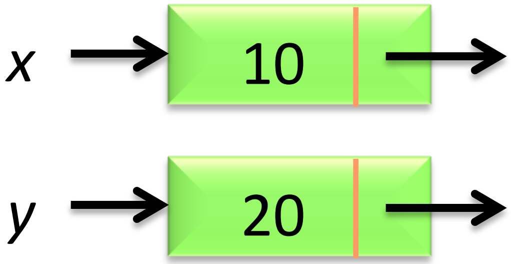
```
class Chain {
 public:
  // chain manipulation operations
  ...
 private:
  class ChainNode {
   public:
    int data;
    ChainNode *link;
   };
  ChainNode *first;
};
```

Pointer Manipulation in C++

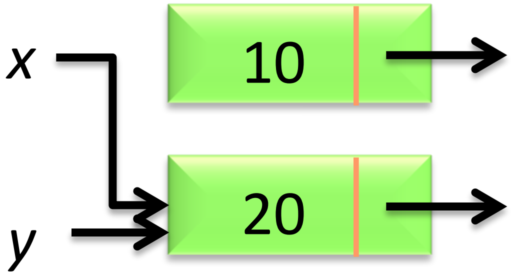
- Declare pointer of object
  - `NodeA *a1=NULL, *a2=NULL;`
- Allocate memory for object
  - `a1 = new NodeA;`
  - `a2 = new NodeA[10];`
- Delete object
  - `delete a1;`
  - `delete [] a2;`

Pointer Assignment

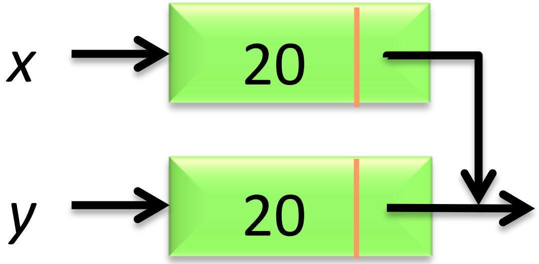
```
ChainNode *x, *y;
```



```
x = y;
```

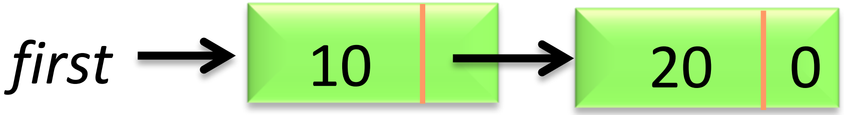


```
*x = *y;
```



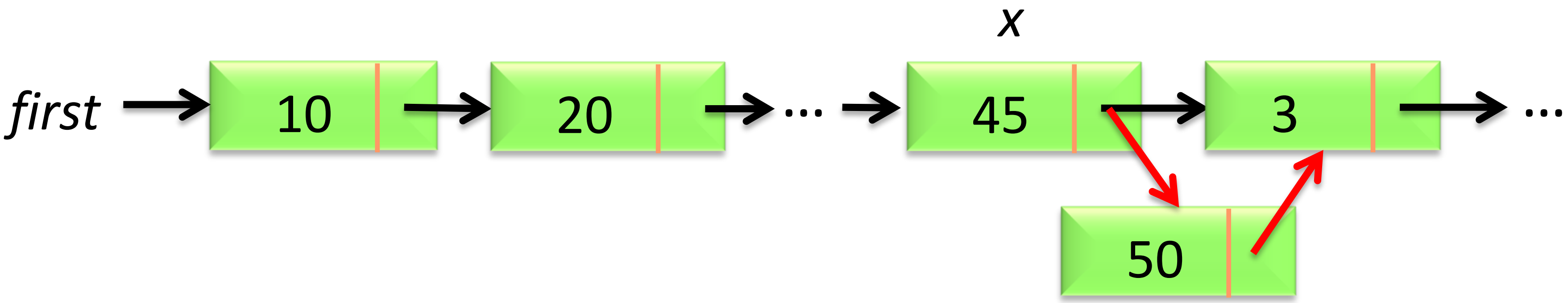
Chain Manipulation Operations

```
void Chain::Create2()
{   // Create a chain of two nodes
    // Create and set the fields of 2nd node
    ChainNode *second = new ChainNode(20,0);

    // Create and set the fields of 1st node
    first = new ChainNode(10,second);
}
```



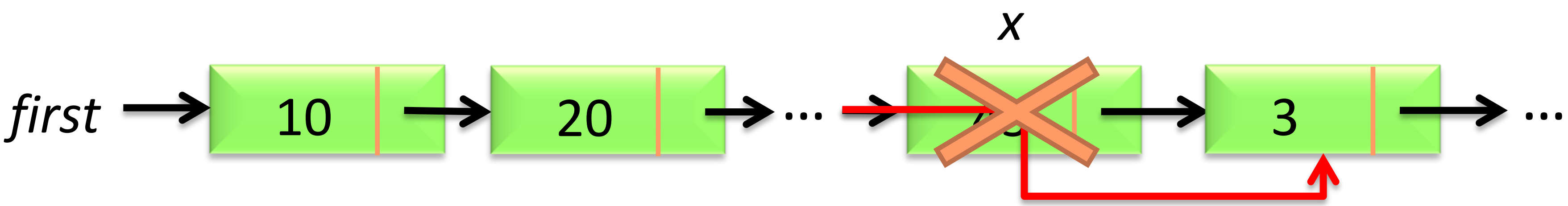
Chain Manipulation Operations

```
void Chain::Insert50(ChainNode *x)
{   // Insert a node with data=50
    if (first) // Insert after x
        x→link = new ChainNode(50, x->link);
    else // Insert into empty list
        first = new ChainNode(50);
}
```



Chain Manipulation Operations

```
void Chain::Delete(ChainNode *x, ChainNode *y)
{   // x is the node to be deleted
    // y is the node preceding x
    if(x==first) first = first->link;
    else y->link = x->link;
    delete x;
}
```



Outline

- Singly linked lists and chains (Sec. 4.1)
- C++ representation of chains (Sec. 4.2)
- The template class chain (Sec. 4.3)
  - C++ Iterator
- Circular lists and doubly linked lists (Sec. 4.4, 4.10)

Software Reuse

- There are urgent needs for reducing the cost of developing software
- How to reduce the number of person-hours in developing software without sacrificing quality?

  → Software reuse
- When initially design and develop software, do so to make it possible to reuse software in the future
- How to enhance chain class so that it becomes more reusable?
  - Use templates, design iterators, decide operations, ...

Implementing Chain Class with Template

```
template <class T> class Chain; // Forward decl.
template <class T>
class ChainNode {
  friend class Chain <T>;
  private:
      T data;
      ChainNode<T>* link;
};
template <class T>
class Chain {
  public:
      // Constructor
      Chain(void) {first = last = NULL;}
      // More chain operations here…
  private:
      ChainNode<T> *first;
      ChainNode<T> *last;
};
```

Please refer to the textbook for more Chain operations

Container Class

- A container class is a class that represents a data structure that contains a number of data objects
  - e.g. **`Chain`** class that contains **`ChainNodes`** objects
- How to visit elements in a container object? Suppose we have a chain L of Chain<int>
  - Output all integers in L
  - Find the maximum, minimum or mean of all integers in L
  - Obtain the sum or product of all integers in L
- All operations require to visit every element in the chain L

Issue: How to Identify Individuals?

- How many birds are there?
- How to visit every bird once?

Issue: How to Identify Individuals?

- How many corals are there?
- How to visit every coral once?

It requires an expert!

It Is Easy to Iterate through an Array

```
  for (int i=0; i<n; i++) {
     int currentItem = a[i];
     // do something with currentItem;
  }
```

- It takes an “expert” to iterate through a linked list

```
for (ChainNode<int> *ptr=first; ptr!=0;
                        ptr=ptr->link) {
   int currentItem = ptr->data;
   // do something with currentItem;
}
```

Towards a Generic “Expert”

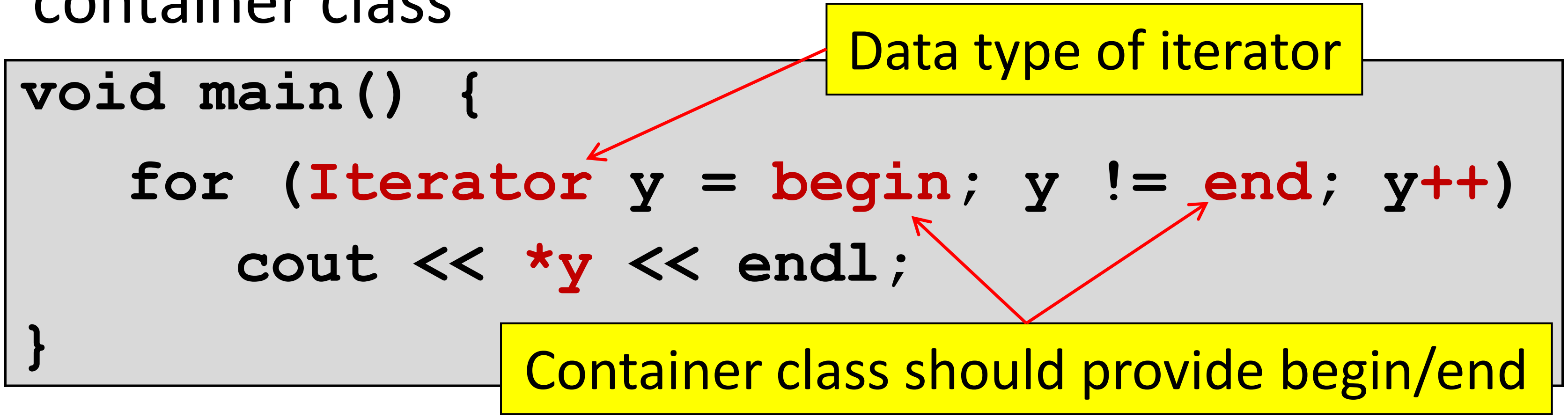
- Which version is easier to generalize to other data types?

```
for (int i=0; i<n; i++) {
   int currentItem = a[i];
   // do something with currentItem;
}
```

```
for (int* ip = a; ip != a+n; ip++) {
   int currentItem = *ip;
   // do something with currentItem;
}
```

Towards a Generic "Expert"

- We need some kind of *pointer* variables (objects) that can point to and iterate through the elements in a container class
  - At least support deferencing (*ip), pre- or post- increment (ip++), and equality (==, !=)
- Such a pointer object is called an iterator of that container class



```
void main() {
   for (Iterator y = begin; y != end; y++)
      cout << *y << endl;
}
```

Data type of iterator

Container class should provide begin/end

Iterators in C++ STL

- Iterators defined in C++ Standard Template Library (STL)
  - All iterators support “==”, “!=” and “*” operators
  - *Input iterator*: read access, pre- and post- “++” operators
  - *Output iterator*: write access, pre- /post- “++” operators
  - *Forward iterator*: pre- and post- “++” operators
  - *Bidirectional iterator*: pre- and post- “++” and “--” operators
  - *Random access iterator*: permit pointer jumps by arbitrary amounts

Forward Iterator for Chain

```
template <class T>
class Chain {
public:
   // Constructor
   Chain(void) {first = last = NULL;}
   // Iterator to Chain
   class ChainIterator{…};
   // Get the first element
   ChainIterator begin() {return ChainIterator(first);}
   // Get the end of the list
   ChainIterator end() {return ChainIterator(last);}
private:
   ChainNode<T> *first;
   ChainNode<T> *last;
};
```

Usage of Forward Iterator for Chain

```
void main() {
   Chain<int> myChain;
   // do operations on myChain here…
   // print out every element in myChain
   Chain<int>::ChainIterator my_it;
   for (my_it = myChain.begin();
              my_it != myChain.end(); ++my_it)
      cout << *my_it << endl;
   // more operations
}
```

```
for (ChainNode<int> *ptr=first; ptr!=0;
                         ptr=ptr->link) {
   cout << ptr->data << endl;
}
```

Forward Iterator for Chain

```
Class ChainIterator{ // nested class within Chain
public:
  // Constructor
  ChainIterator(ChainNode<T>* startNode = 0)
           {current = startNode;}
  // Dereferencing operator
  T& operator*() const {return current->data;}
  T* operator->() const {return &current->data;}
  // Increment operator
  ChainIterator& operator++()  // pre-”++”
  { current = current->link ; return *this; }
  ChainIterator operator++(int) { // post- “++”
    ChainIterator old = *this;
    current = current->link;
    return old;
  }
```

Forward Iterator for Chain

```
  // Equality operators
  bool operator!=(const ChainIterator right) const
  { return current != right.current; }
  bool operator==(const ChainIterator right) const
  { return current == right.current;}
private:
   ChainNode<T>* current;
};
```

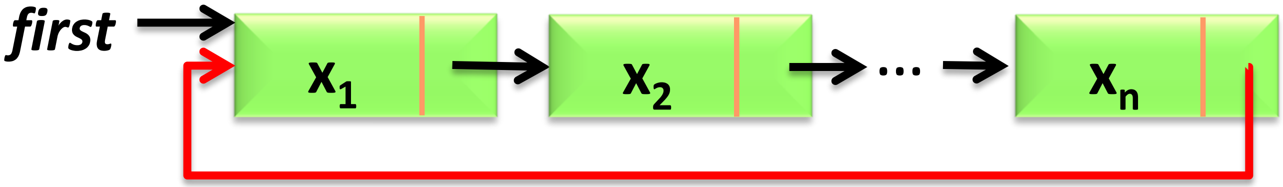
Outline

- Singly linked lists and chains (Sec. 4.1)
- C++ representation of chains (Sec. 4.2)
- The template class chain (Sec. 4.3)
  - C++ Iterator
- Circular lists and doubly linked lists (Sec. 4.4, 4.10)

Circular Lists

A singly-linked circular list

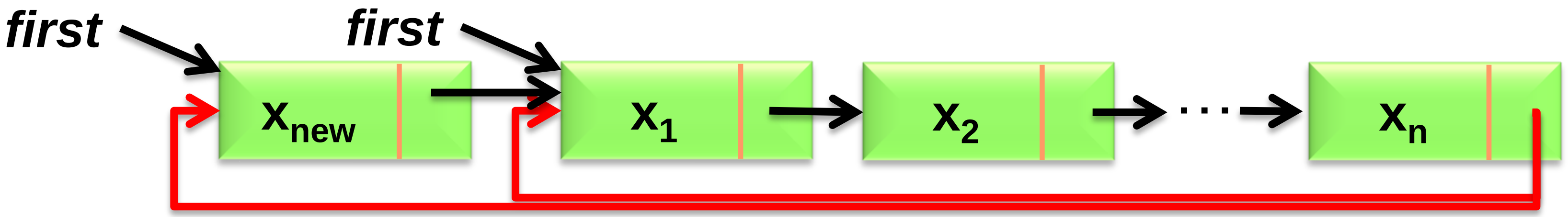
- The link field of the last node points to the first node



- Check for the last node
  - **`If (current->link == first)`**
- Can visit a node from any position

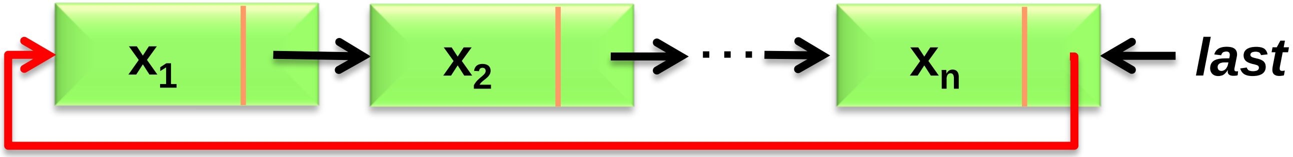
Circular Lists: Insert

- Suppose we want to insert a new node at the front of the list
- Set link field of new node to ***first*** and set ***first*** to new node
- Go to the last node and set the link field to new node



Circular Lists

- Instead of using a pointer to store the first node, it is more convenient to store the last node of a circular list
- We could always access the first node via last->link

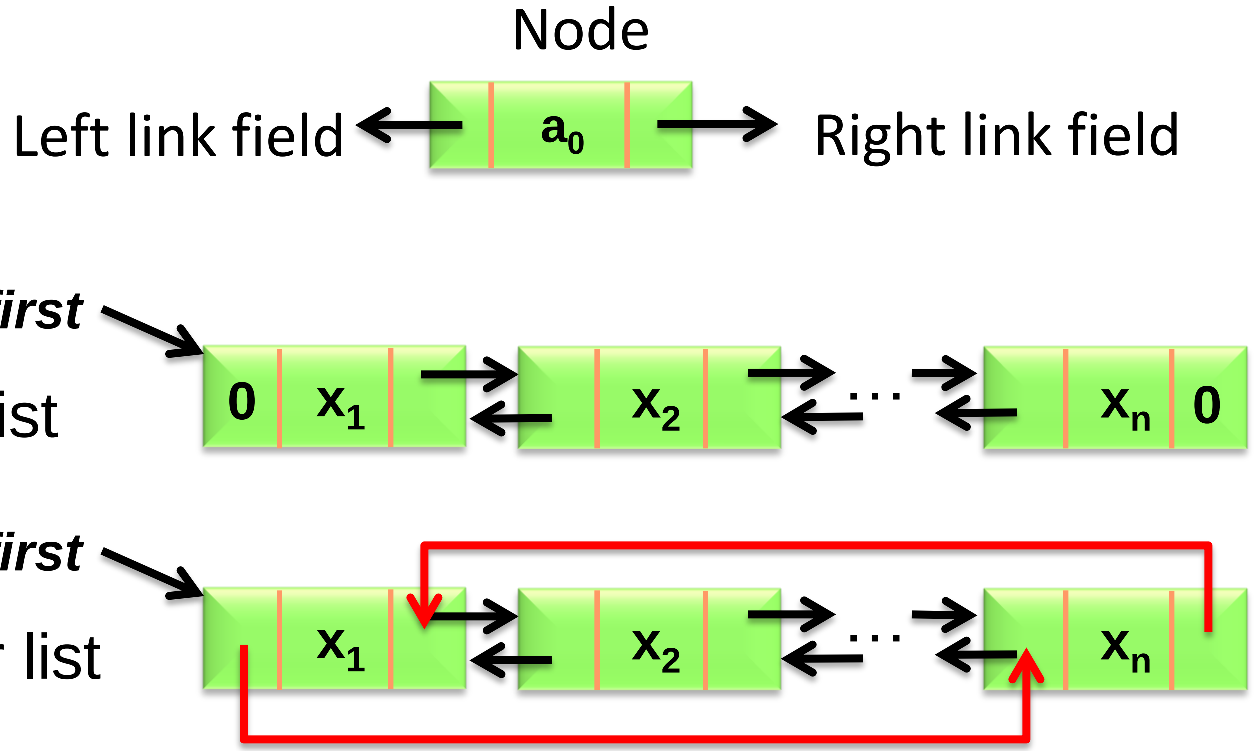


Circular Lists: Insert at Front

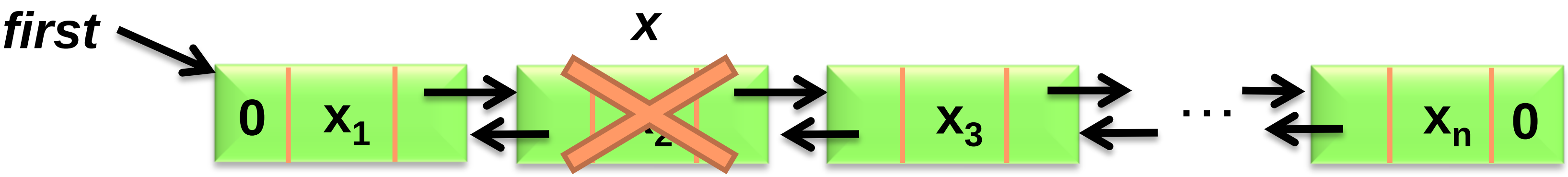
```
Template<class T>
void CircularList<T>::InsertFront(const T& e)
{
   ChainNode<T>* newNode = new ChainNode<T>(e);

   if (last) { // nonempty list
     newNode->link = last->link;
     last->link = newNode;
   }
   else { // empty list
     last = newNode;
     newNode->link = newNode;
   }
}
```

Double Linked Lists

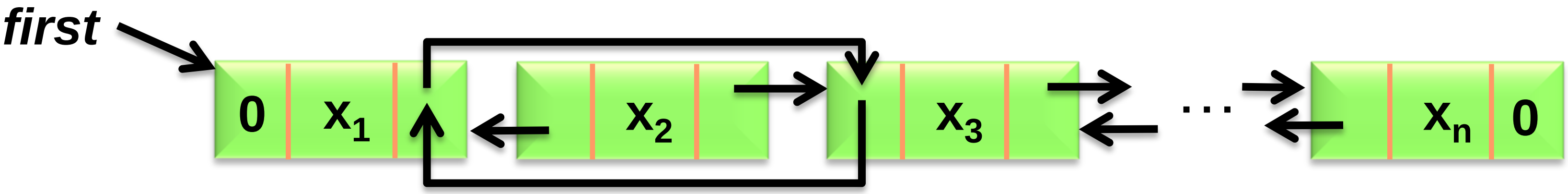
- Each node has **TWO** link fields
- Could move in **TWO directions** to visit nodes



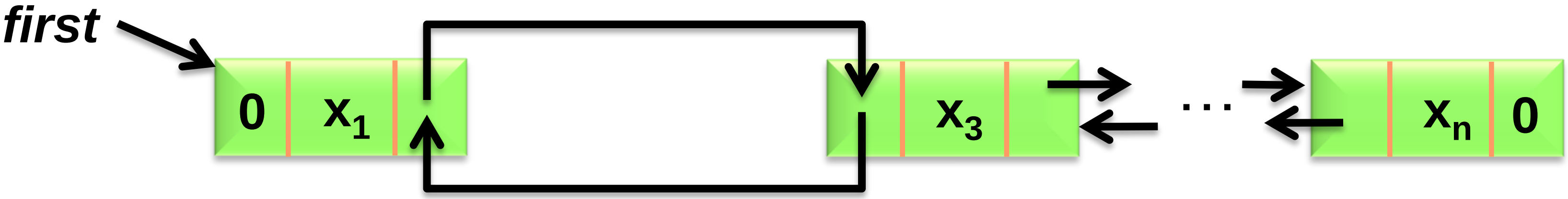
Double Linked Lists: Delete



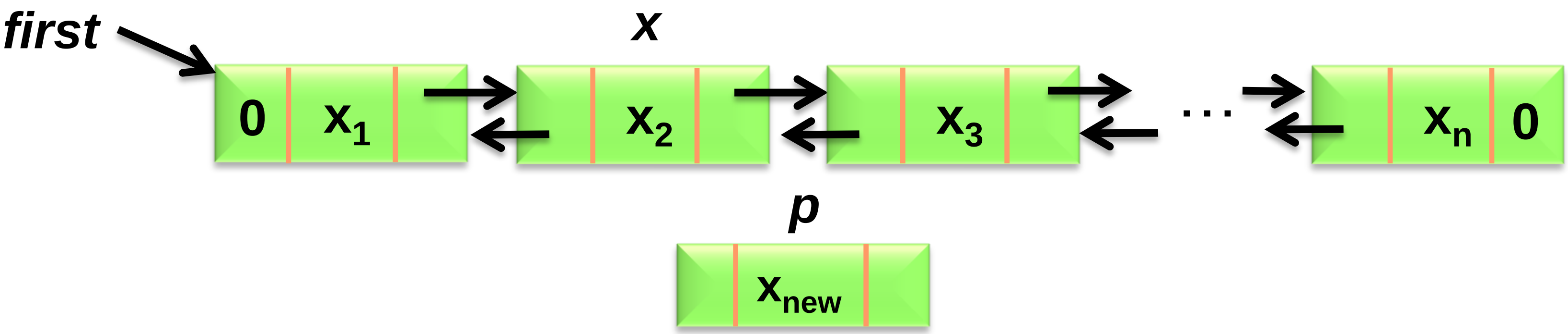
x->left->right = x->right; x->right->left = x->left;



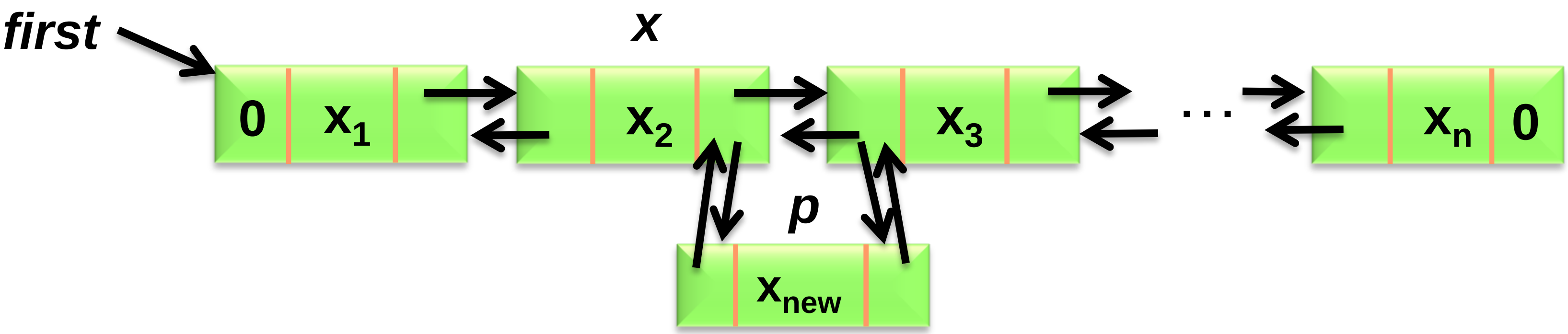
delete x;



Double Linked Lists: Insert



p->left = x; p->right=x->right



x->right->left = p; ⇨ x->right = p;

Summary

- Linked lists need not store data in contiguous space
- Some C++ supports for software reuse: template, iterator
- Circular lists and doubly linked lists
- Self-study topics
  - Polynomial using linked lists
  - Sparse matrix using linked lists
  - Linked stacks and queues