CSIEB0100 Data Structures

Lecture 05 Linked Lists

Shiow-yang Wu 吳秀陽
Department of Computer Science and Information Engineering
National Dong Hwa University

Lecture material is partly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

Introduction

- Array
  - Successive items (homogenous) locate a fixed distance
- Disadvantage
  - Data movements during insertion and deletion
  - Waste space in storing *n* ordered lists of varying size
- Possible solution
  - Linked list

Singly Linked Lists

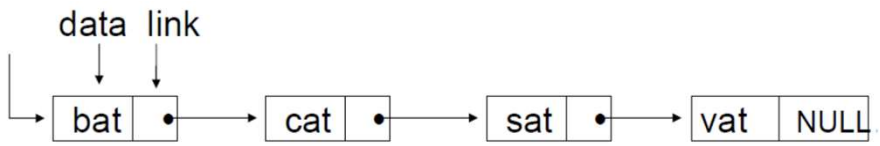
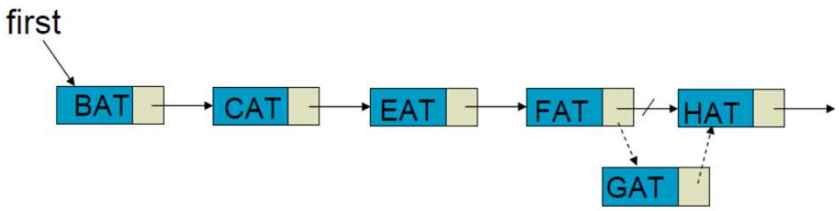


*Figure 4.2: Usual way to draw a linked list

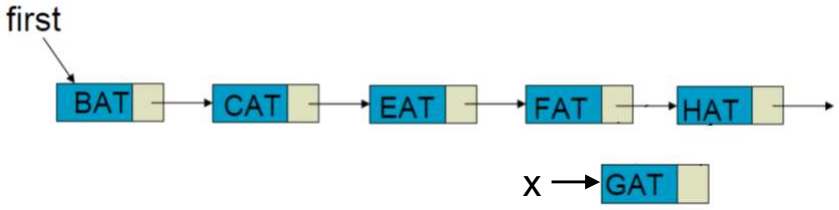
Insertion



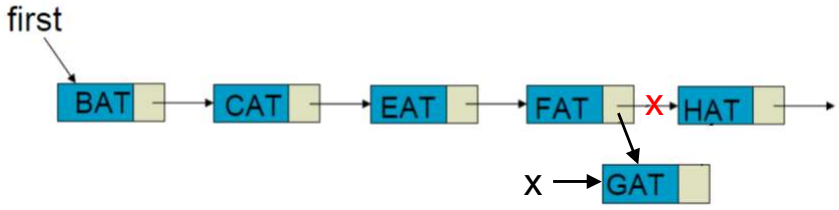
Insert GAT after FAT

Insertion Steps

- What are the correct steps for insertion?

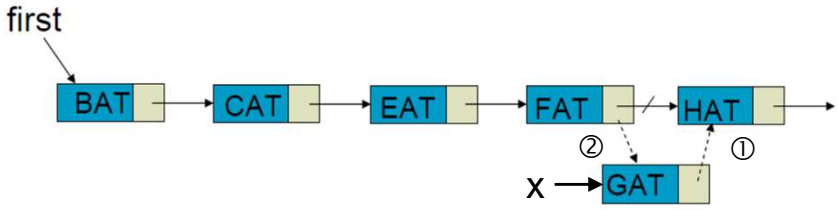


- What happens if we do this?



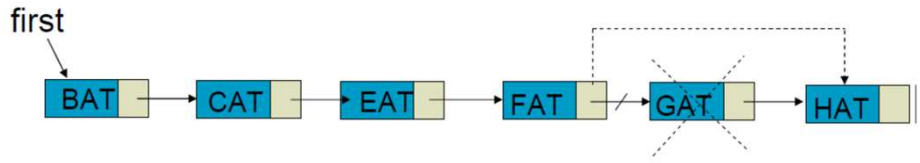
Right Steps of Insertion

- The key is to set up the connection **before** reassign existing connection.



Insert GAT after FAT

Deletion



Delete GAT from list

- What are the correct steps?
- What about the deleted node?

Defining a List Node in C++

```
class ThreeLetterNode
{
private:
  char data[3];
  ThreeLetterNode *link;
}
```

Designing a List in C++

- Design Attempt 1:
  - Use a global variable first which is a pointer of ThreeLetterNode.
  - Unable to access to private data members: data and link.
  - A popular approach in C

```
ThreeLetterNode *first;
first->data, first->link
```


Designing a List in C++

- Design Attempt 2:
  - Make data members public or define public member functions GetLink(), SetLink() and GetData()
  - Defeat the purpose of data encapsulation
    - We should not know how the list is implemented
- An ideal solution should
  - Only grant those functions that perform list manipulation operations (i.e., inserting a node or deleting a node) access to data members

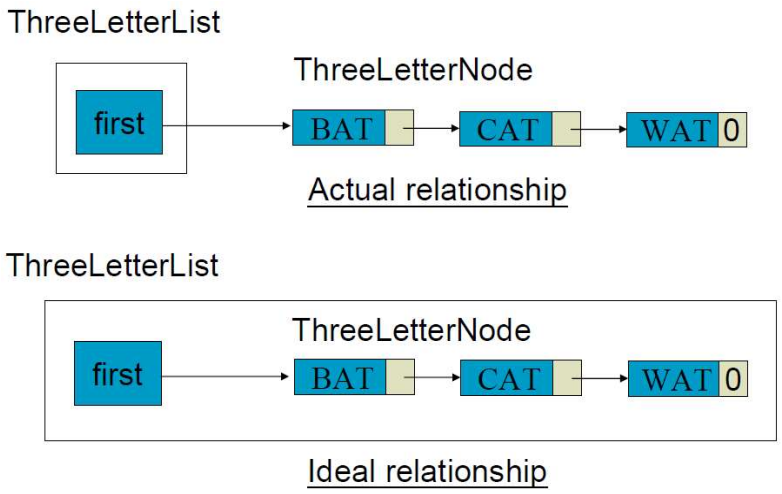
Designing a List in C++

- Design Attempt 3:
  - Use of two classes.
  - Create a class that represents the linked list.
  - The class contains the items (list nodes) of another objects of another class.
- A data object of Type A HAS-A data object of Type B if A conceptually contains B or B is a part of A
  - Computer HAS-A Processor

Composite Classes (initial design)

```
// forward delcarion
class ThreeLetterList;

class ThreeLetterNode {
  friend class ThreeLetterList;
  private:  // Node data
    char data[3];
    ThreeLetterNode * link;
};

class ThreeLetterList {
  public:
  // List Manipulation operations
  ...
  private:
    ThreeLetterNode *first;
};
```

Composite Classes



Nested Classes (safer design)

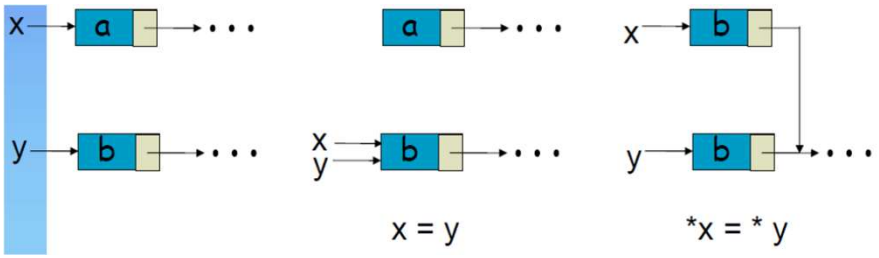
- Nested classes
  - One class is defined inside the definition of another.
- Class *ThreeLetterNode* is defined **inside** the **private** portion of the definition of class *ThreeLetterList*
  - This ensures that *ThreeLetterNode* objects cannot be accessed outside class *ThreeLetterList*

Nested Classes

```
class ThreeLetterList {
public:
  // List Manipulation operations
.
.
private:
  // nested class
  class ThreeLetterNode {
    public:
      char data[3];
      ThreeLetterNode *link;
  };
  ThreeLetterNode *first;
};
```

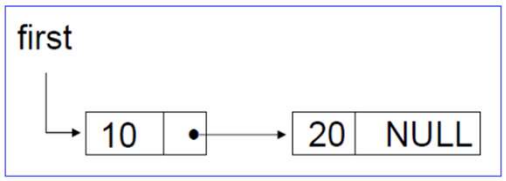

Pointer Manipulation in C++

- Two pointer variables of the same type can be compared.
  - x == y, x != y, x == 0



Create a Two-Node List

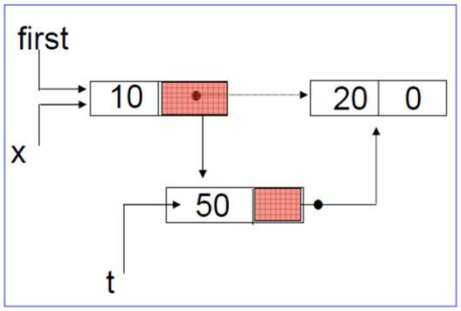
```
void List::Create2( )
{
  // create a linked list with two nodes
  first = new ListNode(10);
  first->link = new ListNode(20);
}
ListNode::ListNode(int element=0)
{
  data=element;
  link=0;
}
```



List Insertion

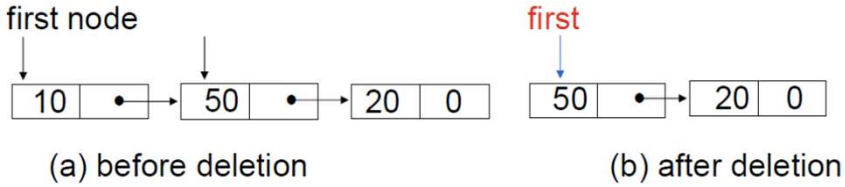
- Insert a node after a specific node

```
void List::Insert50(ListNode *x)  //insert after x
{
  // insert a new node with data=50 into the list
  ListNode *t=new ListNode(50);
  if (!first) { // empty list
    first=t;
    return;
  }
  //insert after x
  t->link=x->link;
  x->link=t;
}
```

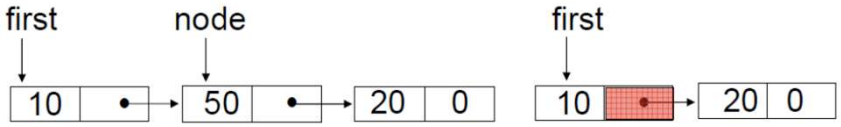


List Deletion

- Delete the first node (first = first->link)



- Delete node other than the first node.



List Iterator

- A list iterator is an object that is used to traverse all elements of a container class.
- `ListIterator<Type>` is declared as a **friend** of both **`List<Type>`** and **`ListNode<Type>`**.
- A `ListIterator<Type>` object is initialized with the name of a `List<Type>` object *L* with which it will be associated.
- The `ListIterator<Type>` object contains a private data member *`current`* of type `ListNode<Type>*`. At all times, *`current`* points to a node of list *L*.
- The `ListIterator<Type>` object defines public member functions *`NotNull()`*, *`NextNotNull()`*, *`First()`*, and *`Next()`* to perform various tests on and to retrieve elements of list *L*.

Template of Linked List

```
enum Boolean { FALSE, TRUE };
template <class Type> class List;
template <class Type> class ListIterator;
template <class Type> class ListNode {
  friend class List<Type>;
  friend class ListIterator<Type>;
  private:
    Type data;
    ListNode *link;
};
```


Template of Linked List

```
template <class Type> class List {
  friend class ListIterator<Type>;
  public:
    List() {first = 0;};
    // List manipulation operations
    ..
  private:
    ListNode<Type> *first, *last;
};
```

Template of ListIterator

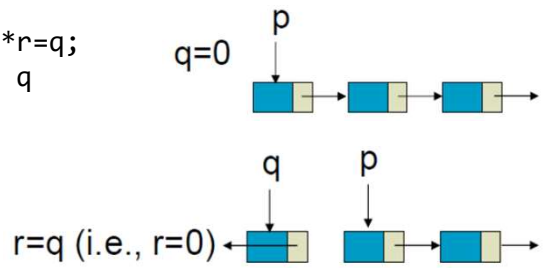
```
template <class Type> class ListIterator {
  public:
    ListIterator(const List<Type> &l):
                 list(l), current(l.first)
    {};
    Boolean NotNull();
    Boolean NextNotNull();
    Type * First();
    Type * Next();
  private:
    const List<Type>& list; // refers to a list
    ListNode<Type>* current; // current node in list
};
```


Attaching a Node at the End

```
Template <class Type>
void List<Type>::Attach(Type k)
{
  ListNode<Type> *newnode =
                        new ListNode<Type>(k);
  if (first == 0)
    first = last = newnode;
  else {
    last->link = newnode;
    last = newnode;
  }
};
```

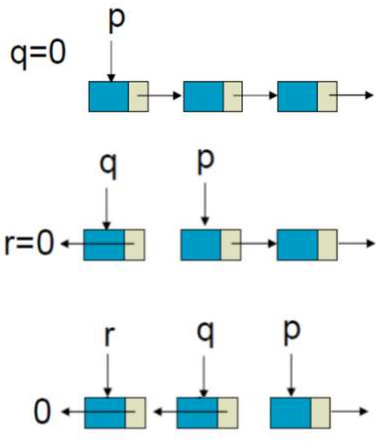
Inverting a List

```
template <class Type>
void List<Type>:: Invert()
// A chain x is inverted so that if x=(a1,…an)
// then, after execution, x=(an,…,a1)
{
  ListNode<Type>*p = first,*q=0; //q trails p
  while (p)
  {
    ListNode<Type> *r=q;
    q=p; //r trails q
    p=p->link;
    q->link=r;
  }
  first=q;
};
```



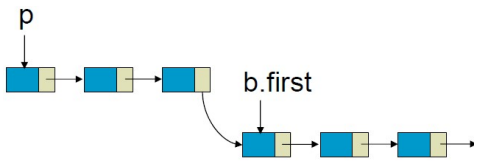
Inverting a List (visually)

- Initial case
- After 1st iteration
- After 2nd iteration
- …
- At the end of the list ?



Concatenating Two Lists

```
Template <class Type>
void List<Type>:: Concatenate(List<Type> b)
// this = (a1, …, am) and b = (b1, …, bn) m, n ≥ ,
// produces the new chain z = (a1, …, am, b1, bn) in this.
{
  if (!first) {
    first = b.first;
    return;
  }
  if (b.first) {
    for (ListNode<Type> *p = first; p->link; p = p->link);
    // no body
    p->link = b.first;
  }
}
```

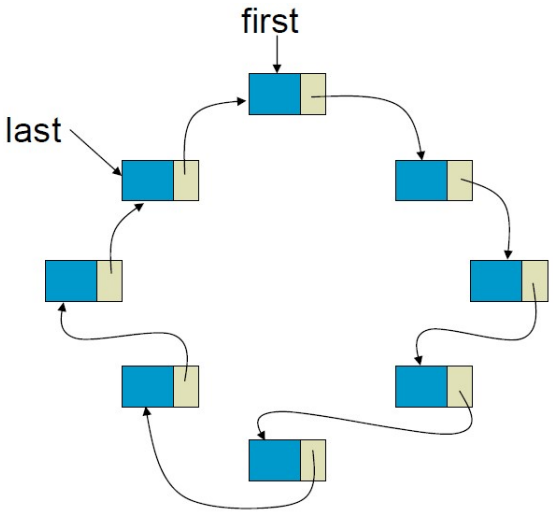


first!=null && b!=null

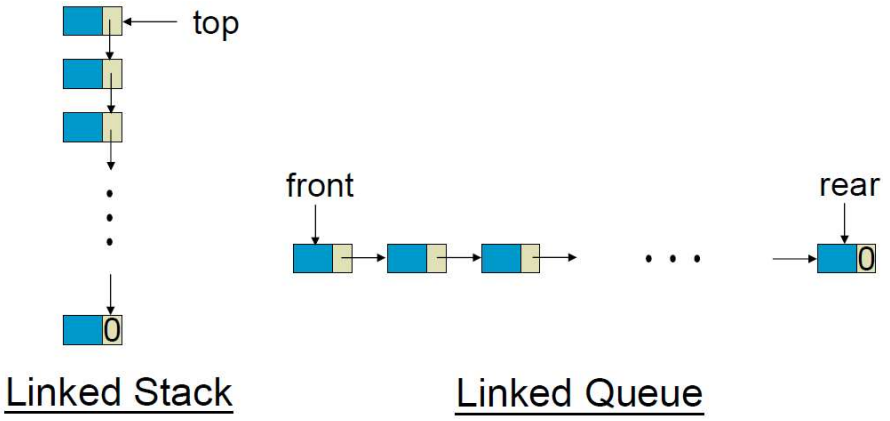
List Destructor

```
Template <class Type>
List<Type>:: ~List()
// Free all nodes in the chain
{
  ListNode<Type>* next;
  for (; first; first = next) {
    next = first->link;
    delete first;
  }
}
```

Diagram of a Circular List

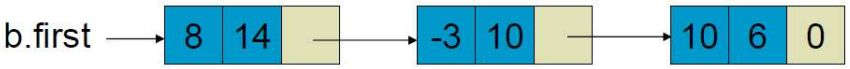


Linked Stacks and Queues



Revisit Polynomials

$$a = 3x^{14} + 2x^{8} + 1$$



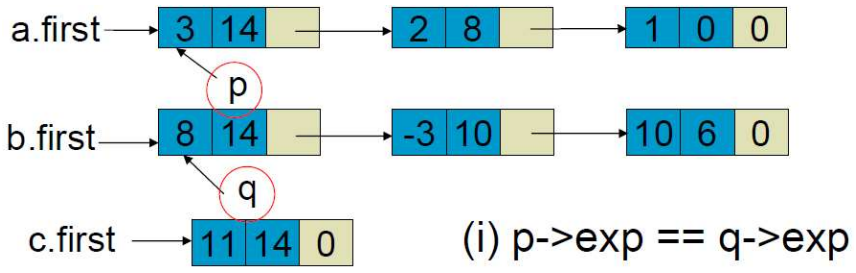
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

Polynomial Class Definition

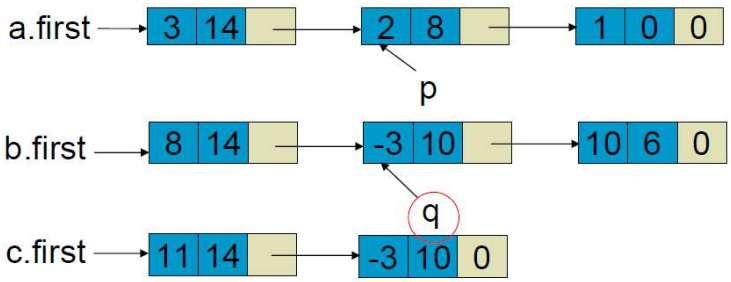
```
struct Term
/* all members of Terms are public by default */
{
  int coef; // coefficient
  int exp; // exponent
  void Init(int c, int e)
  {coef = c; exp = e;};
};

class Polynomial
{
  friend Polynomial operator+(const Polynomial&,
                              const Polynomial&);
  private:
    List<Term> poly;
};
```

Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting.
  - E.g., adding two polynomials *a* and *b*

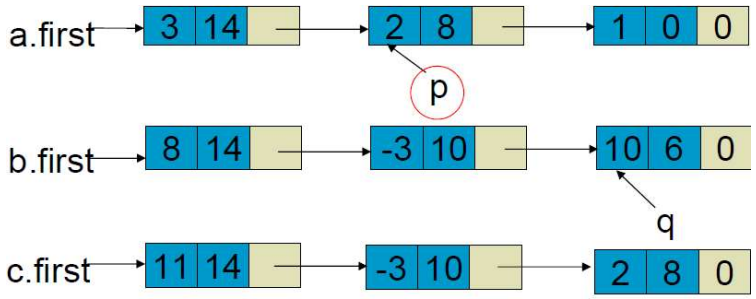


Operating on Polynomials



(ii) p->exp < q->exp

Operating on Polynomials



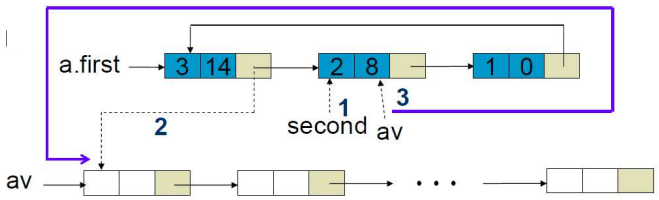
(iii) p->exp > q->exp

Free Pool (of Items)

- When items are created and deleted constantly, it is more efficient to have a circular list to contain all available items.
  - To reduce the times of creating and deleting objects
- When an item is needed, the free pool is checked to see if there is any item available. If yes, then an item is retrieved and assigned for use.
- If the list is empty, then either we stop allocating new items or use **new** to create more items for use.

Template of Circular List

```
template <class Type>
void CircList<Type>:: ~CircList()
// Erase the circular list pointed to by first and add to av
{
  if ( first )
  {
    ListNode* second=first->link; //(1)
    first->link=av; //(2)
    av=second; //(3)
    first=0; //(4)
  }
}
```

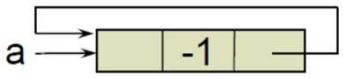



```
template < class Type>
ListNode <Type>* CircList<Type>::GetNode()
// Provide a node in free pool for use
{
  ListNode <Type> *x;
  if ( !av )  // No more free node
    x = new ListNode<Type>;
  else {
    x = av;
    av = av->link;
  }
  return x;
}
```

```
template <class Type>
Void CircList<Type>::RetNode(
ListNode<Type> *x)
//Free the node pointed to by x
{
  x->link = av;
  av = x;
}
```

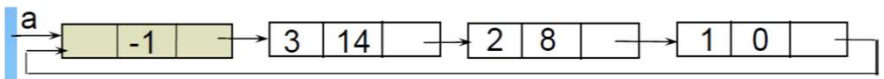

Head Node

- Represent polynomial as circular list
  - Zero



Zero polynomial

  - Others



$$a = 3x^{14} + 2x^{8} + 1$$

Equivalence Relation

- For an arbitrary relation by the symbol ≡
- Reflexive
  - If x ≡ x
- Symmetric
  - If x ≡ y, then y ≡ x
- Transitive
  - If x ≡ y and y ≡ z, then x ≡ z
- A relation over a set, *S*, is said to be an equivalence relation over *S* iff it is symmetric, reflexive, and transitive over *S*.

Examples

- The “equal to” (=) relationship is an equivalence relation since
  - x = x
  - x = y implies y = x
  - x = y and y = z implies x = z
- An equivalence relation is to partition the set *S* into equivalence classes such that two members x and y of *S* are in the same equivalence class iff x ≡ y

Examples

- Treating numbers as symbols
- $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$,
- $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$
- Results in three equivalent classes

  {0,2,4,7,11}; {1,3,5}; {6,8,9,10}

A Rough Algorithm to Find Equivalence Classes

```
void equivalence()
{
   initialize;
   while (there are more pairs) {
      read the next pair <i,j>;
      process this pair;
   }
   initialize the output;
   do {
      output a new equivalence class;
   } while (not done);
}
```

Phase 1

Phase 2

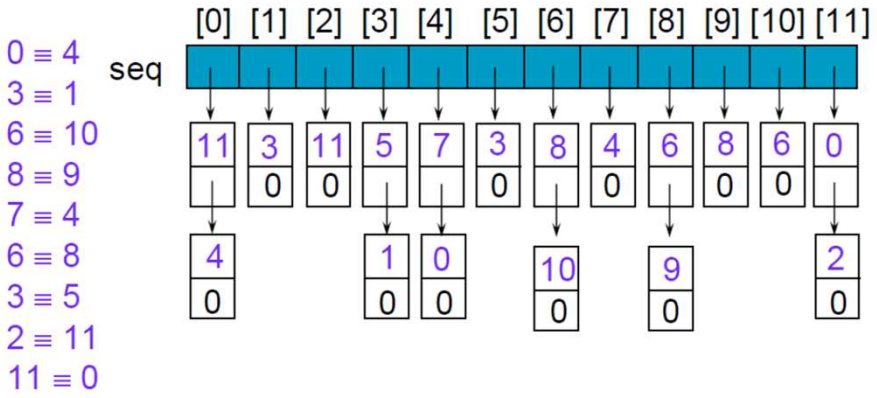
What kinds of data structures to adopt?

```
void equivalence()
{
   read n; // read in number of objects
   initialize seq to 0 and out to FASLE;
   while more pairs // input pairs
   {
      read the next pair (i,j);
      insert j on seq[i] list;
      insert i on seq[j] list;
   }
   for( i = 0; i < n; i++ )
      if ( out[i] == FALSE) {
         out[i] = TRUE;
         output the equivalence class that contains object i
      }
} // end of equivalence
```

direct equivalence

Compute indirect equivalence using transitivity

Lists after pairs have been input



```
enum Boolean { FALSE, TRUE };
class ListNode {
  friend void equivalence();
  private:
    int data;
    ListNode *link;
    ListNode(int);
};
typedef ListNode *ListNodePrt;
/* so we can create an array of pointers
using new */
ListNode::ListNode(int d){
  data = d;
  link = 0;
}
```



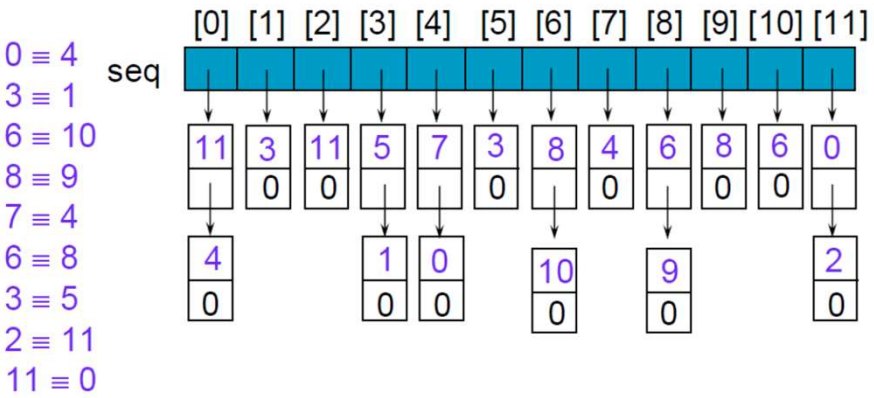
```
void equivalence()
/* Input the equivalence pairs and
output the equivalence classes */
{
  //"equiv.in" is the input file
  ifstream inFile("equiv.in", ios::in);
  if (!inFile) {
    cerr << "Cannot open file " << endl;
    return;
  }
```

```
int i, j, n;
inFile >> n; // read no. of objects
// initialize seq and out
ListNodePtr *seq = new ListNodePtr[n];
Boolean *out = new Boolean[n];
for (i=0; i < n ; i++) {
  seq[i] = 0;
  out[i] = FALSE;
}
```



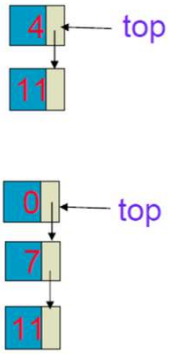
```
// Phase 1: input equivalence pairs
inFile >> i >> j ;
while(inFile.good()) { // check EOF
  ListNode *x = new ListNode(j);
  x->link = seq[i];
  seq[i] = x; // add j to seq[i]
  ListNode *y = new ListNode(i);
  y->link = seq[j];
  seq[j] = y; // add i to seq[j]
  inFile >> i >> j;
}
```

Lists after pairs have been input




```
// Phase 2: output equivalence classes
for( i = 0; i < n; i++)
  if( out[i]==FALSE ){ // needs to be output
    cout << endl << "A new class: " << i;
    out[i] = TRUE;
    ListNode *x = seq[i];
    ListNode *top = 0; //init stack
    while(1){ // find rest of class
      while(x){ // process the list
        j = x->data;
        if( out[j]==FALSE ) {
          cout << "," << j;
          out[j] = TRUE;
          ListNode *y = x->link; //next node
```

```
            x->link = top;
            top = x;
            x = y;
          }
          else x = x->link;
        } // end of while(x)
        if( !top ) break;
        else {
          x = seq[top->data];
          top = top->link; // unstack
        }
      } // end of while(1)
    } // end of if( out[i]==FALSE )
```




```
  for( i = 0; i < n; i++ )
    while( seq[i] ) {
      ListNode *delnode = seq[i];
      seq[i] = delnode->link;
      delete delnode;
    }
  }
  delete [] seq;
  delete [] out;
} // end of equivalence
```

Output of Previous Example

```
A new class: 0, 11, 4, 7, 2
A new class: 1, 3, 5
A new class: 6, 8, 10, 9
```

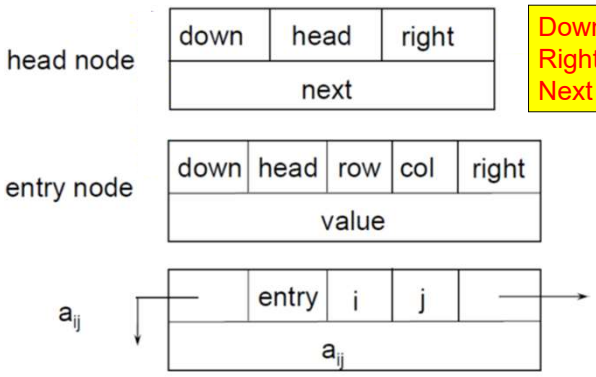

Sparse Matrices

- Inadequates of sequential schemes
  - # of nonzero terms will vary after some matrix computation
  - Matrix just represents intermediate results
- New scheme
  - Each column (or row): a circular linked list with a head node

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

Sparse Matrices Representation

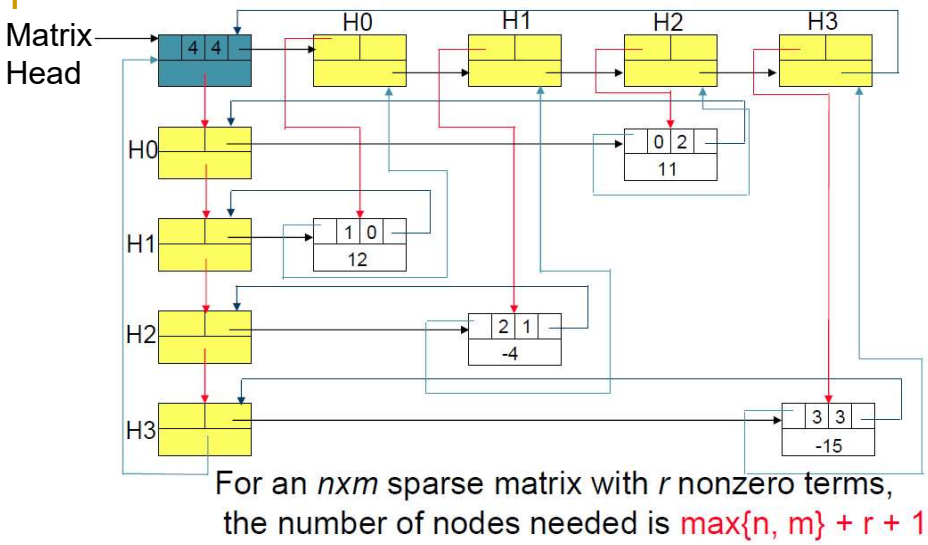
- # of head nodes = max{# of rows, # of columns}
- The field head is used to distinguish between head nodes and entry nodes



Down to link into column list
Right to link into row list
Next links head nodes

Head Nodes of Sparse Matrices

- Each head node is in three lists:
  - A list of rows,
  - A list of columns,
  - A list of head nodes
- The list of head nodes also has a head node (the matrix head) which is in entry node structure and the row and column fields of this node is used to store matrix dimensions.



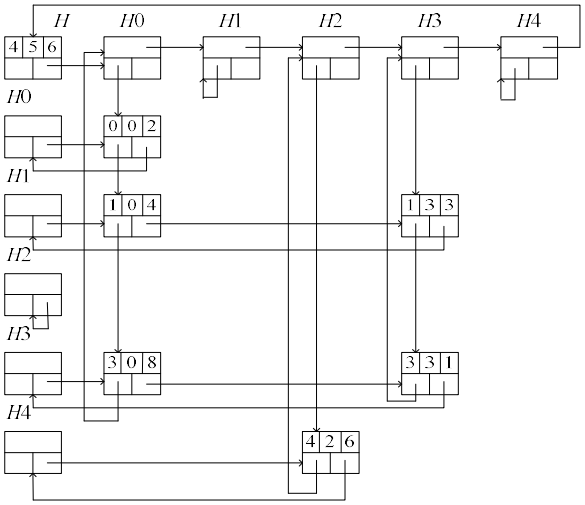
For an *nxm* sparse matrix with *r* nonzero terms, the number of nodes needed is max{n, m} + r + 1.

The head node for row$_i$ is also head node for column$_i$

Another Example

- A 5x4 sparse matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$



```
enum Boolean { FALSE, TRUE };
struct Triple { int value, row, col ; };
class Matrix ; //forward declaration
class MatrixNode {
  friend class Matrix ;
  //for reading in a matrix
  friend istream& operator>>(istream&, Matrix&) ;
  private:
    MatrixNode *down, *right ;
    Boolean head ;
    union { //anonymous union
      MatrixNode *next ; // when it is a head
      Triple triple ;    // when it is an entry
    };
  MatrixNode(Boolean, Triple *) ; //constructor
};
```



```
typedef MatrixNode * MatrixNodePtr ;
// to allow subsequent creation of
// array of pointers
class Matrix{
  friend istream& operator>>(istream&,
Matrix&) ;
  public:
    ~Matrix() ; //destructor
  private:
    MatrixNode *headnode ;
};
```

```
MatrixNode::MatrixNode(Boolean b, Triple *t)
//constructor
{
  head = b ;
  if (b) {
    // row/column head node
    right = next = down = this;
  }
  else
    // head node for list of headnodes OR
    // element node
    triple = *t ;
}
```


Sparse Matrix Example

	[0]	[1]	[2]
[0]	4	4	4
[1]	0	2	11
[2]	1	0	12
[3]	2	1	-4
[4]	3	3	-15

Sample input for sparse matrix

Doubly Linked Lists

- Move in forward and backward direction.
  - Singly linked list in one direction only
- How to get the preceding node during deletion or insertion?
  - Using 2 pointers
- Node in doubly linked list
  - left link field (llink)
  - data field (item)
  - right link field (rlink)

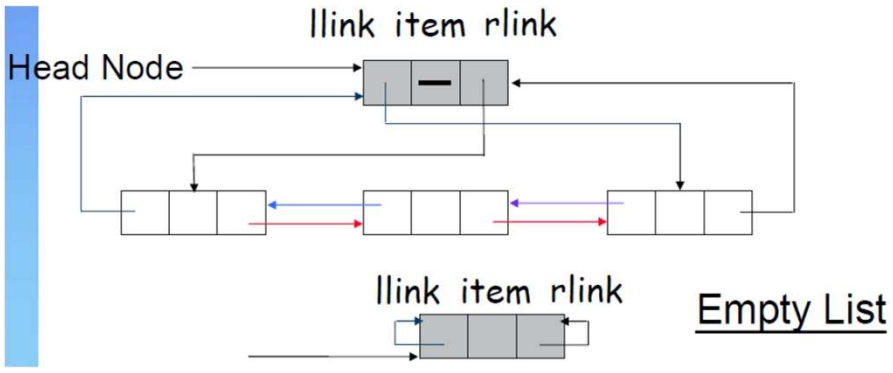
Doubly vs. Singly Linked List

- Can operate on both ends
- Need extra space for additional pointers
- Insertion/Deletion need extra work
- Node deletion requires no additional pointers
- Can work as a Queue and a Stack at the same time. (How?)

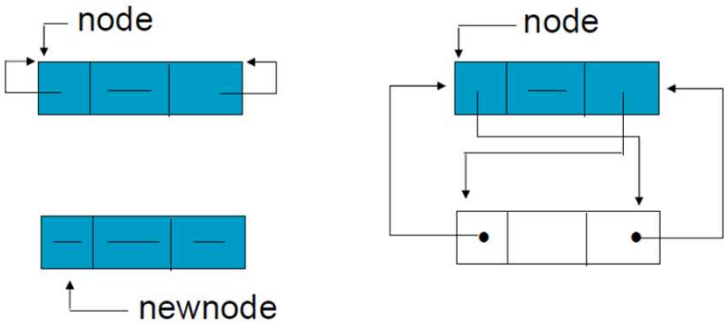
```
class DblList ;
class DblListNode {
  friend class DblList ;
  private:
    int data ;
    DblListNode *llink, *rlink ;
};
class DblList {
  public:
    //List manipulation operations
  private:
    //points to head node
    DblListNode *head ;
};
```


Doubly Linked Lists

- A head node is also used in a doubly linked list to allow us to implement our operations more easily.

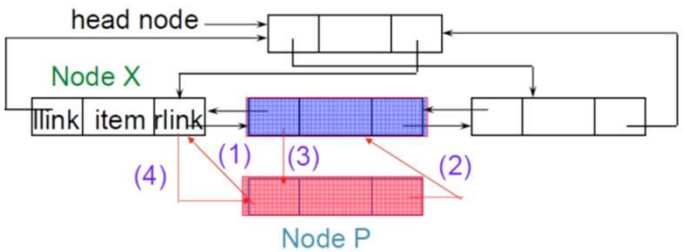


Insertion



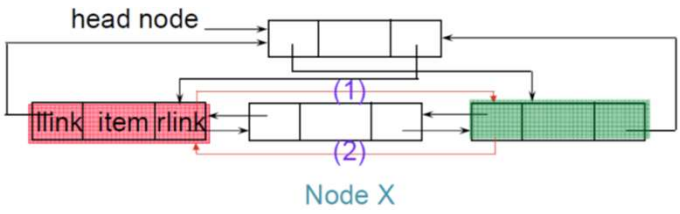
Insertion into an empty doubly linked circular list


```
void DblList::Insert(DblListNode *p,
DblListNode *x)
//insert node p to the right of node x
{
  p->llink = x ; //(1)
  p->rlink = x->rlink ; //(2)
  x->rlink->llink = p ; //(3)
  x->rlink = p ; //(4)
}
```



Deletion

```
void DblList::Delete(DblListNode *x) {
  if(x == first)
    cerr << "Deletion of head node not permitted"
         << endl;
  else {
    x->llink->rlink = x->rlink; //(1)
    x->rlink->llink = x->llink; //(2)
    delete x;
  }
}
```



Reverse

- Reverse a doubly linked list is a good exercise to test your understanding of the structure.

```
void DblList::Reverse() {
  DblListNode *temp = NULL;
  DblListNode *current = ...; //exercise!
  //swap llink and rlink for all nodes
  while (current != ...) {
    temp = current->llink;
    current->llink = current->rlink;
    current->rlink = temp;
    current = current->llink;
  }
  ... //exercise!
}
```

Generalized Lists

- A generalized list is a finite sequence of n elements (a_0, … a_{n-1}) where a_i is either an atom or a list.
- The elements that are not atoms are said to be sublists.
- Self-study.

Inventor of Linked List

- **1953**: Idea of Linked List was published by Hans Peter Luhn from IBM. (also the inventor of Hash Map)
- **1955**: Linked List was implemented in a production software by Allen Newell, Cliff Shaw and Herbert Simon from RAND Corporation.
- The 4 people above are credited as the inventors of Linked List.
- **1958**: LISP was developed by John McCarthy at MIT. Linked List was a major component of LISP design.

Advantages & Disadvantages

- **Advantages**:
  - Dynamic: Linked lists can change size dynamically.
  - Effective insertion and deletion: Inserting or removing elements cab be done quickly and efficiently**. O(1)**
  - Memory efficiency: Linked lists don't require contiguous memory allocation.
  - Flexibility: Linked lists offer a lot of versatility.
- **Disadvantages**:
  - Sequential access: Linked lists have poor cache locality which results in significant overhead.
  - Absence of random access: Accessing entries directly from an index is impossible.
  - Complexity: The implementation of linked lists can be more difficult than those of arrays.

Applications of Linked Lists

- Implementation of other Data Structures**:** Many data structures can be implemented using the linked lists.
- Memory Management**:** Linked Lists can be used in memory management systems for allocating and reallocating memory.
- File Systems**:** File systems can be represented using linked lists. A node represents a file or directory; the links signify the parent-child relationships between the files and directories.
- Graphs and Charts**:** Graphs can be represented by Linked Lists, where nodes are vertices and the links are edges.
- Making music playlists**:** Linked List are frequently used to build music playlists. A node represents a song, and the list indicate the order in which the songs are played.
- Picture Processing Method**:** Picture processing methods can be implemented using linked lists**,** where a node represents each pixel.
- …