APA 254
Data Structures

Lecture 3.2
(LL Representation)

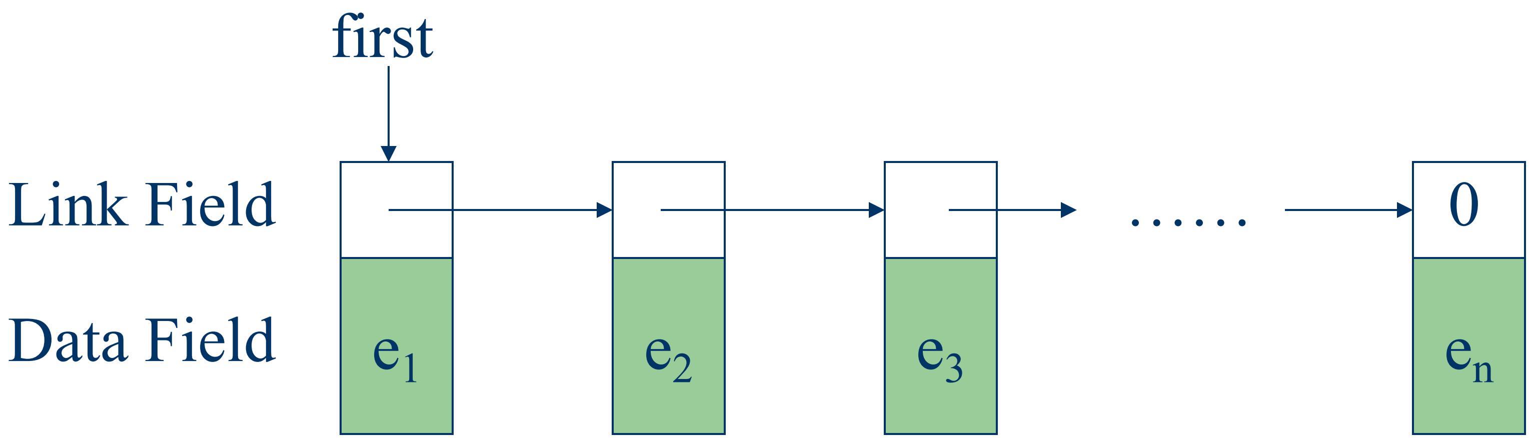
Dept. of Information System
Hanyang University

Linked Representation of Linear List

- Each element is represented in a cell or node.
- Each node keeps explicit information about the location of other relevant nodes.
- This explicit information about the location of another node is called a link or pointer.

Singly Linked List

- Let $L = (e_1, e_2, \ldots, e_n)$
  - Each element e_i is represented in a separate node
  - Each node has exactly one link field that is used to locate the next element in the linear list
  - The last node, e_n, has no node to link to and so its link field is NULL.
- This structure is also called a chain.



Class ‘ChainNode’

```
template <class T>
class ChainNode {
    friend Chain<T>;
    private:
        T data;
        ChainNode<T> *link;
};
```

- Since Chain<T> is a friend of ChainNode<T>, Chain<T> has access to all members (including private members) of ChainNode<T>.

Class ‘Chain’

```
template <class T>
class Chain {
public:         Chain()
                ~Chain();
                bool isEmpty() const
                int Length() const;
                bool Find(int k, T& x) const;
                int Search(const T& x) const;
                Chain<T>& Delete(int k, T& x);
                Chain<T>& Insert(int k, const T& x);
                void Output(ostream& out) const;
private:        ChainNode<T> *first;
                int listSize;
};
```

Operation '~Chain'

```
template <class T>
Chain<T>::~Chain()
{
   ChainNode<T> *next;
   while (first) {
         next = first->link;
         delete first;
         first = next;
   }
}
```

- The time complexity is $\Theta(n)$, where n is the length of the chain.

Operation ‘Length’

```
template <class T>
int Chain<T>::Length() const
{
    ChainNode<T> *current = first;
    int len = 0;
    while (current) {
        len++;
        current = current->link;
    }
    return len;
}
```

- The time complexity is $\Theta(n)$, where n is the length of the chain.

Operation ‘Find’

```
template <class T>
bool Chain<T>::Find(int k, T& x) const
{// Set x to the k’th element in the list if it exists
 // Throw illegal index exception if no such
     element exists
   checkIndex(k);

   // move to desired node
   ChainNode<T>* current = first;
   for (int i = 0; i < k; i++)
      current = current->link;
   x = current->data;
   return true;
}
```

- The time complexity is **O(k)**
- Exercise – write the code for checkIndex() operation & determine its time complexity

Operation ‘checkIndex’

```
template<class T>
void Chain<T>::checkIndex(int Index) const
{  // Verify that Index is between 0 and listSize-1.
   if (Index < 0 || Index >= listSize)
   {ostringstream s;
    s << "index = " << Index << " size = "<< listSize;
    throw illegalIndex(s.str());
   }
}
```

- The time complexity is
  O(1)

Operation 'Search'

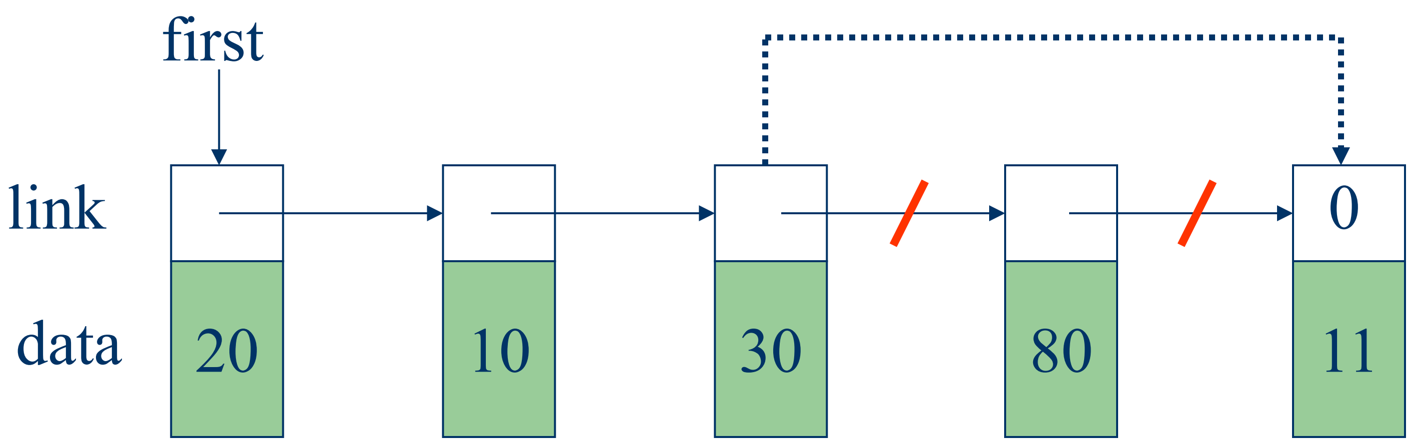
```
template <class T>
int Chain<T>::Search(const T& x) const
{// Locate x and return its position if found else return -1

   // search the chain for x
   ChainNode<T>* current = first;
   int index = 0;  // index of current
   while (current != NULL && current->data != x)
   {
      // move to next node
      current = current->link;
      index++;
   }

   // make sure we found matching element
   if (current == NULL)
      return -1;
   else
      return index;
}
```

- The time complexity is

O(n)

Operation 'Delete'

- To delete the fourth element from the chain, we
  - locate the third and fourth nodes
  - link the third node to the fifth
  - free the fourth node so that it becomes available for reuse



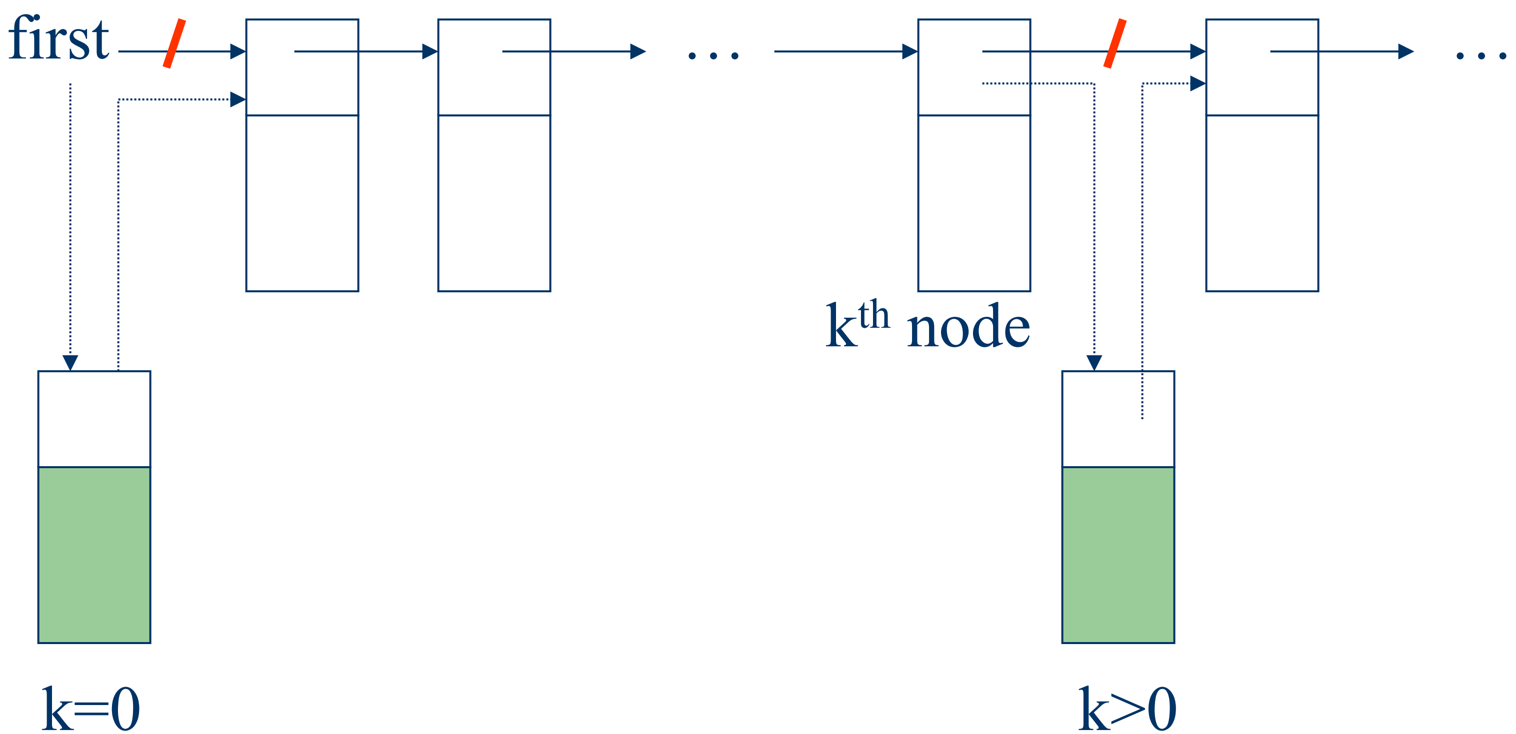
Operation ‘Delete’

- See Program 6.7 for deleting a node from a chain
- The time complexity is

  O(theIndex)

Operation ‘Insert’

- To insert an element following the *k*th in chain, we
  - locate the *k*th node
  - new node’s link points to *k*th node’s link
  - *k*th node’s link now points to the new node



Operation ‘Insert’

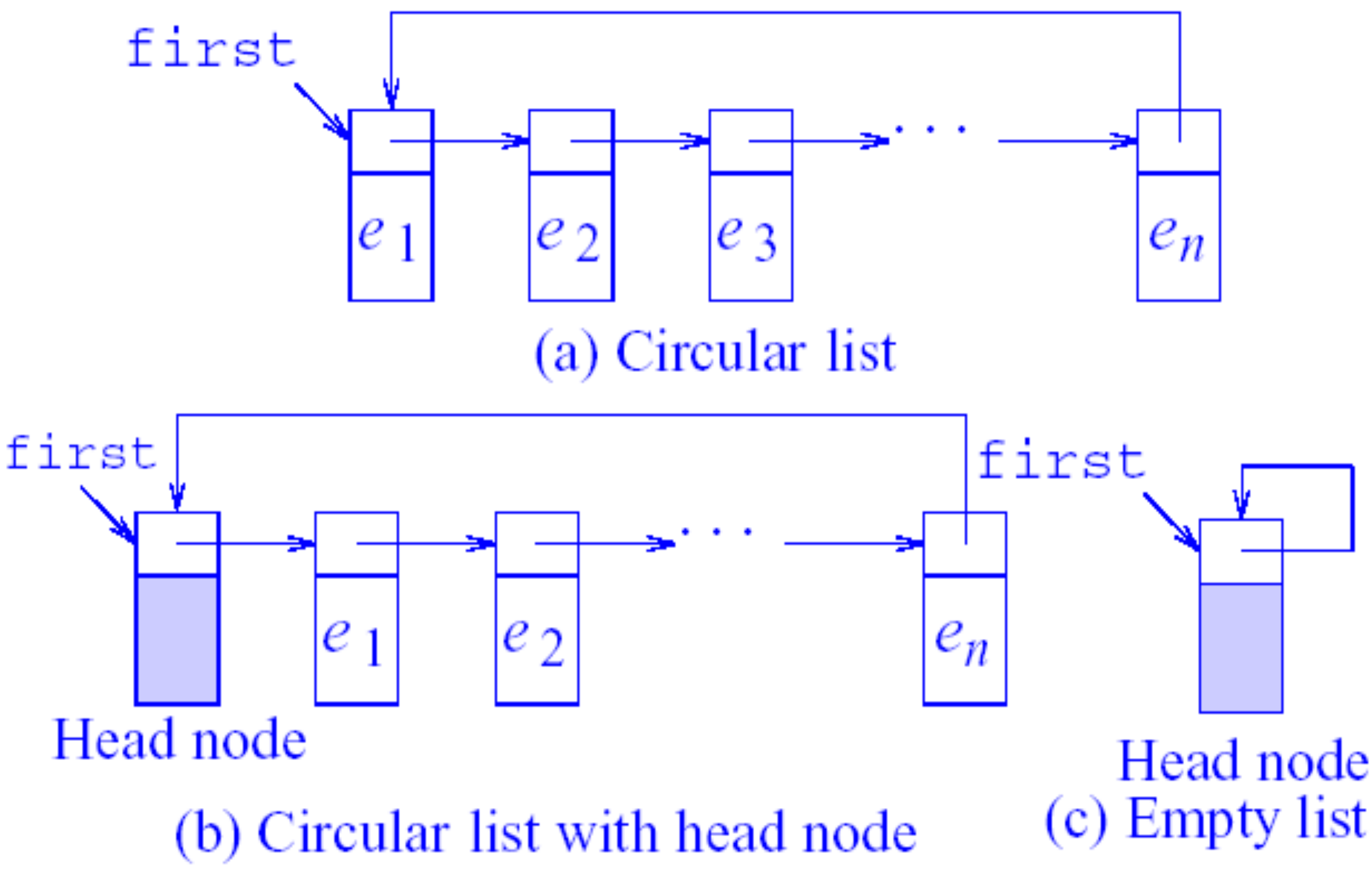
- See Program 6.8 for inserting a node into a chain
- The time complexity is

  O(theIndex)

Circular List Representation

- Programs that use chains can be simplified or run faster by doing one or both of the following:
  1. Represent the linear list as a singly linked circular list (or simply **circular list**) rather than as a chain
  2. Add an additional node, called the **head node**, at the front

Circular List Representation



Circular List Representation

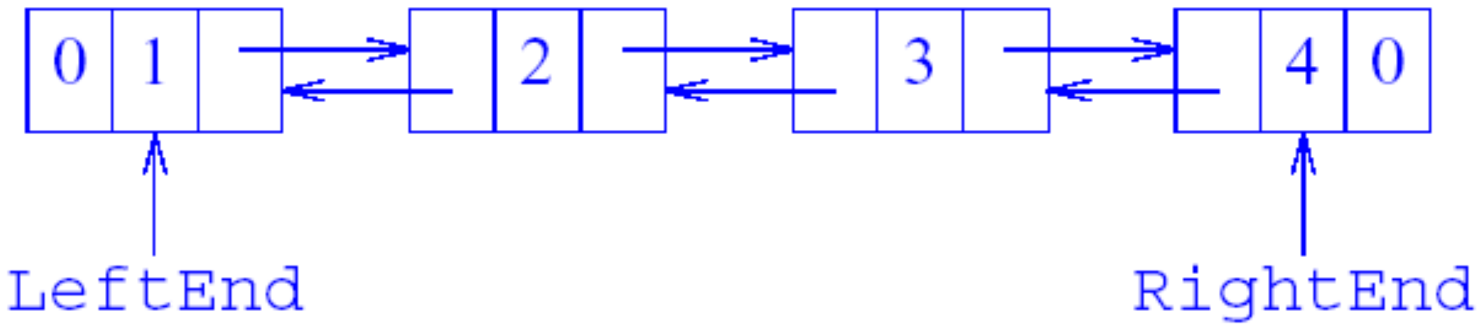
- Why better (run faster) than linear list?
  - Requires fewer comparisons: compare the following two programs

```
template<class T>
int Chain<T>::Search(const T& x) const
{
   ChainNode<T> *current = first;
   int index = 1;  // index of current
   while (current && current->data != x) {
      current = current->link;
      index++;
      }
   if (current) return index;
   return 0;
}
```

```
template<class T>
int CircularList<T>::Search(const T& x) const
{
   ChainNode<T> *current = first->link;
   int index = 1; // index of current
   first->data = x; // put x in head node
   while (current->data != x) {
      current = current->link;
      index++;
      }
   // are we at head?
   return ((current == first) 0 : index);
}
```

Doubly Linked List Representation

- An ordered sequence of nodes in which each node has two pointers: left and right.



Class ‘DoubleNode’

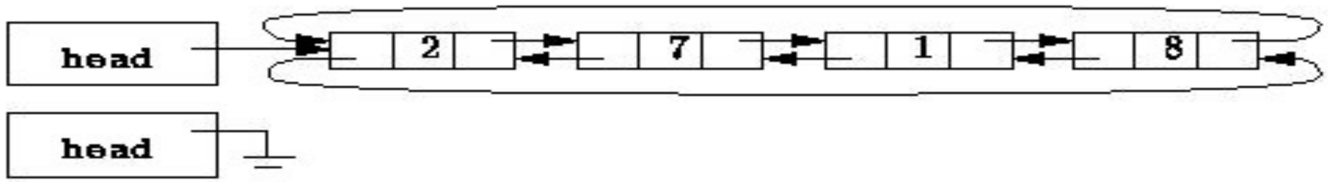
```
template <class T>
class DoubleNode {
    friend Double<T>;
    private:
        T data;
        DoubleNode<T> *left, *right;
};
```

Class ‘Double’

```
template <class T>
class Double {
public:       Double() { LeftEnd = RightEnd = 0; };
              ~Double();
              int Length() const;
              bool Find(int k, T& x) const;
              int Search(const T& x) const;
              Double<T>& Delete(int k, T& x);
              Double<T>& Insert(int k, const T& x);
              void Output(ostream& out) const;
private:      DoubleNode<T> *LeftEnd, *RightEnd;
};
```

Circular Doubly Linked List

- Add a head node at the left and/or right ends
- In a non-empty circular doubly linked list:
  - LeftEnd->left is a pointer to the right-most node (i.e., it equals RightEnd)
  - RightEnd->right is a pointer to the left-most node (i.e., it equals LeftEnd)
- Can you draw a circular doubly linked list with a head at the left end only by modifying Figure 6.7?



- READ Sections 6.1~6.4