EECS 560
DATA STRUCTURES

MODULE I: VECTOR AND LIST

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.

- This document is a copyright of Dr. Cuncong Zhong. Distribution of this document, or use it for any purpose other than what is stated above, is considered as a copyright infringement. Dr. Cuncong Zhong reserves the right to take necessary legal action.

- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Data Structures and Algorithm Analysis in C++***, 4th edition, by Mark Allen Weiss.

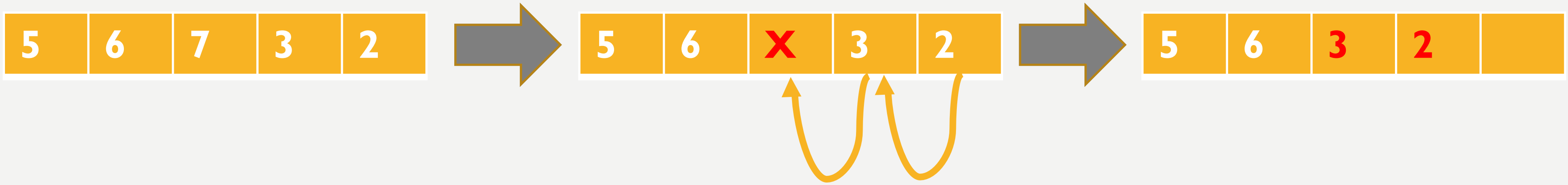
LIST

- List is an ADT that stores a list of elements
- Data
  - Array
  - Linked list
- Methods
  - Insert(x, i): inserts an element x into the ith position of the list
  - Delete(i): deletes the ith element from the list
  - Find(x, i): returns the position of the first occurrence of x in the list after i
  - At(i): returns the ith element
  - Size(): returns the number of the elements in the list
  - Reserve(n): reserves space for n elements

LIST: ARRAY

- The simplest implementation of the list ADT is to use array.

- One potential issue is that the array is fix-sized, while we may be having a large list to store that is beyond the capacity of the array.
- The second issue is that when we insert or delete an element, we may also need to move a large number of elements



LIST: ARRAY CONT.

- Insert(x, i): this operation will first locate move every element after the i-1th (move the nth first, followed by the n-1th, …, at last move the ith one) one position after to reserve the space for x, and insert x at the ith position. In the worst-case scenario (when i=0), we may need to move every element, and the time complexity is O(n).
- Delete(x,i): similar to Insert(), it also requires O(n) time.
- Find(x, i): we will simply scan through the array after position i until we find x. This operation costs O(n) time.
- At(i): because we use an array, the ith element can be immediately retrieved. This operation costs O(1) time (a constant time).
- Size(): we can simply use a counter to store the number of elements. Updating and retrieving the counter both take O(1) time.
- Reserve(n): when overflow, we need to find a consecutive space of n to relocate the information. This could lead to a time complexity of O(n) in the worst case scenario.
- PrintAll(): printing all elements of the list. Given the pre-fetch technique, this can be very efficient with array, although it still takes O(n) time.

LIST: ARRAY CONT.

- Good at quickly retrieving the ith element and printing all elements
- Not so good at insertion, deletion, and information relocation

LIST: LINKED LIST

- A different data type to store a list
- Each element is implemented as a node, and the nodes are connected through pointers

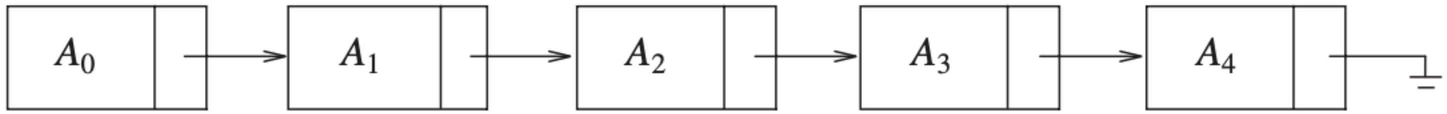


Figure 3.1 A linked list

- To facilitate bi-directional traversal, we can use double-linked list

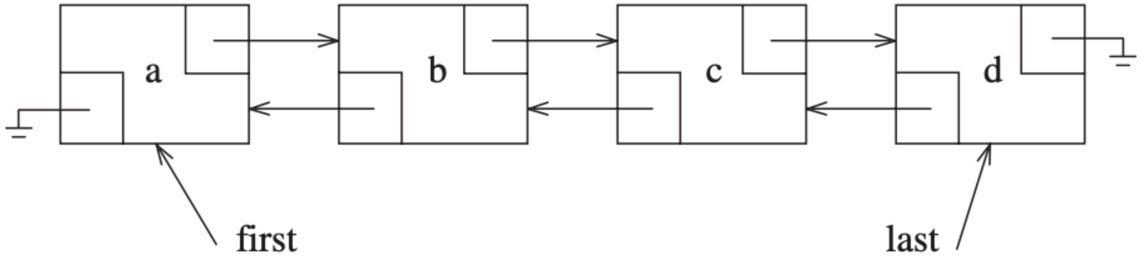


Figure 3.4 A doubly linked list

LIST: LINKED LIST CONT.

- The linked list implementation works better for insertion, deletion, and relocation.
- The main reason is that all elements no longer need to be stored in a consecutive chunk of memory space.
- However, it performs worse in retrieving the ith element and in enumerating all elements in the list.

LIST: LINKED LIST INSERTION

- Set X->next to A2
- Set A1->next to X
- O(1) time

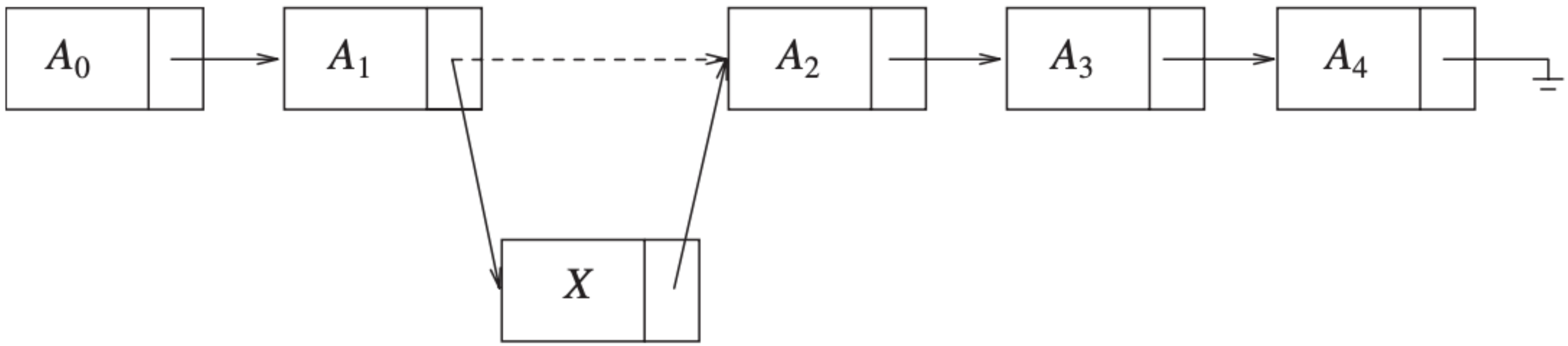


Figure 3.3 Insertion into a linked list

LIST: LINKED LIST DELETION

- Set A1->next to A2->next
- Set A2->next to NULL
- Destruct A2
- O(1) time

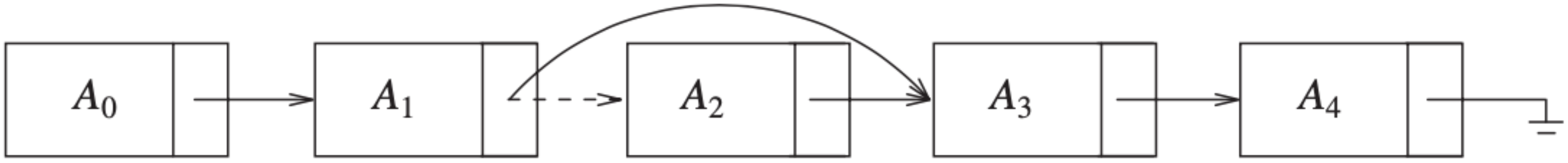


Figure 3.2 Deletion from a linked list

LIST: LINKED LIST OTHER OPERATIONS

- Two special nodes: the head (head->prev = NULL) and the tail (tail-next = NULL)
- Traversal of the entire list: start from the head and keep following the ”->next” pointer

  – Find(), At(), PrintAll() will all be addressed by a traversal of the linked list, and all of them will need O(n) time.

  – Reserve() is no longer needed for linked list, because the overflow problem does not exist.

  – Size() can be retrieved in O(1) time if we keep a counter in the ADT (similar to the array implementation)

LIST: C++ STL VECTOR IMPLEMENTATION

- C++ STL “vector”: uses array implementation
- C++ STL “list”: uses linked list implementation

- We are going to dive into the implementational details of both ADTs

LIST: C++ STL VECTOR IMPLEMENTATION (USING ARRAY)

- These methods are to be supported by all STL containers

- `int size( ) const`: returns the number of elements in the container.
- `void clear( )`: removes all elements from the container.
- `bool empty( ) const`: returns true if the container contains no elements, and false otherwise.

LIST: C++ STL VECTOR IMPLEMENTATION

- These methods are supported by both the vector ADT and the list ADT

- `void push_back( const Object & x )`: adds `x` to the end of the list.
- `void pop_back( )`: removes the object at the end of the list.
- `const Object & back( ) const`: returns the object at the end of the list (a mutator that returns a reference is also provided).
- `const Object & front( ) const`: returns the object at the front of the list (a mutator that returns a reference is also provided).

LIST: C++ STL VECTOR IMPLEMENTATION

- These methods are unique for the vector ADT

- `Object & operator[] ( int idx )`: returns the object at index `idx` in the `vector`, with no bounds-checking (an accessor that returns a constant reference is also provided).
- `Object & at( int idx )`: returns the object at index `idx` in the `vector`, with bounds-checking (an accessor that returns a constant reference is also provided).
- `int capacity( ) const`: returns the internal capacity of the `vector`. (See Section 3.4 for more details.)
- `void reserve( int newCapacity )`: sets the new capacity. If a good estimate is available, it can be used to avoid expansion of the `vector`. (See Section 3.4 for more details.)

LIST: C++ STL VECTOR IMPLEMENTATION

- The data fields

```
private:
  int theSize;            // the number of elements in the list
  int theCapacity;        // the maximum number of elements the array can hold
  Object * objects;       // the array holding the elements
```

LIST: C++ STL VECTOR IMPLEMENTATION

- The constructors

// the explicit keyword disables implicit conversion and copy-initialization

// default parameter setting

```
    explicit Vector( int initSize = 0 ) : theSize{ initSize },
          theCapacity{ initSize + SPARE_CAPACITY }
      { objects = new Object[ theCapacity ]; }

    Vector( const Vector & rhs ) : theSize{ rhs.theSize },
        theCapacity{ rhs.theCapacity }, objects{ nullptr }
    {
        objects = new Object[ theCapacity  ];
        for( int k = 0; k < theSize; ++k )
            objects[ k ] = rhs.objects[ k ];
    }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- The destructor

```
    ~Vector( )
      { delete [ ] objects; }

```

LIST: C++ STL VECTOR IMPLEMENTATION

- The copy/move operation

```
Vector & operator= ( const Vector & rhs )
{
    Vector copy = rhs;
    std::swap( *this, copy );
    return *this;
}
```

// the double ampersand indicates a reference of a rvalue, allowing the parameter to be modified

```
Vector( Vector && rhs ) : theSize{ rhs.theSize },
    theCapacity{ rhs.theCapacity }, objects{ rhs.objects }
{
    rhs.objects = nullptr;
    rhs.theSize = 0;
    rhs.theCapacity = 0;
}
```

```
Vector & operator= ( Vector && rhs )
{
    std::swap( theSize, rhs.theSize );
    std::swap( theCapacity, rhs.theCapacity );
    std::swap( objects, rhs.objects );

    return *this;
}
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Resize and reserve

```
        void resize( int newSize )
        {
            if( newSize > theCapacity )
                reserve( newSize * 2 );
            theSize = newSize;
        }
```

```
        void reserve( int newCapacity )
        {
            if( newCapacity < theSize )
                return;

            Object *newArray = new Object[ newCapacity ];
            for( int k = 0; k < theSize; ++k )
                newArray[ k ] = std::move( objects[ k ] );

            theCapacity = newCapacity;
            std::swap( objects, newArray );
            delete [ ] newArray;
        }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Retrieve an element

```
    Object & operator[]( int index )
      { return objects[ index ]; }
    const Object & operator[]( int index ) const
      { return objects[ index ]; }
```

- Check size and capacity

```
    bool empty( ) const
      { return size( ) == 0; }
    int size( ) const
      { return theSize; }
    int capacity( ) const
      { return theCapacity; }
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Insert/delete/retrieve the last element

```
void push_back( const Object & x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = x;
}

void push_back( Object && x )
{
    if( theSize == theCapacity )
        reserve( 2 * theCapacity + 1 );
    objects[ theSize++ ] = std::move( x );
}
```

```
void pop_back( )
{
    --theSize;
}

const Object & back ( ) const
{
     return objects[ theSize - 1 ];
}
```

LIST: C++ STL VECTOR IMPLEMENTATION

- Return iterators

```
iterator begin( )
  { return &objects[ 0 ]; }
const_iterator begin( ) const
  { return &objects[ 0 ]; }
```

```
iterator end( )
  { return &objects[ size( ) ]; }
const_iterator end( ) const
  { return &objects[ size( ) ]; }
```

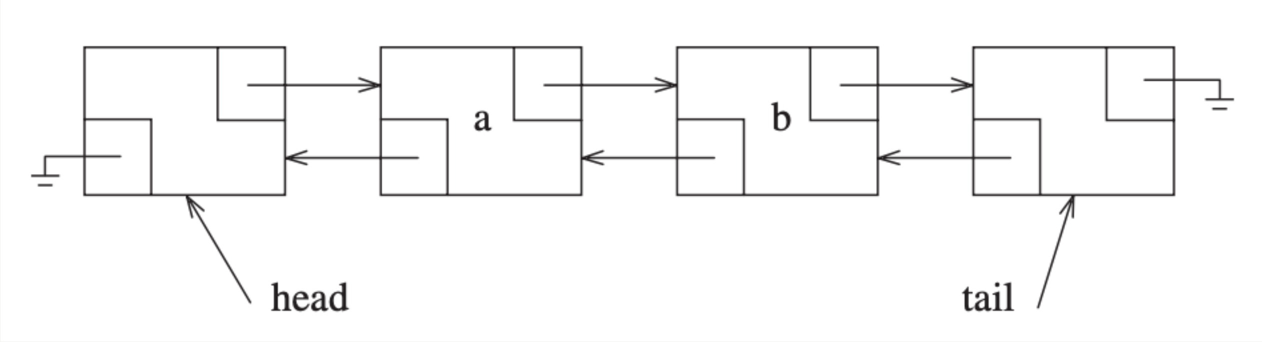
LIST: C++ STL LIST IMPLEMENTATION

- These methods are uniquely available in list

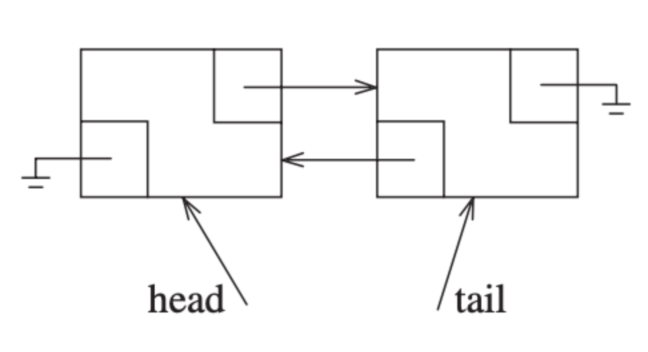
- `void push_front( const Object & x )`: adds `x` to the front of the `list`.
- `void pop_front( )`: removes the object at the front of the `list`.

LIST: C++ STL LIST IMPLEMENTATION

- The head and the tail



- An empty list (which also contains the head and tail)



LIST: C++ STL LIST IMPLEMENTATION

- The node definition

```
struct Node
{
    Object  data;
    Node   *prev;     // pointer to the previous node
    Node   *next;     // pointer to the next node

    Node( const Object & d = Object{ }, Node * p = nullptr,
                                        Node * n = nullptr )
      : data{ d }, prev{ p }, next{ n } { }

    Node( Object && d, Node * p = nullptr, Node * n = nullptr )
      : data{ std::move( d ) }, prev{ p }, next{ n } { }
};
```

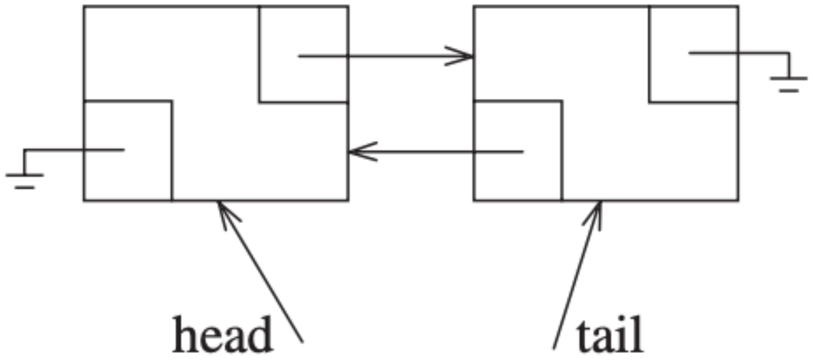
LIST: C++ STL LIST IMPLEMENTATION

- list definition and initialization

```
private:
  int    theSize;
  Node *head;
  Node *tail;
```

// theSize: counter for the number of elements in the list

```
void init( )
{
    theSize = 0;
    head = new Node;
    tail = new Node;
    head->next = tail;
    tail->prev = head;
}
```



LIST: C++ STL LIST IMPLEMENTATION

- constructors and destructor

```
List( )
  { init( ); }

~List( )
{
    clear( );
    delete head;
    delete tail;
}

List( const List & rhs )
{
    init( );
    for( auto & x : rhs )
        push_back( x );
}
```

LIST: C++ STL LIST IMPLEMENTATION

- Copy/assignment

```
List & operator= ( const List & rhs )
{
    List copy = rhs;
    std::swap( *this, copy );
    return *this;
}


List( List && rhs )
  : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
{
    rhs.theSize = 0;
    rhs.head = nullptr;
    rhs.tail = nullptr;
}
```

LIST: C++ STL LIST IMPLEMENTATION

- const_iterator internal definition

```
      protected:
        Node *current;

        Object & retrieve( ) const
          { return current->data; }

        const_iterator( Node *p ) : current{ p }
          { }

        friend class List<Object>;
```

// “friend class” allows to access the private and protected members of List<Object>

LIST: C++ STL LIST IMPLEMENTATION

- const_iterator continued

```
        const_iterator( ) : current{ nullptr }
          { }

        const Object & operator* ( ) const
          { return retrieve( ); }

        const_iterator & operator++ ( )
        {
            current = current->next;
            return *this;
        }

        const_iterator operator++ ( int )
        {
            const_iterator old = *this;
            ++( *this );
            return old;
        }
```

```
        bool operator== ( const const_iterator & rhs ) const
          { return current == rhs.current; }
        bool operator!= ( const const_iterator & rhs ) const
          { return !( *this == rhs ); }
```

LIST: C++ STL LIST IMPLEMENTATION

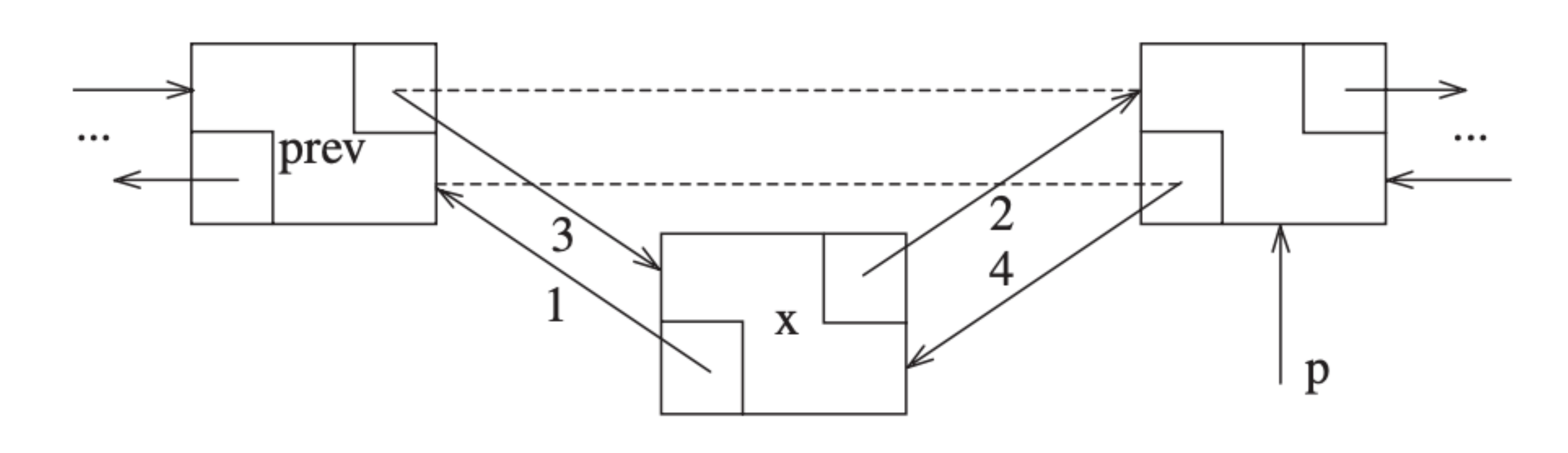
- iterator

```
        iterator( )
          { }

        Object & operator* ( )
          { return const_iterator::retrieve( ); }
        const Object & operator* ( ) const
          { return const_iterator::operator*( ); }

        iterator & operator++ ( )
        {
            this->current = this->current->next;
            return *this;
        }

        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
        }
```

```
      protected:
        iterator( Node *p ) : const_iterator{ p }
          { }

        friend class List<Object>;
```

LIST: C++ STL LIST IMPLEMENTATION

- insertion




```
Node *newNode = new Node{ x, p->prev, p };  // Steps 1 and 2
p->prev = p->prev->next = newNode;          // Steps 3 and 4
```

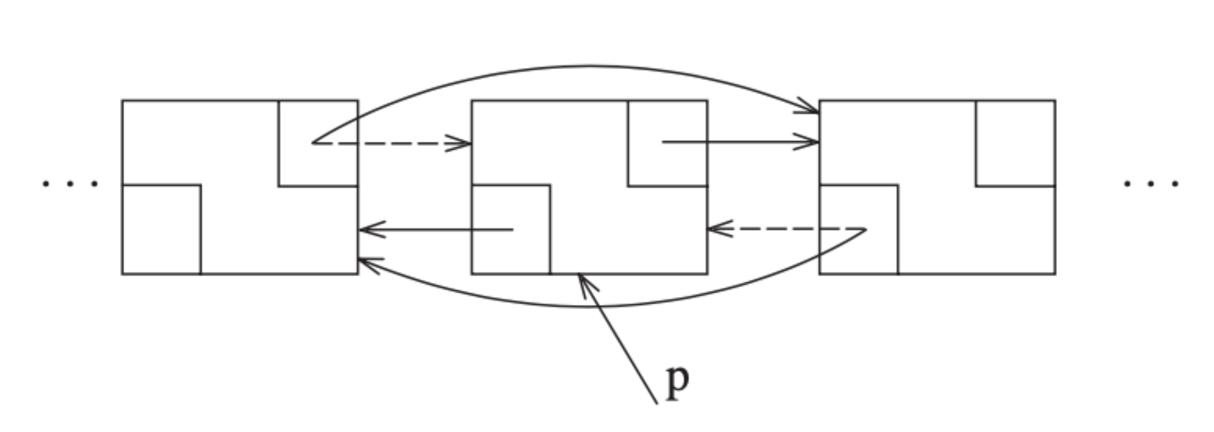
LIST: C++ STL LIST IMPLEMENTATION

- insertion continued

```
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
{
    Node *p = itr.current;
    theSize++;
    return { p->prev = p->prev->next = new Node{ x, p->prev, p } };
}

// Insert x before itr.
iterator insert( iterator itr, Object && x )
{
    Node *p = itr.current;
    theSize++;
    return { p->prev = p->prev->next
                            = new Node{ std::move( x ), p->prev, p } };
}
```

LIST: C++ STL LIST IMPLEMENTATION

- deletion




```
p->prev->next = p->next;
p->next->prev = p->prev;
delete p;
```

LIST: C++ STL LIST IMPLEMENTATION

- deletion continued

```
// Erase item at itr.
iterator erase( iterator itr )
{
    Node *p = itr.current;
    iterator retVal{ p->next };
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    theSize--;

    return retVal;
}

iterator erase( iterator from, iterator to )
{
    for( iterator itr = from; itr != to; )
        itr = erase( itr );

    return to;
}
```

SUMMARY

- the list ADT can be used to represent a list of ordered objects
- list can be implemented using array (STL vector) or linked list (STL list)
- Lab 1: implementation of vector
- Lab 2: implementation of linked list