



Abstract Data Types

CptS 223 – Advanced Data Structures

Larry Holder

School of Electrical Engineering and Computer Science
Washington State University



Purpose

- Abstract Data Types (ADTs)
- Lists
- Stacks
- Queues



Abstract Data Types (ADTs)

- ADT is a set of objects together with a set of operations.
- Abstract in that implementation of operations not specified in ADT definition
- E.g., List
 - Insert, delete, search, sort
- C++ class perfect for ADTs
- Can change ADT implementation details without breaking code using ADT



The List ADT

- List of size N : A_0, A_1, \dots, A_{N-1}
- Each element A_k has a unique position k in the list
- Elements can be arbitrarily complex
- Operations
 - $\text{insert}(X, k)$, $\text{remove}(k)$, $\text{find}(X)$, $\text{findKth}(k)$, $\text{printList}()$

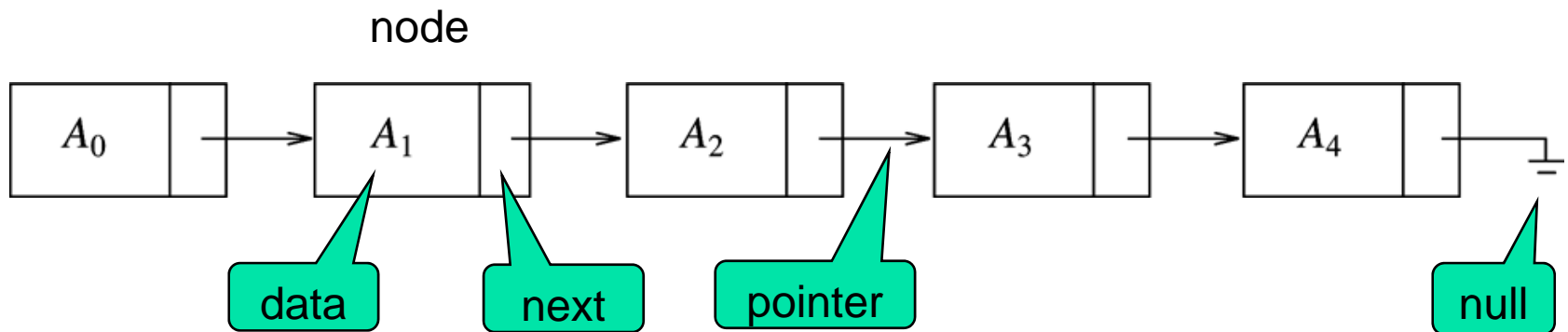


Lists Using Arrays

- Simple array vs. Vector class
 - Estimating maximum size
- Operations
 - `insert(X,k)` – $O(N)$
 - `remove(k)` – $O(N)$
 - `find(X)` – $O(N)$
 - `findKth(k)` – $O(1)$
 - `printList()` – $O(N)$

Linked Lists

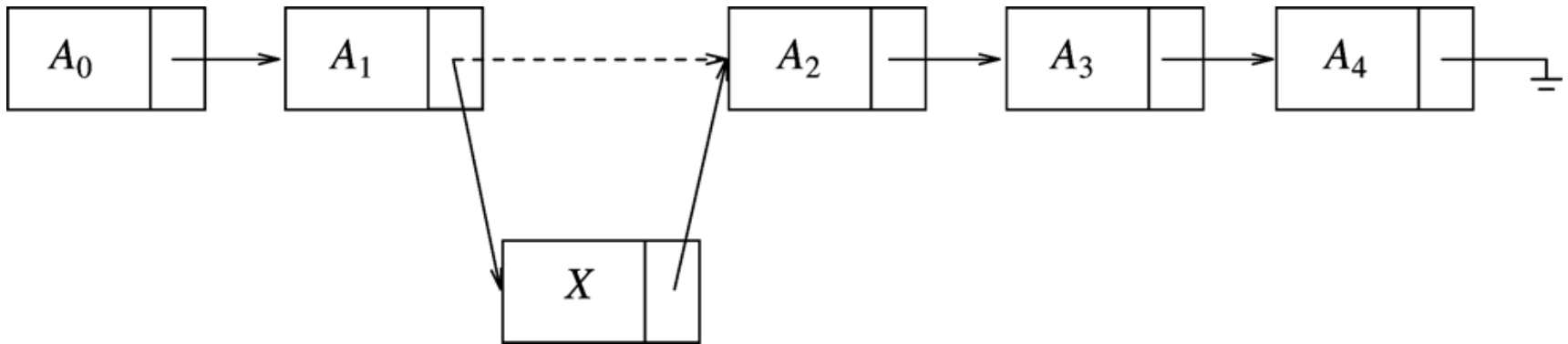
- Elements not stored in contiguous memory
- Nodes in list consist of data element and next pointer



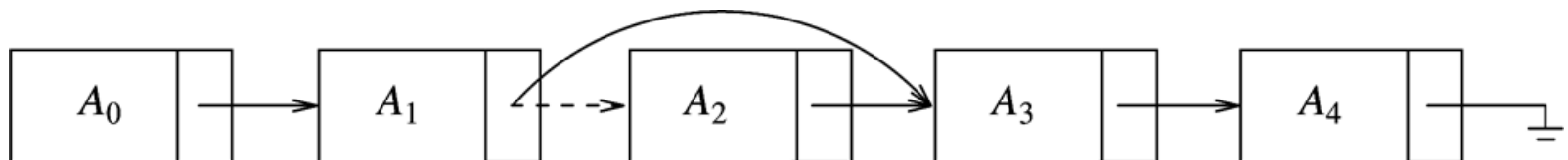
Linked Lists

■ Operations

■ Insert(X, A) – $O(1)$



■ Remove(A) – $O(1)$



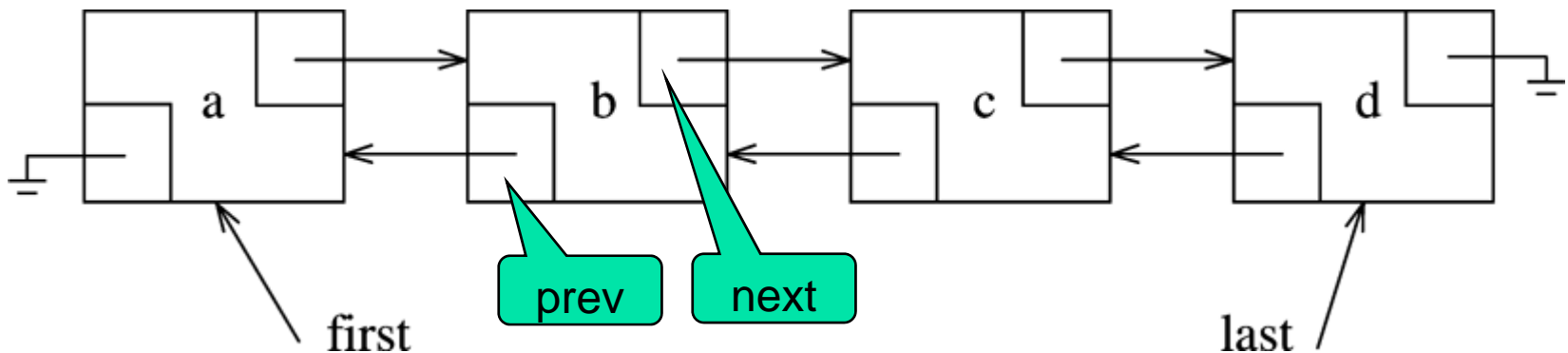


Linked Lists

- Operations
 - `find(X)` – $O(N)$
 - `findKth(k)` – $O(N)$
 - `printList()` – $O(N)$

Doubly-Linked List

- Singly-linked list
 - $\text{insert}(X,A)$ and $\text{remove}(X)$ require pointer to node just before X
- Doubly-linked list
 - Also keep pointer to previous node





Doubly-Linked List

- Insert(X,A)

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

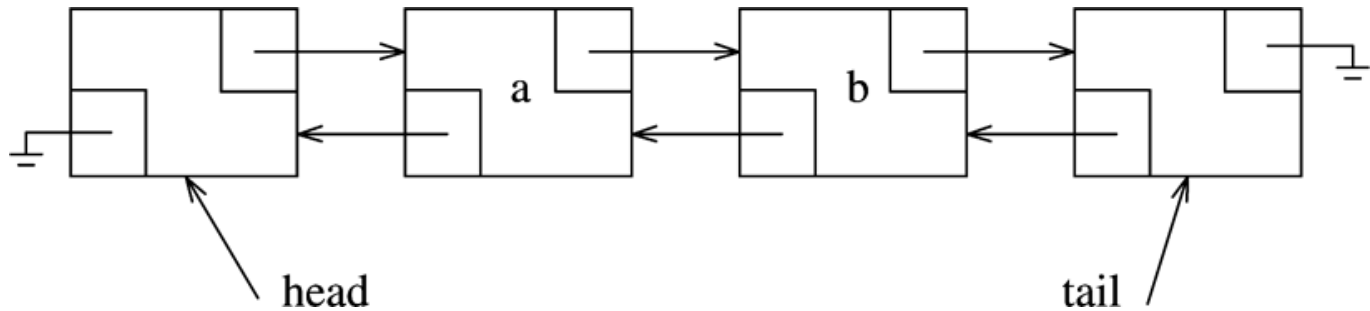
- Remove(X)

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

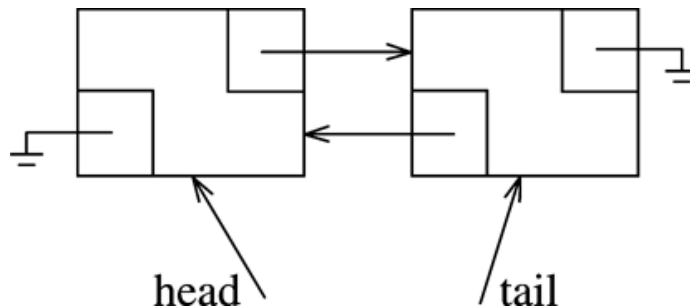
- Problems with operations at ends of list

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





C++ Standard Template Library (STL)

- Implementation of common data structures
 - List, stack, queue, ...
 - Generally called *containers*
- WWW resources
 - www.sgi.com/tech/stl/
 - www.cppreference.com/cppstl.html



Lists using STL

- **vector<Object>**
 - Array-based implementation
 - **findKth** – $O(1)$
 - **insert** and **remove** – $O(N)$
 - Unless change at end of vector
- **list<Object>**
 - Doubly-linked list with sentinel nodes
 - **findKth** – $O(N)$
 - **insert** and **remove** – $O(1)$
 - If position of change is known
- Both require $O(N)$ for search



Container Methods

- **int size() const**
 - Return number of elements in container
- **void clear()**
 - Remove all elements from container
- **bool empty() const**
 - Return true if container has no elements, otherwise return false



Vector and List Methods

- **void push_back (const Object & x)**
 - Add x to end of list
- **void pop_back ()**
 - Remove object at end of list
- **const Object & back () const**
 - Return object at end of list
- **const Object & front () const**
 - Return object at front of list



List-only Methods

- **void push_front (const Object & x)**
 - Add x to front of list
- **void pop_front ()**
 - Remove object at front of list



Vector-only Methods

- **Object & operator[] (int idx)**
 - Return object at index idx in vector with no bounds-checking
- **Object & at (int idx)**
 - Return object at index idx in vector with bounds-checking
- **int capacity () const**
 - Return internal capacity of vector
- **void reserve (int newCapacity)**
 - Set new capacity for vector (avoid expansion)



Iterators

- Represents position in container
- Getting an iterator
 - **iterator begin ()**
 - Return appropriate iterator representing first item in container
 - **iterator end ()**
 - Return appropriate iterator representing end marker in container
 - Position after last item in container



Iterator Methods

- `itr++` and `++itr`
 - Advance iterator `itr` to next location
- `*itr`
 - Return reference to object stored at iterator `itr`'s location
- `itr1 == itr2`
 - Return true if `itr1` and `itr2` refer to same location; otherwise return false
- `itr1 != itr2`
 - Return true if `itr1` and `itr2` refer to different locations; otherwise return false



Example: printList

```
template <typename Container>
void printList (const Container & lst)
{
    for (typename Container::const_iterator itr = lst.begin();
         itr != lst.end();
         ++itr)
    {
        cout << *itr << endl;
    }
}
```



Constant Iterators

- `iterator begin ()`
- `const_iterator begin () const`
- `iterator end ()`
- `const_iterator end () const`
- Appropriate version above returned based on whether container is `const`
- If `const_iterator` used, then `*itr` cannot appear on left-hand side of assignment (e.g., `*itr=0`)



Better printList

```
1  template <typename Container>
2  void printCollection( const Container & c, ostream & out = cout )
3  {
4      if( c.empty( ) )
5          out << "(empty)";
6      else
7          {
8              typename Container::const_iterator itr = c.begin( );
9              out << "[" << *itr++;    // Print first item
10
11              while( itr != c.end( ) )
12                  out << ", " << *itr++;
13              out << "]" << endl;
14          }
15 }
```



Container Operations Requiring Iterators

- **iterator insert (iterator pos, const Object & x)**
 - Add **x** into list, prior to position given by iterator **pos**
 - Return iterator representing position of inserted item
 - $O(1)$ for lists, $O(N)$ for vectors
- **iterator erase (iterator pos)**
 - Remove object whose position is given by iterator **pos**
 - Return iterator representing position of item following **pos**
 - This operation invalidates **pos**
 - $O(1)$ for lists, $O(N)$ for vectors
- **iterator erase (iterator start, iterator end)**
 - Remove all items beginning at position **start**, up to, but not including **end**



Implementation of Vector

```
1  template <typename Object>
2  class Vector
3  {
4  public:
5      explicit Vector( int initSize = 0 )
6          : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
7          { objects = new Object[ theCapacity ]; }
8      Vector( const Vector & rhs ) : objects( NULL )
9          { operator=( rhs ); }
10     ~Vector( )
11         { delete [ ] objects; }
12
13     const Vector & operator= ( const Vector & rhs )
14     {
15         if( this != &rhs )
16         {
17             delete [ ] objects;
18             theSize = rhs.size( );
19             theCapacity = rhs.theCapacity;
20
21             objects = new Object[ capacity( ) ];
22             for( int k = 0; k < size( ); k++ )
23                 objects[ k ] = rhs.objects[ k ];
24         }
25         return *this;
26     }
```

copy constructor

destructor

operator=

Implementation of Vector

```
28 void resize( int newSize )
29 {
30     if( newSize > theCapacity )
31         reserve( newSize * 2 + 1 );
32     theSize = newSize;
33 }
34
35 void reserve( int newCapacity )
36 {
37     if( newCapacity < theSize )
38         return;
39
40     Object *oldArray = objects;
41
42     objects = new Object[ newCapacity ];
43     for( int k = 0; k < theSize; k++ )
44         objects[ k ] = oldArray[ k ];
45
46     theCapacity = newCapacity;
47
48     delete [ ] oldArray;
49 }
50
51 Object & operator[]( int index )
52 { return objects[ index ]; }
53 const Object & operator[]( int index ) const
54 { return objects[ index ]; }
55
56 bool empty( ) const
57 { return size( ) == 0; }
58 int size( ) const
59 { return theSize; }
60 int capacity( ) const
61 { return theCapacity; }
62
63 void push_back( const Object & x )
64 {
65     if( theSize == theCapacity )
66         reserve( 2 * theCapacity + 1 );
67     objects[ theSize++ ] = x;
68 }
69
70 void pop_back( )
71 { theSize--; }
72
73 const Object & back ( ) const
74 { return objects[ theSize - 1 ]; }
```

Automatic
resize



Implementation of Vector

```
75     typedef Object * iterator;
76     typedef const Object * const_iterator;
77
78     iterator begin( )
79         { return &objects[ 0 ]; }
80     const_iterator begin( ) const
81         { return &objects[ 0 ]; }
82     iterator end( )
83         { return &objects[ size( ) ]; }
84     const_iterator end( ) const
85         { return &objects[ size( ) ]; }
86
87     enum { SPARE_CAPACITY = 16 };
88
89     private:
90         int theSize;
91         int theCapacity;
92         Object * objects;
93     };
```

Iterators



Implementation of List

```
1  template <typename Object>
2  class List
3  {
4      private:
5          struct Node
6              { /* See Figure 3.13 */ };
7
8      public: Nested class
9          class const_iterator
10             { /* See Figure 3.14 */ };
11
12         class iterator : public const_iterator
13             { /* See Figure 3.15 */ };
14
15     public:
16         List( )
17             { /* See Figure 3.16 */ }
18         List( const List & rhs )
19             { /* See Figure 3.16 */ }
20         ~List( )
21             { /* See Figure 3.16 */ }
22         const List & operator= ( const List & rhs )
23             { /* See Figure 3.16 */ }
24
25         iterator begin( )
26             { return iterator( head->next ); }
27         const_iterator begin( ) const
28             { return const_iterator( head->next ); }
29         iterator end( )
30             { return iterator( tail ); }
31         const_iterator end( ) const
32             { return const_iterator( tail ); }
33
34         int size( ) const
35             { return theSize; }
36         bool empty( ) const
37             { return size( ) == 0; }
38
39         void clear( )
40         {
41             while( !empty( ) )
42                 pop_front( );
43         }
```

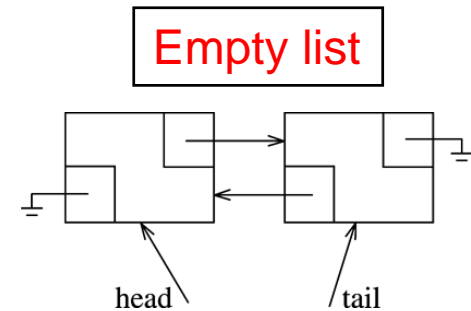
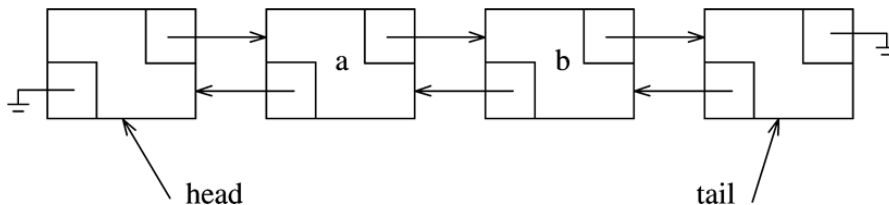


Implementation of List

```
44     Object & front( )
45         { return *begin( ); }
46     const Object & front( ) const
47         { return *begin( ); }
48     Object & back( )
49         { return *--end( ); }
50     const Object & back( ) const
51         { return *--end( ); }
52     void push_front( const Object & x )
53         { insert( begin( ), x ); }
54     void push_back( const Object & x )
55         { insert( end( ), x ); }
56     void pop_front( )
57         { erase( begin( ) ); }
58     void pop_back( )
59         { erase( --end( ) ); }
61     iterator insert( iterator itr, const Object & x )
62         { /* See Figure 3.18 */ }
63
64     iterator erase( iterator itr )
65         { /* See Figure 3.20 */ }
66     iterator erase( iterator start, iterator end )
67         { /* See Figure 3.20 */ }
68
69     private:
70         int theSize;
71         Node *head;
72         Node *tail;
73
74     void init( )
75         { /* See Figure 3.16 */ }
76     };
```

Implementation of List

```
1 struct Node
2 {
3     Object data;
4     Node *prev;
5     Node *next;
6
7     Node( const Object & d = Object( ), Node *p = NULL, Node *n = NULL )
8         : data( d ), prev( p ), next( n ) { }
9 };
```



Implementation of List

```
1  class const_iterator          23
2  {                               24
3  public:                          25
4      const_iterator( ) : current( NULL )  26
5          { }                       27
6
7      const Object & operator* ( ) const  28
8          { return retrieve( ); }      29
9
10     const_iterator & operator++ ( )     30
11     {                                   31
12         current = current->next;      32
13         return *this;                 33
14     }                                  34
15
16     const_iterator operator++ ( int )   35
17     {                                   36
18         const_iterator old = *this;    37
19         ++( *this );                   38
20         return old;
21     }
```

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

protected:

```
Node *current;
```

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

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

```
friend class List<Object>;
```

Allows inheriting classes to access these.

Gives List class access to constructor.



Implementation of List

```
39     class iterator : public const_iterator           63         protected:
40     {                                               64             iterator( Node *p ) : const_iterator( p )
41         public:                                       65                 { }
42             iterator( )                               66
43                 { }                                   67
44                                                     68         friend class List<Object>;
45             Object & operator* ( )
46                 { return retrieve( ); }
47             const Object & operator* ( ) const
48                 { return const_iterator::operator*( ); }
49
50             iterator & operator++ ( )
51             {
52                 current = current->next;
53                 return *this;
54             }
55
56             iterator operator++ ( int )
57             {
58                 iterator old = *this;
59                 ++( *this );
60                 return old;
61             }
```

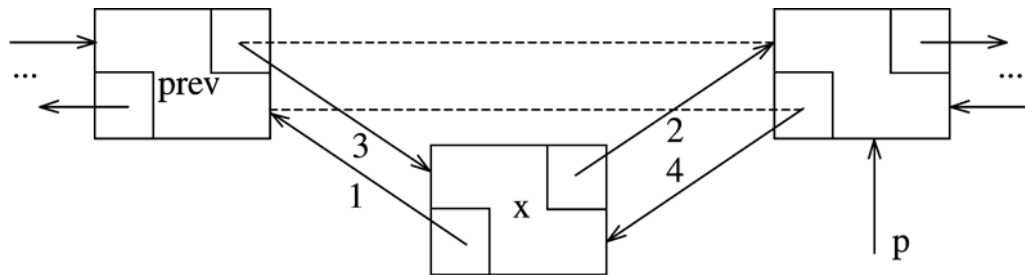


Implementation of List

```
1 List( )
2     { init( ); }
3
4 ~List( )
5 {
6     clear( );
7     delete head;
8     delete tail;
9 }
10
11 List( const List & rhs )
12 {
13     init( );
14     *this = rhs;
15 }
16
17 const List & operator= ( const List & rhs )
18 {
19     if( this == &rhs )
20         return *this;
21     clear( );
22     for( const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
23         push_back( *itr );
24     return *this;
25 }
26
27 void init( )
28 {
29     theSize = 0;
30     head = new Node;
31     tail = new Node;
32     head->next = tail;
33     tail->prev = head;
34 }
```

Implementation of List

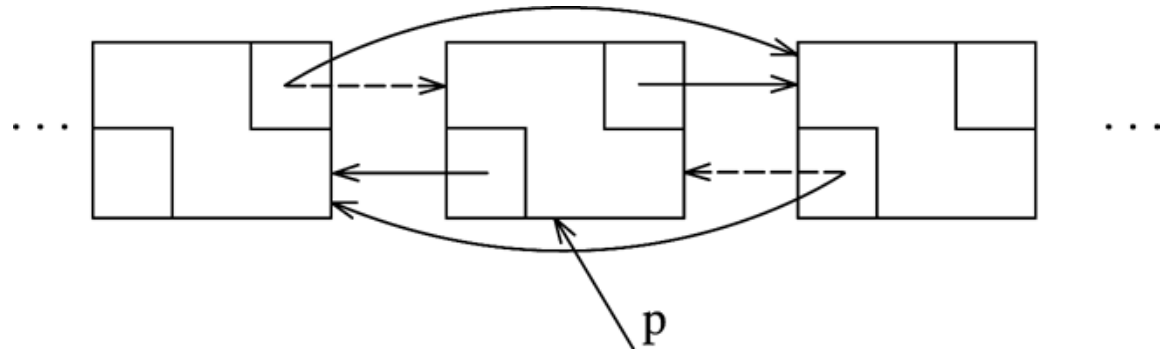
```
1 // Insert x before itr.
2 iterator insert( iterator itr, const Object & x )
3 {
4     Node *p = itr.current;
5     theSize++;
6     return iterator( p->prev = p->prev->next = new Node( x, p->prev, p ) );
7 }
```



Implementation of List

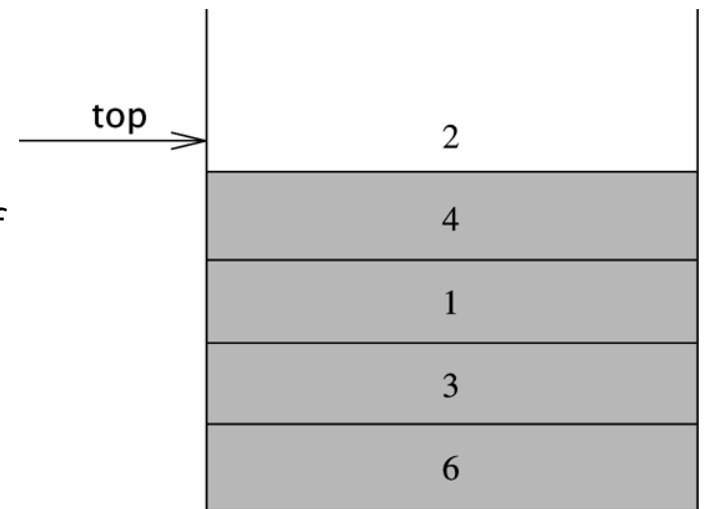
```
1 // Erase item at itr.
2 iterator erase( iterator itr )
3 {
4     Node *p = itr.current;
5     iterator retVal( p->next );
6     p->prev->next = p->next;
7     p->next->prev = p->prev;
8     delete p;
9     theSize--;
10
11     return retVal;
12 }
```

```
14     iterator erase( iterator start, iterator end )
15     {
16         for( iterator itr = from; itr != to; )
17             itr = erase( itr );
18
19         return to;
20     }
```



Stack ADT

- Stack is a list where insert and remove take place only at the “top”
- Operations
 - Push (insert) element on top of stack
 - Pop (remove) element from top of stack
 - Top: return element at top of stack
- LIFO (Last In First Out)





Stack Implementation

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;
}
```

Running Times ?



C++ STL Stack Class

- Methods
 - Push, pop, top
 - Empty, size

```
#include <stack>

stack<int> s;

for (int i = 0; i < 5; i++ )
{
    s.push(i);
}
while (!s.empty())
{
    cout << s.top() << endl;
    s.pop();
}
```



Stack Applications

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

```
stack<char> s;
while not end of file
{
    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
```



Stack Applications

- Postfix expressions
 - $((1 * 2) + 3) + (4 * 5) =$
1 2 * 3 + 4 5 * +
 - HP calculators
- Infix to postfix conversion
 - Output operands
 - Push operators and left parentheses
 - When right parenthesis, output operators up to left parenthesis

```
Class PostFixCalculator
{
    public:
        ...
        void Multiply ()
        {
            int i1 = s.top();
            s.pop();
            int i2 = s.top();
            s.pop();
            s.push (i1 * i2);
        }
    private:
        stack<int> s;
}
```

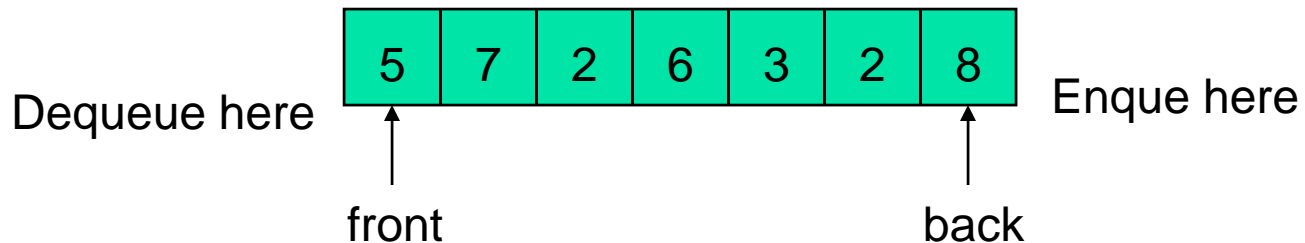


Stack Applications

- Function calls
- Programming languages use stacks 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 (insert) element at the back of the queue
 - Dequeue (remove and return) element from the front of the queue
 - FIFO (First In First Out)





Queue Implementation

Linked List

Vector (?)

```
template <typename Object>
class queue
{
public:
    queue () {}
    void enqueue (Object & x)
        { q.push_back (x); }
    Object & dequeue ()
        {
            Object & x = q.front ();
            q.pop_front ();
            return x;
        }
private:
    list<Object> q;
}
```

Running time ?



C++ STL Queue Class

- Methods
 - Push (at back)
 - Pop (from front)
 - Back, front
 - Empty, size

```
#include <queue>

queue<int> q;

for (int i = 0; i < 5; i++ )
{
    q.push(i);
}
while (!q.empty())
{
    cout << q.front() << endl;
    q.pop();
}
```



Queue Applications

- Job scheduling
- Priority queues
- Graph traversals
- Queuing theory



Summary

- Abstract Data Types (ADTs)
 - Linked list
 - Stack
 - Queue
- C++ Standard Template Library (STL)
- Numerous applications
- Building blocks for more complex data structures