WASHINGTON STATE UNIVERSITY

CPTS 223 Advanced Data Structure C/C++

Abstract Data Type

Topics

- What are Abstract Data Types (ADTs)

- Some basic ADTs:
  - Lists
  - Stacks
  - Queues

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
  - Operations: insert, delete, search, sort
- C++ classes are perfect for ADTs
- Can change ADT implementation details without breaking code using ADT

Image from https://www.geeksforgeeks.org/abstract-data-types/

ADTs

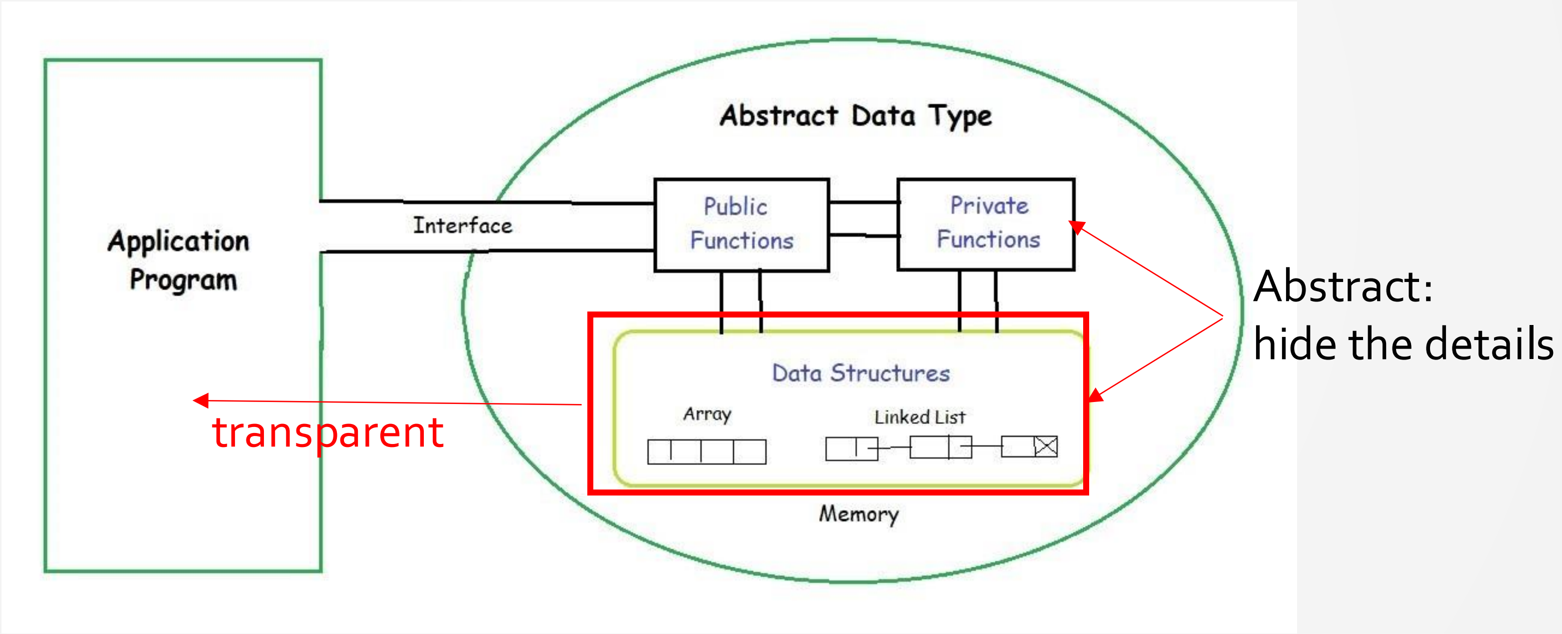
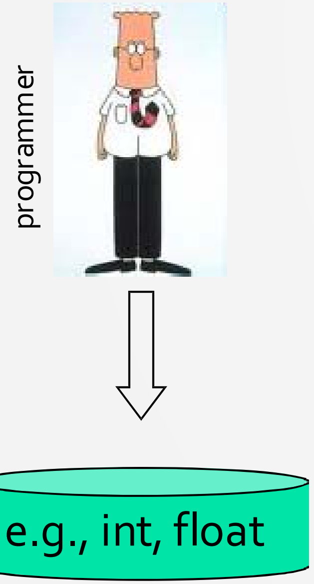


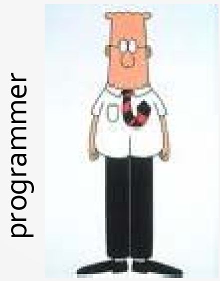
Image from https://www.geeksforgeeks.org/abstract-data-types/

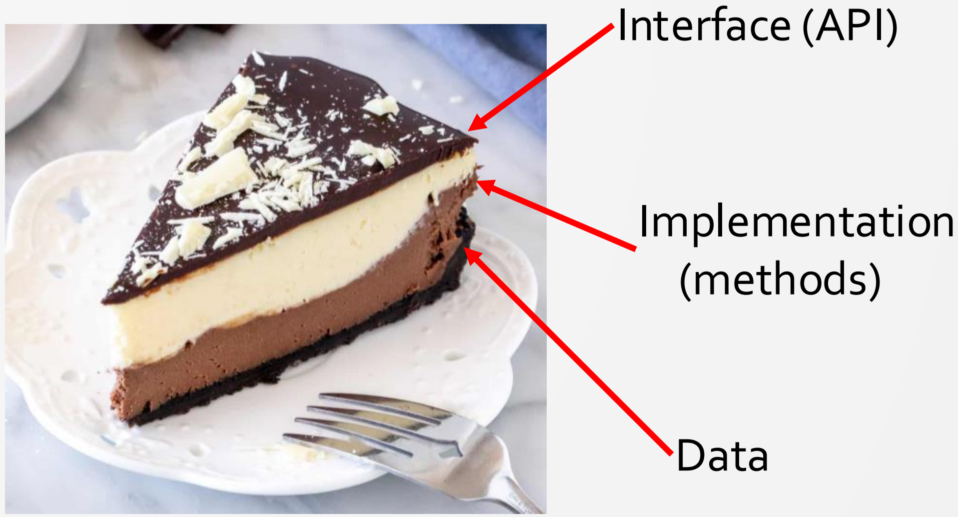
Primitive v.s. Abstract Data Type

- Primitive DT:



- Abstract DT:





Specifications of Basic ADTs

- List
- Stack
- Queue

List ADT

- List of size N: A0, A1, ..., A_{N-1}
- Each element A_k has a unique position k in the list
- Elements can be arbitrarily complex

object

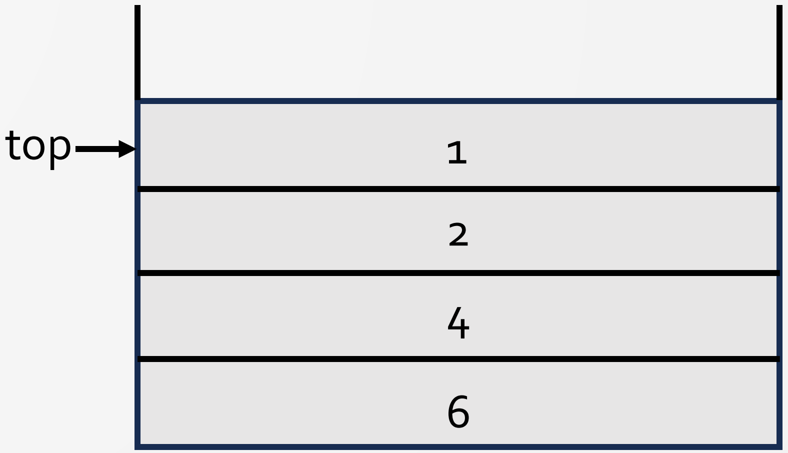
WSU

List ADT

- List of size N: A0, A1, ..., A_{N-1}
- Each element A_k has a unique position k in the list
- Elements can be arbitrarily complex
- Operations
  - insert(X,k)
  - remove(k)
  - find(X)
  - findKth(k)
  - printList()

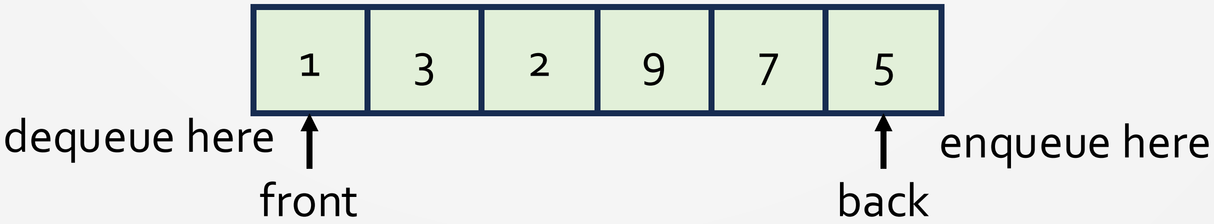
Stack ADT

- Stack == 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 end of a list
- LIFO (Last In First Out)



Queue ADT

- Queue = 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)



Implementation for basic ATDs

List ADT using arrays (fixed size)

A0	A1	A2	A3	A4	...

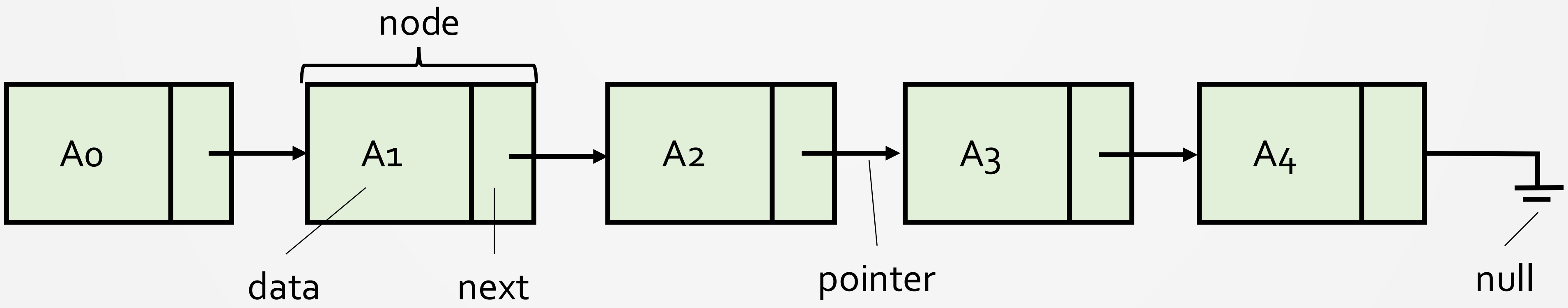
- Operations (k → index/position)
  - insert(X, k): O(N)
  - remove(k): O(N)
  - find(X): O(N)
  - findKth(k): O(1)
  - printList(): O(N)

> Read as "order N"
> →
> runtime is proportional to N

> Read as "order 1"
> →
> runtime is a constant, i.e.,
> not dependent on N

List ADT using linked lists

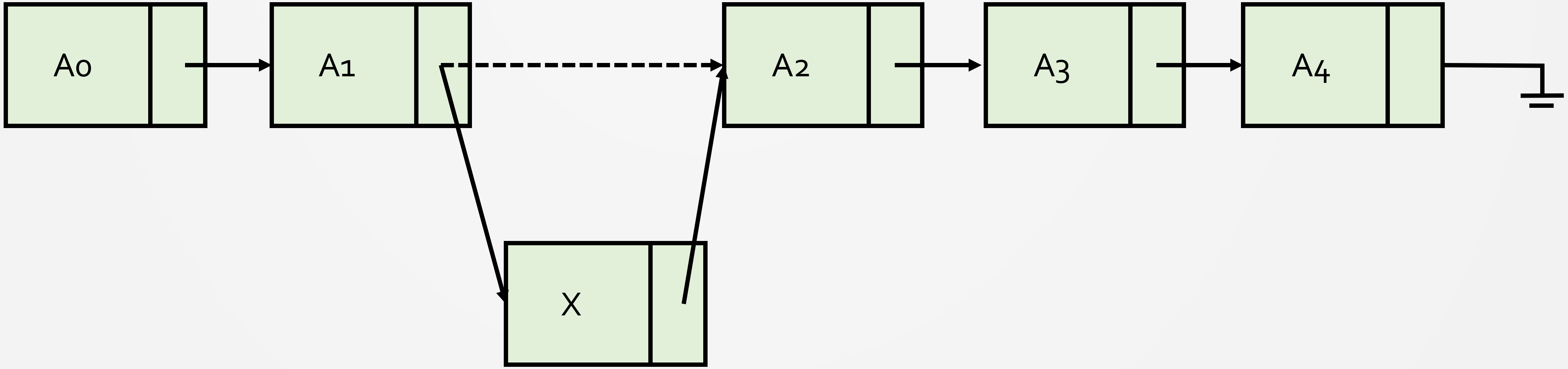
- Elements not stored in contiguous memory
- Nodes in list consist of data element and next pointer



List ADT using linked lists

What about A is unknown?

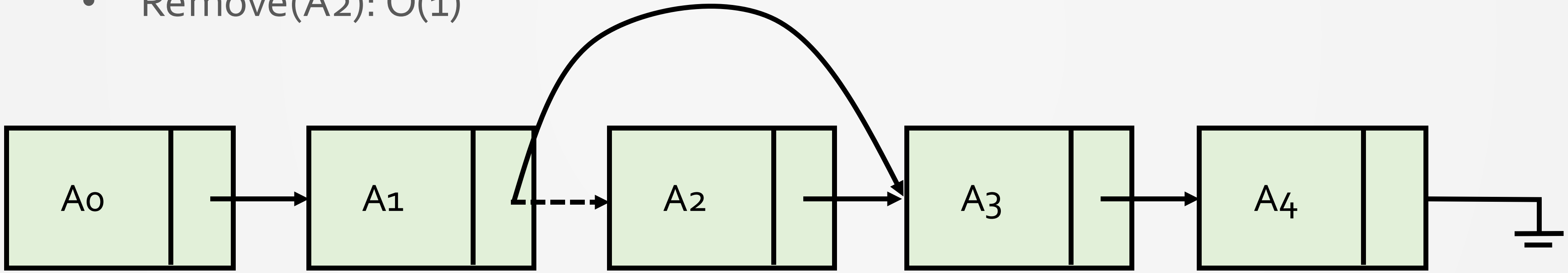
- Operations (A is a known pointer)
  - Insert(X, A): O(1)



WSU

List ADT using linked lists

- Operations
  - Remove(A2): O(1)



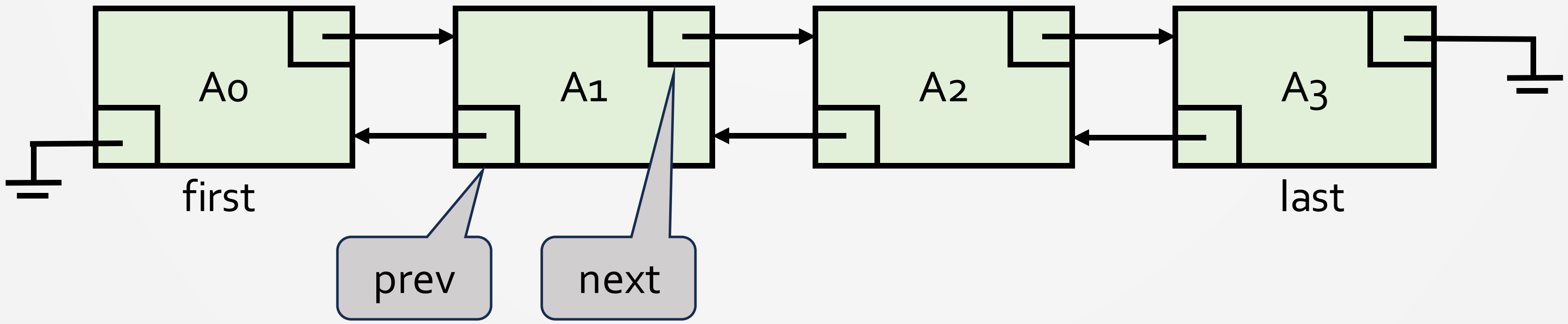
List ADT using linked lists

- Operations
  - find(X): O(N)
  - findKth(k): O(N)
  - printList(): O(N)

WSU

Doubly-linked list

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



Doubly-linked list

- Insert(X,A)

```
newX = new Node(X);
newX->next = A->next;
newX->prev = A;
if (A->next != NULL) {
    A->next->prev = newX;
}
A->next = newX;
```

Insert X after A

- Remove(X)

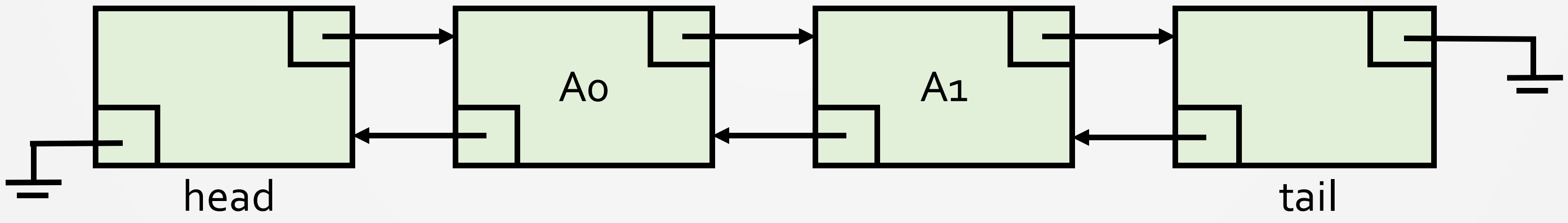
```
X->prev->next = X->next;
X->next->prev = X->prev;
```

- Two-way traversal
- Insert/delete is faster if an existing node pointer is given

WSU

Sentinel nodes

- Not hold any actual data
- To avoid special cases at edge cases
- Example: doubly-linked list with sentinel nodes:



C++ standard template library

- Implementation of **common data structures**
- List, stack, queue, …
- Generally called **containers**
- Online references for STL
  - www.cplusplus.com/reference/stl/
  - www.cppreference.com/cppstl.html

Common container methods

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

Implemented lists in 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
- Search: O(N) for both implementations

Our focus in the following chapters

Methods for both vector and list

- 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

```
#include <list>
std::list<int> myList;
myList.push_back(1);
myList.push_back(2);
myList.push_back(3);
myList.pop_back();
b = myList.back();
F = myList.front();
```

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

"operator square brackets" or "subscript operator"

- Object & operator[] (int idx)
  - Return object at index idx in vector
  - Without 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)

```
#include <vector>
std::vector<int> v = {1, 2, 3};
int value = v[2];
v[1] = 10;
int out_of_bounds = v[5];
int c = v.capacity();
int thirdItem = v.at(2);
int fourthItem = v.at(3);
```

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

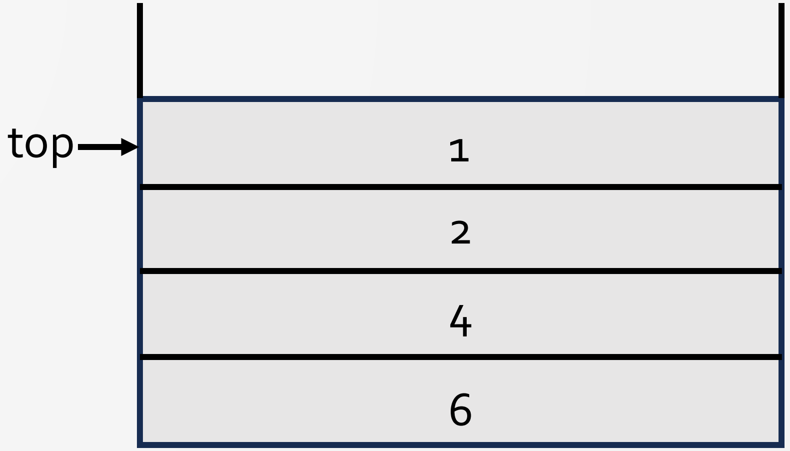
```
template <typename Container>
void printCollection(const Container & c, ostream & out = cout)
{
    if (c.empty())
        std::cout << "(empty)" << std::endl;
    else
    {
        typename Container::const_iterator itr = c.begin();
        std::cout << " [ " << *itr++;

        while (itr != c.end())
            std::cout << ", " << *itr++;
        std::cout << " ]" << std:endl;
    }
}
```

Operations requiring iterator

- 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

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)



WSU

Stack implementation

Linked list

```
template <typename Object>
class stack
{
    public:
        stack () {}
        void push (Object & x)
            { ? }
        void pop ()
            { ? }
        Object & top ()
            { ? }
    private:
        list<Object> s;
}
```

Vector

```
template <typename Object>
class stack
{
    public:
        stack () {}
        void push (Object & x)
            { ? }
        void pop ()
            { ? }
        Object & top ()
            { ? }
    private:
        vector<Object> s;
}
```

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

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;
}
```

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

No parentheses needed

- Postfix expressions
  - 1 2 * 3 + 4 5 * +
    - == ((1 * 2) + 3) + (4 * 5)
  - HP calculators
  - Unambiguous (no need for parenthesis)
    - Infix needs parenthesis or else implicit precedence specification to avoid ambiguity
    - E.g., try a+(b*c) and (a+b)*c
  - Postfix evaluation uses stack

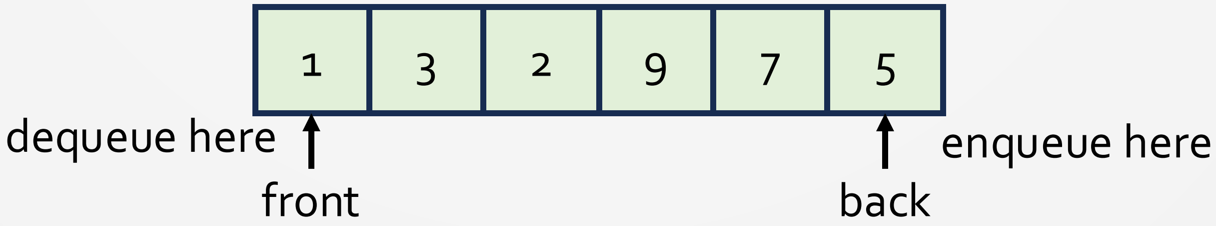
```
Class PostFixCalculator
{
    public:
    ...
    void Multiply ()
    {
        int i1 = s.top();
        s.pop();
        int i2 = s.top();
        s.pop();
        s.push (i1 * i2);
    }
    private:
        stack<int> s;
}
```

Stack applications

- Postfix expressions
- 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

```
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;
```

How would the runtime change if **vector** is used in implementation?

WSU

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
  - A large number of tasks to be performed by the server
  - All tasks have to be put in a queue, first come first serve
- Graph traversals
- Queuing theory

WSU

Summary

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