# Data Structures (in C++)

## - Deques -

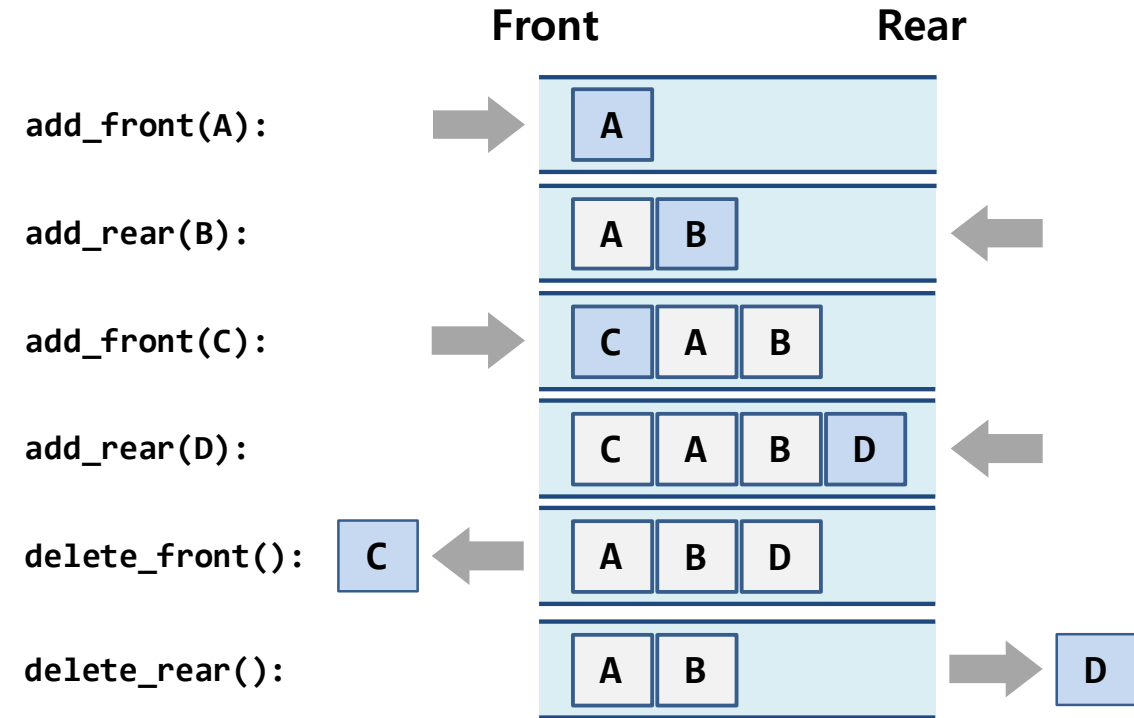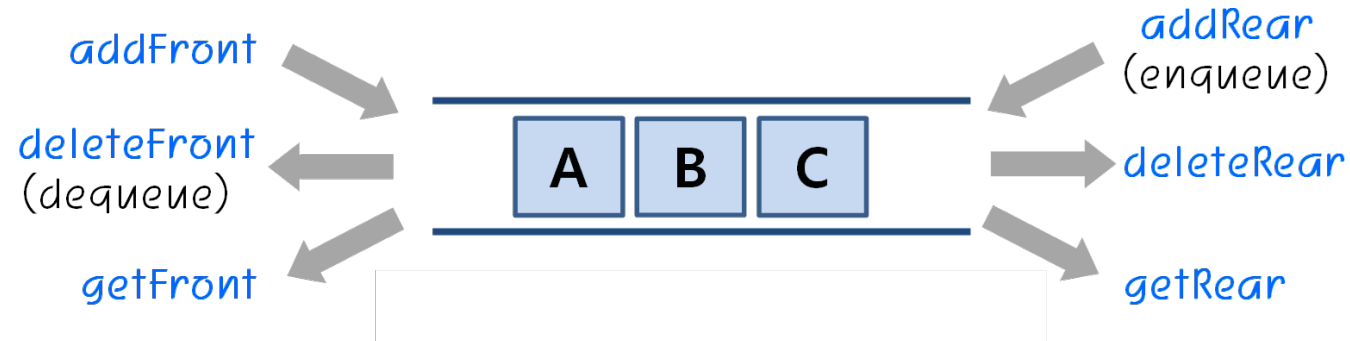**Jinsun Park**

**Visual Intelligence and Perception Lab., CSE, PNU**

Visual Intelligence
and Perception Lab.

School of Computer
Science and Engineering

부산대학교
PUSAN NATIONAL UNIVERSITY

# Deques

# Deques

- **Deque (Double-Ended Queue)**
  - A queue-like data structure

  - Supports insertion and deletion at both ends (*i.e.,* the front and rear sides)

  - Pronounced "*deck*" (not "*D.Q.*")

# Deque ADT

- The deque ADT supports the following operations:

$insertFront(e)$: Insert a new element $e$ at the beginning of the deque.

$insertBack(e)$: Insert a new element $e$ at the end of the deque.

$eraseFront()$: Remove the first element of the deque; an error occurs if the deque is empty.

$eraseBack()$: Remove the last element of the deque; an error occurs if the deque is empty.

- Additional utility functions:

$front()$: Return the first element of the deque; an error occurs if the deque is empty.

$back()$: Return the last element of the deque; an error occurs if the deque is empty.

$size()$: Return the number of elements of the deque.

$empty()$: Return true if the deque is empty and false otherwise.

# Deque Example

| Operation | Output | D |
|---|---|---|
| insertFront(3) | – | (3) |
| insertFront(5) | – | (5,3) |
| front() | 5 | (5,3) |
| eraseFront() | – | (3) |
| insertBack(7) | – | (3,7) |
| back() | 7 | (3,7) |
| eraseFront() | – | (7) |
| eraseBack() | – | () |

Front

Back

# The STL Deque

- STL stack and queue are adapted from the STL deque.

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

$size()$: Return the number of elements in the deque.

$empty()$: Return true if the deque is empty and false otherwise.

$push\_front(e)$: Insert $e$ at the beginning the deque.

$push\_back(e)$: Insert $e$ at the end of the deque.

$pop\_front()$: Remove the first element of the deque.

$pop\_back()$: Remove the last element of the deque.

$front()$: Return a reference to the deque's first element.

$back()$: Return a reference to the deque's last element.

- Applying *front(), back(), pop_front(), or pop_back()* to an empty deque is undefined

# The STL Deque

- **The STL Deque Reference Manual**

## std::deque

Defined in header <deque>

```cpp
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;                                                    (1)

namespace pmr {
    template <class T>
    using deque = std::deque<T, std::pmr::polymorphic_allocator<T>>;   (2)   (since C++17)
}
```

`std::deque` (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

As opposed to `std::vector`, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one.

The storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location. On the other hand, deques typically have large minimal memory cost; a deque holding just one element has to allocate its full internal array (e.g. 8 times the object size on 64-bit libstdc++; 16 times the object size or 4096 bytes, whichever is larger, on 64-bit libc++).

The complexity (efficiency) of common operations on deques is as follows:

- Random access - constant $O(1)$
- Insertion or removal of elements at the end or beginning - constant $O(1)$
- Insertion or removal of elements - linear $O(n)$

`std::deque` meets the requirements of *Container*, *AllocatorAwareContainer*, *SequenceContainer* and *ReversibleContainer*.

### Template parameters

| | | | |
|---|---|---|---|
| T | - | The type of the elements. | |
| | | T must meet the requirements of *CopyAssignable* and *CopyConstructible*. | (until C++11) |
| | | The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of *Erasable*, but many member functions impose stricter requirements. | (since C++11) |
| Allocator | - | An allocator that is used to acquire/release memory and to construct/destroy the elements in that memory. The type must meet the requirements of *Allocator*. The behavior is undefined (until C++20) The program is ill-formed (since C++20) if `Allocator::value_type` is not the same as `T`. | |

### Element access

| | |
|---|---|
| at | access specified element with bounds checking (public member function) |
| operator[] | access specified element (public member function) |
| front | access the first element (public member function) |
| back | access the last element (public member function) |

### Capacity

| | |
|---|---|
| empty | checks whether the container is empty (public member function) |
| size | returns the number of elements (public member function) |
| max_size | returns the maximum possible number of elements (public member function) |
| shrink_to_fit (C++11) | reduces memory usage by freeing unused memory (public member function) |

### Modifiers

| | |
|---|---|
| clear | clears the contents (public member function) |
| insert | inserts elements (public member function) |
| emplace (C++11) | constructs element in-place (public member function) |
| erase | erases elements (public member function) |
| push_back | adds an element to the end (public member function) |
| emplace_back (C++11) | constructs an element in-place at the end (public member function) |
| pop_back | removes the last element (public member function) |
| push_front | inserts an element to the beginning (public member function) |
| emplace_front (C++11) | constructs an element in-place at the beginning (public member function) |
| pop_front | removes the first element (public member function) |
| resize | changes the number of elements stored (public member function) |
| swap | swaps the contents (public member function) |

# Deque Implementation: Doubly Linked List

## ▪ Doubly Linked List
- ▪ A linked list that allows to traverse in both *forward and backward directions*
- ▪ A node stores two links to the *previous* and *next* nodes
- ▪ **Sentinel node** (*i.e.*, header or trailer)
  - ▪ A specifically designated node as a traversal path terminator for convenience
  - ▪ Does not hold any data
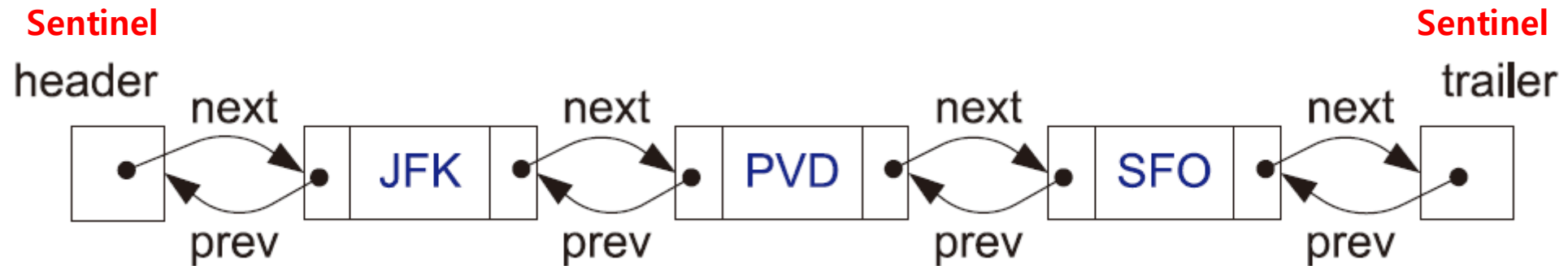
# Deque Implementation: Doubly Linked List

- **Recap: Doubly Linked List**
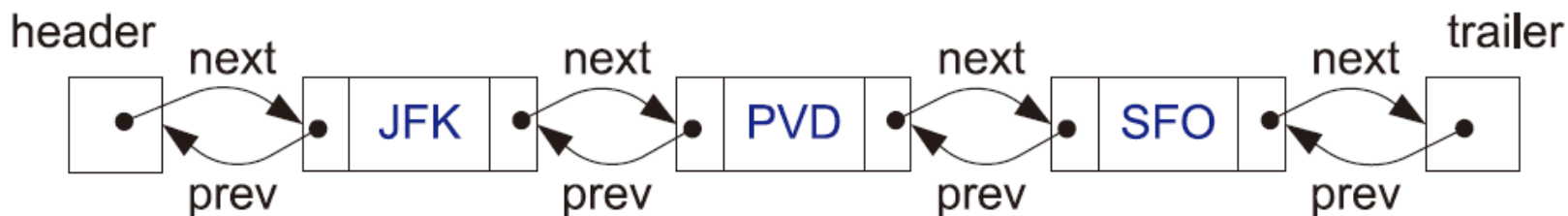


```
                                           // insert new node before v     void DLinkedList::remove(DNode* v) {     // remove node v
void DLinkedList::add(DNode* v, const Elem& e) {                            DNode* u = v->prev;                       // predecessor
  DNode* u = new DNode; u->elem = e;       // create a new node for e       DNode* w = v->next;                       // successor
  u->next = v;                             // link u in between v           u->next = w;                              // unlink v from list
  u->prev = v->prev;                       // ...and v->prev               w->prev = u;
  u->prev ->next = v->prev = u;                                            delete v;
}                                                                        }


void DLinkedList::addFront(const Elem& e) // add to front of list
  { add(header->next, e); }                                              void DLinkedList::removeFront()           // remove from font
                                                                           { remove(header->next); }

void DLinkedList::addBack(const Elem& e)  // add to back of list
  { add(trailer, e); }                                                   void DLinkedList::removeBack()            // remove from back
                                                                           { remove(trailer->prev); }
```

# Deque Implementation: Doubly Linked List

- **C++ Implementation**

```
typedef string Elem;                                    // deque element type
class LinkedDeque {                                     // deque as doubly linked list
public:
    LinkedDeque();                                      // constructor
    int size() const;                                   // number of items in the deque
    bool empty() const;                                 // is the deque empty?
    const Elem& front() const throw(DequeEmpty);        // the first element
    const Elem& back() const throw(DequeEmpty);         // the last element
    void insertFront(const Elem& e);                    // insert new first element
    void insertBack(const Elem& e);                     // insert new last element
    void removeFront() throw(DequeEmpty);               // remove first element
    void removeBack() throw(DequeEmpty);                // remove last element
private:                                                // member data
    DLinkedList D;                                      // linked list of elements
    int n;                                              // number of elements
};
```

**size() is not supported by the DLinkedList**

# Deque Implementation: Doubly Linked List

- **C++ Implementation**

```
                                      // insert new first element
void LinkedDeque::insertFront(const Elem& e) {
  D.addFront(e);
  n++;
}
                                      // insert new last element
void LinkedDeque::insertBack(const Elem& e) {
  D.addBack(e);
  n++;
}
                                      // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeFront of empty deque");
  D.removeFront();
  n--;
}
                                      // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeBack of empty deque");
  D.removeBack();
  n--;
}
```
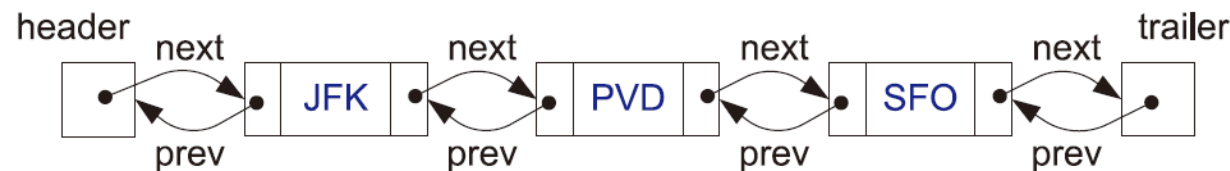


| Operation | Time |
|---|---|
| size | $O(1)$ |
| empty | $O(1)$ |
| front, back | $O(1)$ |
| insertFront, insertBack | $O(1)$ |
| eraseFront, eraseBack | $O(1)$ |

# Adapters

- **Adapters (or Wrappers)**
    - A data structure that translates one interface to another

- **Examples**

| Stack Method | Deque Implementation |
|---|---|
| size() | size() |
| empty() | empty() |
| top() | front() |
| push($o$) | insertFront($o$) |
| pop() | eraseFront() |

Table 5.3: Implementing a stack with a deque.

| Queue Method | Deque Implementation |
|---|---|
| size() | size() |
| empty() | empty() |
| front() | front() |
| enqueue($e$) | insertBack($e$) |
| dequeue() | eraseFront() |

Table 5.4: Implementing a queue with a deque.

```
std::stack

Defined in header <stack>
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

```
std::queue

Defined in header <queue>
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

Visual Intelligence and Perception Lab.

School of Computer Science and Engineering

부산대학교 PUSAN NATIONAL UNIVERSITY

# Adapters

- **Stack Implementation using Deque**

```cpp
typedef string Elem;                              // element type
class DequeStack {                                // stack as a deque
public:
  DequeStack();                                   // constructor
  int size() const;                               // number of elements
  bool empty() const;                             // is the stack empty?
  const Elem& top() const throw(StackEmpty);      // the top element
  void push(const Elem& e);                       // push element onto stack
  void pop() throw(StackEmpty);                   // pop the stack
private:
  LinkedDeque D;                                  // deque of elements
};
```

```cpp
DequeStack::DequeStack()                          // constructor
  : D() { }

int DequeStack::size() const                      // number of elements
  { return D.size(); }

bool DequeStack::empty() const                    // is the stack empty?
  { return D.empty(); }

const Elem& DequeStack::top() const throw(StackEmpty) {   // the top element
  if (empty())
    throw StackEmpty("top of empty stack");
  return D.front();
}

void DequeStack::push(const Elem& e)              // push element onto stack
  { D.insertFront(e); }

void DequeStack::pop() throw(StackEmpty)          // pop the stack
{
  if (empty())
    throw StackEmpty("pop of empty stack");
  D.removeFront();
}
```