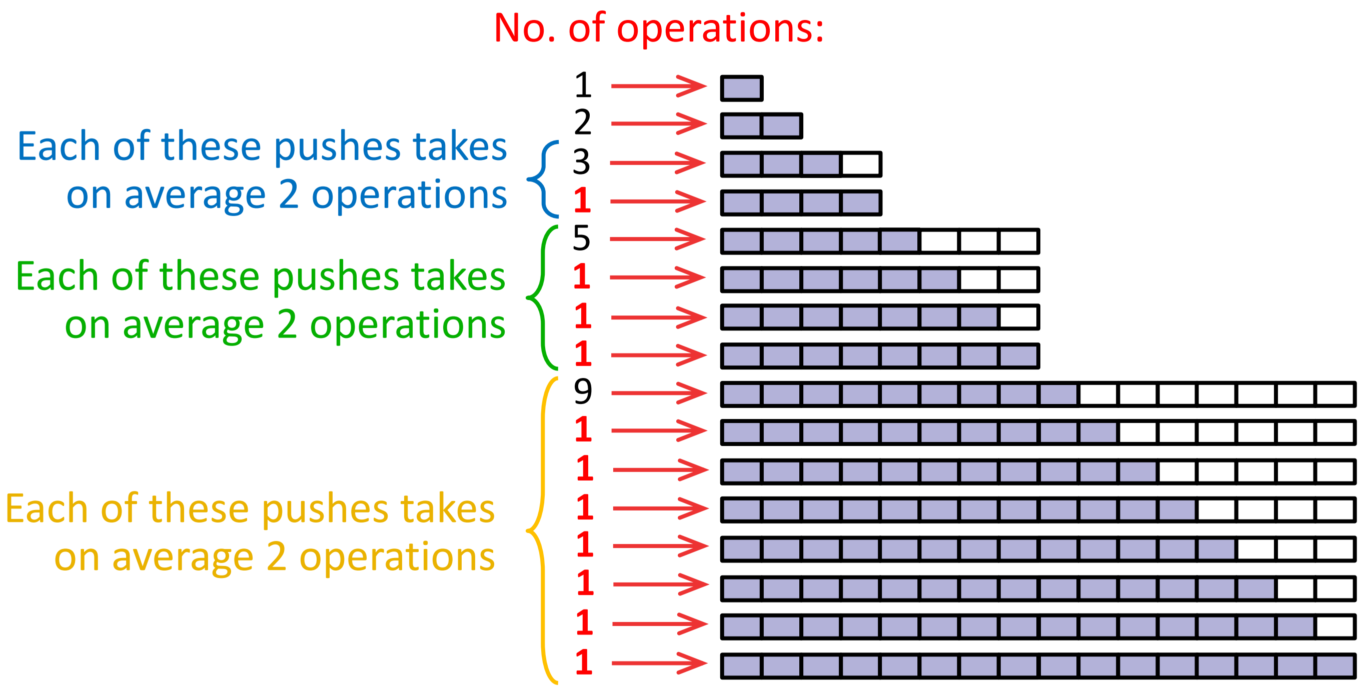
AMORTIZATION

The idea behind amortization is that a single expensive operation will make many subsequent operations much easier.

Based on this, our complexity analysis should **take the average over the expensive operation and all the subsequent, cheap operations**.



Thus, although a push takes O(n) in the worst case, it only takes O(1) time on average. Based on this, **the total time to perform a series of n pushes is just O(n) .**

VECTORS IN C++

- C++ provides a readily-available vector implemented as an extendable array.
- This comes as part of the "**Standard Template Library**" (STL) of C++, e.g., here is how to define a vector of integers:

```
#include <vector>
using std::vector;
vector<int> myVector(100);
```

This line defines `myVector` as a vector in which the elements are of type `int`; in this case, the "**base type**" is `int`

VECTORS IN C++

STL vector are similar to vector ADT, but they provide **additional features**:

- Given a vector, `x`, you can access the ith element either by writing `x[i]` (just like arrays), or by writing `x.at(i)`, which **generates an error exception if the index is out of bounds**, unlike `x[i]`.
- STL vectors provide useful functions that operate on entire vectors, e.g., to **copy** all or part of one vector to another, **compare two vectors**, **insert** and **erase** multiple elements, and **shrink_to_fit** making the capacity equal to the current size.

VECTORS IN C++

Here are the main methods provided by an STL vector:

- vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.
- size$()$: Return the number of elements in V.
- empty$()$: Return true if V is empty and false otherwise.
- resize(n): Resize V, so that it has space for n elements.
- reserve(n): Request that the allocated storage space be large enough to hold n elements.
- **operator**$[i]$: Return a reference to the ith element of V.
- at(i): Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V.\mathsf{size}()$.
- front$()$: Return a reference to the first element of V.
- back$()$: Return a reference to the last element of V.
- push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.
- pop_back$()$: Remove the last element of V, thus reducing its size by one.

24

CONTAINERS, POSITIONS, AND ITERATORS

CONTAINERS

- A container data structure stores a **collection of objects**.
- Think of it as a **generalization** of lists, stacks, queues and vectors
- We assume that the elements of a container can be arranged in a **linear order**.

POSITIONS

- A position is defined to be an abstract data type that is associated with a particular container.
- It refers to the locations of elements within. Therefore, a position in a list is the relative position or place of an element within that list
  - can return a reference to the element stored at that particular position via **overloading the dereferencing operator**, *
- A position, x, **can be thought of as a pointer**
- If the element associated with x was removed, then x is said to be "**invalidated**", i.e., no longer valid.

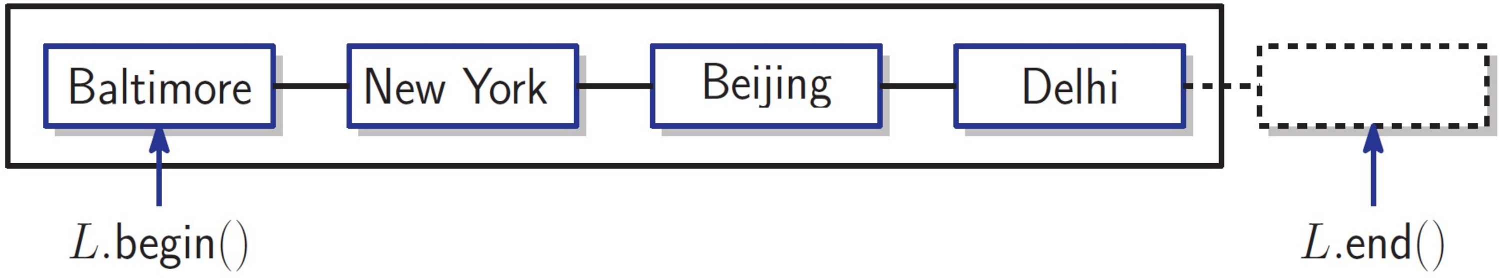
ITERATOR

- An iterator is an **extension/type of a position**.
- An iterator, x, can **access an element** of a container (just like a position), but it can also **navigate forwards** to the next element in the container, by **overloading the increment operator**, `++`; i.e. `x++`
- The same can be done with **the decrement operator**, `--`; i.e. `x--`
- This way, an iterator can be thought of as **a position that can "iterate" through the container**.

ITERATOR

- We assume that each **container** has these methods:
  - `begin()` returns an iterator that refers to the **first element** in the container
  - `end()` returns an iterator that refers to an **imaginary position that lies just after the last element in the container**

Here is an illustration given a container, L, which is a linked list:



- To enumerate all elements of a container `x`, we define an iterator `y` whose value is initialized to `x.begin()`. The associated element is accessed using `*y`. We can enumerate all elements by advancing `y` to the next node using the operation `y++`. We repeat this until `y` becomes equal to `x.end()`

29

LISTS

LISTS

- Here, we formulate the List ADT (Abstract Data Type):
  - ✓ A List ADT is **a singly-linked list** or a **doubly-linked** list that has an "iterator", i.e., a pointer that can jump from node to another in the list.
- The List ADT provides the methods `begin()` and `end()`, which return iterators to the beginning and the end of the list, respectively.
- The List ADT also provide the methods `insert(p,e)` and `erase(p)`, which take an iterator, `p`, as an argument.

LIST - ADT

- Formally, a list is an abstract data type (ADT) that supports :

Iterators

- `begin():` Return an iterator referring to the **first element** of L; this is the same as end() if L is empty.
- `end():` Return an iterator referring to **an imaginary element just after the last element** of L.

Inserters

- `insertFront(e):` **Insert** a new element e into L **as the first element**.
- `insertBack(e):` **Insert** a new element e into L **as the last element**.
- `insert(p,e):` **Insert** a new element e into L **before position p** in L.

Removers

- `eraseFront():` **Remove the first** element of L.
- `eraseBack():` **Remove the last** element of L.
- `erase(p):` **Remove** from L the element **at position p**; invalidates p.

EXAMPLE

- In this example, if we write the list L as follows:

  (x,y,z)

  it means that:
  - x is at the **front** of the list
  - z is at the **back** of the list

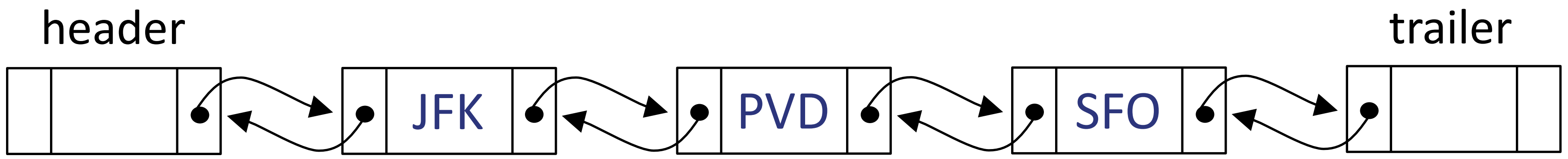
  Now, let's fill this table!

  P.S., under "**Output**" write:

  - "true" or "false" (if there is a logical comparison)
  - The element associated with an iterator (if we change it)
  - "–" otherwise

Operation	***Output***	L
insertFront(8)	–	(8)
$p =$ begin()	$p : (8)$	(8)
insertBack(5)	–	(8, 5)
$q = p;\ \ ++q$	$q : (5)$	(8, 5)
$p ==$ begin()	*true*	(8, 5)
insert(q, 3)	–	(8, 3, 5)
$*q = 7$	–	(8, 3, 7)
insertFront(9)	–	(9, 8, 3, 7)
eraseBack()	–	(9, 8, 3)
erase(p)	–	(9, 3)
eraseFront()	–	(3)

IMPLEMENTATION USING DOUBLY-LINKED LIST



- Let’s see how we can implement a list ADT using a **doubly-linked list**
- Let’s start with the implementation of a **node**, which is straightforward:

```
struct Node {
    Elem elem;
    Node* prev;
    Node* next;
};
```

IMPLEMENTATION USING DOUBLY-LINKED LIST

- The implementation of an **iterator**:

```
class Iterator {
public:
    Elem& operator*();
    bool operator==(const Iterator& p) const;
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();
    Iterator& operator––();
    friend class NodeList;
private:
    Node* v; // pointer to the node
    Iterator(Node* u); // constructor
};
```

```
Elem& NodeList::Iterator::operator*()
{ return v–>elem; }
```

```
bool NodeList::Iterator::operator==(const Iterator& p) const
{ return v == p.v; }
```

```
bool NodeList::Iterator::operator!=(const Iterator& p) const
{ return v != p.v; }
```

```
NodeList::Iterator& NodeList::Iterator::operator++() {
    v = v–>next;
    return *this;
}
```

```
NodeList::Iterator& NodeList::Iterator::operator––() {
    v = v–>prev;
    return *this;
}
```

IMPLEMENTATION USING DOUBLY-LINKED LIST

- In the next slide, we will add the **Node** and the **iterator** declarations into a class called **NodeList**, which represents the entire list.

The **Node** declaration

```
struct Node {
    Elem elem;
    Node* prev;
    Node* next;
};
```

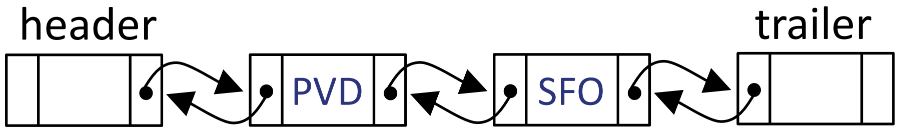
The **Iterator** declaration

```
class Iterator {
public:
    Elem& operator*();
    bool operator==(const Iterator& p) const;
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();
    Iterator& operator--();
    friend class NodeList;
private:
    Node* v; // pointer to the node
    Iterator(Node* u); // constructor
};
```

Implementation of the entire list

```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList(); // default constructor
   int size() const; // list size
   bool empty() const; // is the list empty?
   Iterator begin() const; // beginning position
   Iterator end() const; // (just beyond) last position
   void insertFront(const Elem& e); // insert at front
   void insertBack(const Elem& e); // insert at rear
   void insert(const Iterator& p, const Elem& e); // insert e before p
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```

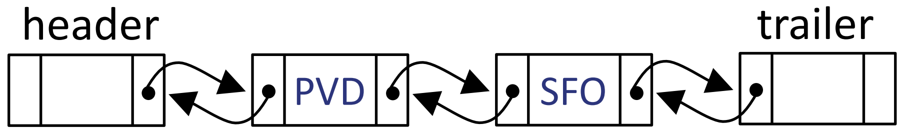
```
struct Node {
   Elem elem;
   Node* prev;
   Node* next;
};
```




```
class Iterator {
public:
   Elem& operator*();
   bool operator==(const Iterator& p) const;
   bool operator!=(const Iterator& p) const;
   Iterator& operator++();
   Iterator& operator--();
   friend class NodeList;
private:
   Node* v; // pointer to the node
   Iterator(Node* u); // constructor
};
```

Implementation of the entire list

```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList();
   int size() const;
   bool empty() const;
   Iterator begin() const;
   Iterator end() const;
   void insertFront(const Elem& e);
   void insertBack(const Elem& e);
   void insert(const Iterator& p, const Elem& e);
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```

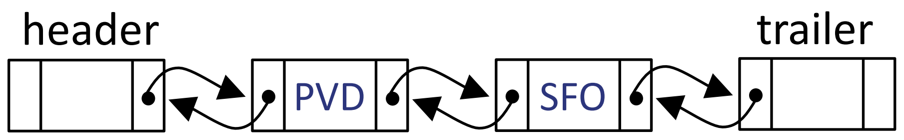



```
NodeList::NodeList() {
   n = 0;
   header = new Node;
   trailer = new Node;
   header−>next = trailer;
   trailer−>prev = header;
}
```

?

Implementation of the entire list

```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList();
   int size() const;
   bool empty() const;
   Iterator begin() const;
   Iterator end() const;
   void insertFront(const Elem& e);
   void insertBack(const Elem& e);
   void insert(const Iterator& p, const Elem& e);
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```



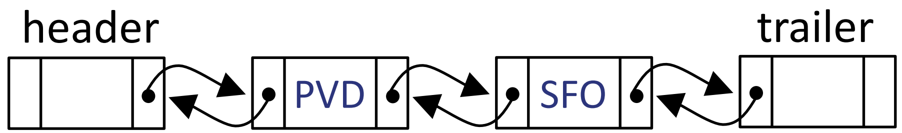

```
NodeList::NodeList() {
   n = 0;
   header = new Node;
   trailer = new Node;
   header−>next = trailer;
   trailer−>prev = header;
}
```

```
int NodeList::size() const
{   return n;   }
```

?

Implementation of the entire list

```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList();
   int size() const;
   bool empty() const;
   Iterator begin() const;
   Iterator end() const;
   void insertFront(const Elem& e);
   void insertBack(const Elem& e);
   void insert(const Iterator& p, const Elem& e);
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```




```
NodeList::NodeList() {
   n = 0;
   header = new Node;
   trailer = new Node;
   header−>next = trailer;
   trailer−>prev = header;
}
```

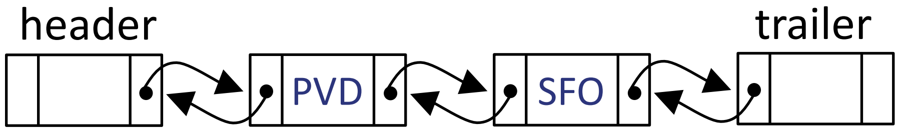
```
int NodeList::size() const
{   return n;   }
```

```
bool NodeList::empty() const
{   return (n == 0);   }
```

?

Implementation of the entire list

```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList();
   int size() const;
   bool empty() const;
   Iterator begin() const;
   Iterator end() const;
   void insertFront(const Elem& e);
   void insertBack(const Elem& e);
   void insert(const Iterator& p, const Elem& e);
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```




```
NodeList::NodeList() {
   n = 0;
   header = new Node;
   trailer = new Node;
   header−>next = trailer;
   trailer−>prev = header;
}
```

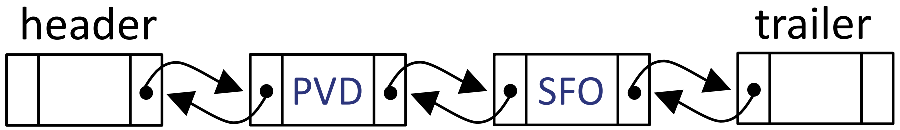
```
int NodeList::size() const
{   return n;   }
```

```
bool NodeList::empty() const
{   return (n == 0);   }
```

```
NodeList::Iterator NodeList::begin() const
{ return Iterator(header−>next); }
```

?

Implementation of the entire list




```
typedef int Elem; // list base element type
class NodeList { // node-based list
private:
   // insert Node declaration here. . .
public:
   // insert Iterator declaration here. . .
public:
   NodeList();
   int size() const;
   bool empty() const;
   Iterator begin() const;
   Iterator end() const;
   void insertFront(const Elem& e);
   void insertBack(const Elem& e);
   void insert(const Iterator& p, const Elem& e);
   void eraseFront(); // remove first
   void eraseBack(); // remove last
   void erase(const Iterator& p); // remove p
private: // data members
   int n; // number of items
   Node* header; // head-of-list sentinel
   Node* trailer; // tail-of-list sentinel
};
```

```
NodeList::NodeList() {
   n = 0;
   header = new Node;
   trailer = new Node;
   header−>next = trailer;
   trailer−>prev = header;
}
```

```
int NodeList::size() const
{   return n;   }
```

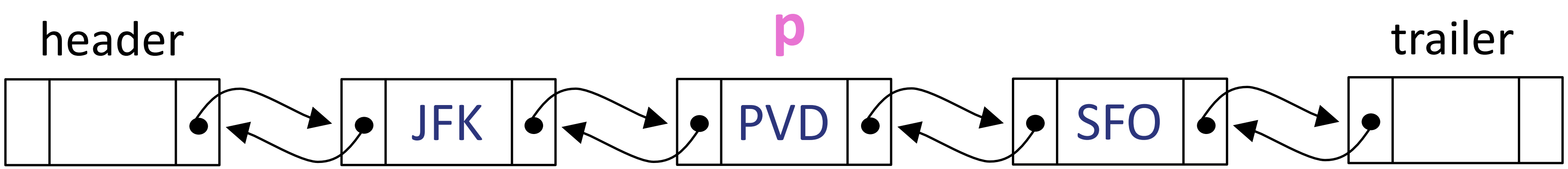
```
bool NodeList::empty() const
{   return (n == 0);   }
```

```
NodeList::Iterator NodeList::begin() const
{ return Iterator(header−>next); }
```

```
NodeList::Iterator NodeList::end() const
{   return Iterator(trailer);   }
```

DOUBLY LINKED LIST: DELETION

How would you implement the **removal** of node associated with p?

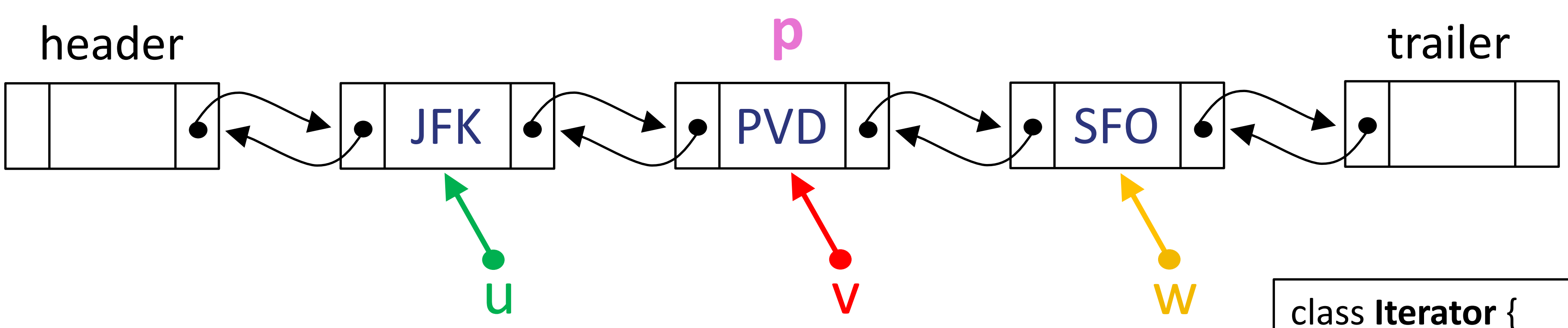



```
void NodeList::erase(const Iterator& p) {



}
```

```
class Iterator {
public:
   Elem& operator*();
   bool operator==(const Iterator& p) const;
   bool operator!=(const Iterator& p) const;
   Iterator& operator++();
   Iterator& operator--();
   friend class NodeList;
private:
   Node* v;
   Iterator(Node* u);
};
```

DOUBLY LINKED LIST: DELETION

How would you implement the **removal** of node associated with p?

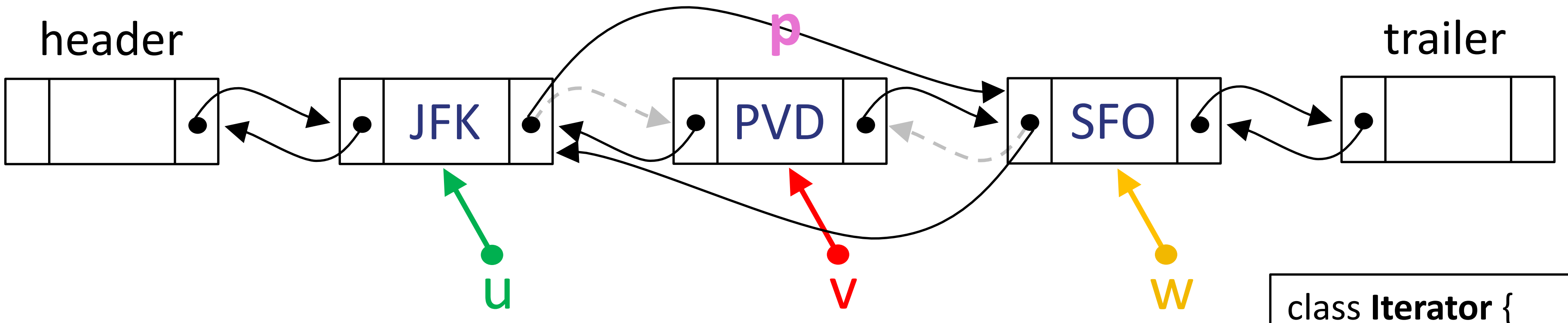



```
void NodeList::erase(const Iterator& p) {
    Node* v = p.v;
    Node* w = v->next;
    Node* u = v->prev;



}
```

```
class Iterator {
public:
   Elem& operator*();
   bool operator==(const Iterator& p) const;
   bool operator!=(const Iterator& p) const;
   Iterator& operator++();
   Iterator& operator--();
   friend class NodeList;
private:
   Node* v;
   Iterator(Node* u);
};
```

DOUBLY LINKED LIST: DELETION

How would you implement the **removal** of node associated with p?

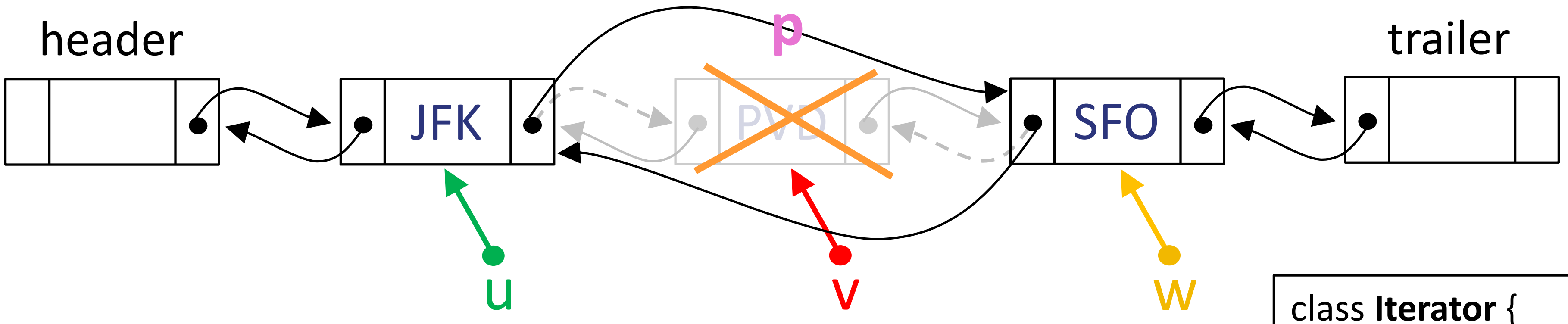



```
void NodeList::erase(const Iterator& p) {
    Node* v = p.v;
    Node* w = v->next;
    Node* u = v->prev;
    u->next = w; w->prev = u;


}
```

```
class Iterator {
public:
    Elem& operator*();
    bool operator==(const Iterator& p) const;
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();
    Iterator& operator--();
    friend class NodeList;
private:
    Node* v;
    Iterator(Node* u);
};
```

DOUBLY LINKED LIST: DELETION

How would you implement the **removal** of node associated with p?

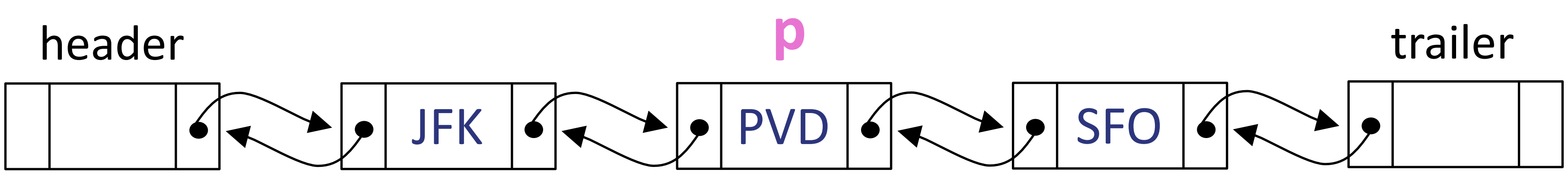



```
void NodeList::erase(const Iterator& p) {
    Node* v = p.v;
    Node* w = v->next;
    Node* u = v->prev;
    u->next = w; w->prev = u;
    delete v;
    n--; //update the number of elements
}
```

```
class Iterator {
public:
    Elem& operator*();
    bool operator==(const Iterator& p) const;
    bool operator!=(const Iterator& p) const;
    Iterator& operator++();
    Iterator& operator--();
    friend class NodeList;
private:
    Node* v;
    Iterator(Node* u);
};
```

DOUBLY LINKED LIST: INSERTION

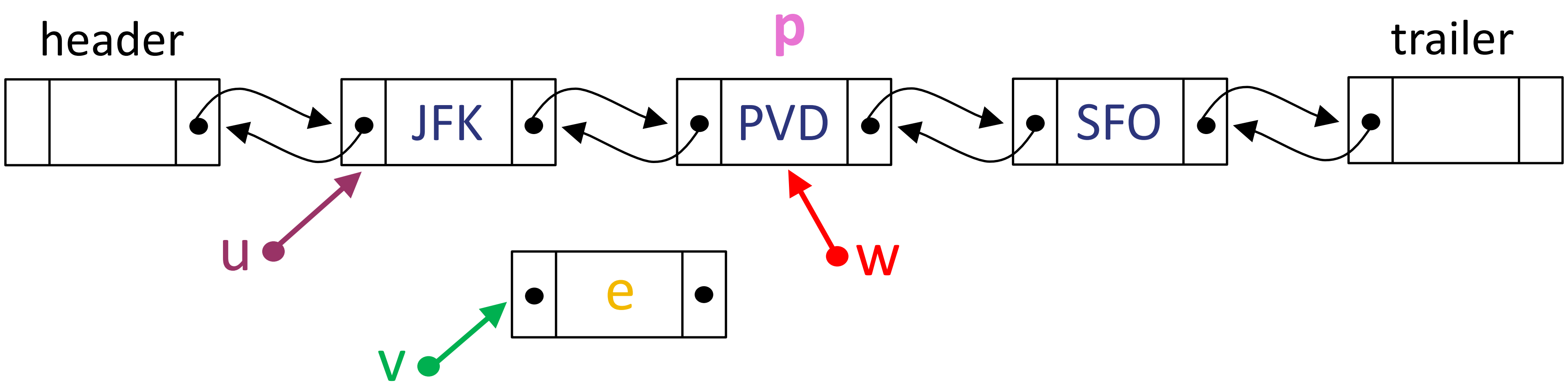
How would you implement the **insertion** of element e just before p?




```
void NodeList::insert(const Iterator& p, const Elem& e) {



}
```

DOUBLY LINKED LIST: INSERTION

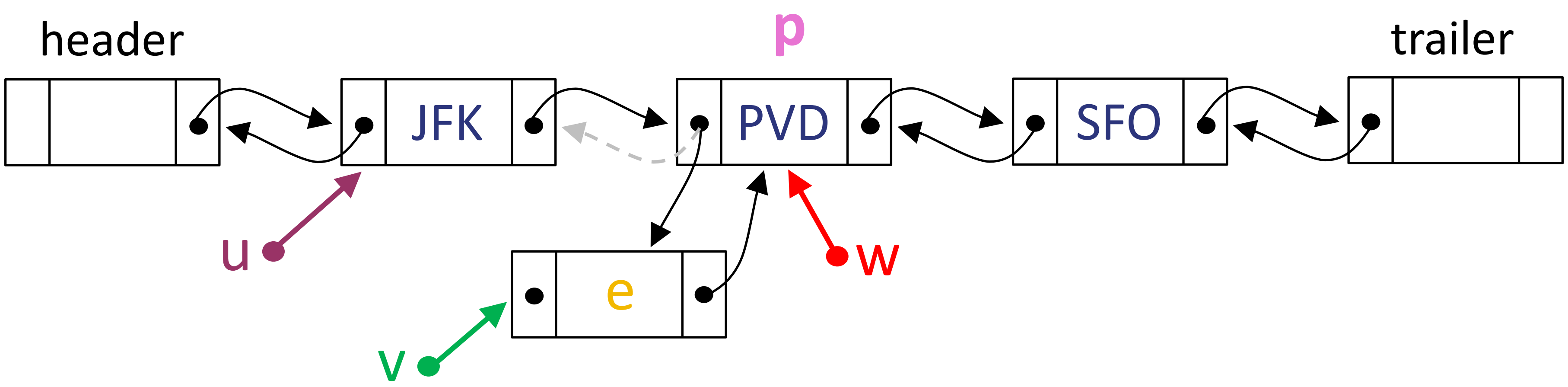
How would you implement the **insertion** of element e just before p?




```
void NodeList::insert(const Iterator& p, const Elem& e) {
    Node* w = p.v;  //w points to the node associated with p
    Node* u = w->prev;
    Node* v = new Node;
    v->elem = e;



}
```

DOUBLY LINKED LIST: INSERTION

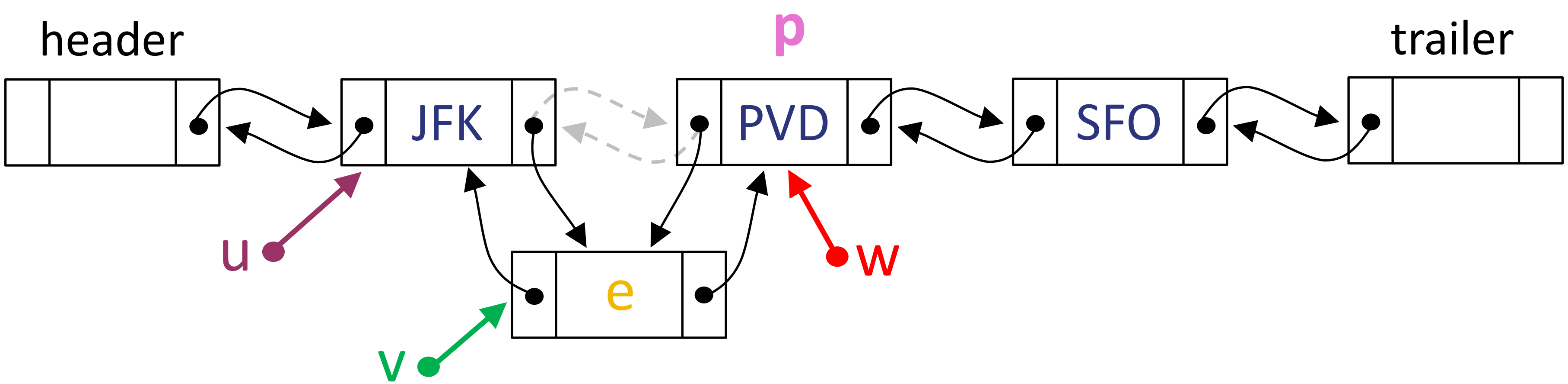
How would you implement the **insertion** of element e just before p?




```
void NodeList::insert(const Iterator& p, const Elem& e) {
    Node* w = p.v;  //w points to the node associated with p
    Node* u = w->prev;
    Node* v = new Node;
    v->elem = e;
    v->next = w; w->prev = v;


}
```

DOUBLY LINKED LIST: INSERTION

How would you implement the **insertion** of element e just before p?




```
void NodeList::insert(const Iterator& p, const Elem& e) {
    Node* w = p.v;  //w points to the node associated with p
    Node* u = w->prev;
    Node* v = new Node;
    v->elem = e;
    v->next = w; w->prev = v;
    v->prev = u; u->next = v;
    n++; //update the number of elements
}
```

LISTS IN C++

- C++ provides a readily-available implementation of a lists.
- This comes as part of the "**Standard Template Library**" (STL) of C++, e.g., here is how to define a list of floats:

```
#include <list>
using std::list;
list<float> myList;
```

This line defines `myList` as a list in which the elements are of type `float`; in this case, the "**base type**" is `float`

LISTS IN C++

- Here are the methods supported by the STL list:

list(n): **Construct** a list with *n* elements; if no argument list is given, an empty list is created.

size(): Return the **number of elements** in *L*.

empty(): **Return true if *L* is empty** and false otherwise.

front(): Return a reference to the **first element** of *L*.

back(): Return a reference to the **last element** of *L*.

push_front(e): **Insert** a copy of *e* at the **beginning** of *L*.

push_back(e): **Insert** a copy of *e* at the **end** of *L*.

pop_front(): **Remove the fist** element of *L*.

pop_back(): **Remove the last** element of L.

52

STL CONTAINERS & ITERATORS

STL CONTAINERS

- C++'s Standard Template Library (STL) provides a variety of containers, many of which will be discussed later…

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

ITERATING THROUGH A CONTAINER

- How would you iterate through a container, say, to sum up all its elements?
- Let’s take **vectors** as an example…

```
int vectorSum1(const vector<int>& V) {
    int sum = 0;
    for (int i = 0; i < V.size(); i++)
        sum += V[i];
    return sum;
}
```

- Unfortunately**, this method is not applicable to other types of containers** because it relies on the fact that the elements of a vector can be accessed efficiently through *indexing*. This is not true for all containers, such as lists.
- An alternative method that works on all containers is to use an ***iterator***!

ITERATING THROUGH A CONTAINER

- How would you iterate through a container, say, to sum up all its elements?
- Let's take **vectors** as an example…

```
int vectorSum1(vector<int>& V) {
    int sum = 0;
    for (int i = 0; i < V.size(); i++)
        sum += V[i];
    return sum;
}
```

Here is an example of how to use an **iterator**:

```
int vectorSum2(vector<int> V) {
    typedef vector<int>::iterator Iterator;
    int sum = 0;
    for (Iterator p = V.begin(); p != V.end(); ++p)
        sum += *p;
    return sum;
}
```

- Notice that the vector on the left was **passed by reference** (see the "&") whereas on the right it was **passed by value**. What difference will this make?

Passing by value means that the function would **create a copy** of the vector, which would needlessly waste memory space!

ITERATING THROUGH A CONTAINER

- We can pass the vector by reference (to avoid working on a copy of it), but this would **allow the function to modify the vector's content**. If we want to avoid this, what should we do? Use a **constant reference**

```
int vectorSum2(vector<int> V) {
   typedef vector<int>::iterator Iterator;
   int sum = 0;
   for (Iterator p = V.begin(); p != V.end(); ++p)
      sum += *p;
   return sum;
}
```

```
int vectorSum3(const vector<int>& V) {
   typedef vector<int>::const_iterator Iterator;
   int sum = 0;
   for (Iterator p = V.begin(); p != V.end(); ++p)
      sum += *p;
   return sum;
}
```