DATA STRUCTURES
CS-UH 1050, SPRING 2020

Lecture IV – Vectors and Linked lists

Prof. Mai Oudah

LECTURE OUTLINES

I. Vectors
II. Linked lists

VECTORS

STL (STANDARD TEMPLATE LIBRARY) VECTORS

- Vectors are same as dynamic arrays, but with the ability to resize itself automatically when an element is inserted or deleted
  - Compared to array, vector consumes more memory in exchange for the ability to grow/shrink dynamically
- You need to load <vector> library header file:

```
#include <vector>
```

VECTOR ITERATORS FUNCTIONS

begin()	Returns an iterator pointing to the first element in the vector
end()	Returns an iterator pointing to the theoretical element that follows the last element in the vector
rbegin()	Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
rend()	Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
cbegin()	Returns a constant iterator pointing to the first element in the vector.
cend()	Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
crbegin()	Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
crend()	Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

VECTOR **CAPACITY** FUNCTIONS

size()	Returns the number of elements in the vector.
max_size()	Returns the maximum number of elements that the vector can hold.
capacity()	Returns the size of the storage space currently allocated to the vector expressed as number of elements.
resize()	Resizes the container so that it contains ‘n’ elements.
empty()	Returns whether the container is empty.
shrink_to_fit()	Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
reverse()	Requests that the vector capacity be at least enough to contain n elements.

VECTOR MODIFIER FUNCTIONS

assign()	It assigns new values to the vector.
push_back()	It adds a new element at the end.
pop_back()	It removes a last element from the vector.
insert()	It inserts new elements before the element at the specified position.
erase()	It deletes the specified element(s).
swap()	It is used to swap the contents of one vector with another vector of same type.
clear()	It removes all the elements from the vector.

EXAMPLE

```
#include<iostream>
#include<vector>
…
int main()
{
vector<string> v1; //declare the vector (see, it's template library)
v1.push_back("Data ");  //add element to the vector
v1.push_back("Structures");

for(int i=0; i<v1.size(); i++)
        cout<<v1[i];
return 0;
}
```

Output:
Data Structures

LINKED LISTS

LINKED LISTS

- A **linked list** is a data structure, which contains a collection of **nodes**
- Each node stores data, in addition to a pointer (called *next*) to the node containing the next item
- This structure is called a ***singly linked list*** because each node stores a single link.

SINGLY LINKED LISTS

- The first node = Head
- The last node = Tail (has a null *next* pointer)

- The order is determined by the chain of *next* links
- No predetermined fixed size

SINGLY LINKED LISTS: NODE CLASS DEFINITION

Example:

```
class IntNode {
private:
        int elem; // element value
        IntNode* next; // pointer to the next item in the list


        friend class IntLinkedList; //provide access to the linkedlist
    };
```

SINGLY LINKED LISTS: LINKED LIST CLASS DEFINITION

Example:

```
class IntLinkedList {
public:
        IntLinkedList(): head(NULL) {};    // empty list constructor, points the head to NULL
        ~IntLinkedList();                  // destructor
        bool empty() const;                // is list empty?
        const int& front() const;          // get front element
        void addFront(const int& e);       // add to front of list
        void removeFront();                // remove front item list
private:
        IntNode* head;
};
```

SINGLY LINKED LISTS: LINKED LIST CLASS - METHODS

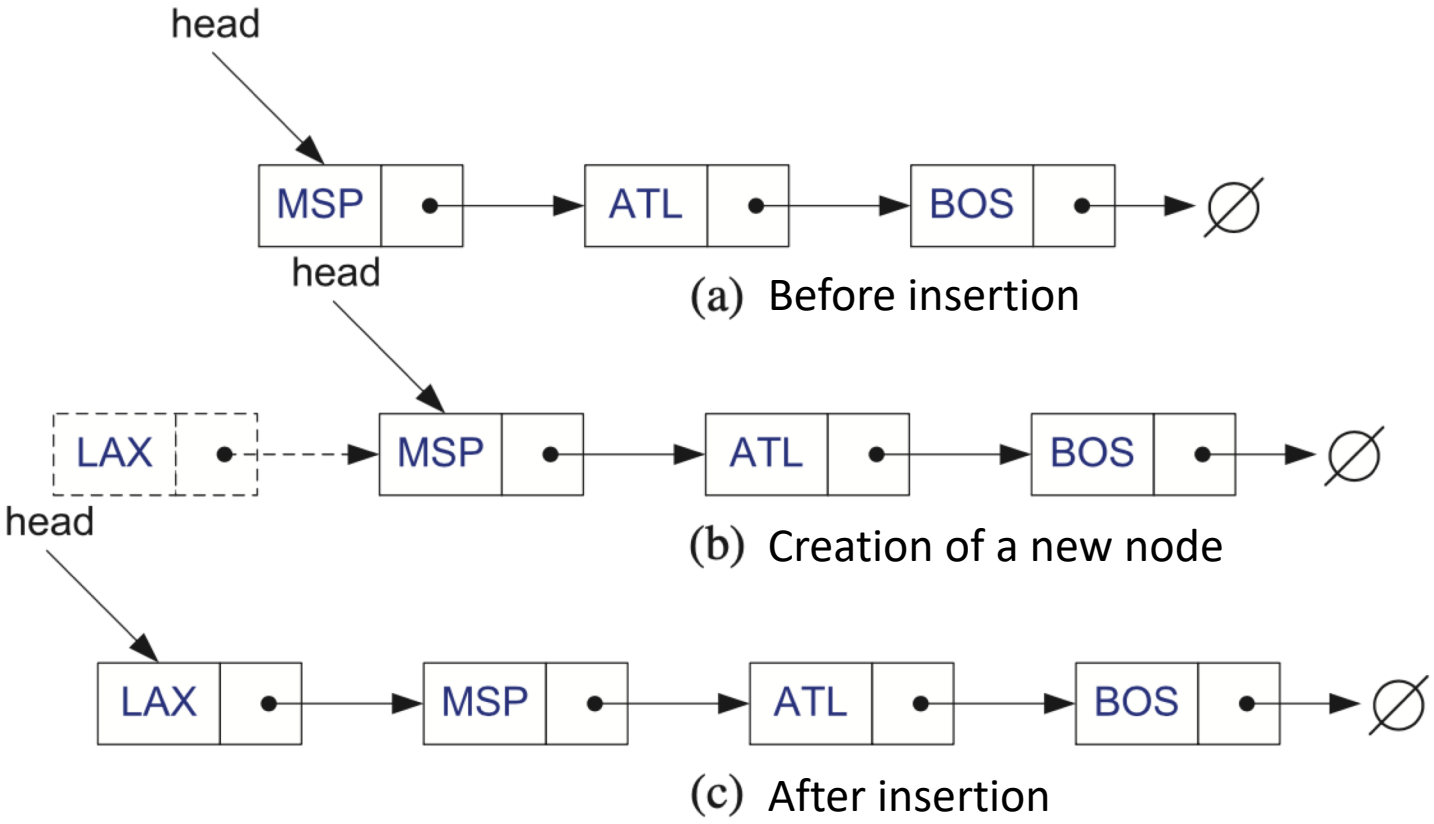
Example (some of the methods):

```
IntLinkedList::~IntLinkedList()
      { while (!empty()) removeFront(); }   // destructor


bool IntLinkedList::empty() const
      { return head == NULL; }        // checks if list is empty


const int& IntLinkedList::front() const
      { return head->elem; }          // get front element
```

SINGLY LINKED LIST: INSERTION TO THE FRONT (HEAD)

1. Create a new node
2. Set its *elem* value to the desired one and set its *next* link to point to the current head of the list.
3. Set *head* to point to the new node.

SINGLY LINKED LIST: INSERTION TO THE FRONT (HEAD)



SINGLY LINKED LIST: INSERTION TO THE FRONT (HEAD)

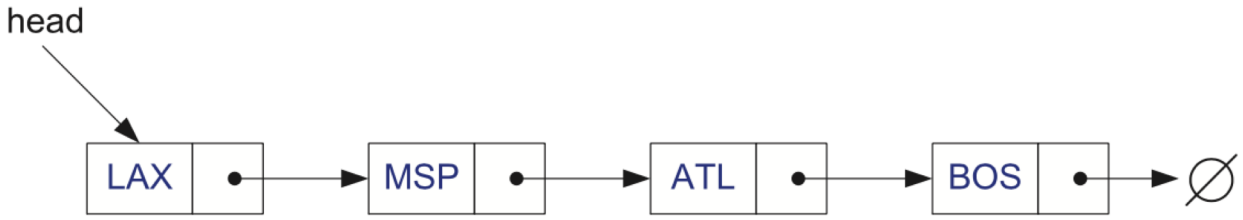
Example:

```
void IntLinkedList::addFront(const int& e) {

IntNode* v = new IntNode;              // create new node
v->elem = e;                           // store data
v->next = head;                        // head now follows v

head = v;                              // v is now the head

}
```

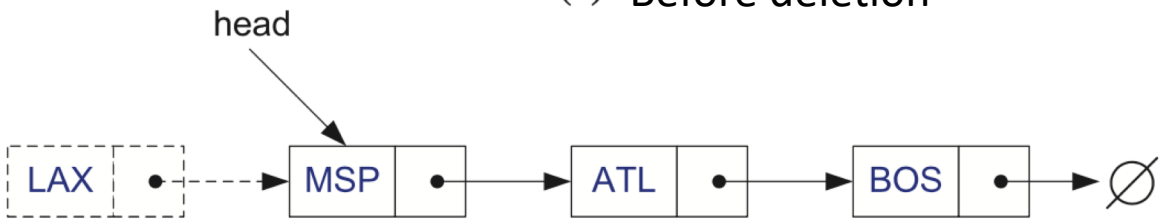
SINGLY LINKED LIST: DELETION FROM THE FRONT (HEAD)

1. Save a pointer to the old head node
2. Advance the head pointer to the next node in the list
3. Delete the old head node

SINGLY LINKED LIST: DELETION FROM THE FRONT (HEAD)



(a) Before deletion



(b) "linking out" the old new node

(c) After deletion

SINGLY LINKED LIST: DELETION FROM THE FRONT (HEAD)

Example:

```
void IntLinkedList::removeFront() {

IntNode* old = head;              // save current head
head = old->next;                 // skip over old head
delete old;                       // delete the old head

}
```

SINGLY LINKED LIST: TEMPLATE IMPLEMENTATION

- We can generate singly linked lists of *various types*

1. Present the template for node class (e.g. Node)
2. The element type associated with each node is parameterized by the type variable, e.g. E
3. Present the template for linked list class. Use references to Node and E.

SINGLY LINKED LIST: TEMPLATE NODE CLASS DEFINITION

Example:

```
template <typename E>  //typename or class
class Node {
private:

      E elem;              // linked list element value
      Node<E>* next;       // next item in the list

      friend class LinkedList<E>; // provide LinkedList access
}
```

SINGLY LINKED LIST: TEMPLATE LINKED LIST CLASS DEFINITION

Example:

```
template <typename E> //typename or class
class LinkedList {
public:
        LinkedList();           //constructor
        ~LinkedList();          //destructor
        bool empty() const;
        const E& front() const;
        void addFront(const E& e);
        void removeFront();
private:
        Node<E>* head; }        // head of the list
```

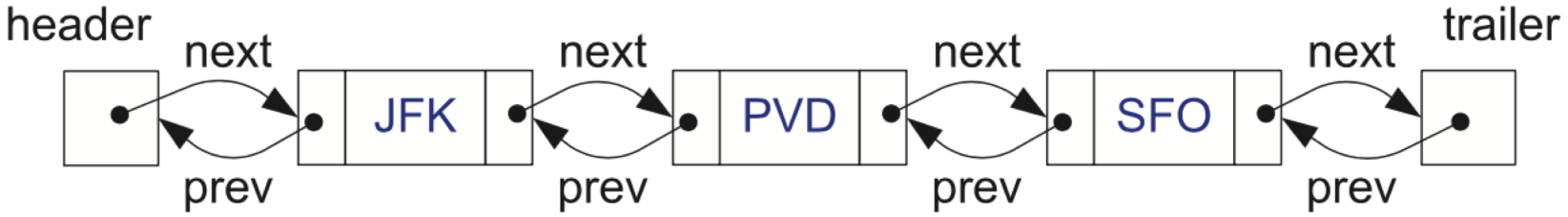
SINGLY LINKED LIST: TEMPLATE LINKED LIST CLASS - METHODS

Example (some of the methods):

```
template <typename E>
const E& LinkedList::front() const
      { return head->elem; }          // get front element

template <typename E>
void LinkedList::addFront(const E& e) {
Node<E>* v = new Node<E>;             // create new node
v->elem = e;                          // store data
v->next = head;                       // head now follows v
head = v; }         // v is now the head
```

DOUBLY LINKED LISTS

- It allows us to go in both directions—forward and reverse—in a linked list. Such lists allow for quick update operations (insertion/deletion at any point)
- A node in a doubly linked list stores two pointers:
  - *next:* points to the next node in the list
  - *prev:* points to the previous node in the list
- It has two special nodes that do not store any elements:
  - *header* node: just before the head of the list
  - *trailer/tail* node: just after the tail of the list



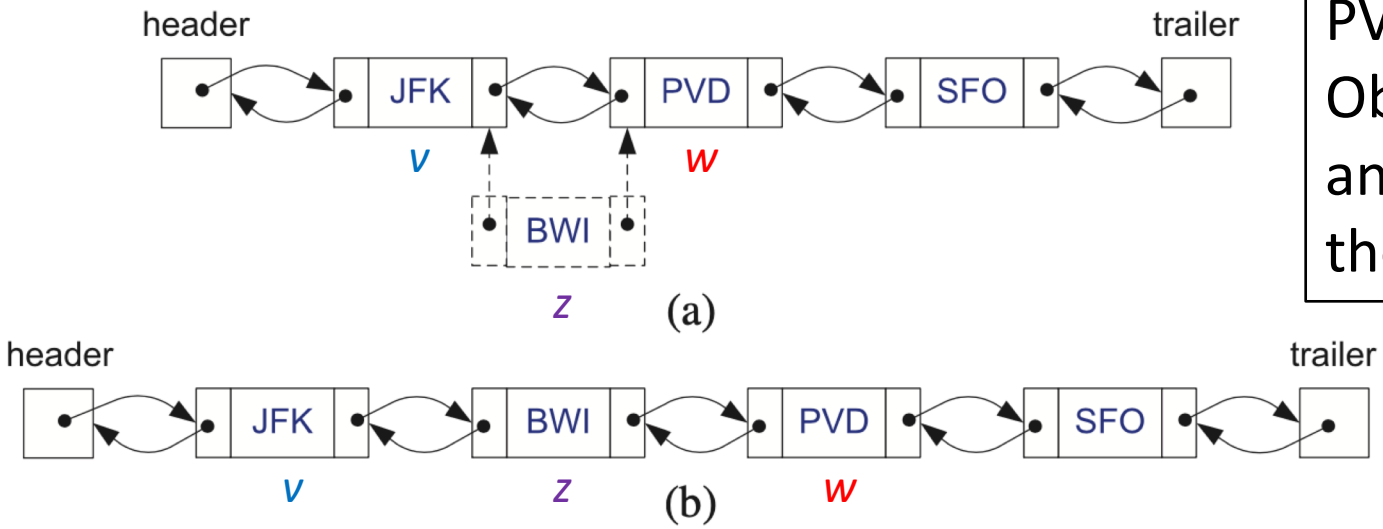
DOUBLY LINKED LIST: INSERTION

- It is possible to insert a node at any position.
- Let *w* be the node after *v*. To insert *z* after *v*, we link it into the current list as follows:

1. Make *z*'s *prev* link point to *v*
2. Make *z*'s *next* link point to *w*
3. Make *w*'s *prev* link point to *z*
4. Make *v*'s *next* link point to *z*

DOUBLY LINKED LIST: INSERTION

- Example:

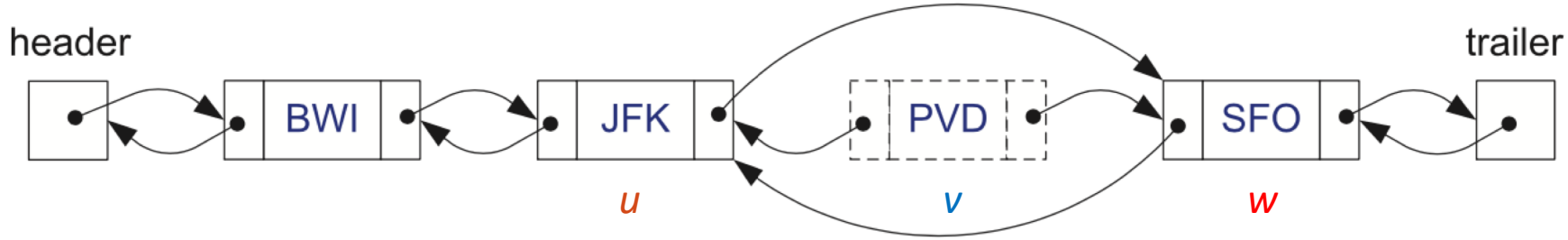


where *v* is the node JFK, *w* is the node PVD, and *z* is the new node BWI. Observe that this process works if *v* is any node ranging from the header to the node just prior to the trailer!

DOUBLY LINKED LIST: DELETION

- Let *u* be the node before *v*, and *w* be the node after *v*. To remove node *v*, we have *u* and *w* point to each other instead of to *v*, *i.e.* ***linking out*** of *v:*
  - Make *w*'s *prev* link point to *u*
  - Make *u*'s *next* link point to *w*
  - Delete node *v*

Example:



DOUBLY LINKED LIST: NODE CLASS DEFINITION

Example:

```
typedef string Elem; // define the type Elem to be string (can be any data type)
class DNode {
private:
        Elem elem;  // node element value of the type defined earlier
        DNode* prev;              // previous node in the list
        DNode* next;               // next node in the list
        friend class DLinkedList;
};
```

DOUBLY LINKED LIST: LINKED LIST CLASS DEFINITION

Example:

```
class DLinkedList {
public:
        DLinkedList();          // constructor
        ~DLinkedList();         // destructor
        bool empty() const;
        const Elem& front() const;
        const Elem& back() const;
        void addFront(const Elem& e);
        void addBack(const Elem& e);
        void removeFront();
        void removeBack();
private:
        DNode* header;
        DNode* trailer;
protected:
        void add(DNode* v, const Elem& e); //insert new node before v
        void remove(DNode* v); };          // remove node v
```

DOUBLY LINKED LISTS: LINKED LIST CLASS - METHODS

Example (some of the methods):

```
DLinkedList::DLinkedList() {          // constructor
header = new DNode;
trailer = new DNode;
header->next = trailer;
trailer->prev = header; }

DLinkedList::~DLinkedList() {         // destructor
while (!empty()) removeFront();
delete header;
delete trailer;  }

const Elem& DLinkedList::front() const {   // get the front element
return header->next->elem; }

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

DOUBLY LINKED LISTS: LINKED LIST CLASS – ADD NODE

Example:

```
// insert new node before v
void DLinkedList::add(DNode* v, const Elem& e) {
// create a new node for e
DNode* u = new DNode;
u->elem = e;
// link u in between v and v->prev
u->next = v;
u->prev = v->prev;
v->prev->next = v->prev = u; // make u the next for the
                             // node before v
}
```

DOUBLY LINKED LISTS: LINKED LIST CLASS – DELETE NODE

Example:

```
// remove node v
void DLinkedList::remove(DNode* v) {
DNode* u = v->prev; // the previous node
DNode* w = v->next; // the next node

// unlink v from list
u->next = w;
w->prev = u;
delete v;

}
```

NEXT LECTURE

- Continue with Linked lists and Recursion

- Readings:
  - Chapter 3