بسم الله الرحمن الرحیم

ساختمان‌های داده

جلسه ۱۲

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

پیاده‌سازی لیست

- ADT فقط درباره عملگرها صحبت می‌کند و درباره پیاده‌سازی حرفی نمیزند.

- دو پیاده‌سازی متداول:
  - ArrayList
  - LinkedList

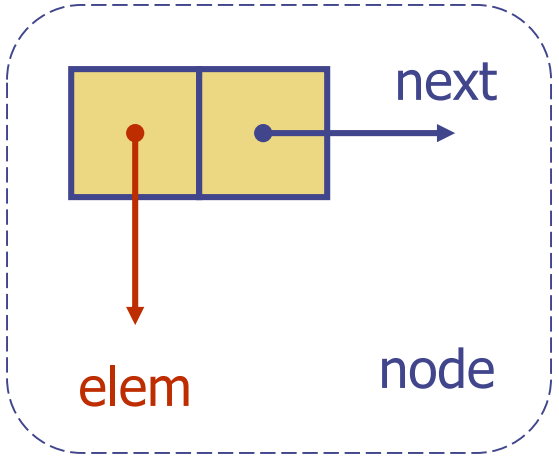
لیست پیوندی (LinkedList)

- یک ساختمان داده دیگر که می‌توان از آن هم برای پیاده‌سازی لیست استفاده کرد.
  - با همان رابط‌های قبلی (اضافه به ابتدا، حذف، ...)
  - با بده بستان‌های متفاوت

لیست پیوندی (LinkedList)

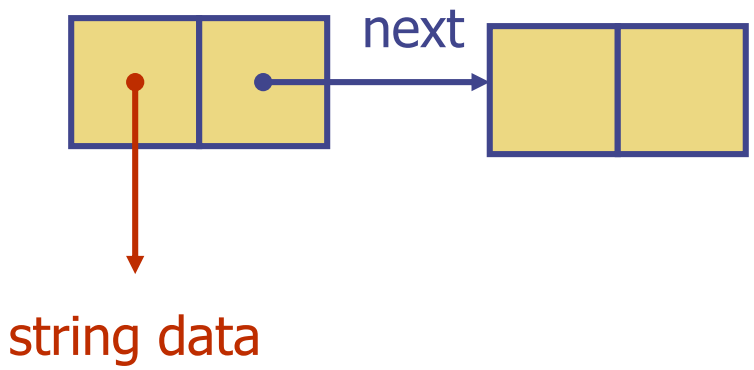
IUT-ECE

- لیست پیوندی شامل دنباله‌ای از گره‌هاست (خطی).
- هر گره شامل دو بخش است:
  - داده
  - اشاره‌گر به گره بعدی
- هر گره در هرجایی از حافظه می‌تواند ذخیره شود.



لیست پیوندی (LinkedList)

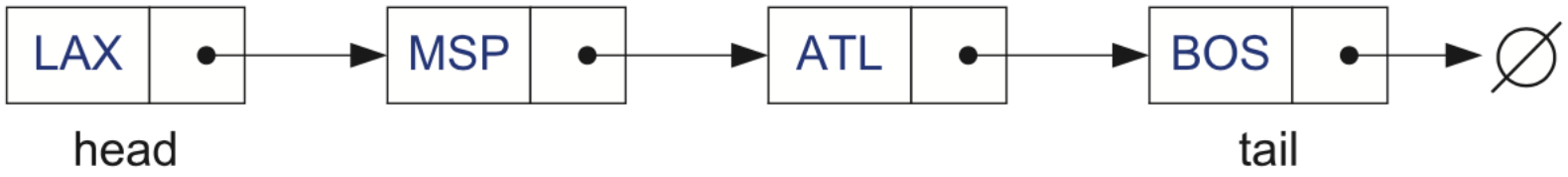
IUT-ECE




```
class Node {
   string data;
   Node* next;
}
```

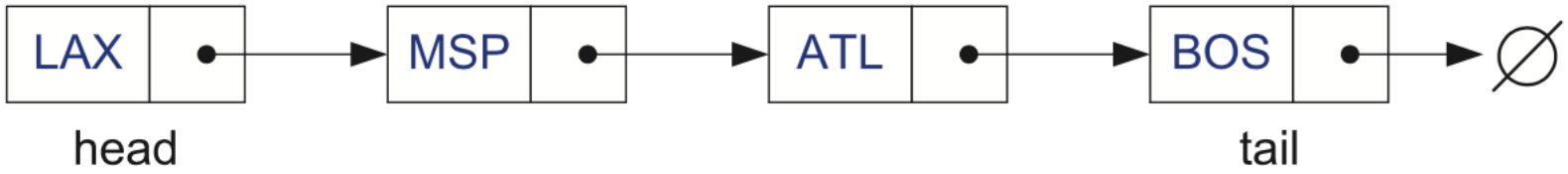
لیست پیوندی (LinkedList)

- لیست پیوندی شامل دنباله‌ای از گره‌هاست (خطی).
- بنابراین برای نمایش این لیست نیاز است:
  - head
  - tail



لیست پیوندی (LinkedList)

- فقط نیاز است head را دنبال کنیم تا به گره‌های بعدی برسیم.
- برای اینکه متوجه شویم به انتهای لیست رسیده‌ایم از nullptr استفاده میکنیم.
- خبری از اندیس نیست.

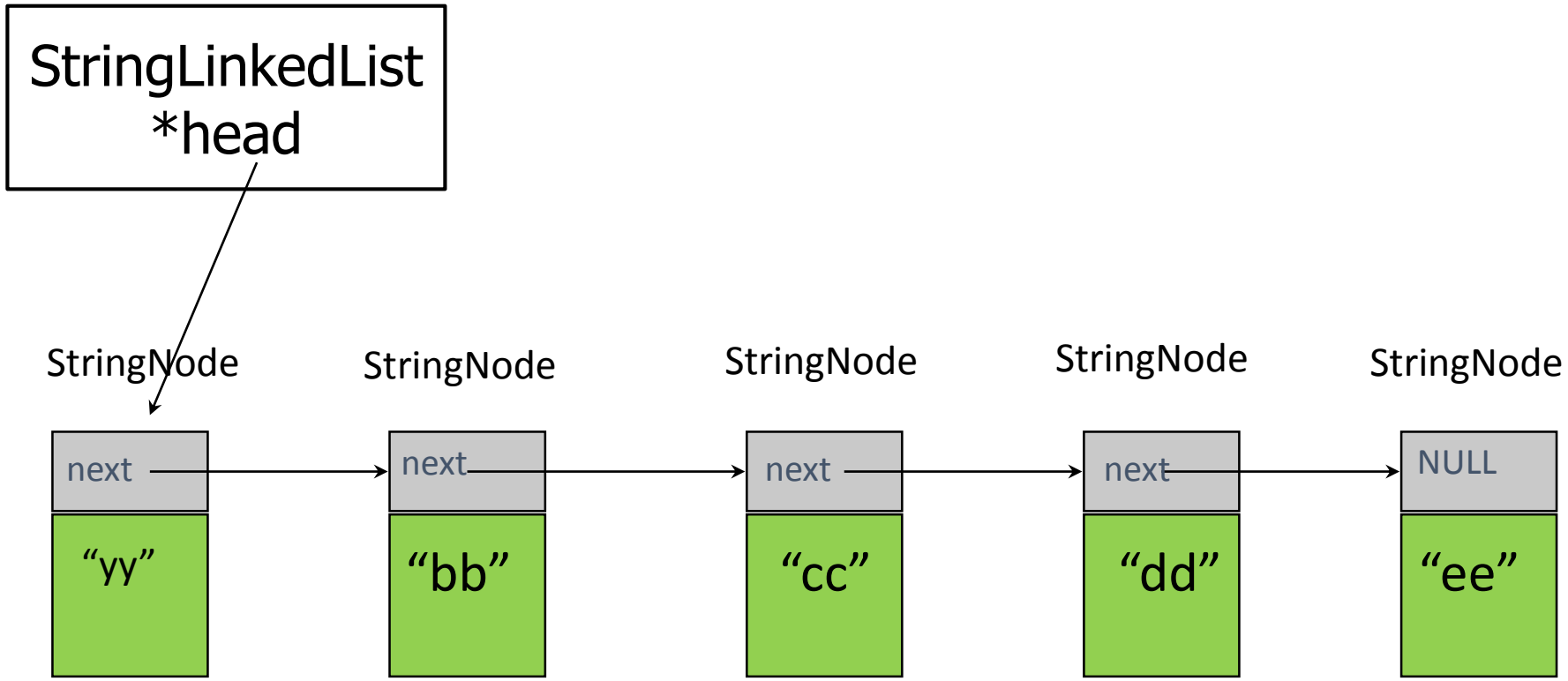


مثال: لیست پیوندی رشته‌ها


```
class StringNode {                          // a node in a list of strings
private:
  string elem;                              // element value
  StringNode* next;                         // next item in the list

  friend class StringLinkedList;            // provide StringLinkedList access
};
```

```
class StringLinkedList {                    // a linked list of strings
public:
  StringLinkedList();                       // empty list constructor
  ~StringLinkedList();                      // destructor
  bool empty() const;                       // is list empty?
  const string& front() const;              // get front element
  void addFront(const string& e);           // add to front of list
  void removeFront();                       // remove front item list
private:
  StringNode* head;                         // pointer to the head of list
};
```

مثال: لیست پیوندی رشته‌ها



لیست پیوندی (LinkedList)

- زمان مورد نیاز برای محاسبه طول لیست؟

$$O(n)$$

لیست پیوندی (LinkedList)

- زمان مورد نیاز برای پیدا کردن tail؟

$$O(n)$$

لیست پیوندی (LinkedList)

- برای برخی از کاربردها نیاز دسترسی به tail و دانستن اندازه لیست داریم.
- چگونه این زمان را بهبود دهیم؟

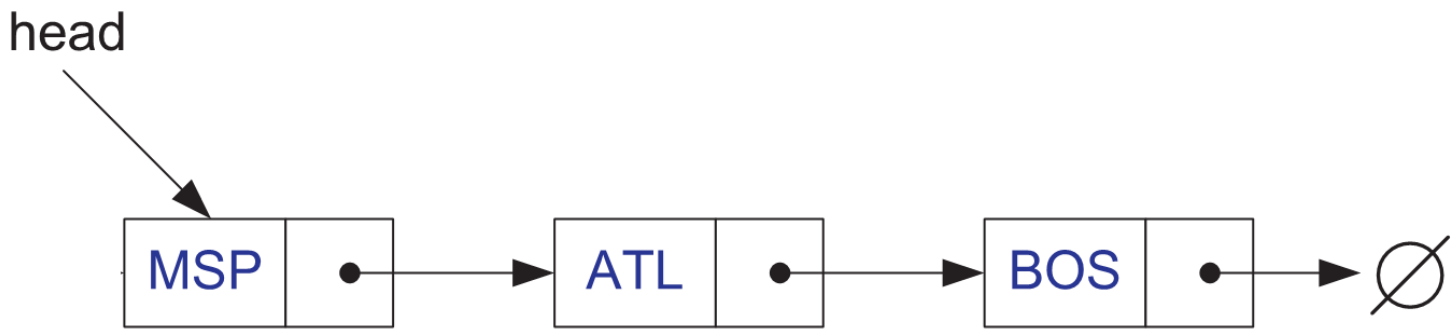
```
class StringLinkedList {     // a linked list of strings
public:
...
private:
StringNode* head;          // pointer to the head of list
StringNode* tail;          // pointer to the tail of list
int size
};
```

$$O(1)$$

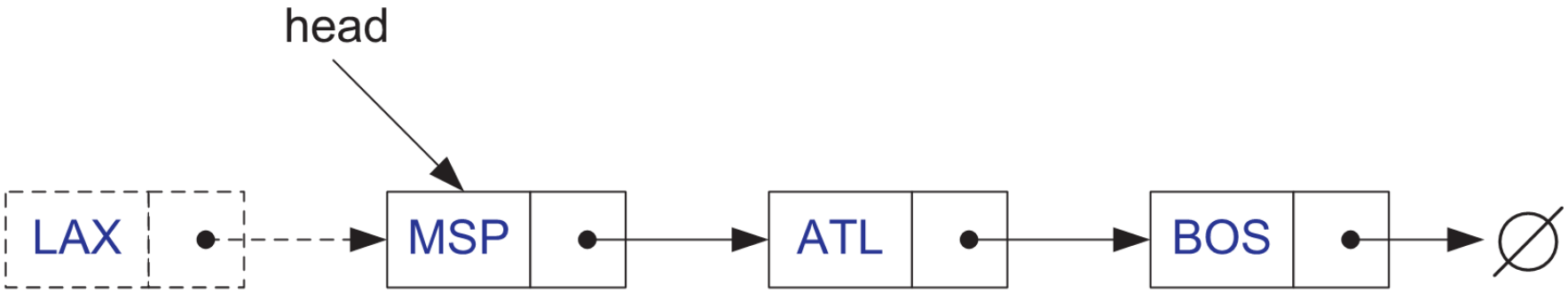
Insertion to the Front of a Singly Linked List

1. Allocate a new node
2. Insert a new element
3. Have the new node point to the old head
4. Update head to point to new node

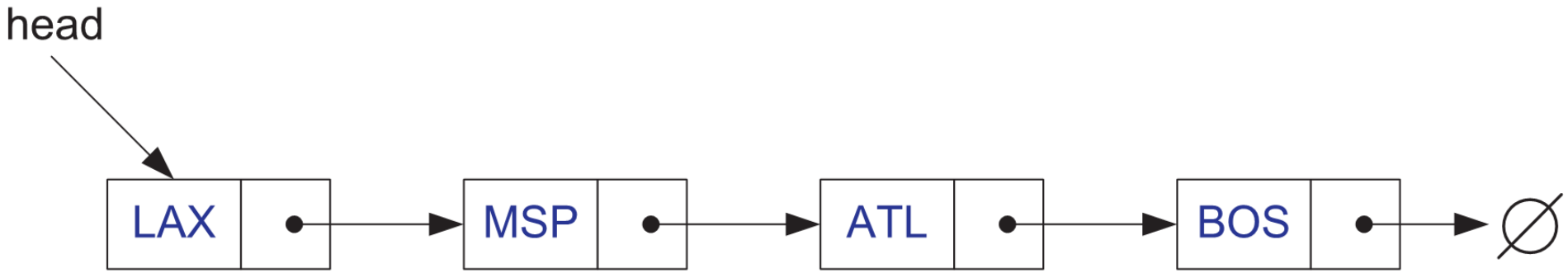
Insertion to the Front of a Singly Linked List



Insertion to the Front of a Singly Linked List



Insertion to the Front of a Singly Linked List



Insertion to the Front of a Singly Linked List

IUT-ECE

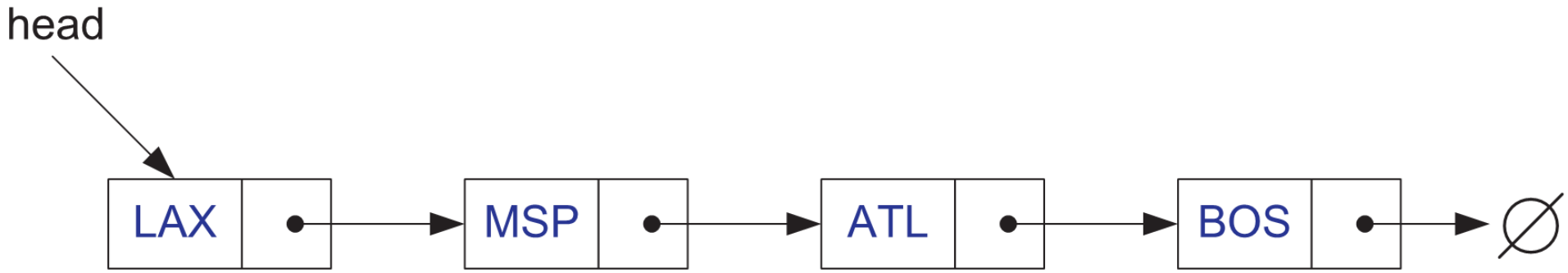
```
void StringLinkedList::addFront(const string& e) {  // add to front of list
  StringNode* v = new StringNode;                    // create new node
  v->elem = e;                                       // store data
  v->next = head;                                    // head now follows v
  head = v;                                          // v is now the head
}
```

Insertion to the Front of a Singly Linked List

- زمان لازم؟

$$O(1)$$

Removal from the Front of a Singly Linked List

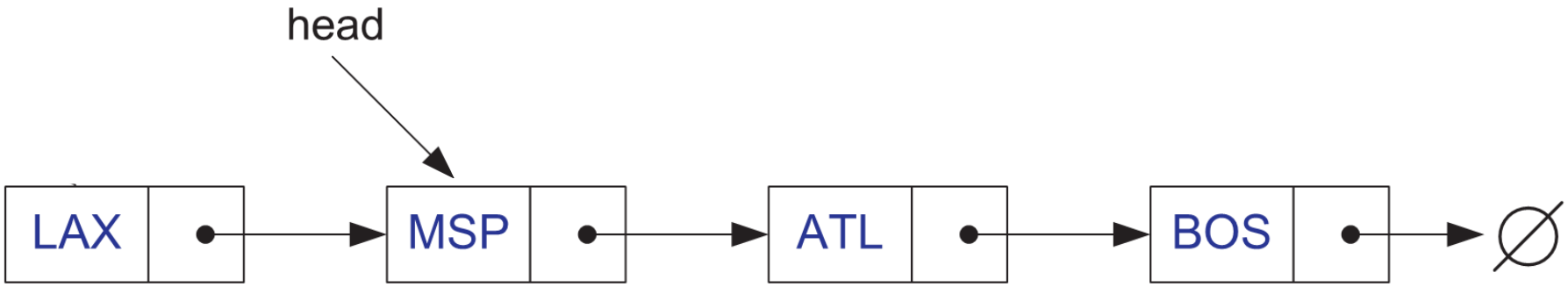


Removal from the Front of a Singly Linked List

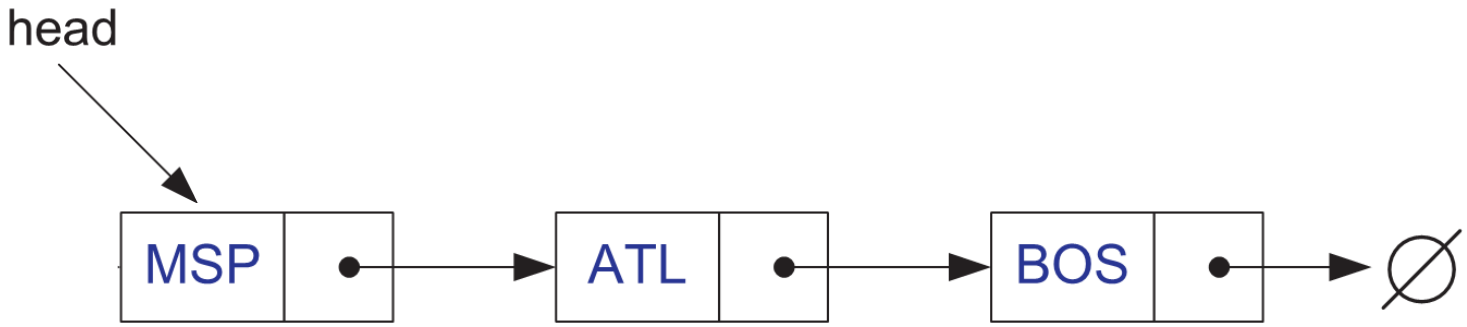
1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

   (typically done by calling “delete” in C++)

Removal from the Front of a Singly Linked List



Removal from the Front of a Singly Linked List



Removal from the Front of a Singly Linked List

IUT-ECE

```
void StringLinkedList::removeFront() {          // remove front item
  StringNode* old = head;                       // save current head
  head = old->next;                             // skip over old head
  delete old;                                   // delete the old head
}
```

دسترسی به گره‌های میانی

- یافتن گره i ام در لیست چقدر زمان میخواهد؟
- بدترین حالت؟ عنصر یکی به آخر. پس O(n)

درج در بین گره‌های میانی

- درج کردن در مکان i ام در لیست چقدر زمان میخواهد؟
- ابتدا باید به آن مکان دسترسی پیدا کنیم.

$$O(1) + O(n) = O(n)$$

حذف از انتهای لیست

- چقدر زمان میخواهد؟

$$O(n)$$

Generic Singly Linked List

- A node in a generic singly linked list:

```
template <typename E>
class SNode {                                   // singly linked list node
private:
  E elem;                                       // linked list element value
  SNode<E>* next;                               // next item in the list
  friend class SLinkedList<E>;                  // provide SLinkedList access
};
```

Generic Singly Linked List

- A class definition for a generic singly linked list:

```
template <typename E>
class SLinkedList {                          // a singly linked list
public:
  SLinkedList();                             // empty list constructor
  ~SLinkedList();                            // destructor
  bool empty() const;                        // is list empty?
  const E& front() const;                    // return front element
  void addFront(const E& e);                 // add to front of list
  void removeFront();                        // remove front item list
private:
  SNode<E>* head;                            // head of the list
};
```

Generic Singly Linked List

- Other member functions for a generic singly linked list:

```
template <typename E>
SLinkedList<E>::SLinkedList()                    // constructor
  : head(NULL) { }

template <typename E>
bool SLinkedList<E>::empty() const               // is list empty?
  { return head == NULL; }

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

template <typename E>
SLinkedList<E>::~SLinkedList()                   // destructor
  { while (!empty()) removeFront(); }
```

Generic Singly Linked List

- Other member functions for a generic singly linked list:

```
template <typename E>
void SLinkedList<E>::addFront(const E& e) {      // add to front of list
  SNode<E>* v = new SNode<E>;                    // create new node
  v->elem = e;                                   // store data
  v->next = head;                                // head now follows v
  head = v;                                      // v is now the head
}

template <typename E>
void SLinkedList<E>::removeFront() {             // remove front item
  SNode<E>* old = head;                          // save current head
  head = old->next;                              // skip over old head
  delete old;                                    // delete the old head
}
```

Generic Singly Linked List

- Examples using the generic singly linked list class:

```
SLinkedList<string> a;                    // list of strings
a.addFront("MSP");
// ...
SLinkedList<int> b;                       // list of integers
b.addFront(13);
```

درباره لیست تک پیوندی

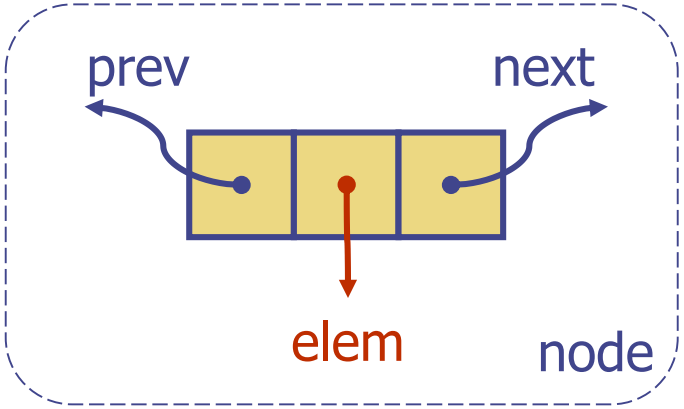
- میدانیم برای موارد زیر خوب است:
  - درج کردن در ابتدا/انتهای لیست
  - حذف کردن از ابتدای لیست
  - درج کردن در میان لیست (به شرط داشتن رفرنس به آن مکان)
- میدانیم برای موارد زیر خوب نیست:
  - دسترسی به میان لیست
  - حذف کردن از انتهای لیست

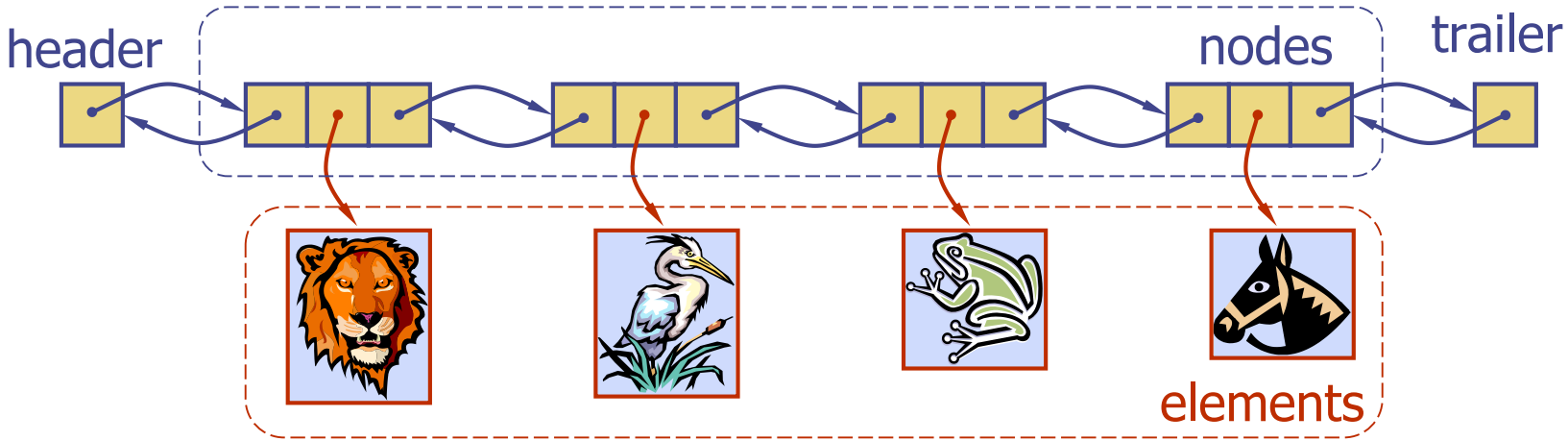
درباره لیست تک پیوندی

- راهکار؟ بده بستان؟

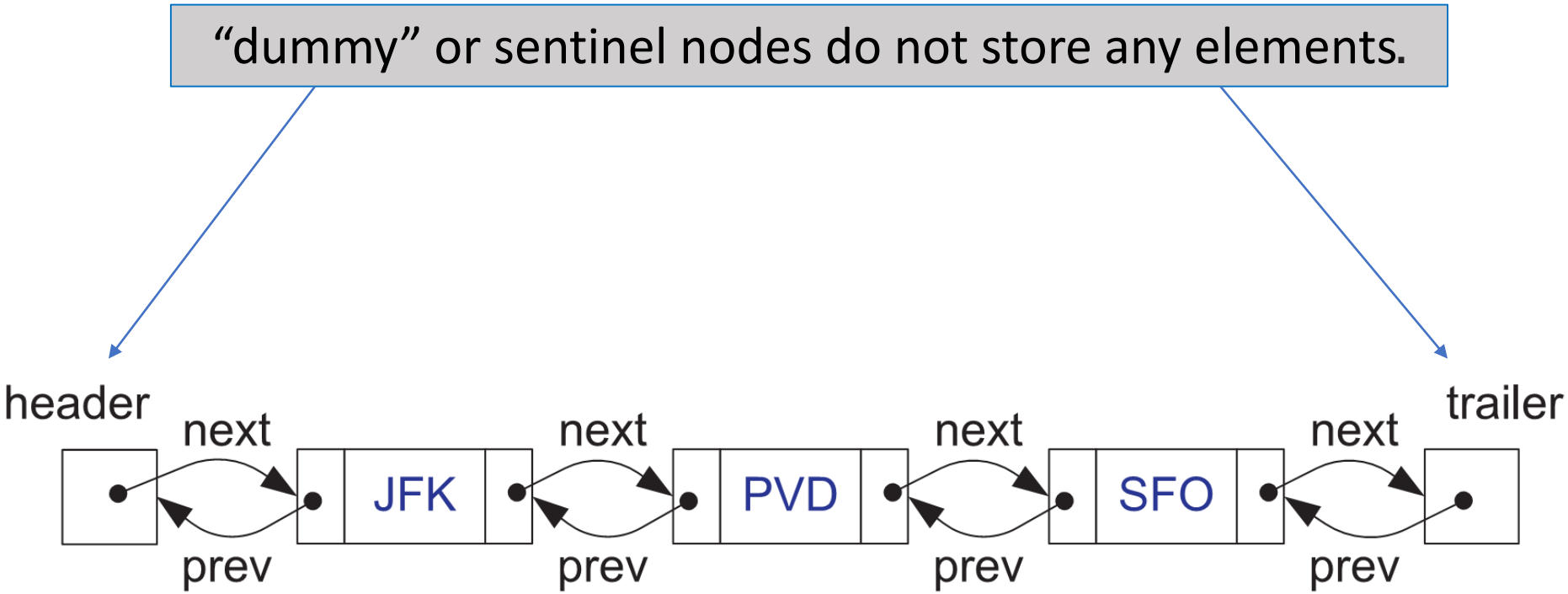
Doubly Linked Lists

- Singly Linked List
  - Not easy to remove an elem. at the tail (or any other node)
- Trailer/header: Dummy sentinel
- Previous link





Doubly Linked Lists



Doubly Linked Lists

- C++ implementation of a doubly linked list node:

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

```
class DLinkedList {                              // doubly linked list
public:
  DLinkedList();                                 // constructor
  ~DLinkedList();                                // destructor
  bool empty() const;                            // is list empty?
  const Elem& front() const;                     // get front element
  const Elem& back() const;                      // get back element
  void addFront(const Elem& e);                  // add to front of list
  void addBack(const Elem& e);                   // add to back of list
  void removeFront();                            // remove from front
  void removeBack();                             // remove from back
private:                                         // local type definitions
  DNode* header;                                 // list sentinels
  DNode* trailer;
protected:                                       // local utilities
  void add(DNode* v, const Elem& e);             // insert new node before v
  void remove(DNode* v);                         // remove node v
};
```

- Implementation of a doubly linked list class.

Doubly Linked Lists

- Class constructor and destructor:

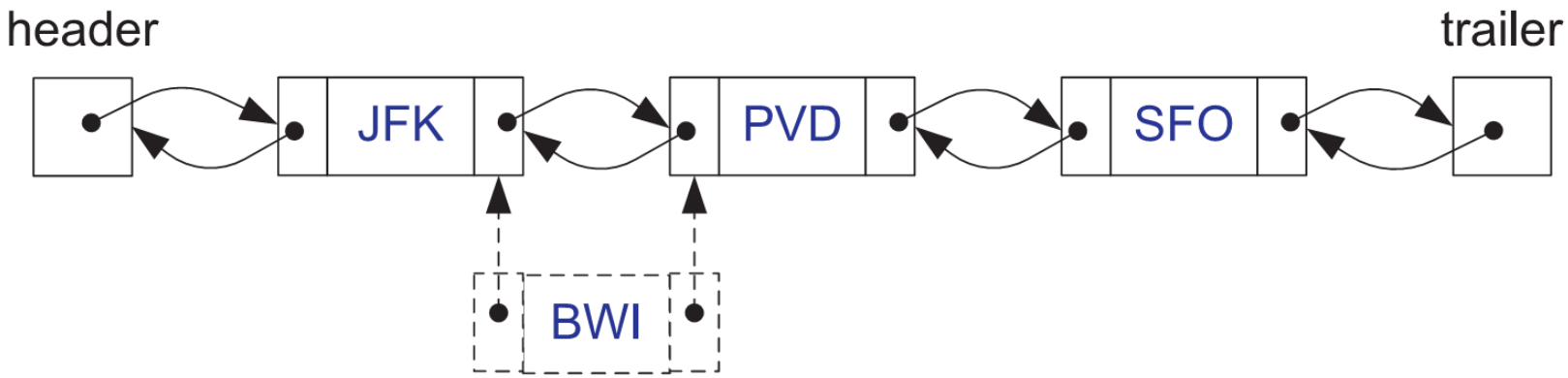
```
DLinkedList::DLinkedList() {                    // constructor
  header = new DNode;                           // create sentinels
  trailer = new DNode;
  header->next = trailer;                       // have them point to each other
  trailer->prev = header;
}

DLinkedList::~DLinkedList() {                   // destructor
  while (!empty()) removeFront();               // remove all but sentinels
  delete header;                                // remove the sentinels
  delete trailer;
}
```

Doubly Linked Lists

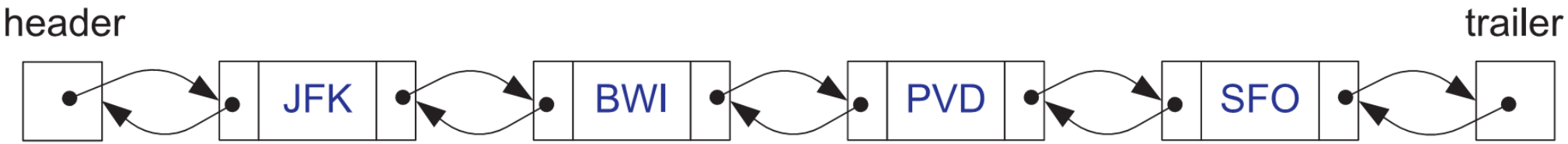
- Accessor functions for the doubly linked list class:

```
bool DLinkedList::empty() const                    // is list empty?
  { return (header->next == trailer); }

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

const Elem& DLinkedList::back() const              // get back element
  { return trailer->prev->elem; }
```

Insertion into a Doubly Linked List



(a)

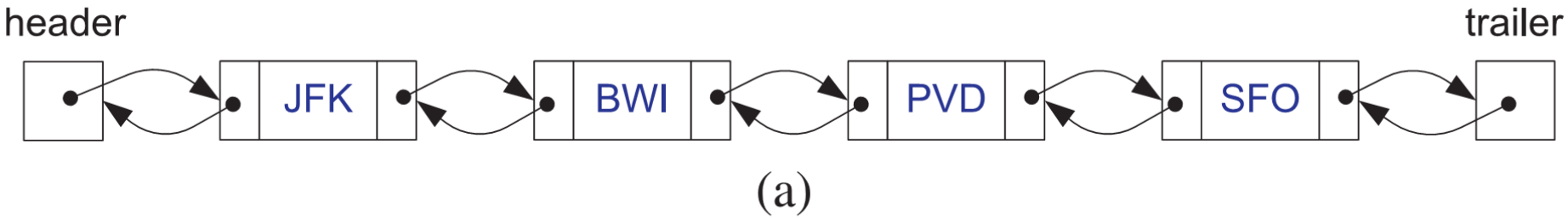


(b)

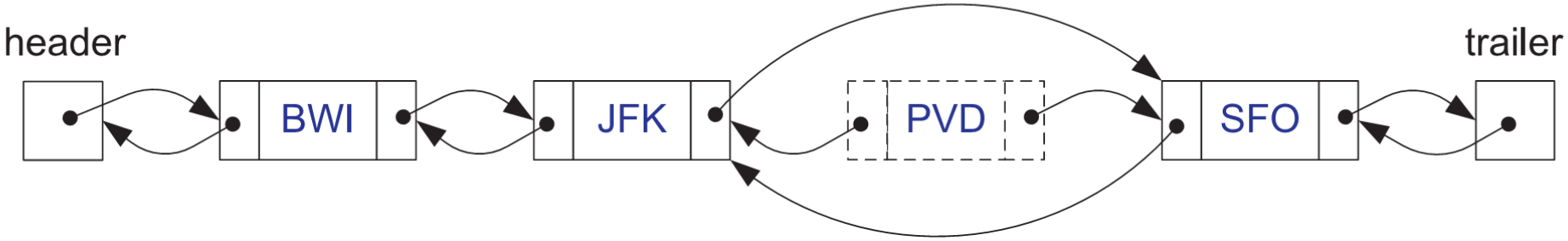
زمان درج

- درج در ابتدای لیست: $O(1)$
- درج در انتهای لیست: $O(1)$
- درج در میان لیست: $O(n)$
- درج در میان لیست (به شرط داشتن رفرنس): $O(1)$

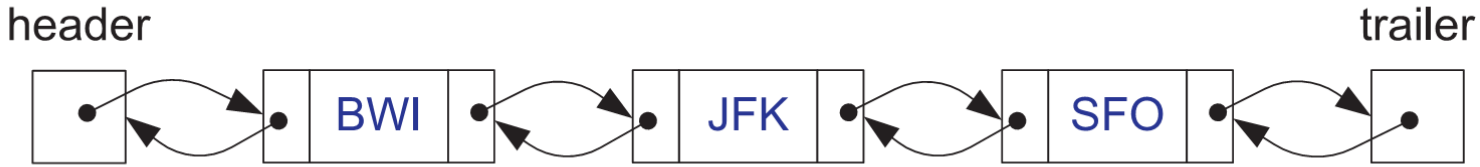
Removal from a Doubly Linked List



(a)



(b)



(c)

زمان حذف

- حذف از ابتدای لیست: $O(1)$
- حذف از انتهای لیست: $O(1)$
- حذف از میان لیست: $O(n)$
- حذف از میان لیست (به شرط داشتن رفرنس): $O(1)$

مقایسه با arraylist

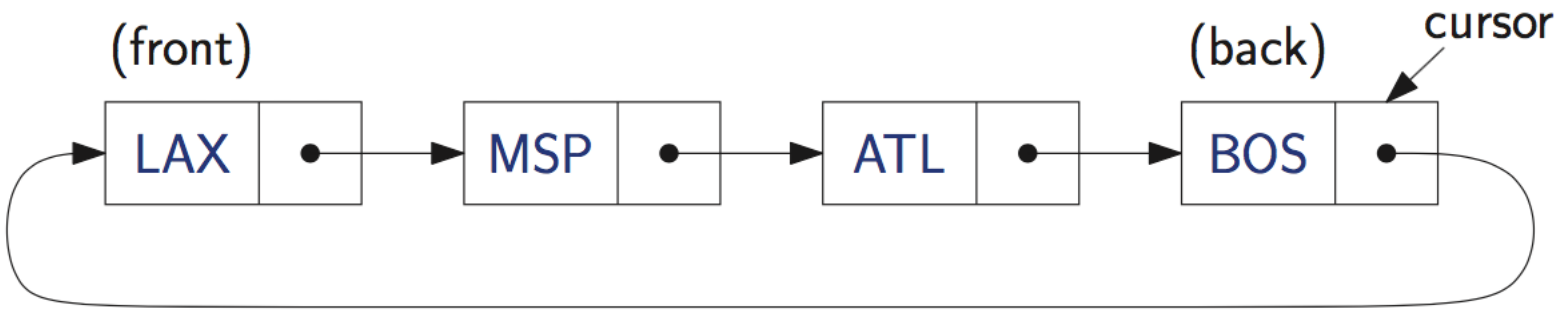
- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time (by condition)

مقایسه با arraylist

- جستجو

Circular Linked List

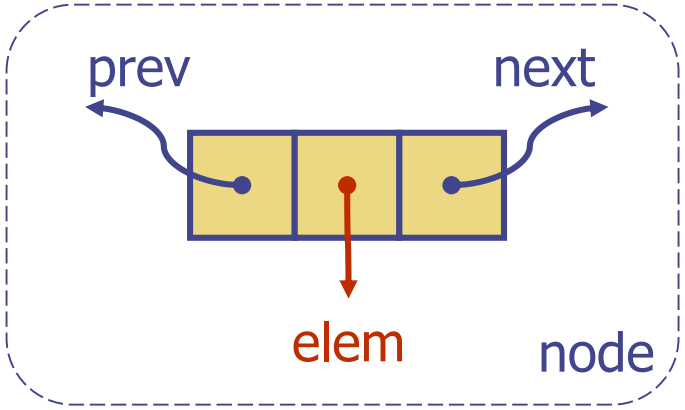
- A kind of Singly Linked List
- Rather than having a head or a tail, it forms a cycle
- Cursor
  - A virtual starting node
  - This can be varying as we perform operations

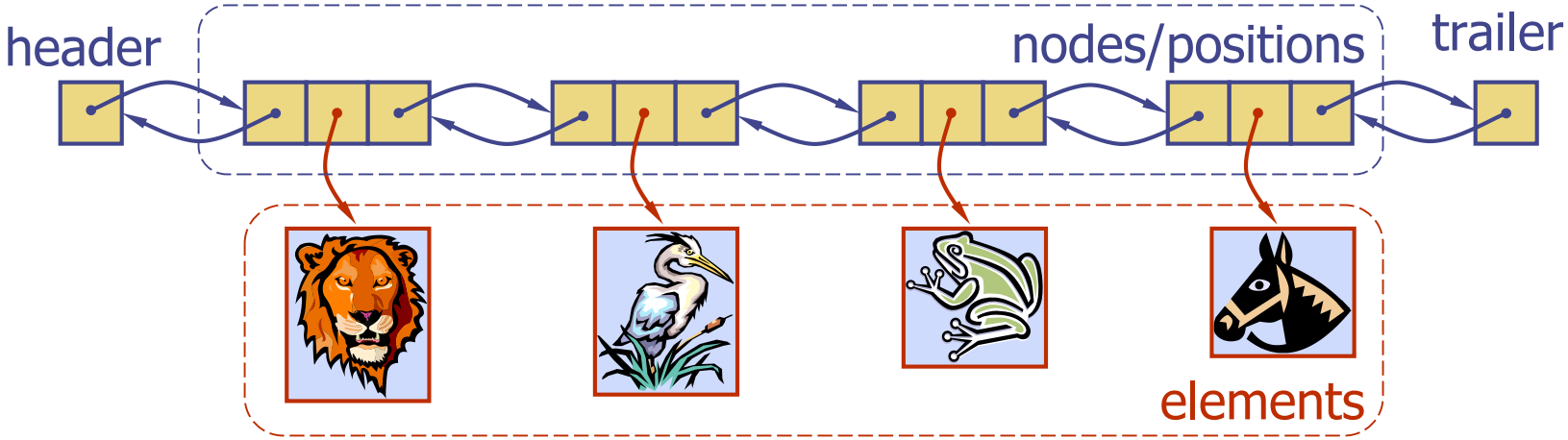


List

IUT-ECE

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





List in STL

IUT-ECE

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

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.

STL

- I want to find “text” in Vector or List objects

```
vector<string>  V(100);
list<string> L(100);
// some data insertion to V and L

//Design 1: different function
find_vector(&V);
find_list(&L);

//Design 2: function overloading
find(&V);
find(&L);
```

STL

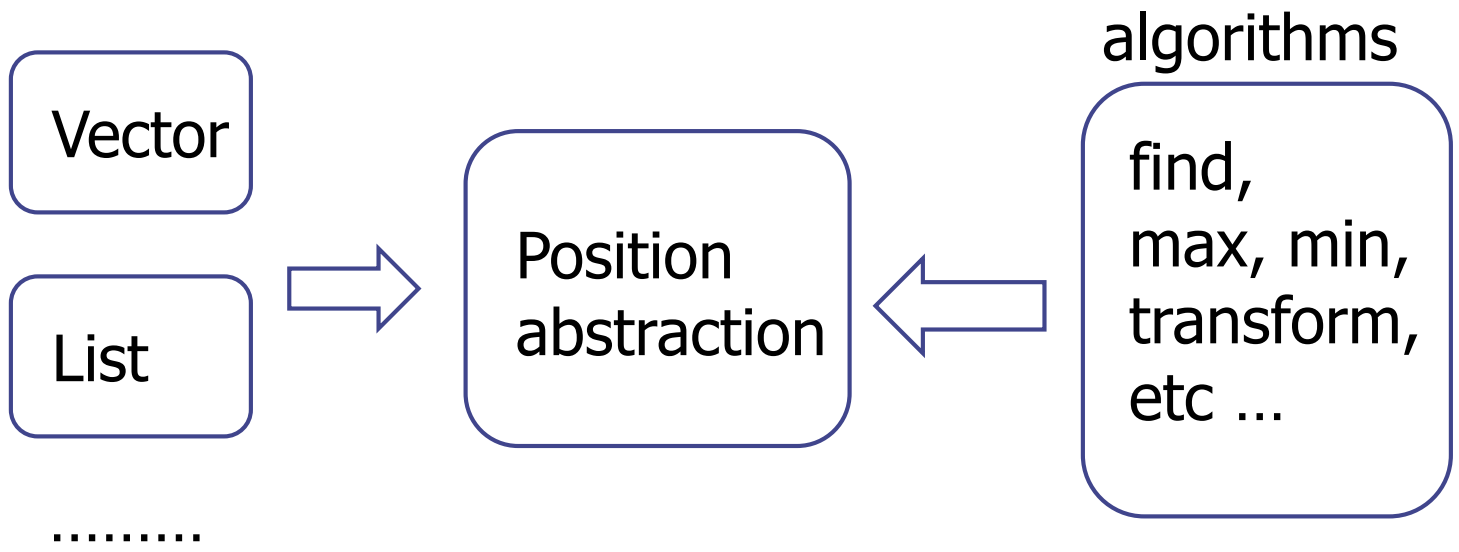

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  vector<string> vec_str;
  vec_str.push_back("is");
  vec_str.push_back("of");
  vec_str.push_back("the");
  vec_str.push_back("hello");

  vector<string>::iterator it;

  it =
    find(vec_str.begin(), vec_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

```cpp
#include <iostream>
#include <list>
#include <string>
#include <algorithm>

using namespace::std;

int main()
{
  list<string> list_str;
  list_str.push_back("is");
  list_str.push_back("of");
  list_str.push_back("the");
  list_str.push_back("hello");

  list<string>::iterator it;

  it =
    find(list_str.begin(), list_str.end(), "the");
  cout << "Print: " << *it << endl;

  it++;
  cout << "Print: " << *it << endl;

  return 0;
}
```

http://www.cplusplus.com/reference/algorithm/

STL

- Lots of data structures (or classes in C++) that can contain various types of elements
  - "Container"
  - Examples: Vector, List, deque, set, map, etc …



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- "A" method of accessing the element at position p:
  - object p.element(): returns the element at position
  - In C++ it is convenient to implement this as *p
    - Operator overloading
- Implemented as "iterator" in C++

Containers and Iterators in C++

- An iterator abstracts the process of scanning through a collection of elements
- A container is an abstract data structure that supports element access through iterators
  - Data structures that support iterators
  - Examples include Vector, List
  - begin(): returns an iterator to the first element
  - end(): return an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - *p: returns the element referenced by this iterator
  - ++p: advances to the next element
- Extends the concept of position by adding a traversal capability

Containers and Iterators in C++


```
for (list<double>::iterator it = list1.begin(); it
!= list1.end(); ++it){
    sum += *it;
}
```