Data Structures (in C++)

- Arrays and Linked Lists -

부산대학교 정보·의생명공학대학
정보컴퓨터공학부

Arrays and Lists

Arrays

❖ A collection of elements of the same type

❖ Each element of the array is referenced by its index

```
double f[5];        // array of 5 doubles: f[0], ..., f[4]
int m[10];          // array of 10 ints: m[0], ..., m[9]
f[4] = 2.5;
m[2] = 4;
cout << f[m[2]];    // outputs f[4], which is 2.5
```

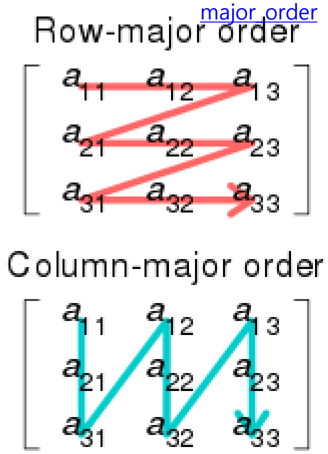
- Not possible to adjust the number of elements in an array once declared
- Common mistake: Indexing an array outside of its boundary

```
double vect[10];              // Possible Index range:
[0, 1, ..., 9]
cout << vect[10] << endl;     // Error
```

Arrays

❖ Multidimensional Arrays

https://en.wikipedia.org/wiki/Row-_and_column-major_order



- Implemented as an array of arrays
- Row-major indexing

double vect[10][20]; // A 10-element array of 20-element arrays

- **Initialization**
  - Arrays can be initialized by using **curly braces**
  - The compiler figures out the size

```
int a[]   = {10, 11, 12, 13};        // declares and initializes a[4]
bool b[]  = {false, true};           // declares and initializes b[2]
char c[]  = {'c', 'a', 't'};         // declares and initializes c[3]
```

Arrays

❖ Initialization of multidimensional arrays

```
int matrix[3][4] = {     // A 3-element array of 4-element arrays
    {1, 2, 3, 4},        // Row 0
    {5, 6, 7, 8},        // Row 1
    {9, 0, 1, 2}         // Row 2
};

int matrix[3][4] = {     // Missing entries are initialized to 0
    {1, 2},              // Row 0: {1, 2, 0, 0}
    {5, 6, 7},           // Row 1: {5, 6, 7, 0}
    {9}                  // Row 2: {9, 0, 0, 0}
};

int matrix[][4] = {      // The size is determined by the compiler
    {1, 2, 3, 4},        // Row 0
    {5, 6, 7, 8},        // Row 1
    {9, 0, 1, 2}         // Row 2
};

int matrix[][] = {       // This is not allowed
    {1, 2, 3, 4},        // Row 0
    {5, 6, 7, 8},        // Row 1
    {9, 0, 1, 2}         // Row 2
};
```

Arrays

❖ Pointers and Arrays

- The name of an array is equivalent to a pointer to the first element of the array

```
char c[] = {'c', 'a', 't'};
char* p = c;                          // p points to c[0]
char* q = &c[0];                      // q also points to c[0]
cout << c[2] << p[2] << q[2];         // outputs "ttt"
```

Caution

This equivalence between array names and pointers can be confusing, but it helps to explain many of C++'s apparent mysteries. For example, given two arrays c and d, the comparison (c == d) does not test whether the contents of the two arrays are equal. Rather it compares the addresses of their initial elements, which is probably not what the programmer had in mind. If there is a need to perform operations on entire arrays (such as copying one array to another) it is a good idea to use the vector class, which is part of C++'s Standard Template Library. We discuss these concepts in Section 1.5.5.

Array Application

❖ Storing Game Entries in an Array

- Store game scores using an array in descending score order
- Define an object to represent a game score entry
  - Name and score of a player

```
class GameEntry {                                  // a game score entry
public:
  GameEntry(const string& n="", int s=0); // constructor
  string getName() const;                          // get player name
  int getScore() const;                            // get score
private:
  string name;                                     // player's name
  int score;                                       // player's score
};

  GameEntry::GameEntry(const string& n, int s) // constructor
    : name(n), score(s) { }
                                                  // accessors
  string GameEntry::getName() const { return name; }
  int GameEntry::getScore() const { return score; }
```

Array Application

❖ A Class for High Scores

- Store the highest scores in an array
- Need to trace the number of current elements

```
class Scores {                                  // stores game high scores
public:
  Scores(int maxEnt = 10);                      // constructor
  ~Scores();                                    // destructor
  void add(const GameEntry& e);                 // add a game entry
  GameEntry remove(int i)                       // remove the ith entry
      throw(IndexOutOfBounds);
private:
  int maxEntries;                               // maximum number of entries
  int numEntries;                               // actual number of entries
  GameEntry* entries;                           // array of game entries
};
```

Array Application

❖ A Class for High Scores

- Store the highest scores in an array
- Need to trace the number of current elements

```
Scores::Scores(int maxEnt) {                      // constructor
  maxEntries = maxEnt;                            // save the max size
  entries = new GameEntry[maxEntries];            // allocate array storage
  numEntries = 0;                                 // initially no elements
}

Scores::~Scores() {                               // destructor
  delete[] entries;
}
```

Array Application

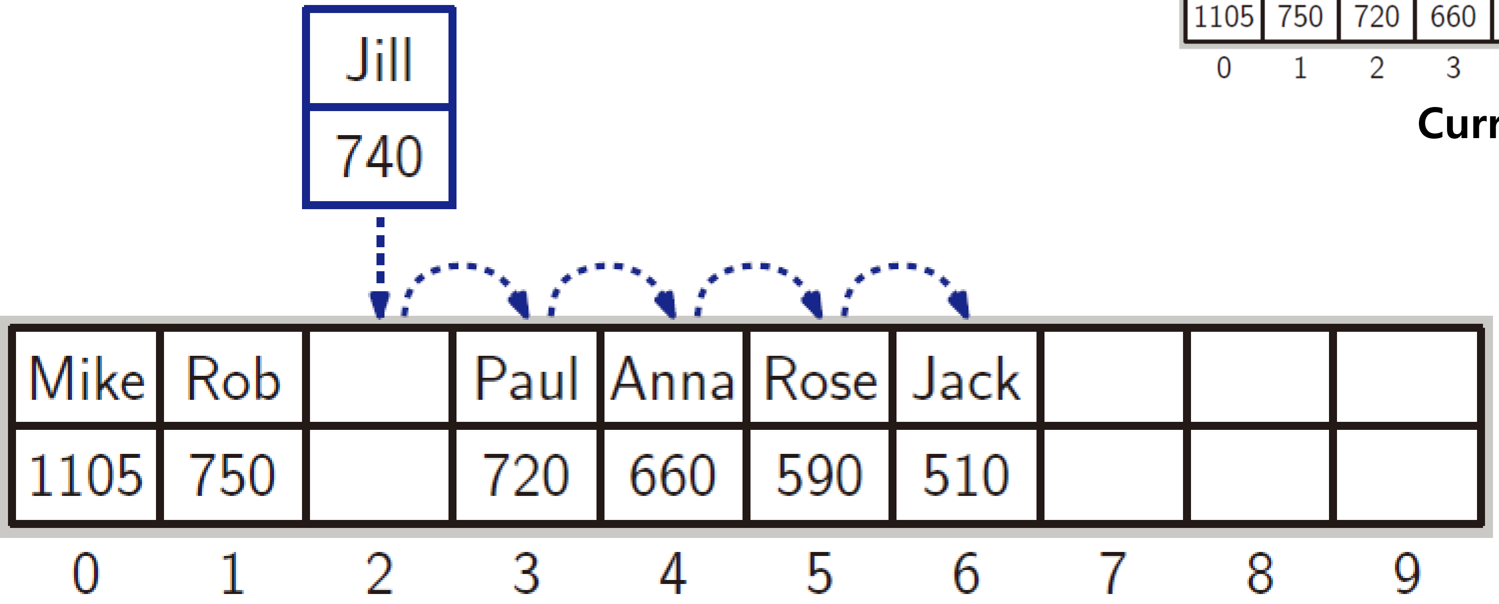
❖ Insertion

- *GameEntry* objects are ordered by their score values from highest to lowest

add(e): Insert game entry e into the collection of high scores. If this causes the number of entries to exceed *maxEntries*, the smallest is removed.

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

Current entries



Array Application

❖ Insertion

```
void Scores::add(const GameEntry& e) {     // add a game entry
  int newScore = e.getScore();             // score to add
  if (numEntries == maxEntries) {          // the array is full
    if (newScore <= entries[maxEntries-1].getScore())
      return;                              // not high enough - ignore
  }
  else numEntries++;                       // if not full, one more entry

  int i = numEntries-2;                    // start with the next to last
  while ( i >= 0 && newScore > entries[i].getScore() ) {
    entries[i+1] = entries[i];             // shift right if smaller
    i--;
  }
  entries[i+1] = e;  // put e in the empty spot
}
```

Mike	Rob	Paul	Anna	Rose	Jack				
1105	750	720	660	590	510				
0	1	2	3	4	5	6	7	8	9

Array Application

❖ **Removal**

remove(i): Remove and return the game entry e at index i in the *entries* array. If index i is outside the bounds of the *entries* array, then this function throws an exception; otherwise, the *entries* array is updated to remove the object at index i and all objects previously stored at indices higher than i are "shifted left" to fill in for the removed object.

Return:

Paul
720

Mike	Rob	Jill	Anna	Rose	Jack				
1105	750	740	660	590	510				
0	1	2	3	4	5	6	7	8	9

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			
0	1	2	3	4	5	6	7	8	9

Current entries

Array Application

❖ **Removal**

```
GameEntry Scores::remove(int i) throw(IndexOutOfBounds) {
  if ((i < 0) || (i >= numEntries))             // invalid index
    throw IndexOutOfBounds("Invalid index");
  GameEntry e = entries[i];                     // save the removed object
  for (int j = i+1; j < numEntries; j++)
    entries[j−1] = entries[j];                  // shift entries left
  numEntries−−;                                 // one fewer entry
  return e;                                     // return the removed object
}
```

Mike	Rob	Jill	Paul	Anna	Rose	Jack			
1105	750	740	720	660	590	510			
0	1	2	3	4	5	6	7	8	9

Sorting an Array

❖ Sorting

- Rearrange objects of an array to be ordered by some criterion (*e.g.*, ascending order)

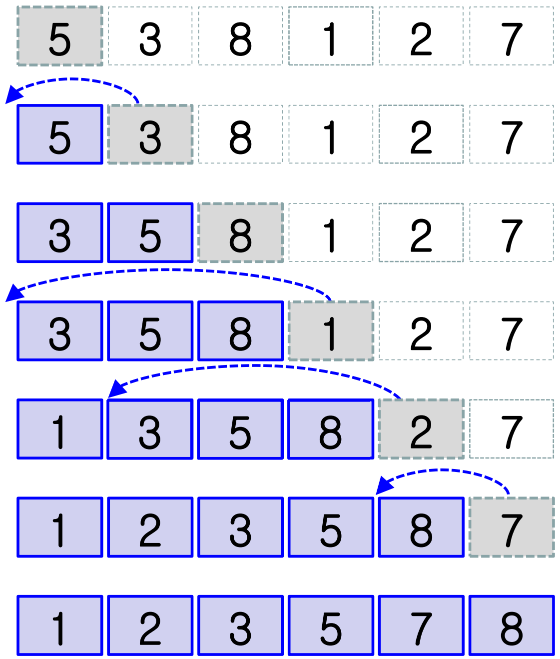
what we already have done for the insertion

❖ Insertion Sort

- Each iteration of the algorithm inserts the next element into the current sorted part of the array

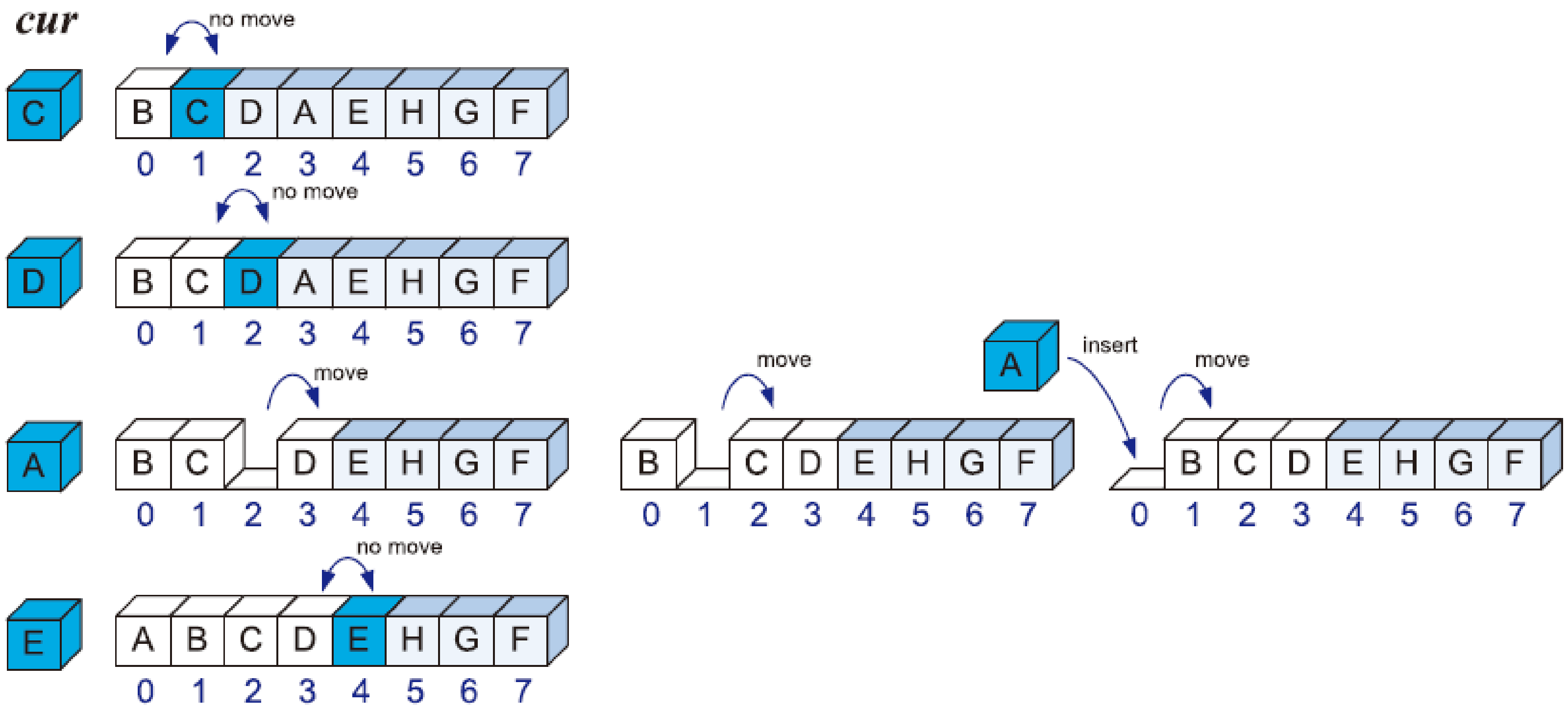
Algorithm InsertionSort(A):
    Input: An array A of n comparable elements
    Output: The array A with elements rearranged in nondecreasing order
    for $i \leftarrow 1$ to $n-1$ **do**
        {Insert $A[i]$ at its proper location in $A[0], A[1], \ldots, A[i-1]$}
        $cur \leftarrow A[i]$
        $j \leftarrow i-1$
        while $j \geq 0$ and $A[j] > cur$ **do**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow cur$ {cur is now in the right place}

Pseudocode



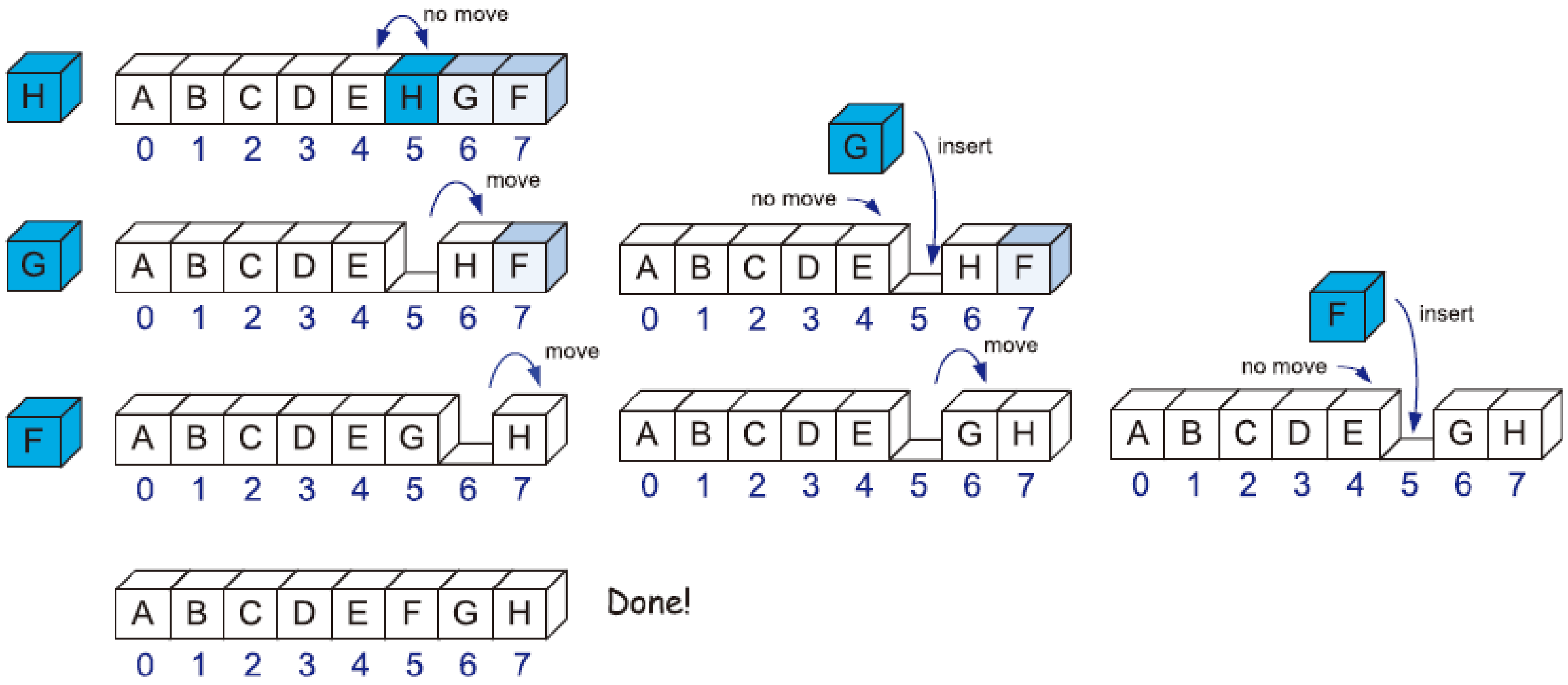
Sorting an Array

- ❖ **Insertion Sort**



Sorting an Array

❖ Insertion Sort



Two-Dimensional Arrays (Matrix)

❖ we can create a two-dimensional array as an array of arrays

```
int M[8][10];          // matrix with 8 rows and 10 columns
```

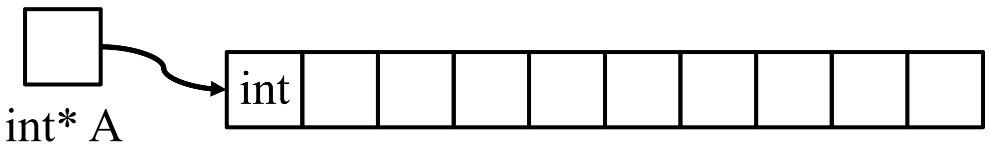
M **is an array of length 8**

Each element of ***M*** **is an array of length 10 of integers**

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

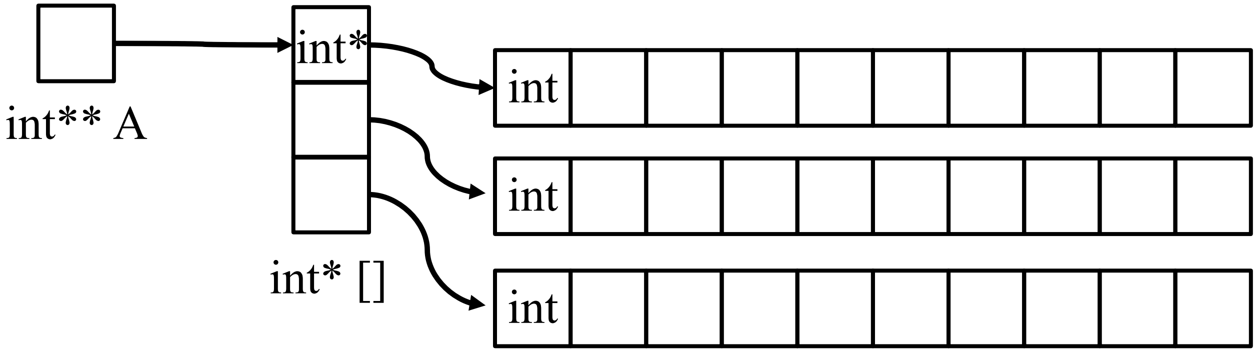
Dynamic Allocation of Matrices

❖ Recall that a dynamic array is represented as a <u>pointer to its first element.</u>



int* A = **new int**[m];

❖ Since each row pointer is of type int*, the matrix is of type int**




```
int** M = new int*[n];            // allocate an array of row pointers
for (int i = 0; i < n; i++)
  M[i] = new int[m];              // allocate the i-th row
```

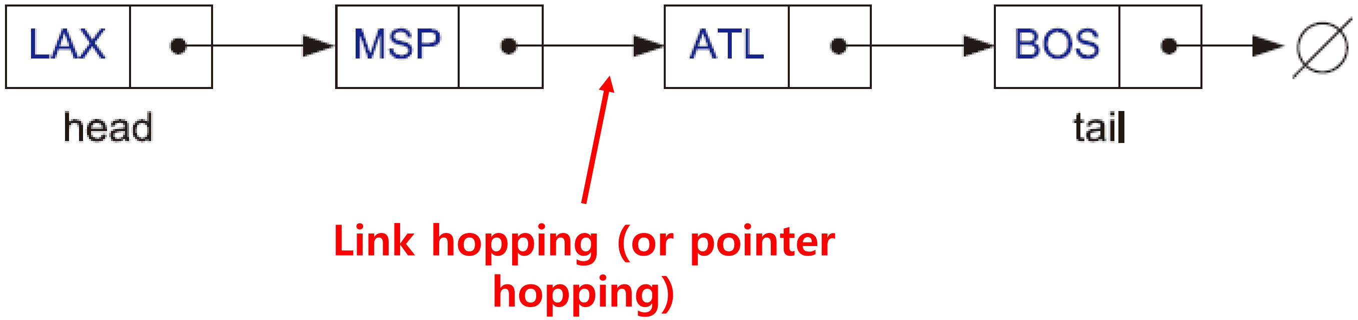
Linked Lists

❖ **Arrays are not very adaptable**

- difficult to resize
- Insertions and deletions are difficult

❖ **Linked List**

- A collection of ***nodes*** that together form a linear ordering
- Each node contains ***links*** to other nodes



❖ **Singly Linked List**

- Each node stores a single link to its *successor*

Singly Linked Lists

❖ Singly Linked List Implementation

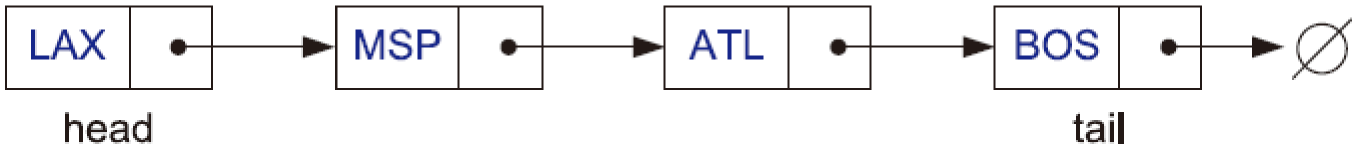
```
class StringNode {                    // a node in a list of strings
private:
  string elem;                        // element value
  StringNode* next;                   // next item in the list

  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
};
```

Singly Linked Lists

❖ **Simple Member Functions**

```
StringLinkedList::StringLinkedList()                    // constructor
  : head(NULL) { }

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



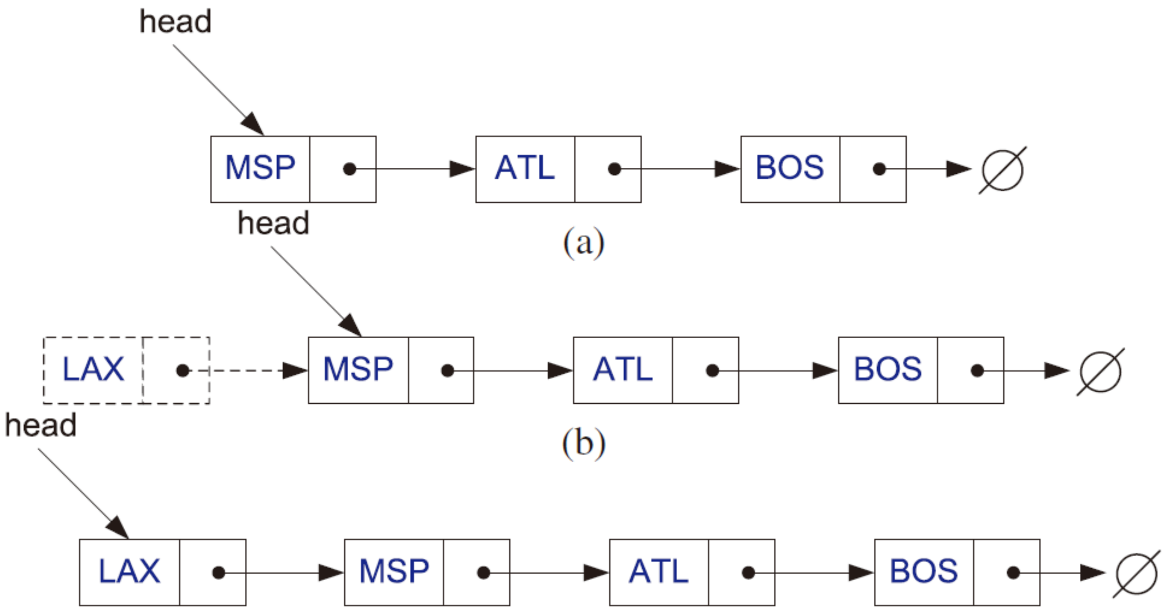

```
bool StringLinkedList::empty() const                    // is list empty?
  { return head == NULL; }

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

Singly Linked Lists

❖ Insertion to the Front

- The easiest way to insert an element

```
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
}
```



(a)

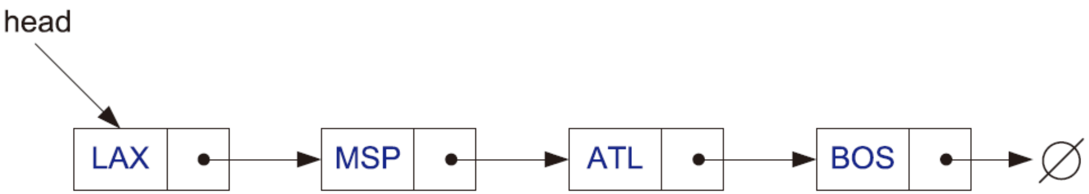
(b)

(c)

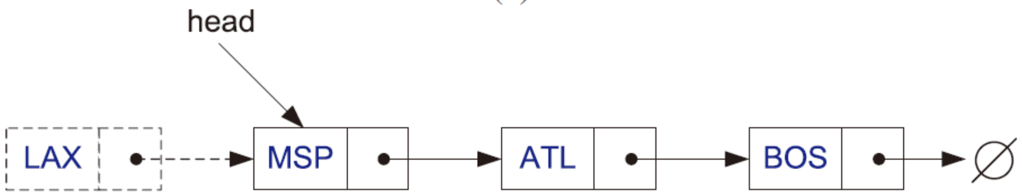
Singly Linked Lists

❖ Removal from the Front

```
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
}
```



(a)



(b)

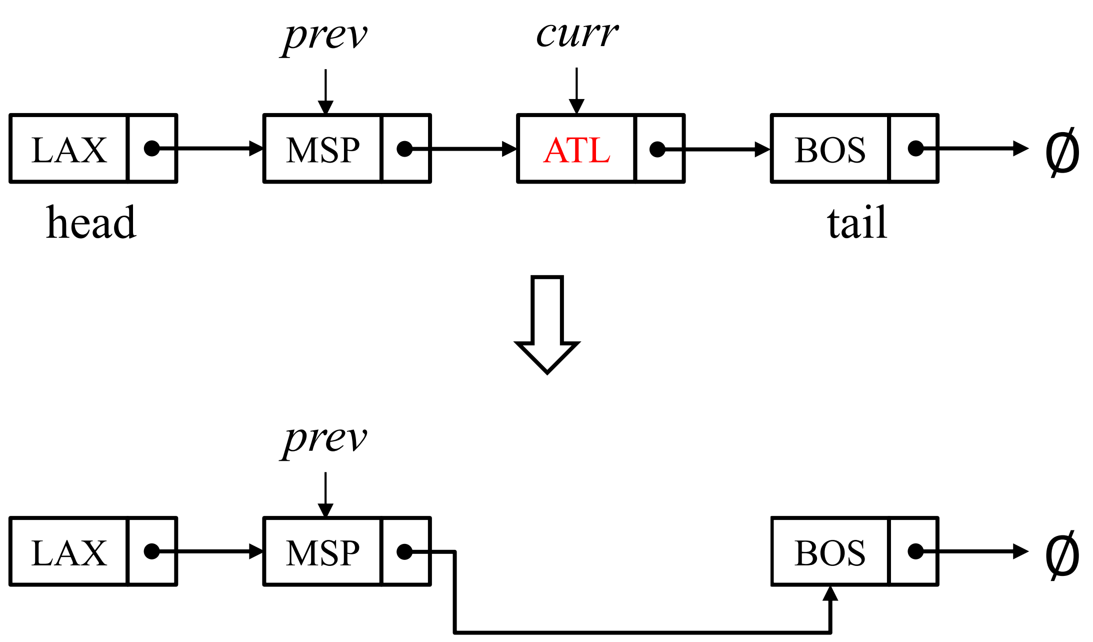
(c)

Singly Linked Lists

❖ Removal of an Intermediate Node

- Must connect the previous and next nodes correctly
- What happens if we want to *remove the last node frequently*?

```
/*
Pseudocode for the removal of an intermediate node

Input:
- name: element value of a node to be removed

curr <- head    // current node
prev <- NULL    // previous node

while curr->next != NULL
    if curr->elem == name
        prev->next <- curr->next    // connect the prev and next
        delete curr                 // delete the curr node
    else
        prev <- curr                // curr becomes prev
        curr <- curr->next          // next becomes curr for the next iteration
*/
```



Generic Singly Linked List

```
class StringNode {
private:
  string elem;
  StringNode* next;

  friend class StringLinkedList;
};


class StringLinkedList {
public:
  StringLinkedList();
  ~StringLinkedList();
  bool empty() const;
  const string& front() const;
  void addFront(const string& e);
  void removeFront();
private:
  StringNode* head;
};
```

```
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
};

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

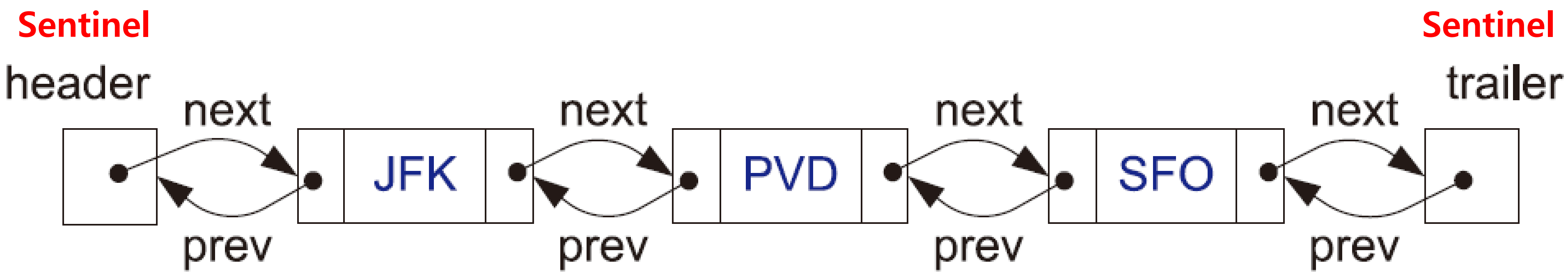
- We can generate singly linked lists of various types by simply setting the template parameter as desired

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

Doubly Linked Lists

❖ 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(Dummy) node** (*i.e.*, header or trailer)
  - A specifically designated node as a traversal path terminator for convenience
  - Does not hold any data

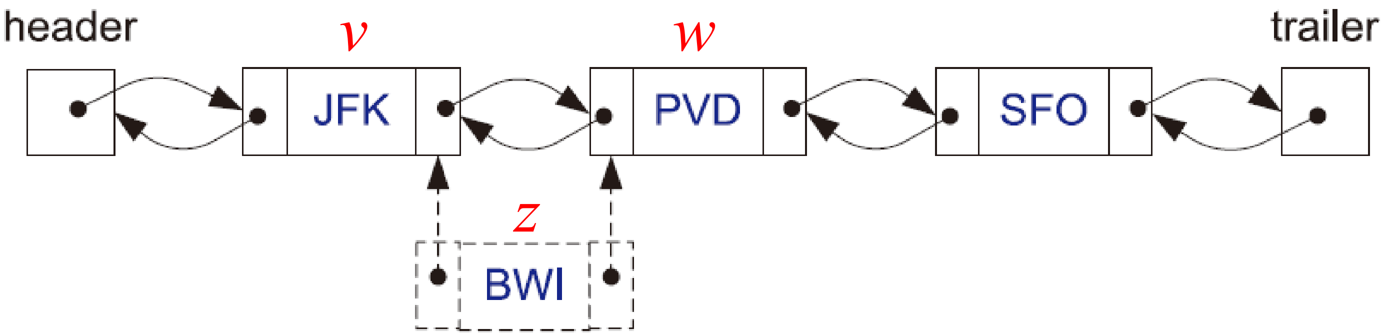


Doubly Linked Lists

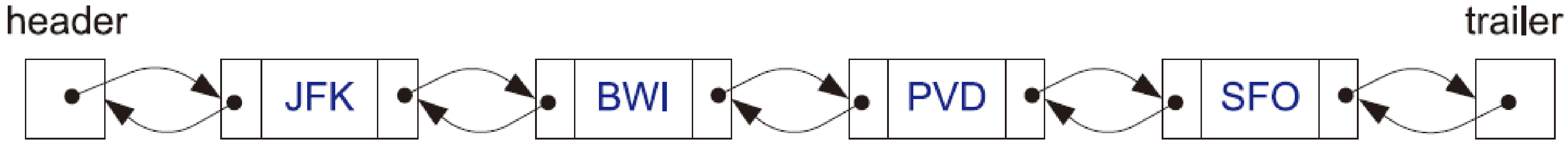
❖ **Insertion at Any Position**

v: a node in a doubly linked list
z: a new node to be inserted after v
w: the next node of v

- Make z's *prev* link point to v
- Make z's *next* link point to w
- Make w's *prev* link point to z
- Make v's *next* link point to z



(a)

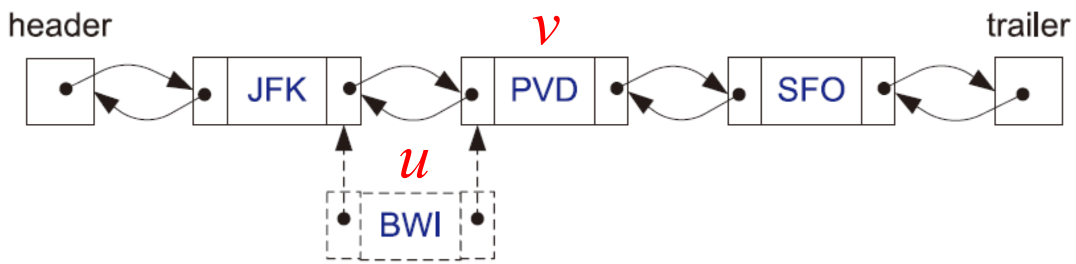


(b)

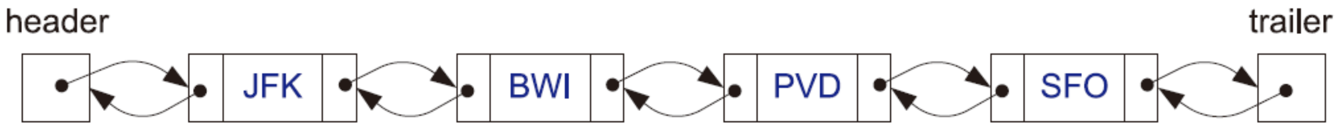
Doubly Linked Lists

❖ Insertion at Any Position

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

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

void DLinkedList::addBack(const Elem& e)  // add to back of list
  { add(trailer, e); }
```



(a)



(b)

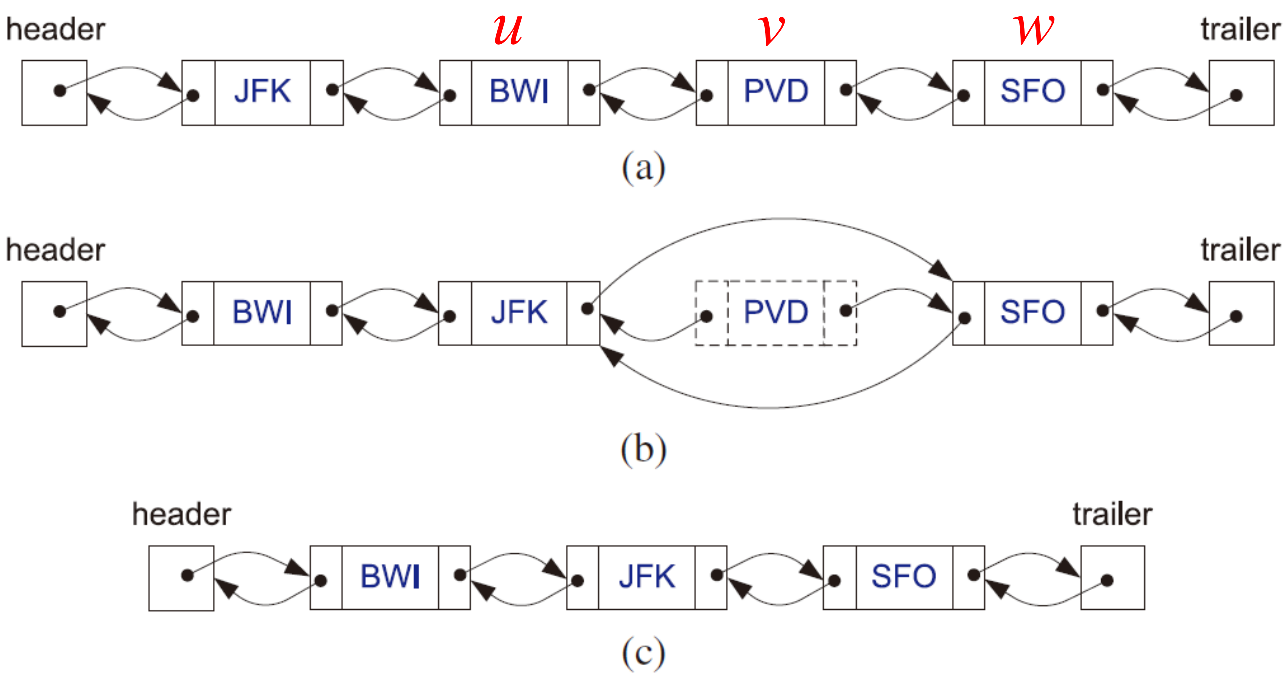
Doubly Linked Lists

❖ Removal of an Intermediate Node

- Refer to this operation as the **linking out** of v

v: a node in a doubly linked list to be removed
w: the next node of v
u: the previous node of v

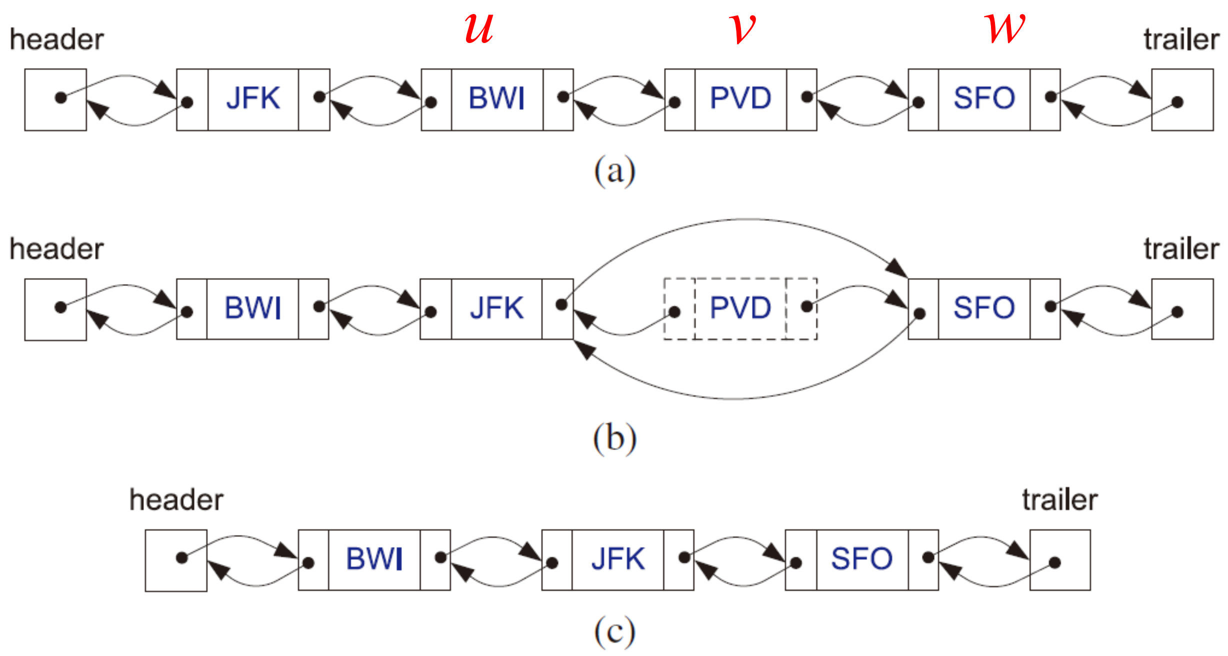
- Make w's *prev* link point to u
- Make u's *next* link point to w
- Delete node v



Doubly Linked Lists

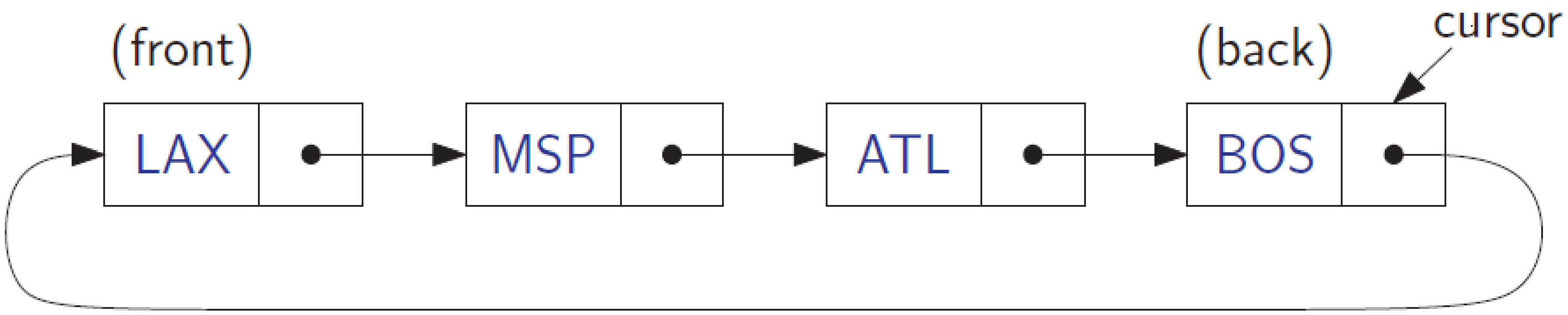
❖ Removal of an Intermediate Node

```
void DLinkedList::remove(DNode* v) {      // remove node v
  DNode* u = v->prev;                     // predecessor
  DNode* w = v->next;                     // successor
  u->next = w;                            // unlink v from list
  w->prev = u;
  delete v;
}

void DLinkedList::removeFront()           // remove from font
  { remove(header->next); }

void DLinkedList::removeBack()            // remove from back
  { remove(trailer->prev); }
```



Circularly Linked Lists

- Same kind of nodes as a singly linked list
  - The node structure is essentially identical to that of a singly linked list
- But, rather than having a head or tail, the nodes of a circularly linked list are linked into a cycle.



Circularly Linked Lists

❖ Circularly Linked Lists Implementation

```
typedef string Elem;                    // element type
class CNode {                           // circularly linked list node
private:
  Elem elem;                            // linked list element value
  CNode* next;                          // next item in the list

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

The node structure is essentially identical to that of a singly linked list.

```
class CircleList {                      // a circularly linked list
public:
  CircleList();                         // constructor
  ~CircleList();                        // destructor
  bool empty() const;                   // is list empty?
  const Elem& back() const;             // element at cursor
  const Elem& front() const;            // element following cursor
  void advance();                       // advance cursor
  void add(const Elem& e);              // add after cursor
  void remove();                        // remove node after cursor
private:
  CNode* cursor;                        // the cursor
};
```

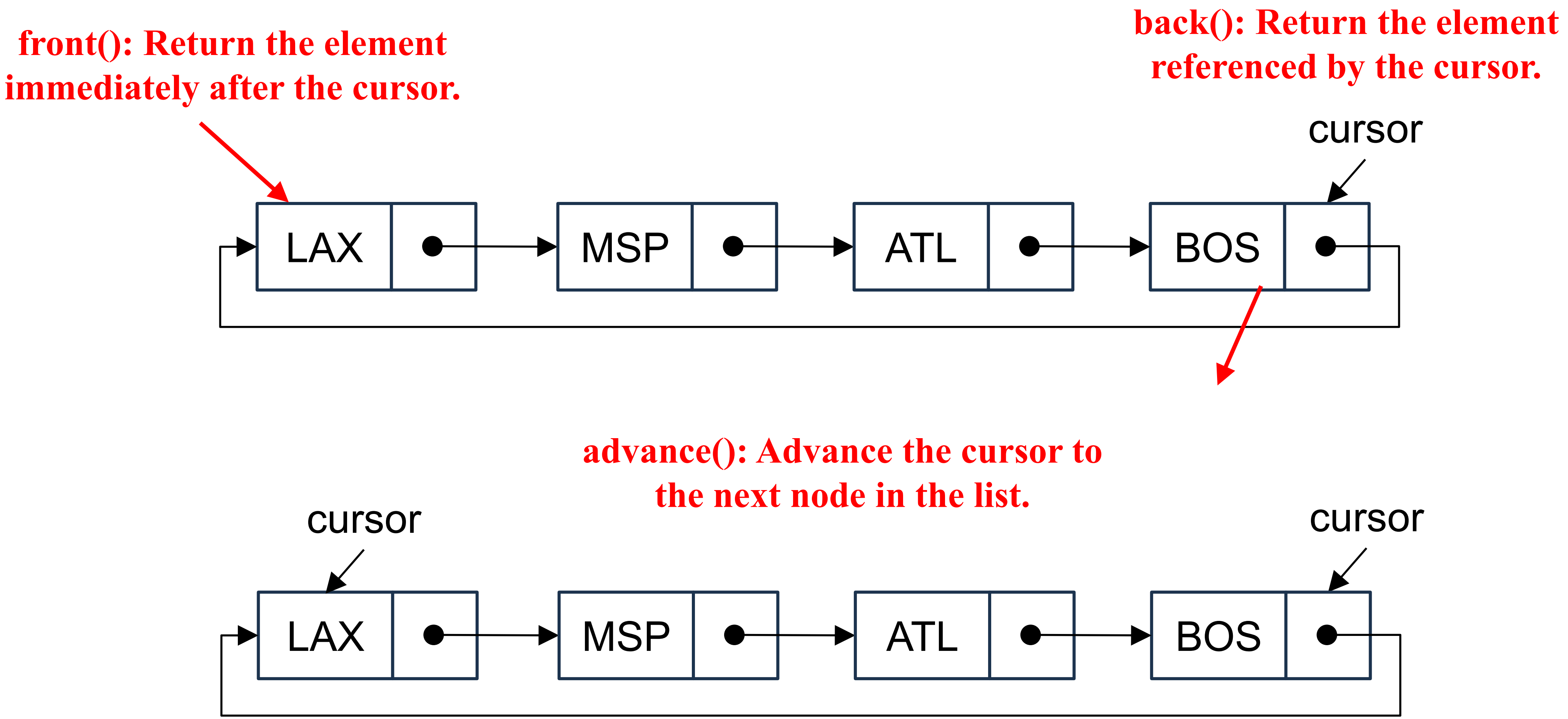
back(): Return the element referenced by the cursor

front(): Return the element immediately after the cursor

advance(): Advance the cursor to the next node in the list

Circularly Linked Lists

❖ **Member Functions of Circularly Linked Lists**



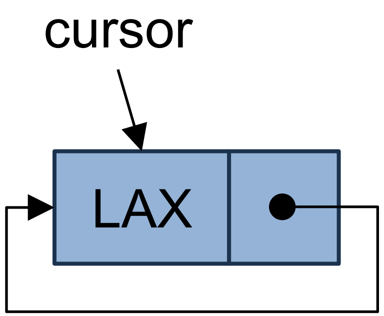
Circularly Linked Lists

❖ void CircleList::add(const Elem& e)

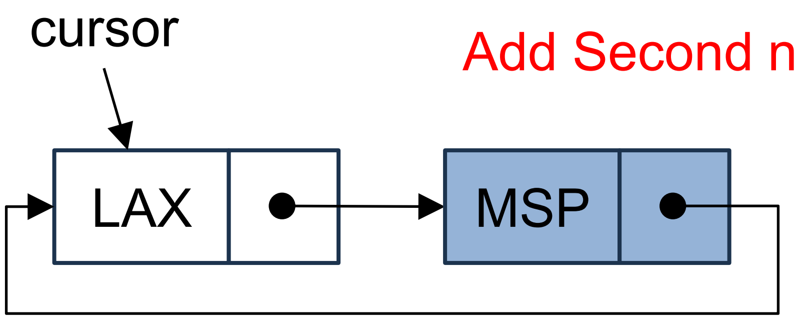
```
void CircleList::add(const Elem& e) {          // add after cursor
  CNode* v = new CNode;                        // create a new node
  v->elem = e;
  if (cursor == NULL) {                        // list is empty?
    v->next = v;                               // v points to itself
    cursor = v;                                // cursor points to v
  }
  else {                                       // list is nonempty?
    v->next = cursor->next;                    // link in v after cursor
    cursor->next = v;
  }
}
```

cursor → NULL

Empty List



Add First node



Add Second node

Circularly Linked Lists

❖ void CircleList::remove()

```
CircleList::CircleList()                    // constructor
  : cursor(NULL) { }
CircleList::~CircleList()                   // destructor
  { while (!empty()) remove(); }


void CircleList::remove() {                  // remove node after cursor
  CNode* old = cursor->next;                 // the node being removed
  if (old == cursor)                         // removing the only node?
    cursor = NULL;                           // list is now empty
  else
    cursor->next = old->next;                // link out the old node
  delete old;                                // delete the old node
}
```