# ECE 2400 Computer Systems Programming
## Fall 2017
### Topic 7: C Lists and Vectors

School of Electrical and Computer Engineering  
Cornell University

revision: 2017-09-29-08-04

<table>
<thead>
<tr>
<th></th>
<th>Lists</th>
<th></th>
<th>Vectors</th>
<th></th>
<th>Interaction Between Data Structures and Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lists</td>
<td>2</td>
<td>Vectors</td>
<td>13</td>
<td>Interaction Between Data Structures and Algorithms</td>
</tr>
<tr>
<td></td>
<td>1.1. List Interface</td>
<td>2</td>
<td>2.1. Vector Interface</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2. List Implementation</td>
<td>3</td>
<td>2.2. Vector Implementation</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

1  Lists  
   1.1. List Interface  
   1.2. List Implementation  

2  Vectors  
   2.1. Vector Interface  
   2.2. Vector Implementation  

3 Interaction Between Data Structures and Algorithms  

1
• **Data structures** are: (1) a way of organizing data; and (2) a collection of operations for accessing and manipulating this data

• **Algorithms** are used to implement the operations for accessing and manipulating the data structure

• Algorithms and data structures are tightly connected

• Data structures have an **interface** and an **implementation**

• Use abstraction (i.e., **data encapsulation, information hiding**) to hide the implementation details (i.e., the algorithm details) from the user of the interface

1. **Lists**

• Recall our example of a chain of dynamically allocated nodes

• Let’s transform this idea into a **list** data structure

• Also called a **linked list**, or more specifically a **singly linked list**

1.1. **List Interface**

```c
typedef struct _node_t {
    int value;
    struct _node_t* next_p;
} node_t;

typedef struct {
    node_t* head_p;
} list_t;

void list_construct ( list_t* list_p );
void list_destruct ( list_t* list_p );
void list_push_front ( list_t* list_p, int v );
void list_insert ( list_t* list_p, node_t* node_p, int v );
void list_sorted_insert ( list_t* list_p, int v );
void list_sort ( list_t* list_p );
```
- **`void list_construct( list_t* list );`**
  Construct the list initializing all fields in the given `list_t`.

- **`void list_destruct( list_t* list );`**
  Destruct the list by freeing any dynamically allocated memory used by `list_t`.

- **`void list_push_front( list_t* list, int v );`**
  Push a new node with the given value (`v`) at the front of the list (the head end).

- **`void list_insert( list_t* list, node_t* node_p, int v );`**
  Insert a new node with the given value (`v`) after the node pointed to by `node_p`.

- **`void list_sorted_insert( list_t* list, int v );`**
  Assume the list is already sorted in increasing order. Search the list to find the proper place to insert a new node with the given value (`v`) such that the list remains in sorted in increasing order.

- **`void list_sort( list_t* list );`**
  Sort the given list in increasing order. Any pointers to nodes within the list will be invalidated.

**Abstraction**

- Implementation details about `node_t` “leak” into the interface
- User will likely need to directly manipulate nodes

**1.2. List Implementation**

- For each function we will use a combination of pseudo-code and “visual” pseudo-code to explain high-level approach

- Then we will translate the pseudo-code to actual C code
**Pseudo-code for** list_construct

1. void list_construct( list_t* list_p )
2. set head ptr to NULL

**Pseudo-code for** list_push_front

1. void list_push_front( list_t* list_p, int v )
2. allocate new node
3. set new node’s value to v
4. set new node’s next ptr to head ptr
5. set head ptr to point to new node

After push front of value 12

After push front of value 11
After push front of value 10

Pseudo-code for `list_destruct`

1. `void list_destruct( list_t* list_p )`
2. set curr node ptr to head ptr
3. while curr node ptr is not NULL
   4. set temp node ptr to curr node’s next ptr
   5. free curr node
   6. set curr node ptr to temp node ptr

After first iteration of while loop

After second iteration of while loop

And so on ...
1. Lists

1.2. List Implementation

// Construct list data structure
void list_construct( list_t* list_p )
{
    list_p->head_p = NULL;
}

// Push value on front of list
void list_push_front( list_t* list_p,
                      int v )
{
    node_t* new_node_p
        = malloc( sizeof(node_t) );
    new_node_p->value = v;
    new_node_p->next_p = list_p->head_p;
    list_p->head_p = new_node_p;
}

// Destruct list data structure
void list_destruct( list_t* list_p )
{
    node_t* curr_node_p = list_p->head_p;
    while ( curr_node_p != NULL ) {
        node_t* next_node_p
            = curr_node_p->next_p;
        free( curr_node_p );
        curr_node_p = next_node_p;
    }
    list_p->head_p = NULL;
}

// Main function
int main( void )
{
    list_t list;
    list_construct( &list );
    list_push_front( &list, 12 );
    list_push_front( &list, 11 );
    list_push_front( &list, 10 );
    list_destruct( &list );
    return 0;
}
**Pseudo-code for** list_insert

```c
void list_insert( list_t* list_p, node_t* node_p, int v )
if list is empty
    list_push_front( list_p, v )
else
    allocate new node
    set new node’s value to v
    set new node’s next ptr to node’s next ptr
    set node’s next ptr to point to new node
```

Initial state of the list

After insert of value of 4 after first node

After insert of value 8 after last node
void list_insert( list_t* list_p, node_t* node_p, int v )
{
    if ( node_p == NULL ) {
        list_push_front( list_p, v );
    } else {
        node_t* new_node_p = malloc( sizeof(node_t) );
        new_node_p->value = v;
        new_node_p->next_p = node_p->next_p;
        node_p->next_p = new_node_p;
    }
}

int main( void )
{
    list_t list;
    list_construct( &list );
    list_push_front( &list, 6 );
    list_push_front( &list, 2 );
    list_insert( &list, list.head_p, 4 );

    node_t* tail_p = list.head_p->next_p->next_p;
    list_insert( &list, tail_p, 8 );

    list_destruct( &list );
    return 0;
}
Pseudo-code for list_sorted_insert

This pseudo-code ignores the corner cases when the list is empty and when the value needs to be inserted into the beginning or end of the list.

```c
void list_sorted_insert( list_t* list_p, int v )
{
    set prev node ptr to head ptr
    set curr node ptr to head node’s next ptr

    while curr node ptr is not NULL
        if v is less than curr node’s value
            list_insert( list_p, prev node ptr, v )
            return

    set prev node ptr to curr node ptr
    set curr node ptr to curr node’s next ptr
}
```

Moving the node pointers for sorted insert of value 5

After sorted insert of value 5
void list_sorted_insert( list_t* list_p, int v )
{
    // Insert into empty list
    if ( list_p->head_p == NULL ) {
        list_push_front( list_p, v );
        return;
    }

    // Insert at beginning of list
    if ( v < list_p->head_p->value ) {
        list_push_front( list_p, v );
        return;
    }

    // Insert in middle of list
    node_t* prev_node_p = list_p->head_p;
    node_t* curr_node_p = list_p->head_p->next_p;
    while ( curr_node_p != NULL ) {
        if ( v < curr_node_p->value ) {
            list_insert( list_p, prev_node_p, v );
            return;
        }
        prev_node_p = curr_node_p;
        curr_node_p = curr_node_p->next_p;
    }

    // Insert at end of list
    list_insert( list_p, prev_node_p, v );
}

int main( void )
{
    list_t list;
    list_construct( &list );
    list_push_front( &list, 6 );
    list_push_front( &list, 4 );
    list_push_front( &list, 2 );
    list_sorted_insert( &list, 5 );
    list_destruct( &list );
    return 0;
}
**Pseudo-code for list_sort**

1. void list_sort( list_t* list_p )
2.     construct output list
3.    
4.    set curr node ptr to input list’s head ptr
5.    while curr node ptr is not NULL
6.        list_sorted_insert( output list, curr node’s value )
7.        set curr node ptr to curr node’s next ptr
8.    
9.    destruct input list
10.   set input list’s head ptr to output list’s head ptr

Unsorted input list

Sorted output list
void list_sort( list_t* list_p )
{
    list_t new_list;
    list_construct( &new_list );

    node_t* curr_node_p = list_p->head_p;
    while ( curr_node_p != NULL ) {
        list_sorted_insert( &new_list, curr_node_p->value );
        curr_node_p = curr_node_p->next_p;
    }

    list_destruct( list_p );
    list_p->head_p = new_list->head_p;
}
2. Vectors

- Recall the constraints on allocating arrays on the stack, and the need to explicitly pass the array size
- Let’s transform a dynamically allocated array along with its maximum size and actual size into a data structure

2.1. Vector Interface

```c
typedef struct
{
    int* data;
    size_t maxsize;
    size_t size;
} vector_t;

void vector_construct ( vector_t* vec_p, size_t maxsize, size_t size );
void vector_destruct ( vector_t* vec_p );
void vector_push_front ( vector_t* vec_p, int v );
void vector_insert ( vector_t* vec_p, size_t idx, int v );
void vector_sorted_insert ( vector_t* vec_p, int v );
void vector_sort ( vector_t* vec_p )
```

- `void vector_construct( vector_t* vector, size_t maxsize, size_t size );`
  Construct the vector initializing all fields in the given `vector_t`.

- `void vector_destruct( vector_t* vector );`
  Destruct the vector by freeing any dynamically allocated memory used by `vector_t`.
2. Vectors

2.2. Vector Implementation

- **void vector_push_front( vector_t* vector, int v );**
  Push a new element with the given value (v) at the front of the vector (i.e., index 0).

- **void vector_insert( vector_t* vector, size_t idx, int v );**
  Insert a new element with the given value (v) after the element with the given index idx.

- **void vector_sorted_insert( vector_t* vector, int v );**
  Assume the vector is already sorted in increasing order. Search the vector to find the proper place to insert a new node with the given value (v) such that the vector remains in sorted in increasing order.

- **void vector_sort( vector_t* vector );**
  Sort the given vector in increasing order. Any pointers to elements within the vector will be invalidated.

**Abstraction**

- User will likely need to directly manipulate the internal array
- Must directly access data to set/get values

2.2. **Vector Implementation**

- For each function we will use a combination of pseudo-code and “visual” pseudo-code to explain high-level approach
- Then we will translate the pseudo-code to actual C code
2. Vectors

2.2. Vector Implementation

**Pseudo-code for vector_construct**

```c
void vector_construct( vector_t* vec_p,
                      size_t maxsize, size_t size )

allocate new array with maxsize elements
set vector’s data to point to new array
set vector’s maxsize to given maxsize
set vector’s size to given size
```

Construct vector with a maxsize of 8 and 3 elements

**Pseudo-code for vector_push_front**

```c
void vector_push_front( vector_t* vec_p, int v )

set prev value to v
for i in 0 to vector’s size (inclusive)
    set temp value to vector’s data[i]
    set vector’s data[i] to prev value
    set prev value to temp value
set vector’s size to size + 1
```

Initial state of vector
After push front of value 8

After push front of value 9

**Pseudo-code for vector_destruct**

```c
1 void vector_destruct( vector_t* vec_p )
2   free vector’s data
```
// Construct vector data structure
void vector_construct( vector_t* vec_p,
    size_t maxsize,
    size_t size )
{
    vec_p->data = malloc( maxsize * sizeof(int) );
    vec_p->maxsize = maxsize;
    vec_p->size = size;
}

// Push value on front of vector
void vector_push_front( vector_t* vec_p,
    int v )
{
    assert((vec_p->maxsize-vec_p->size) >= 1);
    int prev_value = v;
    for ( size_t i=0; i<=vec_p->size; i++ ) {
        int temp_value = vec_p->data[i];
        vec_p->data[i] = prev_value;
        prev_value = temp_value;
    }
    vec_p->size += 1;
}

// Destruct vector data structure
void vector_destruct( vector_t* vec_p )
{
    free( vec_p->data );
}

// Main function
int main( void )
{
    vector_t vector;
    vector_construct( &vector, 8, 3 );
    vector.data[0] = 2;
    vector.data[1] = 4;
    vector.data[2] = 6;
    vector_push_front( &vector, 8 );
    vector_push_front( &vector, 9 );
    vector_destruct( &vector );
    return 0;
}
Pseudo-code for vector_insert

```c
void vector_insert( vector_t* vec_p, size_t idx, int v )
if vector is empty
    vector_push_front( vec_p, v )
else
    set prev value to v
    for i in idx+1 to vector’s size (inclusive)
        set temp value to vector’s data[i]
        set vector’s data[i] to prev value
        set prev value to temp value
    set vector’s size to size + 1
```

Pseudo-code for vector_sorted_insert

This pseudo-code ignores the corner cases when the list is empty and when the value needs to be inserted into the beginning or end of the list

```c
void vector_sorted_insert( vector_t* vec_p, int v )
for i in 0 to vector’s size
    if v is less than vector’s data[i]
        vector_insert( vec_p, i-1, v )
    return
```

Initial state of vector

After sorted insert of value 5
void vector_sorted_insert( vector_t* vec_p,  
    int v )  
{  
    assert((vec_p->maxsize-vec_p->size) >= 1);  

    // Insert into empty vector  
    if ( vec_p->size == 0 ) {  
        vector_push_front( vec_p, v );  
        return;  
    }  

    // Insert into beginning of vector  
    if ( v < vec_p->data[0] ) {  
        vector_push_front( vec_p, v );  
        return;  
    }  

    // Insert into middle of vector  
    for ( size_t i=0; i<vec_p->size; i++ ) {  
        if ( v < vec_p->data[i] ) {  
            vector_insert( vec_p, i-1, v );  
            return;  
        }  
    }  

    // Insert into end of vector  
    vector_insert( vec_p, vec_p->size-1, v );  
}  

int main( void )  
{  
    vector_t vector;  
    vector_construct( &vector, 8, 0 );  
    vector_push_front( &vector, 8 );  
    vector_push_front( &vector, 6 );  
    vector_push_front( &vector, 4 );  
    vector_push_front( &vector, 2 );  
    vector_sorted_insert( &vector, 5 );  
    vector_destruct( &vector );  
    return 0;  
}  

http://cpp.sh/4i23s
Pseudo-code for `vector_sort`:

```c
void vector_sort( vector_t* vec_p )
{
    construct output vector

    for i in 0 to vector’s size
        vector_sorted_insert( output vector,
                               input vector’s data[i] )

    destruct input vector
    set input vectors data ptr to output list’s data ptr
}
```

Unsorted input vector

```
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
```

Sorted output vector

```
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
...........
```

20
```c
void vector_sort( vector_t* vec_p )
{
    vector_t new_vector;
    vector_construct( &new_vector, vec_p->maxsize, 0 );

    for ( int i = 0; i < vec_p->size; i++ ) {
        vector_sorted_insert( &new_vector, vec_p->data[i] );
    }

    vector_destruct( vec_p );
    vec_p->data = new_vector->data;
}
```
3. Interaction Between Data Structures and Algorithms

- **Data structures** are: (1) a way of organizing data; and (2) a collection of operations for accessing and manipulating this data

- **Algorithms** are used to implement the operations for accessing and manipulating the data structure

- Algorithms and data structures are tightly connected

```c
void list_sort( list_t* list_p )
{
    list_t new_list;
    list_construct( &new_list );

    node_t* node_p = list_p->head_p;
    while ( node_p != NULL ) {
        list_sorted_insert( &new_list, node_p->value );
        node_p = node_p->next_p;
    }

    list_destruct( list_p );
    *list_p = new_list;
}

void vector_sort( vector_t* vec_p )
{
    vector_t new_vector;
    vector_construct( &new_vector, vec_p->maxsize, 0 );

    for ( int i = 0; i < vec_p->size; i++ ) {
        vector_sorted_insert( &new_vector, vec_p->data[i] );
    }

    vector_destruct( vec_p );
    *vec_p = new_vector;
}
```