# Topic 7: Lists and Vectors

School of Electrical and Computer Engineering  
Cornell University  

revision: 2019-10-02-11-29

## 1 Lists

1.1. Singly Linked List Interface ............................................. 5  
1.2. Singly Linked List Implementation ................................. 6  
1.3. Singly Linked Lists vs. Doubly Linked Lists ..................... 12

## 2 Vectors

2.1. Bounded Vector Interface ............................................. 13  
2.2. Bounded Vector Implementation ................................. 14  
2.3. Bounded Vectors vs. Resizable Vectors ......................... 19

## 3 Comparing Lists and Vectors

Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2019 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
• An **algorithm** is a clear set of steps to solve any problem in a particular problem class

```python
def fib( n):
    if ( n == 0 ): return 0
    if ( n == 1 ): return 1
    return fib( n-1 ) + fib( n-2 )
```

• A **data structure** is a structured way of storing data and the operations that can be applied to the data
  
  – *chain* of nodes each storing one integer
  
  – *array* of elements each storing one integer

• The `fib` algorithms do not involve a data structure
• The chain and array data structures do not involve an algorithm
• Most interesting programs involve a combination of algorithms and data structures

• Think of algorithms as **verbs** and data structures as **nouns**
• Most interesting stories involve a combination of verbs and nouns
### Algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mul</code></td>
<td>iter, single step</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>iter, recur</td>
</tr>
<tr>
<td><code>pow</code></td>
<td>iter, recur</td>
</tr>
</tbody>
</table>

### Data Structures

- chain of nodes
- array of elements
- list, vector
- stack, queue, set, map
- tree, table, graph

### Key Points

- Simple algorithms do not use a non-trivial data structure
- Simple data structures do not provide non-trivial operations
- Many algorithms operate on a simple data structure
- Many data structures provide operations which are implemented using an algorithm that operates on a simple data structure
- Sometimes our programs are more **algorithm centric**, sometimes they are more **data-structure centric**, but they almost always use both algorithms **and** data structures

**Algorithm + Data Structure = Program**

---

**Topic 7: Lists and Vectors**
• A data structure includes both an interface and an implementation
  – The interface specifies the “what”
  – The implementation specifies the “how”

• Separating interface from implementation is called data encapsulation or information hiding

Brainstorm other non-programming examples of interfaces and implementations. What are some reasons to separate the interface from the implementation?
1. Lists

- Recall our example of a chain of dynamically allocated nodes
- Let’s combine this data structure with a few simple algorithms to create a new concrete data type called a **singly linked list**

### 1.1. Singly Linked List Interface

```c
typedef struct {
    // implementation specific
} slist_int_t;

void slist_int_construct ( slist_int_t* this );
void slist_int_destruct ( slist_int_t* this );
void slist_int_push_front ( slist_int_t* this, int v );
void slist_int_reverse ( slist_int_t* this );
```

- **void slist_int_construct( slist_int_t* this );**
  Construct the slist initializing all fields in this slist_int_t. Undefined to call this function more than once on same slist.

- **void slist_int_destruct( slist_int_t* this );**
  Destruct the slist by freeing any dynamically allocated memory used by this slist_int_t. Undefined to call this function more than once on same slist.

- **void slist_int_push_front( slist_int_t* this, int v );**
  Push a new value (v) at the front of this slist_int_t. Undefined to call this function before construct or after destruct.

- **void slist_int_reverse( slist_int_t* this );**
  Reverse all values in this slist_int_t. Undefined to call this function before construct or after destruct.
1. Lists

1.2. Singly Linked List Implementation

Example of using list interface

```c
int main( void )
{
    slist_int_t lst;
    slist_int_construct( &lst );
    slist_int_push_front( &lst, 12 );
    slist_int_push_front( &lst, 11 );
    slist_int_push_front( &lst, 10 );
    slist_int_reverse ( &lst );
    slist_int_destruct ( &lst );
    return 0;
}
```

1.2. Singly Linked List Implementation

```c
typedef struct _slist_int_node_t
{
    int value;
    struct _slist_int_node_t* next_p;
} slist_int_node_t;
```

```c
typedef struct
{
    slist_int_node_t* head_p;
} slist_int_t;
```
Approach for implementing functions

1. Draw figure to explore high-level approach
2. Develop pseudo-code to capture high-level approach
3. Translate the pseudo-code to actual C code

**Pseudo-code for** slist_int_construct

```c
void slist_int_construct( slist_int_t* this )

set head ptr to NULL
```

**Pseudo-code for** slist_int_push_front

After push front of value 12

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

```c
void slist_int_push_front( slist_int_t* this, int v )
  allocate new node
  set new node’s value to v
  set new node’s next ptr to head ptr
  set head ptr to point to new node
```

**Pseudo-code for** \texttt{slist\_int\_destruct}

Dellocate head node?

Dellocate head node’s next pointer?
Need temporary pointer to point to next node!

```c
void slist_int_destruct( slist_int_t* this )
    while head ptr is not NULL
        set temp ptr to head node’s next ptr
        free head node
        set head node ptr to temp ptr
```
1. Lists

1.2. Singly Linked List Implementation

```c
// Construct slist
void slist_int_construct(
    slist_int_t* this )
{
    this->head_p = NULL;
}

// Push value on front of slist
void slist_int_push_front(
    slist_int_t* this,
    int v )
{
    slist_int_node_t* new_node_p = malloc( sizeof(slist_int_node_t) );
    new_node_p->value = v;
    new_node_p->next_p = this->head_p;
    this->head_p = new_node_p;
}

// Destruct slist
void slist_int_destruct(
    slist_int_t* this )
{
    while ( this->head_p != NULL ) {
        slist_int_node_t* temp_p = this->head_p->next_p;
        free( this->head_p );
        this->head_p = temp_p;
    }
}

// Main function
int main( void )
{
    slist_int_t lst;
    slist_int_construct ( &lst );
    slist_int_push_front( &lst, 12 );
    slist_int_push_front( &lst, 11 );
    slist_int_push_front( &lst, 10 );
    slist_int_destruct ( &lst );
    return 0;
}
```

https://repl.it/@cbatten/ece2400-T07-ex1
Interface vs. Implementation

- Implementation details are exposed in slist_int_t
- A user can freely manipulate fields in slist_int_t
- C does not provide any mechanism to enforce encapsulation

Develop an algorithm for slist_int_reverse
1.3. Singly Linked Lists vs. Doubly Linked Lists

• When programmers say “list” they usually mean a doubly linked list

• We will use slist for singly linked list, and just list for a doubly linked list

• We will try and be explicit in the course about the kind of list
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. Bounded Vector Interface

```c
typedef struct
{
    // implementation specific
}
bvector_int_t;

void bvector_int_construct ( bvector_int_t* this, size_t maxsize );
void bvector_int_destruct ( bvector_int_t* this );
void bvector_int_push_front ( bvector_int_t* this, int v );
void bvector_int_reverse ( bvector_int_t* this );
```

- **void bvector_int_construct( bvector_int_t* this,**
  ```
  size_t maxsize );
  ```

  Construct the bvector initializing all fields in this bvector_int_t. Undefined to call this function more than once on same bvector.

- **void bvector_int_destruct( bvector_int_t* this );**

  Destruct the bvector by freeing any dynamically allocated memory used by this bvector_int_t. Undefined to call this function more than once on same bvector.

- **void bvector_int_push_front( bvector_int_t* this,**
  ```
  int v );
  ```

  Push a new value (v) at the front of this bvector_int_t. Undefined to push more than maxsize values. Undefined to call this function before construct or after destruct.
• `void bvector_int_reverse( bvector_int_t* this );`
  Reverse all values in this `bvector_int_t`. Undefined to call this function before construct or after destruct.

**Example of using vector interface**

```c
int main( void )
{
  bvector_int_t vec;
  bvector_int_construct ( &vec, 4 );
  bvector_int_push_front( &vec, 12 );
  bvector_int_push_front( &vec, 11 );
  bvector_int_push_front( &vec, 10 );
  bvector_int_reverse ( &vec );
  bvector_int_destruct ( &vec );
  return 0;
}
```

### 2.2. Bounded Vector Implementation

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

• `data` is pointer to dynamically allocated array of `maxsize` elements
• `maxsize` is max number of elements we can store in `bvector`
• `size` is how many elements currently stored in `bvector`
2. Vectors

2.2. Bounded Vector Implementation

Approach for implementing functions

1. Draw figure to explore high-level approach
2. Develop pseudo-code to capture high-level approach
3. Translate the pseudo-code to actual C code

**Pseudo-code for `bvector_int_construct`**

```c
void vector_construct( bvector_int_t* this, size_t maxsize )
{
    allocate new array with maxsize elements
    set bvector's data to point to new array
    set bvector's maxsize to maxsize
    set bvector's size to zero
}
```

**Pseudo-code for `bvector_int_push_front`**

Initial state of `bvector`

After push front of value 9
After push front of value 8

Implement moving down all of the elements

```c
void vector_push_front( bvector_int_t* this, int v )
  set prev value to v
  for i in 0 to bvector’s size (inclusive)
    set temp value to bvector’s data[i]
    set bvector’s data[i] to prev value
    set prev value to temp value
  set bvector’s size to size + 1
```

**Pseudo-code for bvector_int_destruct**

```c
void vector_destruct( bvector_int_t* this )
  free bvector’s data
```
2. Vectors

2.2. Bounded Vector Implementation

1. // Construct bvector
2. void bvector_int_construct(
3.     bvector_int_t* this,
4.     size_t maxsize )
5. {
6.     this->data = malloc( maxsize * sizeof(int) );
7.     this->maxsize = maxsize;
8.     this->size = 0;
9. }
10. // Push value on front of bvector
11. void bvector_int_push_front(
12.     bvector_int_t* this, int v )
13. {
14.     int prev_value = v;
15.     for ( size_t i=0; i<=this->size; i++ ) {
16.         int temp_value = this->data[i];
17.         this->data[i] = prev_value;
18.         prev_value = temp_value;
19.     }
20.     this->size += 1;
21. }
22. // Destruct bvector
23. void bvector_int_destruct(
24.     bvector_int_t* this )
25. {
26.     free( this->data );
27. }
28. // Main function
29. int main( void )
30. {
31.     bvector_int_t vec;
32.     bvector_int_construct ( &vec, 4 );
33.     bvector_int_push_front( &vec, 12 );
34.     bvector_int_push_front( &vec, 11 );
35.     bvector_int_push_front( &vec, 10 );
36.     bvector_int_destruct ( &vec );
37.     return 0;
38. }
39. https://repl.it/@cbatten/ece2400-T07-ex2
Interface vs. Implementation

- Implementation details are exposed in `bvector_int_t`
- A user can freely manipulate fields in `bvector_int_t`
- C does not provide any mechanism to enforce encapsulation

Develop an algorithm for `bvector_int_reverse`
2.3. Bounded Vectors vs. Resizable Vectors

• When programmers say “vector” they usually mean a resizable vector.

• We will use `bvector` for bounded vector, and `just vector` for a resizable vector.

• We will try and be explicit in the course about the kind of vector.
3. Comparing Lists and Vectors

- Many more functions are possible for both lists and vectors

```c
void ds_int_construct ( ds_int_t* this );
void ds_int_destruct ( ds_int_t* this );
void ds_int_push_front ( ds_int_t* this, int v );
void ds_int_reverse ( ds_int_t* this );

void ds_int_push_back ( ds_int_t* this, int v );
size_t ds_int_size ( ds_int_t* this );
int ds_int_at ( ds_int_t* this, size_t idx );
int ds_int_find ( ds_int_t* this, int v );
void ds_int_print ( ds_int_t* this );

void ds_int_insert ( ds_int_t* this, ptr_t* ptr, int v );
void ds_int_remove ( ds_int_t* this, ptr_t* ptr );
```

- The list and vector data structures ...
  - ... have similar interfaces, but
  - ... very different execution times, and
  - ... very different space usage.
3. Comparing Lists and Vectors

- Compare the execution time and space usage of the algorithms?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution Time</th>
<th>Space Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>slist</td>
<td>bvector</td>
</tr>
<tr>
<td>push_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push_back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>at</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What about comparing a doubly linked list or a resizable vector?
- Compare the space usage of the data structure itself?