1 List CDTs
   1.1. List CDT Interface ........................................ 4
   1.2. List CDT Implementation .................................. 5

2 Vector CDTs
   2.1. Vector CDT Interface ..................................... 11
   2.2. Vector CDT Implementation .............................. 12

3 Comparing List and Vector CDTs .............................. 17
• 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
  – chain of nodes each storing one integer
  – linear array of integers

• 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
• We can use a combination of algorithms and data structures to implement a **concrete data type** (CDT)

• A CDT 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**
1. List CDTs

- 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 linked list

1.1. List CDT Interface

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

void list_int_construct ( list_int_t* this );
void list_int_destruct ( list_int_t* this );
void list_int_push_front ( list_int_t* this, int v );
void list_int_reverse ( list_int_t* this );
```

- **void list_int_construct( list_int_t* this );**
  Construct the list initializing all fields in this list_int_t.

- **void list_int_destruct( list_int_t* this );**
  Destruct the list by freeing any dynamically allocated memory used by this list_int_t.

- **void list_int_push_front( list_int_t* this, int v );**
  Push a new value (v) at the front of this list_int_t.

- **void list_int_reverse( list_int_t* this );**
  Reverse all values in this list_int_t.
Example of using list interface

```c
int main( void )
{
    list_int_t list;
    list_int_construct( &list );
    list_int_push_front( &list, 12 );
    list_int_push_front( &list, 11 );
    list_int_push_front( &list, 10 );
    list_int_reverse ( &list );
    list_int_destruct ( &list );
    return 0;
}
```

1.2. List CDT Implementation

```c
typedef struct _list_int_node_t
{
    int value;
    struct _list_int_node_t* next_p;
} list_int_node_t;

typedef struct
{
    list_int_node_t* head_p;
} list_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 list_int_construct**

```c
1 void list_int_construct( list_int_t* this )
2   set head ptr to NULL
```

**Pseudo-code for list_int_push_front**

After push front of value 12

After push front of value 11
1. List CDTs

1.2. List CDT Implementation

After push front of value 10

```c
1 void list_int_push_front( list_int_t* this, 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
```

Pseudo-code for `list_int_destruct`

Dellocate head node?

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

```c
void list_int_destruct( list_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. List CDTS

1.2. List CDT Implementation

```c
// Construct list
void list_int_construct( list_int_t* this ) {
    this->head_p = NULL;
}

// Push value on front of list
void list_int_push_front( list_int_t* this, int v ) {
    list_int_node_t* new_node_p = malloc( sizeof(list_int_node_t) );
    new_node_p->value = v;
    new_node_p->next_p = this->head_p;
    this->head_p = new_node_p;
}

// Destruct list
void list_int_destruct( list_int_t* this ) {
    while ( this->head_p != NULL ) {
        list_int_node_t* temp_p = this->head_p->next_p;
        free( this->head_p );
        this->head_p = temp_p;
    }
}

// Main function
int main( void ) {
    list_int_t list;
    list_int_construct ( &list );
    list_int_push_front( &list, 12 );
    list_int_push_front( &list, 11 );
    list_int_push_front( &list, 10 );
    list_int_destruct ( &list );
    return 0;
}
https://repl.it/@cbatten/ece2400-T07-ex1
```
Interface vs. Implementation

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

Develop an algorithm for `list_int_reverse`
2. Vector CDTs

- 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 CDT

2.1. Vector CDT Interface

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

void vector_int_construct ( vector_int_t* this );
void vector_int_destruct ( vector_int_t* this );
void vector_int_push_front ( vector_int_t* this, int v );
void vector_int_reverse ( vector_int_t* this );
```

- `void vector_int_construct( vector_int_t* this );` Construct the vector initializing all fields in this vector_int_t.
- `void vector_int_destruct( vector_int_t* this );` Destruct the vector by freeing any dynamically allocated memory used by this vector_int_t.
- `void vector_int_push_front( vector_int_t* this, int v );` Push a new value (v) at the front of this vector_int_t.
- `void vector_int_reverse( vector_int_t* this );` Reverse all values in this vector_int_t.
Example of using vector interface

```c
int main( void )
{
    vector_int_t vector;
    vector_int_construct ( &vector );
    vector_int_push_front( &vector, 12 );
    vector_int_push_front( &vector, 11 );
    vector_int_push_front( &vector, 10 );
    vector_int_reverse ( &vector );
    vector_int_destruct ( &vector );
    return 0;
}
```

2.2. Vector CDT Implementation

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

- To simplify our discussion in lecture, we will allocate a small array in the constructor, and then we assume we only push back a limited number of values
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** vector_int_construct

```c
void vector_construct( vector_int_t* this )
2 allocate new array with eight elements
3 set vector’s data to point to new array
4 set vector’s maxsize to eight
5 set vector’s size to zero
```

**Pseudo-code for** vector_int_push_front

Initial state of vector

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

Implement moving down all of the elements

```c
void vector_push_front( vector_int_t* this, 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
```

Pseudo-code for `vector_int_destruct`

```c
void vector_destruct( vector_int_t* this )
    free vector’s data
```
// Construct vector
void vector_int_construct(vector_int_t* this) {
    this->data = malloc(8 * sizeof(int));
    this->maxsize = 8;
    this->size = 0;
}

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

// Destruct vector
void vector_int_destruct(vector_int_t* this) {
    free(this->data);
}

// Main function
int main(void) {
    vector_int_t vector;
    vector_int_construct( &vector );
    vector_int_push_front( &vector, 12 );
    vector_int_push_front( &vector, 11 );
    vector_int_push_front( &vector, 10 );
    vector_int_destruct( &vector );
    return 0;
}

https://repl.it/@cbatten/ece2400-T07-ex2
2. Vector CDTs

2.2. Vector CDT Implementation

Interface vs. Implementation

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

Develop an algorithm for `vector_int_reverse`
3. Comparing List and Vector CDTs

- The list and vector CDTs ...
  - ... have similar interfaces, but
  - ... very different execution times, and
  - ... very different space usage.

- 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>List</td>
<td>Vector</td>
</tr>
<tr>
<td>push_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push_back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find</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>

- Compare the space usage of the data structure itself?

An analysis of a CDT’s execution time and space usage is usually a key part of the CDT’s high-level specification