1. Lists

1.1. List Interface

A data structure is: (1) a way of organizing data; and (2) a collection of operations for accessing and manipulating this data.

An algorithm is a clear set of steps to solve any problem instance in a particular class of problems; algorithms are used to implement the operations for accessing and manipulating the data structure.

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

```
typedef struct _node_t
{
    int value;
    struct _node_t* next_p;
} node_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 );
```
1. Lists

- void list_construct( list_t* list );
  Construct the list initializing all fields in the given list_t.

- void list_destruct( list_t* list );
  Destroy 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
After push front of value 10

![Diagram of list structure after push front]

**Pseudo-code for list_destroy**

```c
void list_destroy(list_t* list_p)
{
    void* node_p;
    while (list_p) {
        node_p = list_p;
        list_p = list_p->next;
        free(node_p);
    }
}
```

After first iteration of while loop

![Diagram of list structure after first iteration]

After second iteration of while loop

![Diagram of list structure after second iteration]

And so on...
1. Lists

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

1.2. List Implementation

```c
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;
    }
}
```

```c
int main( void )
{
    list_t list;
    list_construct( &list );
    list_push_back( &list, 9 );
    list_push_back( &list, 8 );
    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_insert( &list, list.head_p, 4 );
    return 0;
}
```

http://cpp.sh/83gnp
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, int v )
{
    // Insert into empty list
    if (list->head_p == NULL) {
        list_push_front(list, v);
        return;
    }
    // Insert at beginning of list
    if (v < list->head_p->value) {
        list_push_front(list, v);
        return;
    }
    // Insert in middle of list
    node_t* prev_node_p = list->head_p;
    node_t* node_p = list->head_p->next_p;
    while (node_p != NULL) {
        if (v < node_p->value) {
            list_insert(list, prev_node_p, v);
            return;
        }
        prev_node_p = node_p;
        node_p = node_p->next_p;
    }
    // Insert at end of list
    list_insert(list, prev_node_p, v);
}
```

Moving the node pointers:

After sorted insert of value 5:

```
1. Lists

1.2. List Implementation

1. Lists

1.2. List Implementation

http://cpp.sh/2douz
```
Pseudo-code for list_sort

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

2.2. 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
**Pseudo-code for vector_construct**

1. `void vector_construct( vector_t* vec_p, size_t maxsize, size_t size )`
2. allocate new array with maxsize elements
3. set vector's data to point to new array
4. set vector's maxsize to given maxsize
5. set vector's size to given size

Construct vector with a maxsize of 8 and 3 elements

**Pseudo-code for vector_push_front**

1. `void vector_push_front( vector_t* vec_p, int v )`
2. set prev value to v
3. 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**

1. `void vector_destruct( vector_t* vec_p )`
2. free vector's data
2. Vectors

2.2. Vector Implementation

Pseudo-code for vector_insert

```c
void vector_sorted_insert( vector_t* vec_p, size_t idx, int v )
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

```
8 4 5 1
```

After sorted insert of value 5

```
5 8 4 5 6 1
```
2. Vectors

2.2. Vector Implementation

```c
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_destruct( &vector );
    return 0;
}
```

### 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

```
```
2. Vectors

2.2. Vector Implementation

```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

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;
}
```