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

- 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

```
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`

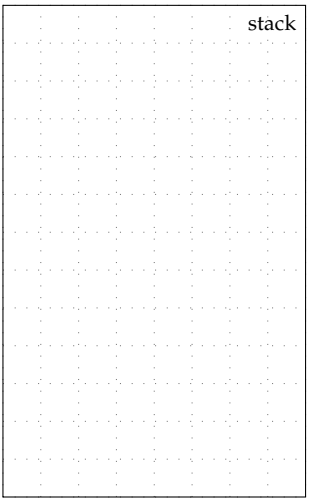
```
1 void list_destruct( list_t* list_p )
2   set curr node ptr to head ptr
3   while curr node ptr is not NULL
4     set temp node ptr to curr node's next ptr
5     free curr node
6     set curr node ptr to temp node ptr
```

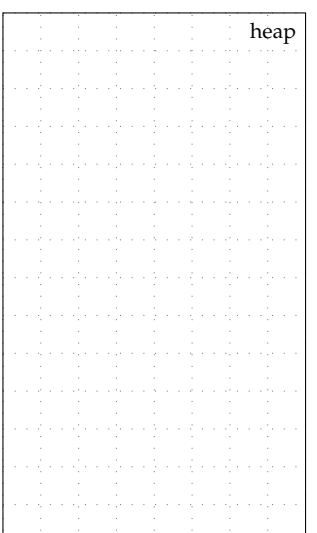
After first iteration of while loop

After second iteration of while loop

And so on ...

```
1  // Construct list data structure
2  void list_construct( list_t* list_p )
3  {
4    list_p->head_p = NULL;
5  }
6
7  // Push value on front of list
8  void list_push_front( list_t* list_p,
9                        int v )
10 {
11   node_t* new_node_p
12     = malloc( sizeof(node_t) );
13
14   new_node_p->value  = v;
15   new_node_p->next_p = list_p->head_p;
16   list_p->head_p     = new_node_p;
17 }
18
19 // Destruct list data structure
20 void list_destruct( list_t* list_p )
21 {
22   node_t* curr_node_p = list_p->head_p;
23   while ( curr_node_p != NULL ) {
24     node_t* next_node_p
25       = curr_node_p->next_p;
26     free( curr_node_p );
27     curr_node_p = next_node_p;
28   }
29   list_p->head_p = NULL;
30 }
31
32 // Main function
33 int main( void )
34 {
35   list_t list;
36   list_construct( &list );
37   list_push_front( &list, 12 );
38   list_push_front( &list, 11 );
39   list_push_front( &list, 10 );
40   list_destruct( &list );
41   return 0;
42 }
```





http://cpp.sh/9ggkr

Pseudo-code for `list_insert`

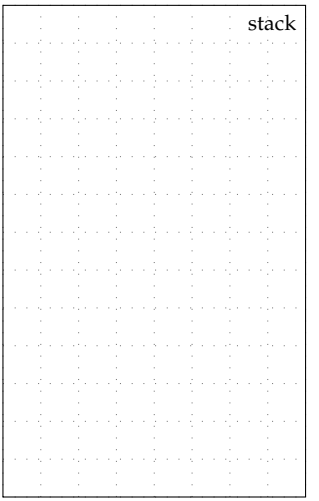
```
1 void list_insert( list_t* list_p, node_t* node_p, int v )
2   if list is empty
3     list_push_front( list_p, v )
4   else
5     allocate new node
6     set new node's value to v
7     set new node's next ptr to node's next ptr
8     set node's next ptr to point to new node
```

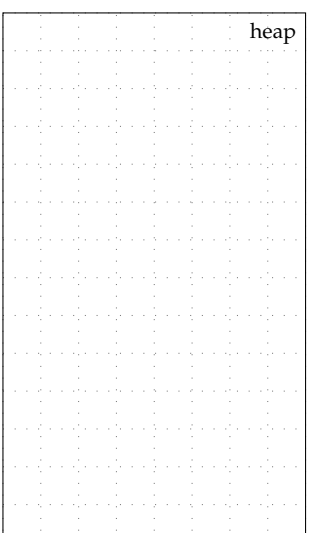
Initial state of the list

After insert of value of 4 after first node

After insert of value 8 after last node

```
1  void list_insert( list_t* list_p,
2                    node_t* node_p,
3                    int v )
4  {
5    if ( node_p == NULL ) {
6      list_push_front( list_p, v );
7    }
8    else {
9      node_t* new_node_p
10       = malloc( sizeof(node_t) );
11
12     new_node_p->value  = v;
13     new_node_p->next_p = node_p->next_p;
14     node_p->next_p     = new_node_p;
15   }
16 }
17
18 int main( void )
19 {
20   list_t list;
21   list_construct( &list );
22   list_push_front( &list, 6 );
23   list_push_front( &list, 2 );
24
25   list_insert( &list, list.head_p, 4 );
26
27   node_t* tail_p
28     = list.head_p->next_p->next_p;
29   list_insert( &list, tail_p, 8 );
30
31   list_destruct( &list );
32   return 0;
33 }
```





http://cpp.sh/83gnp

Pseudo-code for `list_sorted_insert`

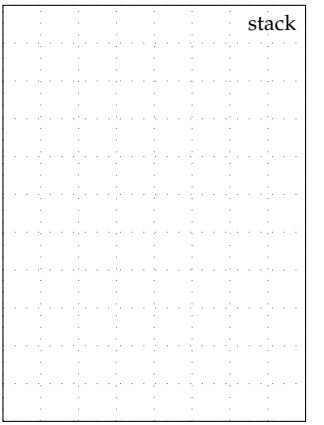
This pseudo-code ignores the corner cases when the list is empty and when the value needs to be inserted into the beginning or end of the list

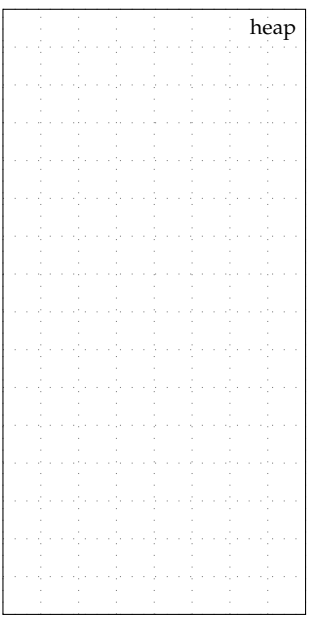
```
1  void list_sorted_insert( list_t* list_p, int v )
2    set prev node ptr to head ptr
3    set curr node ptr to head node's next ptr
4
5    while curr node ptr is not NULL
6      if v is less than curr node's value
7        list_insert( list_p, prev node ptr, v )
8        return
9
10     set prev node ptr to curr node ptr
11     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

```
1  void list_sorted_insert( list_t* list_p,
2                           int v )
3  {
4    // Insert into empty list
5    if ( list_p->head_p == NULL ) {
6      list_push_front( list_p, v );
7      return;
8    }
9
10   // Insert at beginning of list
11   if ( v < list_p->head_p->value ) {
12     list_push_front( list_p, v );
13     return;
14   }
15
16   // Insert in middle of list
17
18   node_t* prev_node_p = list_p->head_p;
19   node_t* curr_node_p
20     = list_p->head_p->next_p;
21
22   while ( curr_node_p != NULL ) {
23     if ( v < curr_node_p->value ) {
24       list_insert( list_p, prev_node_p, v );
25       return;
26     }
27     prev_node_p = curr_node_p;
28     curr_node_p = curr_node_p->next_p;
29   }
30
31   // Insert at end of list
32   list_insert( list_p, prev_node_p, v );
33 }
34
35 int main( void )
36 {
37   list_t list;
38   list_construct( &list );
39   list_push_front( &list, 6 );
40   list_push_front( &list, 4 );
41   list_push_front( &list, 2 );
42   list_sorted_insert( &list, 5 );
43   list_destruct( &list );
44   return 0;
45 }
```





http://cpp.sh/2douz

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

```
1  void list_sort( list_t* list_p )
2  {
3    list_t new_list;
4    list_construct( &new_list );
5
6    node_t* curr_node_p = list_p->head_p;
7    while ( curr_node_p != NULL ) {
8      list_sorted_insert( &new_list, curr_node_p->value );
9      curr_node_p = curr_node_p->next_p;
10   }
11
12   list_destruct( list_p );
13   list_p->head_p = new_list->head_p;
14 }
```

2. Vectors

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

2.1. Vector Interface

```
1  typedef struct
2  {
3    int*   data;
4    size_t maxsize;
5    size_t size;
6  }
7  vector_t;
8
9  void vector_construct     ( vector_t* vec_p,
10                             size_t maxsize, size_t size );
11 void vector_destruct      ( vector_t* vec_p );
12 void vector_push_front    ( vector_t* vec_p, int v );
13 void vector_insert        ( vector_t* vec_p, size_t idx, int v );
14 void vector_sorted_insert ( vector_t* vec_p, int v )
15 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`.

- `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`

```
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`

```
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

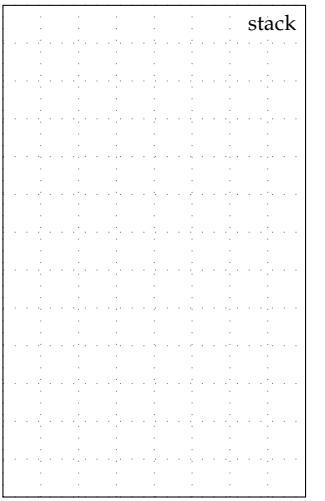
After push front of value 8

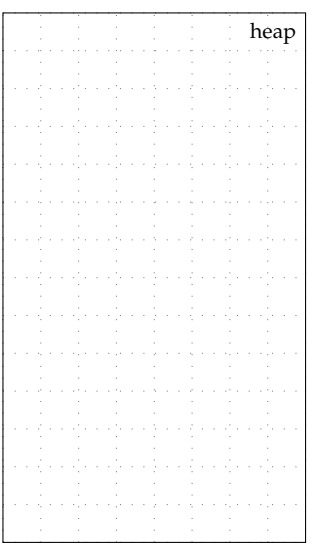
After push front of value 9

Pseudo-code for `vector_destruct`

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

```
1  // Construct vector data structure
2  void vector_construct( vector_t* vec_p,
3                         size_t maxsize,
4                         size_t size )
5  {
6    vec_p->data
7      = malloc( maxsize * sizeof(int) );
8    vec_p->maxsize = maxsize;
9    vec_p->size    = size;
10 }
11
12 // Push value on front of vector
13 void vector_push_front( vector_t* vec_p,
14                         int v )
15 {
16   assert((vec_p->maxsize-vec_p->size) >= 1);
17   int prev_value = v;
18   for ( size_t i=0; i<=vec_p->size; i++ ) {
19     int temp_value = vec_p->data[i];
20     vec_p->data[i] = prev_value;
21     prev_value = temp_value;
22   }
23   vec_p->size += 1;
24 }
25
26 // Destruct vector data structure
27 void vector_destruct( vector_t* vec_p )
28 {
29   free( vec_p->data );
30 }
31
32 // Main function
33 int main( void )
34 {
35   vector_t vector;
36   vector_construct( &vector, 8, 3 );
37   vector.data[0] = 2;
38   vector.data[1] = 4;
39   vector.data[2] = 6;
40   vector_push_front( &vector, 8 );
41   vector_push_front( &vector, 9 );
42   vector_destruct( &vector );
43   return 0;
44 }
```





http://cpp.sh/4wtv6

Pseudo-code for `vector_insert`

```
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`

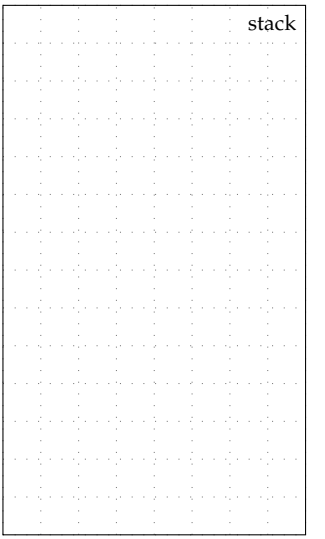
This pseudo-code ignores the corner cases when the list is empty and when the value needs to be inserted into the beginning or end of the list

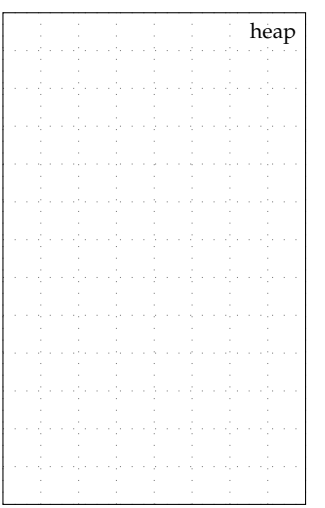
```
void vector_sorted_insert( vector_t* vec_p, int v )
  for i in 0 to vector's size
    if v is less than vector's data[i]
      vector_insert( vec_p, i-1, v )
      return
```

Initial state of vector

After sorted insert of value 5

```
1  void vector_sorted_insert( vector_t* vec_p,
2                             int v )
3  {
4    assert((vec_p->maxsize-vec_p->size) >= 1);
5
6    // Insert into empty vector
7    if ( vec_p->size == 0 ) {
8      vector_push_front( vec_p, v );
9      return;
10   }
11
12   // Insert into beginning of vector
13   if ( v < vec_p->data[0] ) {
14     vector_push_front( vec_p, v );
15     return;
16   }
17
18   // Insert into middle of vector
19   for ( size_t i=0; i<vec_p->size; i++ ) {
20     if ( v < vec_p->data[i] ) {
21       vector_insert( vec_p, i-1, v );
22       return;
23     }
24   }
25
26   // Insert into end of vector
27   vector_insert( vec_p, vec_p->size-1, v );
28 }
29
30 int main( void )
31 {
32   vector_t vector;
33   vector_construct( &vector, 8, 0 );
34   vector_push_front( &vector, 8 );
35   vector_push_front( &vector, 6 );
36   vector_push_front( &vector, 4 );
37   vector_push_front( &vector, 2 );
38   vector_sorted_insert( &vector, 5 );
39   vector_destruct( &vector );
40   return 0;
41 }
```





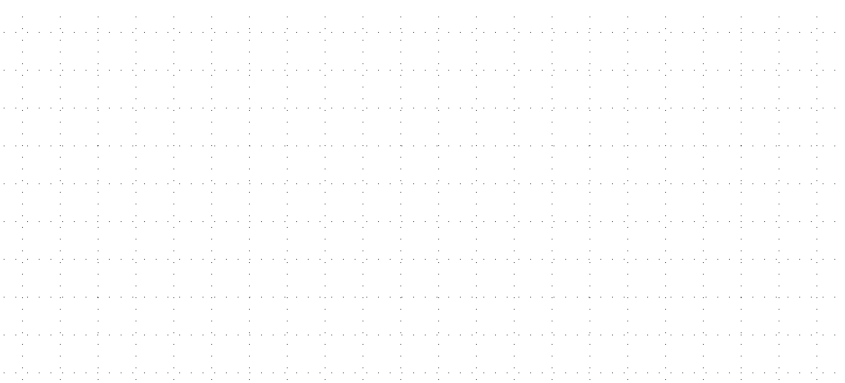
http://cpp.sh/4i23s

Pseudo-code for `vector_sort`

```
1 void vector_sort( vector_t* vec_p )
2   construct output vector
3
4   for i in 0 to vector's size
5     vector_sorted_insert( output vector,
6                           input vector's data[i] )
7
8   destruct input vector
9   set input vectors data ptr to output list's data ptr
```

Unsorted input vector

Sorted output vector



```
1  void vector_sort( vector_t* vec_p )
2  {
3    vector_t new_vector;
4    vector_construct( &new_vector, vec_p->maxsize, 0 );
5
6    for ( int i = 0; i < vec_p->size; i++ ) {
7      vector_sorted_insert( &new_vector, vec_p->data[i] );
8    }
9
10   vector_destruct( vec_p );
11   vec_p->data = new_vector->data;
12 }
```

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

```
1  void list_sort( list_t* list_p )
2  {
3    list_t new_list;
4    list_construct( &new_list );
5
6    node_t* node_p = list_p->head_p;
7    while ( node_p != NULL ) {
8      list_sorted_insert( &new_list, node_p->value );
9      node_p = node_p->next_p;
10   }
11
12   list_destruct( list_p );
13   *list_p = new_list;
14 }
```

```
1  void vector_sort( vector_t* vec_p )
2  {
3    vector_t new_vector;
4    vector_construct( &new_vector, vec_p->maxsize, 0 );
5
6    for ( int i = 0; i < vec_p->size; i++ ) {
7      vector_sorted_insert( &new_vector, vec_p->data[i] );
8    }
9
10   vector_destruct( vec_p );
11   *vec_p = new_vector;
12 }
```