ECE 2400 Computer Systems Programming

Topic 7: Lists and Vectors

`http://www.csl.cornell.edu/courses/ece2400`
School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-14-11-04

Please do not ask for solutions. Students should compare their solutions to solutions from their fellow students, discuss their solutions with the instructors during lab/office hours, and/or post their solutions on Ed for discussion.

List of Problems

Problem 1. Short Answer

Carefully plan your solution before starting to write your response. Please be brief and to the point; if at all possible, limit your answers to the space provided.

Part 1.A Circular Linked Lists

In lecture, we discussed a singly linked list with the following interface. We have also included the implementation of the `slist_int_t` struct, and the implementation-specific node definition.

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

typedef struct
{
  slist_int_node_t* head_p;
}
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 );
int  slist_int_contains   ( slist_int_t* this, int v );
```

Notice we have added a new function `slist_int_contains` which takes as input two parameters: `this` is a pointer to an `slist_int_t` to operate on, and `v` is a value to search for in this list. The function returns 1 if the list contains the value and 0 if the list does not contain the value. **Implement the `slist_int_contains` function.** *Your implementation must handle any reasonable corner cases correctly.* You cannot add fields to `slist_int_t`. Your implementation should be correct and efficient in terms of both execution time and space usage. While you are welcome to use pseudo-code to plan your approach, your final solution must be written using valid C syntax.

```
int slist_int_contains( slist_int_t* this, int v ) {
  assert( this != NULL );
```

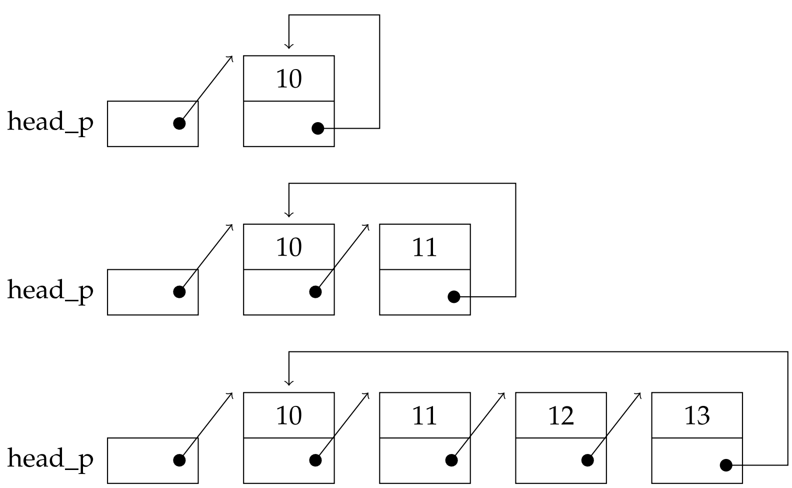
Consider the following (partial) interface for a circular list. A *circular list* is a kind of linked list where the next pointer of the tail node points to the head node. We have also included the implementation of the `clist_int_t` struct, and the implementation-specific node definition.

```
typedef struct _clist_int_node_t
{
  int                       value;
  struct _clist_int_node_t* next_p;
}
clist_int_node_t;

typedef struct
{
  clist_int_node_t* head_p;
}
clist_int_t;

void clist_int_construct  ( clist_int_t* this );
void clist_int_destruct   ( clist_int_t* this );
void clist_int_push_front ( clist_int_t* this, int v );
int  clist_int_contains   ( clist_int_t* this, int v );
```

Here is an example of three different circular lists with one, two, and four nodes respectively.



Notice that this data structure also includes a function `clist_int_contains` with a similar interface as `slist_int_contains`.

Implement the `clist_int_contains` **function.** *Your implementation must handle any reasonable corner cases correctly.* You cannot add fields to `clist_int_t`. Your implementation should be correct and efficient in terms of both execution time and space usage. While you are welcome to use pseudo-code to plan your approach, your final solution must be written using valid C syntax.

```
int clist_int_contains( clist_int_t* this, int v ) {
  assert( this != NULL );
```

Problem 2. Removing Elements from Lists and Vectors

In this problem, you will explore implementing a new function for each of three different data structures: a singly linked list, a doubly linked list, and a bounded vector. More specifically, we will implement a `remove` function that can remove an element from any position in the data structure. Unless otherwise specified, the baseline implementation of the data structures is identical to what was discussed in lecture. For reference, the interface for each data structure along with a new `remove` function is shown below.

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

```
typedef struct
{
  slist_int_node_t* head_p;
}
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_remove     ( slist_int_t* this, slist_int_node_t* node_p );
```

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

```
typedef struct
{
  list_int_node_t* head_p;
  list_int_node_t* tail_p;
}
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_remove     ( list_int_t* this, list_int_node_t* node_p );
```

```
typedef struct
{
  int* data;
  int  maxsize;
  int  size;
}
bvector_t;

void bvector_int_construct  ( bvector_t* this, int maxsize );
void bvector_int_destruct   ( bvector_t* this );
void bvector_int_push_front ( bvector_t* this, int v );
void bvector_int_remove     ( bvector_t* this, int idx );
```

Part 2.A Removing Element from Singly Linked List

The first implementation of the `remove` function is for a singly linked list data structure. This function should take a pointer to the list data structure (`this`) and a pointer to a node (`node_p`) as parameters. This function should remove the node pointed to by `node_p`. **Implement the `slist_int_remove` C function. Clearly identify any corner cases and choose a reasonable approach to handle those corner cases.** While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C syntax. Avoid memory leaks.

```
void slist_int_remove( slist_int_t* this, slist_int_node_t* node_p ) {
```

Part 2.B Removing Element from Doubly Linked List

The second implementation of the `remove` function is for the doubly linked list data structure. This function should take a pointer to the list data structure (`this`) and a pointer to a node (`node_p`) as parameters. This function should remove the node pointed to by `node_p`. **Implement the `list_int_remove` C function. Clearly identify any corner cases and choose a reasonable approach to handle those corner cases.** While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C syntax. Avoid memory leaks.

```
void list_int_remove( list_int_t* this, list_int_node_t* node_p ) {
```

Part 2.C Removing Element from Bounded Vector

The third implementation of the `remove` function is for a bounded vector data structure. This function should take a pointer to the vector data structure (`bvector_t`) and an index to an element in the vector (`idx`). This function should remove the element at the given `idx`. **Implement the `bvector_int_remove` C function. Clearly identify any corner cases and choose a reasonable approach to handle those corner cases.** While you are welcome to use pseudocode to plan your approach, your final solution must be written using valid C syntax. Avoid memory leaks.

```
void bvector_int_remove( bvector_t* this, int idx ) {
```

Part 2.D Comparing Remove Algorithms

Note: This problem involves material on complexity analysis from Topic 8.

In this problem, you will be qualitatively comparing the three remove algorithms. **Begin by filling in the following table.**

	Wost Case Time Complexity big-O	Worst Case Aux Heap Space Complexity big-O	Inherent Data Structure Space Usage $S(N)$	Inherent Data Structure Space Complexity big-O
`slist_int_remove`				
`list_int_remove`				
`bvector_int_remove`				

Use these results along with deeper insights to perform a comparative analysis of these two search algorithms, with the ultimate goal of **making a compelling argument for which data structure will perform better across a large number of usage scenarios where we need to do many remove operations.** While you are free to use whatever approach you like, we recommend you structure your response in several paragraphs. The **first paragraph** might discuss the performance of each data structure and remove algorithm using time complexity analysis. Justify the entries in the table. Remember that time complexity analysis is not the entire story; it is just the starting point for performance analysis. The **second paragraph** might discuss the space requirements of each data structure and algorithm using space complexity analysis. Consider discussing both the inherent space usage of the data structures themselves along with the auxillary heap space usage of the actual remove algorithm. Justify the entries in the table. Remember that space complexity analysis is not the entire story; it is just the starting point for storage requirement analysis. The **third paragraph** might discuss other qualitative metrics such as generality, maintainability, and design complexity. The **final paragraph** can conclude by making a compelling argument for which data structure and remove algorithm will perform better in the general case when we need to do many remove operations, or if you cannot strongly argue for either algorithm explain why. Your answer will be assessed on how well you argue your position.