CS 261: Data Structures

Linked Lists

List Stack
Dynamic Array -- Problems

- Data kept in a single large block of memory
- Often more memory used than necessary
  - especially when repeatedly growing and shrinking the dynamic array
Linked List

- A good alternative
- The memory use is always proportional to the number of elements in the collection
Characteristics of Linked Lists

• Elements are held in objects called **Links**

• Links are 1-to-1 with data elements, allocated and released as necessary
Single and Double Linked Lists

Each link points to

only next link $\Rightarrow$ single linked list

next and previous links $\Rightarrow$ double linked list

in the sequence
Link Structure

```c
struct Link {
    TYPE   value;
    struct Link *next;
};
```

![Diagram of Link Structure]
Elements of Linked Lists

- **Sentinel** -- special link for start or end

- Points to
  - first **or** last link only (single linked list)
  - first **and** last link (double linked list)
List Stack

- Sentinel points to the first element
- Sentinel points to NULL if stack empty
- Add or remove elements only from front
- Allow only singly linked list
- Can access only first element
List Stack

stk

sentinel
  next:

  val: 2
  next:

  val: 7
  next:

  val: 4
  next: null
Implementation of List Stack

```c
struct Link {
    TYPE value;
    struct Link * next;
};

struct ListStack {
    struct Link * sentinel;
};
```

How to initialize List Stack?
void InitStack(struct ListStack * stk){
    /* initialize the sentinel */
    struct Link *sentinel =
        (struct Link *)malloc(sizeof(struct Link));

    assert(sentinel != 0);

    /* linked list is empty */
    sentinel->next = NULL;
    stk->sentinel = sentinel;
}
Push List Stack: 3 Steps

Step 1: allocate new link
Push List Stack: 3 Steps

Step 2: link the new element to the next data element

stk
sentinel

val: 5
next:

val: 2
next:

val: 7
next:

val: 4
next: null
Push List Stack: 3 Steps

Step 3: add the new element to the top

```
stk
sentinel

val: 5
next:

val: 2
next:

val: 7
next:

val: 4
next: null
```
void pushStack (struct listStack *stk, TYPE val){
    struct Link * new =
        (struct Link *) malloc(sizeof(struct Link));
    assert (new != 0);
    new->value = val;
}

Step 1: allocate new link
Push List Stack: 3 Steps

```c
void pushStack (struct listStack *stk, TYPE val){
    struct Link * new =
        (struct Link *) malloc(sizeof(struct Link));
    assert (new != 0);
    new->value = val;
    new->next = stk->sentinel->next;
}
```

Step 2: link the new element to the next data element.
void pushStack (struct listStack *stk, TYPE val){

    struct Link * new =
          (struct Link *) malloc(sizeof(struct Link));

    assert (new != 0);

    new->value = val;

    new->next = stk->sentinel->next;

    stk->sentinel->next = new;
}

Step 3: add the new element to the top
PopStack

/*move the top to the next element*/
void PopStack (struct ListStack *stk) {
    struct Link * lnk = stk->sentinel->next;
    if(lnk!=NULL){ /*the top element exists*/

    }

    stk

    sentinel

    lnk

    val: 5
    next:

    val: 2
    next:

    val: 7
    next:

    val: 4
    next: null
/*move the top to the next element*/
void PopStack (struct ListStack *stk) {
    struct Link * lnk = stk->sentinel->next;
    if(lnk!=NULL) { /*the top element exists*/
        stk->sentinel->next = lnk->next;
        free(lnk);
    }
}

/*move the top to the next element*/
void PopStack (struct ListStack *stk) {
    struct Link * lnk = stk->sentinel->next;
    if(lnk!=NULL) { /*the top element exists*/
        stk->sentinel->next = lnk->next;
        free(lnk);
    }
}

/*move the top to the next element*/
void PopStack (struct ListStack *stk) {
    struct Link * lnk = stk->sentinel->next;
    if(lnk!=NULL) { /*the top element exists*/
        stk->sentinel->next = lnk->next;
        free(lnk);
    }
}
topStack, isEmpty...

• Should be done on your own
• Worksheet 17
List Stack vs. Dyn. Array Stack

<table>
<thead>
<tr>
<th></th>
<th>List</th>
<th>Dyn. Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>popStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>topStack</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
List Bag

- Init, Add operations are similar to List Stack
- Contains and Remove operations are tricky
- How to patch up links after removing an element?
void removeListBag(struct ListBag *b, TYPE val) {
    // Code implementation
}

void removeListBag(struct ListBag *b, TYPE val) {
    // Code implementation
}
Remove

void removeListBag(struct ListBag *b, TYPE val) {
    struct Link *previous = b->sentinel;
    struct Link *current = b->sentinel->next;
    while (current != NULL){
        // Diagram:
        // previous
        // sentinel
        // current
        // val: 2
        // next:
        // val: 7
        // next:
        // val: 4
        // next: null
    }
}
void removeListBag(struct ListBag *b, TYPE val) {
    struct Link *previous = b->sentinel;
    struct Link *current = b->sentinel->next;
    while (current != NULL) {
        if (EQ(current->value, val)) {
            previous->next = current->next;
        }
    }
}

val: 2
previous

val: 7
current

val: 4
next: null
Remove – First Occurrence

```c
void removeListBag(struct ListBag *b, TYPE val) {
    struct Link *previous = b->sentinel;
    struct Link *current = b->sentinel->next;
    while (current != NULL) {
        if (EQ(current->value, val)) {
            previous->next = current->next;
            free(current);
            return; /*removes only the first occurrence*/
        }
        previous = current;
        current = current->next;
    }
}
```
Remove -- All Occurrences

```c
void removeListBag(struct ListBag *b, TYPE val) {
    struct Link *previous = b->sentinel;
    struct Link *current = b->sentinel->next;
    while (current != NULL) {
        if (EQ(current->value, val)) {
            previous->next = current->next;
            free(current);
            current = previous;
        }
        previous = current;
        current = current->next;
    }
}
```
When you find it

• When you find the element to be deleted, what does previous point to?

• What if the element to be deleted is at the front of the list? Does this matter?