ADT Lists

CS580u Fall ‘17

Problems with Arrays

Arrays in C/C++ are:

- inflexible
  - static sizes - once the size is allocated it cannot change
- wasteful
  - over-allocation happens ‘in case you need it’
- cumbersome
  - to insert or expand requires reallocation

Vectors only hide the unpleasant truth...

Example problem

- Suppose I have an array: [1,4,10,19,6]
- I want to insert a 7 between the 4 and the 10
  - new_size = old_array.size + 1
    allocate new_array[new_size]
    copy from old_array -> new_array adding the 7
    delete old_array

Linked Lists solve Everything

- The problem with arrays is that they must be contiguous in memory
  - So expanding them is entirely dependant of the current state of memory and cannot be dynamic
- What if we could link non-contiguous segments of memory?
  - This means insertion and deletion are less cumbersome
  - expanding or shrinking the list is just a matter of changing links

Doubly Linked List

- We can create our a list of objects we'll call **nodes**
  - A node represents one 'chunk' of memory
  - The order of the nodes is determined by the insertion order and the next 'chunk', called the ***link***, is stored in each node
    - *The link is a pointer to a node class*
- Every node (except the last node) contains the address of the next node and the previous node
  - Hence the doubly part

Nodes

- Components of a node
  - Data: stores the relevant information
  - Link: stores the address of the next node

```
struct Node {
        Data data;
        Node * next, * prev;
};
```

What goes in the Node?

- No matter what kind of data is stored in our Node, the algorithms for our CRUD operations will always be the same
  - CRUD: Create, Read, Update, Delete
  - This is true for all data structures. A dynamic array is going to work the same whether it is storing chars or ints
- We need a wrapper struct for our data so we can dynamically change our data type

Data Struct

- If we write everything to a Data struct, then we need only change the Data struct, not our data structure implementation when our data type changes
    - ```
      struct Data{
              <type> data;
      };
      ```
      ```

Templating in C

- By creating one more level of indirection, and wrapping our data in a generic struct, we can implement a crude form of templating
    - Both Java and C++ have language features that allow templating classes
    - Since C does not have this feature, we have to come up with other ways to template

List Struct

- The List class's only required instance variable is a pointer to the head of the list
    - ```
      struct List{
              Node * head, * tail;
      };
      ```
      ```

List Data Structure

- A head pointer is really the only required instance variables for your list
  - starts as null
  - The tail pointer is a convenience (nice-to-have) attribute
- It must always hold the address of the first node in the list
  - requires a check for null to determine if the list is empty

Traversal and Iteration

- Basic operations of a linked list
    - Read the list for items
    - Insert an item in the list
    - Delete an item from the list
- The means we should have the following function points in our list struct
    - <type> (*read)(List *)
    - void (*insert)(List *, <type>)
    - void (*delete)(List *, <type>)

Reading from the list

- All Linked List operations require list traversal
    - Remember: You cannot use head to traverse the list
        - ```
          if(l->head == NULL) return NULL;
          Node * current = l->head;
          while(current != NULL){
                  process(current->data);
                  current = current->next;
          }
          ```
- the list attribute *head* must always point to the first item in the list
          ```

Insert/Append

```
void insert(List * l, Data data){
	Node * new_node = newNode(data);
	l->tail->next = new_node;
	new_node->prev = l->tail;
	l->tail = new_node;
	//Why don't we need to do anything with the new_node next?
	return;
}
```

Copying a linked list

- What happens in the following code?
  - struct List new_list = old_list;
    - *Both lists point to the same nodes. why?*
- Shallow Copy
  - Shallow copy only copies the pointer addresses
- Deep Copy
  - Deep copy traverses pointers and copies values

Making Two Lists

Classwork

```
List elist, olist;
...
int i = 0;
for(Node * c = list.head;  c != NULL; c = c->next, i++){
	if(i % 2 == 1)
		olist.insert(c->data);
	else
		elist.insert(c->data);
--
List * join(List * list1, List * list2){
	list1->tail = other_list->head;
	return list1;
}
```

Circular Linked List

- A Circular Linked is a list in which last node points to the first node
- How do we know when we have finished traversing the list?
    - check if the pointer of the current node is equal to the head.
- Eliminates checks for null
    - Changes method implementation, not class definition

Delete in a Circular Linked List

```
void pop(List * l){ //removes the last node
    Node * to_delete = l->tail;
    l->tail = tail->previous; //set the new tail
    //update pointers
    l->tail->next = l->head;
    l->head->previous = l->tail;
    //remove the tail
    free(to_delete);
}
```

Linked List Algorithms

- Linked List algorithms generally require a well formed linked list.
  - That is a linked list without loops or cycles in it.
    - *If a linked list has a cycle:*
      - The malformed linked list has no end (no node ever has a null next_node pointer)
      - The malformed linked list contains two links to some node
      - Iterating through the malformed linked list will yield all nodes in the loop multiple times

Malformed Lists

- A malformed linked list with a loop causes iteration over the list to fail because the iteration will never reach the end of the list.
- Solution?
    - be able to detect that a linked list is malformed before trying an iteration.

The Two Iterator Algorithm

- Simultaneously go through the list by ones (slow iterator) and by twos (fast iterators).
    - If there is a loop the fast iterators will go around that loop twice as fast as the slow iterator.
    - The fast iterator will lap the slow iterator within a single pass through the cycle, O(n).
- Detecting a loop is detecting that the slow iterator has been lapped by the fast iterator.

Two Iterator Check

```
int hasLoop(List * l){
	Node * slowNode = l->head,
		* fastNode1 = l->head,
		* fastNode2 = l->head;
	while (slowNode &&
			fastNode1 = fastNode2.next() &&
			fastNode2 = fastNode1.next()){
		if (slowNode == fastNode1 ||
			slowNode == fastNode2)
				return true;
		slowNode = slowNode->next();
	}
	return false;
}
```

Classwork

Another design

- Can we alter our design to remove special cases?
  - Adding dummy nodes allows you to get rid of the special cases
    - *Except you have to check for the dummy nodes*
- How does this change your constructor?
  - You have to allocate space for a head node and a tail node in the constructor
    - *Pros*
      - Wasted space
    - *Cons*
      - Less complexity in the methods

Dummy Nodes

- Eliminates the need to check if the head is empty
  - No need for long while conditions
    - *if(current != null && current->next != null && current->next->next != null)*
  - always at least two items in the list
- Should reserve an invalid value for the dummy node