ADT Lists

CS240 Spring ‘19

Problems with Arrays

Arrays in C/C++ are:

- inflexible
- wasteful
- cumbersome

Example problem

Suppose I have an array: [1,4,10,19,6]

I want to insert a 7 between the 4 and the 10. What do I need to do?

C++ Solution Performance

- What is the performance of the C++ solution?
  - How can we evaluate this array insertion algorithm?
- How can we evaluate the effectiveness of any data structure or algorithm?
  - We need several evaluation tools
    - *growth rate*
- Growth rate of what? what are we measuring?

Complexity

- The efficiency of an algorithm is called its complexity
  - Memory complexity
  - Time complexity
- The complexity of an algorithm is not exact
  - Different hardware or languages affect the time and memory usage of an algorithm or data structure
- Complexity also does not tell us anything about the merits of two programs where one is "slightly faster".

Big O

- We can refer to the complexity of an algorithm using Big O notation.
- Why is it called Big O
    - Best guess is that it stands for Order of Complexity.
    - Typically, Big O describes the time required for an algorithm and the space required for a data structure.

Factors Requiring Resources

- How do we determine the memory complexity of an algorithm or data structure?
- How do we determine the time complexity?
  - Basic Operations
  - input size

Resource Example

```
int largest(int A[], int size) {
        int currlarge = 0;              // Position of largest element seen
        for (int i=1; i<size; i++)      // For each element
                if (A[currlarge] < A[i])      // if A[i] is larger
                        currlarge = i;              // remember its position
        return A[currlarge];            // Return largest
}
```

- The memory complexity would be size + 2
    - (the size of the array, plus the size of the two variables in the function)
- The time complexity is ‘size’ (repetition factor) multiplied by the if statement (the operations factor)

Expressing Resource Usage

- For a given input size of *n* we often express the time to run the algorithm as a function of *n*
    - written as $O(n)$.
    - We will always assume $O(n)$ is a non-negative value.

Calculating Time

- If ‘***c***’ = *the amount of time required to compare two integers*
  - we only count operations that happen every time
- The total time to run largest is therefore approximately ‘cn’

Time Function

- We say that function ***largest()*** has a running time expressed by the equation O(n) = cn
    - We do not care right now what the precise value of c might be.
    - We do not care about the time required to increment variable **i**
    - We do not care about the time for the actual assignment when a larger value is found
    - We do not care about the little bit of extra time taken to initialize **currlarge**.

Growth Rate

- $O(n)=cn$.
  - This equation describes the growth rate for the running time of the largest-value sequential search algorithm.
- When we discuss Growth rates, we often have several scenarios
  - best case, average case, and worst case

Constant Time Operations

- How many operations to read a value from an array?
  - 1 operation.
- c = 1 the amount of time necessary to read a value.
- Thus, the equation for this algorithm is simply $T(n)=1$
    - This is called a constant running time, or $O(1)$.

Linear Growth Rate

- If you have a set of operations, each one running in constant time, but only run once.
    - Example: Your morning routine
        - *wakeup*
        - *brush teeth*
        - *shower*
        - *make coffee*
        - *get dressed*
        - *leave house*
- 6 operations, each taking roughly the same amount of time
    - not really, but close enough
- So we have 6 constant time operations performed once per day
- A growth rate of c*n (for c any positive constant) is referred to as a *linear growth* rate
    - Also referred to as running time.
    - Our morning routine function is:
        - $T(n) = 6c$
- As the value of n grows, the running time of the algorithm grows in the same proportion.
    - Doubling the value of n roughly doubles the running time.
- So if you add another task to your morning routine, eat breakfast, the function becomes:
    - $T(n) = 7c$

What is the complexity of ‘Largest()’?

- O(n) = size * number of operations
  - size: the size of the array
  - number of operations: 2
- Big O for this code is:
  - O(n) = 2n

```
int largest(int A[], int size) {
	int currlarge = 0;
	for (int i=1; i<size; i++)
		if (A[currlarge] < A[i])
			currlarge = i;
	return currlarge;
}
```

Simplifying Growth Rates

- When working with algorithm complexity, we are concerned with rate of growth
    - We just want to know how it will grow when the input size grows
- When analyzing growth rates, we can simplify the results
    - Ignore Constants
    - Higher order term dominates

BogoSort

- Simple sorting algorithm
- Algorithm
  - ```
    bogoSort(list):
        while( listSorted(list) == FALSE )
            shuffle(list)
    ```
    ```

Evaluating Bogosort

- What is the best case scenario for Bogosort?
  - Constant Time
- How long this sort technique takes grows rapidly with each additional input
  - Average Case: O((n+1)!)
  - Worst Case: O(unbounded)

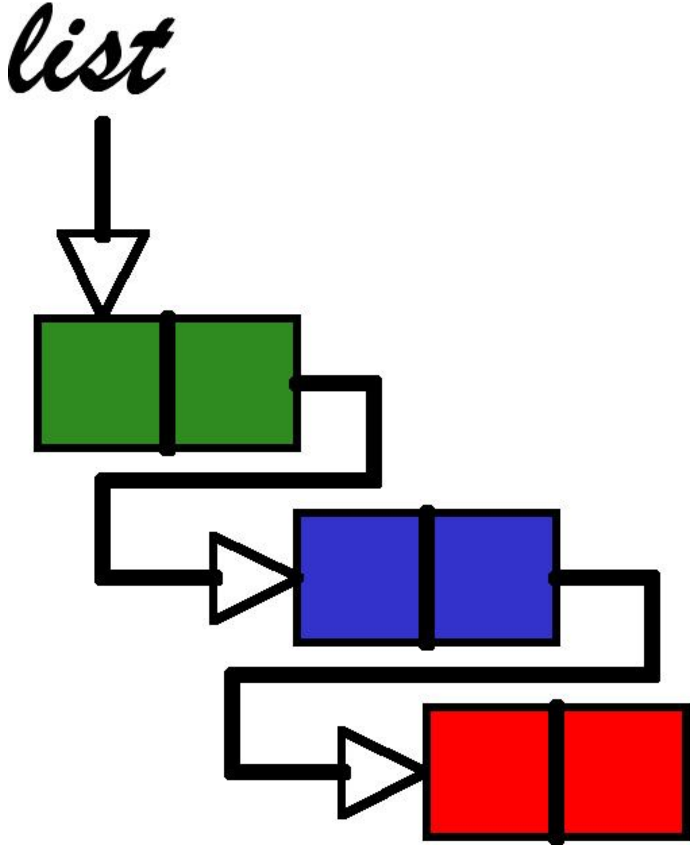
Never Use BogoSort!

Bogosort is an academic sort, used only to illustrate an idea. It is never used in any real world application.

Remember When We Were Talking About C Arrays?

- What is the Memory and Time complexity of the following solution:
  - new_size = old_array.size + 1;
    allocate new_array[new_size]
    copy from old_array -> new_array adding the 7
    delete old_array
    - *You can assume array allocation and deallocation is 1 operation*
- Time and Memory: We have to copy over every value in the array, which is O(n) for both

Classwork: Making a Train



Linked Lists

Linked Lists

- What if we need to insert and delete values frequently during runtime?
- Arrays would be very slow for these data requirements
- What if we could break up each element of the array into its own separate object?

Linked Lists solve Everything

- We would need a mechanism to go from one element to the other, but they wouldn’t have to be contiguous in memory
    - This means insertion and deletion are less cumbersome

Singly Linked List

- We can create our own list of objects
  - we’ll call them **nodes**
- The order of the **nodes** is determined by the insertion order, called the link, stored in each **node**
- Every **node** (except the last node) contains the address of the next node

Nodes

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

```
class Node {
    public:
        <type> data;
        Node * next;
};
```

List

The List class's only required instance variable is a pointer to the head of the list

```
class List{
    private:
        Node * head;
    public:
};
```

Head Node

- Only required instance variable for your list
  - initialized to null
- Holds the address of the first node in the list
  - how can we determine if the list is empty?
- Advanced versions of the linked list can have more
  - size
  - pointer to the end

Traversal and Iteration

- Basic operations of a linked list that require the list to be traversed
  - Read the list for items
  - Insert an item in the list
  - Delete an item from the list

Example of Traversal

```
Node * current = this->head;
while (current != NULL) {
    //Process current
     current = this->current->next;
}
```

What is the time complexity of traversing the linked list?

Traversal and Iteration

- How do we traverse the list?
  - You cannot use the **head** node to traverse the list
  - Create a new variable to traverse every time?
    - *Pros: encapsulated, no state*
    - *Cons: more complexity in the methods, **no state***
  - Add another instance variable of the same type as head, Node * current, to your List class
    - *Pros - can increase efficiency*
    - *Cons - more complexity in the class*

Current

- To allow reading items from the list without exposing implementation, we will need a pointer to the ‘current’ item in the list
- We could use **current** to iterate through the list instead of a temp variable
    - wrap iteration in methods

Example of Traversal with Iterator

```
this->current = this->head;
while (this->current != NULL) {
    //Process current
   this->current = this->current->next;
}
```

- What is the time complexity of traversing the linked list?

Reading from the list

- In order to read from the list we need to start at the beginning and iterate through the list with each call
  - ```
    <type> * LList::get(){
        if(head == NULL) return NULL;
        if(current == NULL) {
            current = head;
            return NULL;
        }
        <type> * temp = current->data;
        current = current->next;
        return temp;
    }
    ```
    ```

Resetting Current

- Internally, we should have a way to set **current** back to the beginning of the list

```
bool LList::reset(){
        if(head == NULL) return false;
        else current = head;
        return true;
}
```

- We should use reset anywhere we need to reset **current**
    - Prefer method calls over direct manipulation

Next

- We could also wrap the iteration itself in a method
    - ```
      bool LList::next(){
              if(current != NULL) current = current->next;
              (current == NULL) ? return false: return true;
      }
      ```
- For the same reason reset should be wrapped in a method, we should hide the implementation of our iteration
      ```

Updated get() method

```
<type> * LList::get(){
    if(this->next()) return NULL;
    return &(current->data);
}
```

- Internally, use reset() and next() to iterate through the list.
    - This is an example of a design choice that must be documented
    - The user can only iterate through the list items. **No Random Access.**

Doubly Linked List

Doubly Linked List

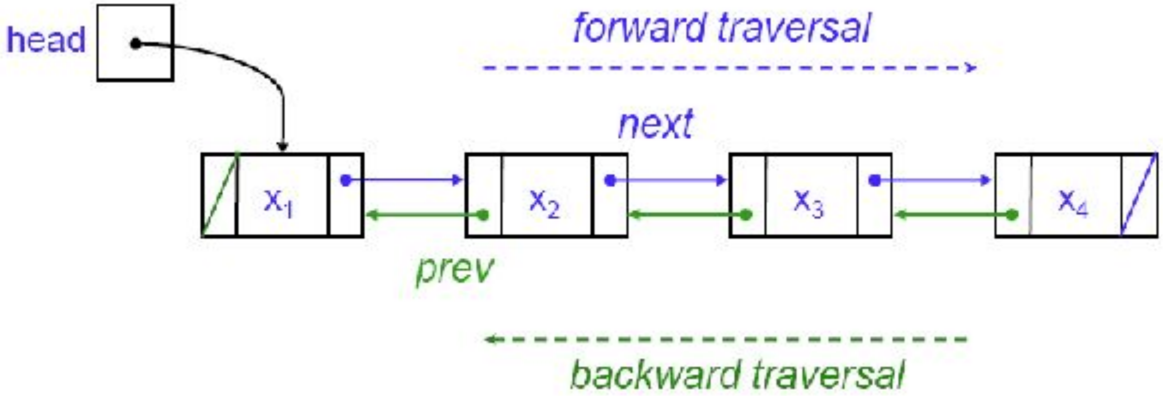
- Linked list in which every node has a next pointer and a previous pointer
    - must add previous pointer to Node class
    - Must refactor most methods to update/use the previous pointer
- A doubly linked list can be traversed in either direction
- List class must also contains a pointer to last node

Doubly Linked List Attributes

- What needs to be changed?
  - Add previous pointer
  - How does this affect the iterator, **current**
- What benefits does this provide?
- How can this be detrimental?

Changes to our CRUD

How does the extra pointer affect our CRUD operations?



Doubly Linked List Insertion

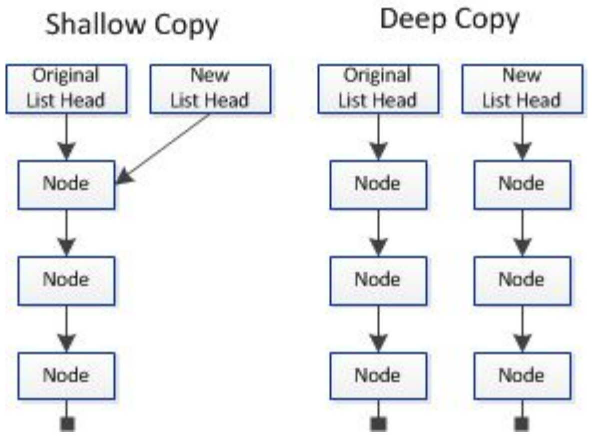
- Adds an element to the end of the list
  - ```
    bool insert(<type> data){
        Node * new_node = new Node(data);
        tail->next = new node;
        return true;
    }
    ```
- What is the Big O of appending to the doubly linked list?
    ```

Classwork

Two Lists

Shallow vs Deep Copy

- Shallow Copy
  - Copies all instance variables into a cloned object
  - DOES NOT copy pointer reference values
- Deep Copy
  - Copies all instance variables **and** referenced values into a cloned object
  - In C++ you must write the code to perform the deep copy



Copy Constructor

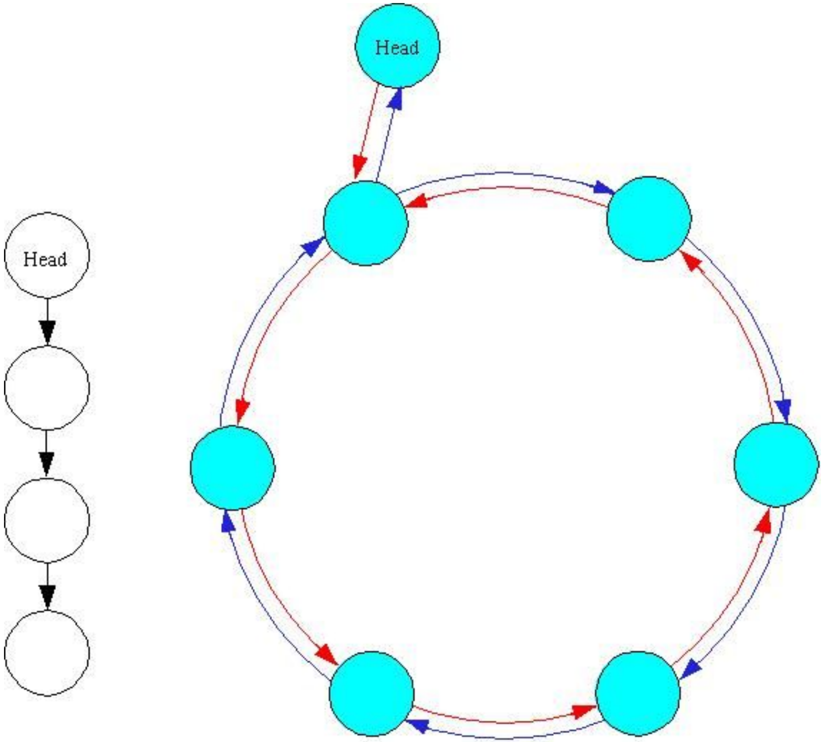
- Copy Constructors
  - <Class>(const <Class> &o){
    //deep copy object
    }
- 3 Scenarios where Copy Constructor is called:
  - Using the copy constructor explicitly
    - *Both invoke the copy constructor*
      - List copy_list(old_list);
      - List copy_list = old_list;
  - Passing parameters by value

Circularly Linked List

Circular Linked List

- Linked list in which last node points to the first node
- How do we know when we have finished traversing the list?
- Eliminates checks for null

Circular Doubly Linked List



Deletion

- Standard Deletion
  - removes an element from the end of the list

```
void remove(){
        Node * to_delete = tail;
        tail = tail->previous; //set the new tail
        //update pointers
        tail->next = head;
        head->previous = tail;
        //remove the tail
        delete to_delete;
}
```

Deletion Complexity

- What is the Big O of deleting the last item from the circularly linked list?
- What about a singly linked list?

Another design

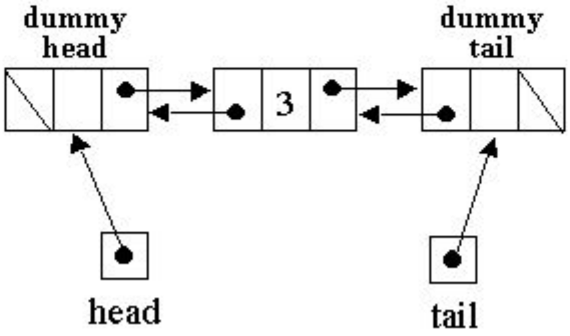
- How can we alter our design to remove special cases?
- Adding dummy nodes allows you to get rid of the special cases
  - You never have to worry about an empty list
- How does this change your constructor?
  - You have to allocate space for a head node and a tail node in the constructor

Dummy Nodes

- The dummy node is very useful as a “lag” pointer when inserting or removing nodes.
  - For instance, to remove all the nodes whose data is 0, one can write:
    - ```
      Node lag = head;
      for (Node p = lag.next; p != null; p = p.next) {
          if (p.data == 0) {
              lag.next = p.next;
              delete p;
          } else {
              lag = p;
          }
      }
      ```
      ```

Dummy Nodes

- Eliminates the need to check if the head is empty
- Eliminates long boolean conditions
    - if(current != null && current->next != null && current->next->next != null)
- How would a dummy node work for a circular doubly linked list?



Linked List Algorithms

- Linked List algorithms generally require a well formed linked list.
- If a linked list has a cycle:
  - The malformed linked list has no end
  - Iterating through the malformed linked list will yield all or some 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?

Classwork

Well-Formed Lists

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 Pseudocode

```
malFormed(List  l){
        Node slow = l->head, fast1 = l->head, fast2 = l->head;
        while (slow && fast1 = fast2->next && fast2 = fast1->next){
            if (slow == fast1 || slow == fast2)
                return true;
            slow = slow->next;
        }
        return false;
}
```