Data Structures

Topic #5

Today’s Agenda

- **Other types of linked lists**
  - discuss algorithms to manage circular and doubly linked lists
  - should we use a dummy head node? What are the advantages and disadvantages
  - what about arrays of linked lists, or linked lists of arrays?
  - evaluate the benefits/drawbacks of a doubly “threaded” list

Dynamic Linked Lists

- Wisely controlled dynamic memory can save considerable memory that may be wasted by other implementations
- We are not limited to fixed size restrictions
- Certain operations are simpler with linked structures (inserting into a linked list consists of only 2 assignment statements, once we have found the location)…no shifting!

Dynamic Linked Lists

- Of course, algorithms may be more complex, harder to read, and harder to debug than similar algorithms with statically allocated structures
- Think about program #1…how would have an “array” changed the debugging process? Would it have provided all of the necessary functionality?

Dynamic Linked Lists

- And, don’t forget that in some cases dynamic linked lists can waste memory. It is possible to store many pointers compared to the quantity of data. This pointer space must be considered as overhead, which is accentuated when the nodes contain a small amount of data (like a LLL of single characters!)

Dynamic Linked Lists

- For example, a list with a single character data member (one byte)
  - may require a 4-byte pointer as its link
  - resulting in 80% overhead (4 bytes out of 5) in each list node
- Lastly, allocating and deallocating memory at run-time is overhead and can overshadow the time saved by simple list-processing algorithms.
- ** no hard and fast rules! **

Doubly Linked Lists

- We have already discussed
  - the benefits and drawbacks of doubly linked lists in relation to various “position oriented” ADTs
  - avoids the need to manage a previous pointer when we traverse
  - think about this: when you traverse a singly linked list to remove a node -- what happens?
  - yes! two pointers must be managed (current, previous) or a look-ahead approach is used which requires 2 dereferences!

Traversing...singly LLL

- Let’s examine this further:

```
node * current=head;
node * previous= NULL;
while (current && current->data
  != match) {
  previous = current;
  current = current->next;    }
```

- Count the number of operations, fetches...

Traversing...singly LLL

- With the look ahead approach:

```
node * current=head;
if (current)
  while (current->next &&
             current->next->data != match) {
    current = current->next;   }
```

- Count the number of operations, fetches...
- Compare these two techniques

Traversing...doubly LLL

- But, with a doubly linked list, we have:

```
node * current=head;
 while (current &&
         current->data != match) {
  current = current->next;  }
```

- Count the number of operations, fetches...
- Compare this with the last two techniques

Updating...doubly LLL

- When we update the pointers for a singly linked list, we need to:

```
if (previous)
  previous->next = current->next;
else head = current->next;
```

- Versus:

```
if (current->prev) {
 current->prev->next = current->next;
else head = current->next;
```

Updating...doubly LLL

- But, this is not all…we have to update the previous pointer in the “next” node too:

```
 if (current->next)
    current->next->prev =current->prev;
```

  - anything else? (draw a picture)
  - why did we have to check if current->next?

Doubly Linked Lists

- What we should have learned from these last few slides is that
  - while doubly linked lists reduce the need to manage two pointers (or use the look ahead)
  - they do not necessarily improve our overall efficiency dramatically for normal deletion
  - instead, they add an extra pointer required for every single node
  - but they can minimize the need for traversals if used in more complicated searches

Doubly Linked Lists

- Remember with a doubly linked list,
  - there are two pointers in each node
  - a next pointer, and a previous pointer
  - the previous pointer should point to the node's immediate successor, and should be null if this is the first node
  - a node with both next and previous as null means that there is just one node in the list

Doubly Linked Lists

- Compared to a singly linked list
  - inserting and deleting nodes is a bit slower
  - this is because both the next and the previous members must be updated
  - however, updating the extra pointer in each node inserted/removed is still much faster than doing a complete list traversal to find a predecessor (or to backup 3 nodes...)

Doubly Linked Lists

- Given this, we know that insert will not be as elegant as our LLL code:

```
//add as the first node:
node * temp = head;
head = new node;
head->data = new_data;
head->prev = NULL;
head->next = temp;
//anything else?
```

Doubly Linked Lists

- Yes!

```
head->next->prev = head;
```

- Anything wrong with this? Yes!
  - if this is the first node in the list, we'd have a seg fault with the code above.

```
if (temp)  //why not if (head->next)?
  head->next->prev = head;
```

Doubly Linked Lists

- Let’s do one more. Add at the end of a doubly linked list <u>without a tail ptr:</u>
- What is wrong with this code:

```
node * current = head;
while (current)
  current= current->next;
current->next = new node;
current->next->prev = current
current->next->next = NULL;
```

Doubly Linked Lists

- We can still go “too far” with a doubly LL

```
node * current = head;
if (!current) //insert at the head
else while (current->next)
  current= current->next;
current->next = new node;
current->next->prev = current;
current->next->next = NULL;
```

  – **Any better approaches? Anything missing?**

Doubly Linked Lists

- Is the “ideal” solution to have a tail pointer instead? are there any drawbacks to a tail pointer?

```
tail->next = new node;
tail->next->prev = tail;
tail->next->next = NULL;
```

  - **every time the list is altered the tail pointer must be updated**

Traversing...Circular LLL

- How would circular linked lists compare with singly and doubly for traversal
- Do we still have to check for null? why?
- What should the stopping condition be for traversal?

```
 if (!head) //no items in list
 else while (current->data != match) {
   prev=current;
  current = current->next;
   if (current == head) break; }
```

Traversing...Circular LLL

- Why, instead couldn't we have said:

```
 else while (current != head &&current->data != match) {
   previous=current;
  current = current->next;  }
```

- or:

```
 else while (current->next != head &&current->data != match) {
   previous = current;
  current = current->next;  }
```

Circular Linked Lists

- Can we avoid having a previous pointer in our traversals with a circular linked list? No! (unless a look ahead is used)
- Count the number of operations/fetches and compare with the other approaches for today
- What about deallocating all nodes? How does that work?

Deallocating all in Circular LL

- Remember, with a circular linked list
  - it is the stopping condition that changes
  - if we check for it too soon...we won’t get anywhere!

```
//for example, this is wrong
node * current = head;
while (current != head) {...}
```

Deallocating all in Circular LL

- But, waiting to check can also be wrong:

```
//for example, this is wrong
node * current = head;
node * temp;
do {
  temp = current->next;
  delete current;
  current = temp;
} while (current != head);
```

Deallocating all in Circular LL

- The previous slide would have caused a seg fault (dereferencing a null pointer) if the list was already empty…
- By adding the following at the beginning, would we have solved this problem?

```
if (!head) return;
```

```
– yes, but there is another choice.
```

Deallocating all in Circular LL

- Is this better or worse?

```
if (!head) return;
node * current = head;
node * temp;
while (current->next != head) {
  temp = current->next;
  delete current;
  current = temp;
}  //now what needs to get done?
```

Deallocating all in Circular LL

- Yes, we have one node left…oops!

```
delete head;
head = NULL;
```

- Compare this approach with the one before, which is better and why?
- Also realize…that with both approaches
  - we had 3 pointers (head, current, temp)
  - in addition, the stopping condition requires more work than just checking for null!

Deallocating all in Circular LL

```
//An alternate approach
if (!head) return;
node * current = head->next;
head->next = NULL;     ///say what?
while (current){
  head = current->next;
  delete current;
  current = head;
}
```

Dummy Head Nodes

- Variations to singly linked lists have been designed to decrease the complexity and increase the efficiency of specific algorithms
- For many list processing algs, the first node of the list is a special case
  - we’ve seen this with inserting and deleting
  - because updating the head pointer is different from updating a next pointer of another node

Dummy Head Nodes

- The result is that many of our algorithms have the form:
  - if the node is the first node being processed
    - update the head appropriately
  - otherwise
    - process the node normally
- One way to eliminate this special case is to include a head node or list header (dummy head node)

Dummy Head Nodes

- A head node is an extra node placed at the beginning of the list
- It has the same data type as all other nodes in the list
  - but, its data member is unused
  - eliminating the special case for the first node - because every list has this empty node.

Dummy Head Nodes

- So, to insert at the end would not require a special case for if the list existed or not:

```
node * current = head;
while (current->next) //no seg fault!
  current = current->next;
current->next = new node;
current = current->next;
current->data = new_data;
current->next = NULL;
```

Dummy Head Nodes

- This means your constructor would NOT set head to null
  - in fact, there should be no situation where head is null!!

```
//constructor:
head = new node;
head->next = NULL;
```

  - problems occur with a destructor if it is ever explicitly invoked. Why?

Other Data Structures

- We are not limited to these data structures
  - why not combine what we have learned about linked lists and arrays to create list that draws off of the strengths of both?
  - if we had a linked list of arrays, where each node is an array we could dynamically grow it (no fixed size limitations), we could easily insert/remove nodes (blocks of memory), and we could directly access within an array once found

Other Data Structures

- For example, let’s manage a linked list of arrays, where each array contains 10 data items
  - we figure that even if all 10 are not used in a given node, that wasting 9-0 data “cells” is trivial
  - commonly called a “flexible array”

```
struct node {
    data fixed_array[SIZE];
    node * next;
    };
```

Other Data Structures

- So, assume that we have built a flexible array and we are interested in accessing the 15th data item (i.e., by position)

```
    node * current = head;
   int traverse = dposition/SIZE;
   while (--traverse && current) {
     current = current->next;
   if (current) cout <<
 current->fixed_array[dposition%SIZE];
```

Other Data Structures

- Discuss the benefits and drawbacks of this approach...
  - How do the number of operations/fetches compare?
  - How does the use of memory compare?
  - Are there any problems with the direct access in the previous code? will it work in all cases?
  - Could we have avoided traversal all together?

Next Time...

- Next time we will begin discussing
  - how to measure the efficiency of our algorithms in a more precise manner
- Then, we will move on and begin discussing abstractions that are value oriented instead of position oriented
  - and begin applying non-linear data structures to improve our insertion, deletion, retrieval performance

Data Structures

Programming Assignment Discussion