Data Structures

Topic #3

Today’s Agenda

- **Ordered List ADTs**
  - What are they
  - Discuss two different interpretations of an “ordered list”
  - Are manipulated by “position”
- **Use of Data Structures for Ordered Lists**
  - arrays (statically & dynamically allocated)
  - linear linked lists

Ordered Lists

- Now that we have seen a simple list example, and started to examine how we can use different data structures to solve the same problem
- We will move on to examine an **ordered list ADT**
  - implemented using a linear linked list, circular linked list, linked list of linked lists, doubly linked lists, etc.

Ordered Lists

- So, what is an ordered list?
- Immediately, a sorted list comes to mind
- But…this isn’t the only definition!
  - think of lists ordered by time,
  - grocery lists? In the order you think about it
  - to-do lists? In priority order...
- In fact, typically an <u>ordered list</u> is:
  - ordered by “position”
  - whereas a “sorted” list is a “value oriented list”, this is a “position oriented list”

Ordered Lists

- But, an ordered list has many many different interpretations
- We will discuss two:
  - absolute lists
  - relative lists
- Why? Well, there are a variety of interpretations of where data is inserted and whether or not the list has “holes”

Absolute Ordered Lists

- An absolute ordered list
  - may have holes
  - which means if the first data inserted is at position 12, there are 11 holes (1-11) prior
  - may “replace” data if inserted at the same position (another insert at position 12 could replace the previously inserted data at that position...it never shifts the data)
  - similar to “forms” such as a tax form

Relative Ordered Lists

- A relative ordered list
  - may **not** have holes
  - which means if the first data inserted is at position 12, it would actually be inserted at the first position!
  - may “shift” data if inserted at the same position (another insert at position 1 would shift what had been at position 1 -- now to be at position 2)
  - similar to “editors” such as vi

Absolute List Operations

- For an absolute ordered list, what are the operations?
  - insert, retrieve, remove, create, destroy, display
  - insert, retrieve, and remove would all require the client program to supply a position number
  - notice we are not inserting in sorted order, or retrieving by a “value”
  - instead we insert at an <u>absolute</u> position, retrieve the data at that <u>position</u>, remove data at a given <u>position</u> --- not affecting the rest of the list!

Relative List Operations

- For a relative ordered list, what are the operations? (the same!)
  - insert, retrieve, remove, create, destroy, display
  - insert, retrieve, and remove would all require the client program to supply a position number
  - instead we insert at a <u>relative position</u>, retrieve the data at that <u>position</u>, remove data at a given <u>position</u> --- this time affecting the rest of the list!
  - A remove at position 1 would require every other piece of data to shift down (logically)

Absolute/Relative Operations

- Notice what was interesting about the last two slides
- The operations for a relative and absolute list are the same
- One exception is that a relative list might also have an “append” function
- And, an absolute list might restrict insert from “replacing”, and add another function to specifically “replace”

Client Interface

- Therefore, the client interface for these two interpretations might be identical!
  - so...how would the application writer know which type of list the ADT supports?
  - Documentation! Critical whenever you implement a list
  - What does it mean to insert? Where?
  - What implications does insert and remove have on the rest of the data in the list?

Client Interface

```
class ordered_list {
  public:
      ordered_list();
      ~ordered_list();
      int insert(int, const data & );
      int retrieve(int, data &);
      int display();
      int remove(int);
```

Client Interface

- With the previous class public section
  - the constructor might be changed to have an integer argument if we were implementing this abstraction with an array
  - the int return types for each member function represent the "success/failure" situation; if more than two states are used (to represent the error-code) ints are a good choice; otherwise, select a bool return type

Data Structures

- Now let's examine various data structures for an ordered list and discuss the efficiency tradeoffs, based on:
  - run-time performance
  - memory usage
- We will examine:
  - statically/dynamically allocated arrays
  - linked lists (linear, circular, doubly)

Data Structures

- Statically Allocated array…

```
    private:
        data array[SIZE];
        int number_of_items;
```

- Absolute lists:
  - direct access (insert at pos12 : array[11] = ...
  - remove only alters one element (resetting it?)
  - problem: memory limitations (fixed size)
  - problem: must “guess” at the SIZE at compile time

Data Structures

- Relative lists: (statically allocated arrays)
  - direct access for retrieve
  - problem: searching! Insert might cause the data to be inserted somewhere other than what the position specifies (i.e., if the position # is greater than the next "open" position)
  - problem: shifting! Remove, insert alters all subsequent data
  - problem: memory limitations (fixed size)
  - problem: must "guess" at the SIZE at compile time

Data Structures

- Dynamically Allocated array…

```
   private:
         data * array;
         int number_of_items;
         int size_of_array;
```

- Absolute lists:
  - direct access (insert at pos12 : array[11] = ...
  - remove only alters one element (resetting it?)
  - problem: memory limitations (fixed size)

Data Structures

- Relative lists: (dynamically allocated arrays)
  - direct access for retrieve
  - problem: searching for the correct position for insert
  - problem: shifting with insert and remove
  - problem: memory limitations (fixed size)

Data Structures

- What this tells us is that a dynamically allocated list is better than a statically allocated list (one less problem)
  - if the cost of memory allocation for the array is manageable at run-time.
  - may not be reasonable if a large quantity of instances of an ordered_list are formed
  - is not required if the size of the data is known up-front at compile time (and is the same for each instance of the class)

Data Structures

- We also should have noticed from the previous discussion that…
  - absolute ordered list are well suited for array implementations, since they are truly direct access abstractions
  - relative ordered list are rather poorly suited for arrays, since they require that data be shifted
    - therefore, hopefully the array consists of <u>pointers to our data, so that at least when we shift we are only moving pointers rather than the actual data!!</u>

Data Structures

- Linear Linked list…

```
    private:
            node * head;
            node * tail;  //???helpful?
```

- Absolute lists: (a poor choice)
  - holes: how to deal with them? add a position number to the contents of each node....don't really allocate nodes for each hole!!!!
  - insert, retrieve, removal requires traversal
  - how can a tail pointer help? if data is entered in order by position!

Data Structures

- Relative lists: (linear linked lists)
  - no holes -- so no extra position needed in each node
  - insert, retrieve, remove requires traversal
  - a tail pointer assists if appending at the end
  - no shifting!!
- So, while we still have to "search", and the search may be more expensive than with an array -- this is greatly improved for a relative list, since there is not shifting!!

Data Structures

- Circular Linked list…

```
private:
        node * head;
        node * tail;  //???helpful?
```

- There is nothing in an ordered list that will benefit from the last node pointing to the first node
- A circular linked list will require additional code to manage, with no additional benefits

Data Structures

- Doubly Linked list…

```
    private:     (each node has a node * prev)
         node * head;
         node * tail;  //???helpful?
```

- Again, there is nothing in an ordered list that will benefit from each node having a pointer to the previous node.
- UNLESS, there were operations to backup or go forward from the “current” position. In this case a doubly linked list would be ideal

Data Structures

- What about a linked list of arrays
  - where every n items are stored in the first node, the next n items are stored in the second node, etc.
  - This still requires traversal to get to the right node, but then from there direct access can be used to insert, retrieve, remove the data
  - May be the best of both worlds for relative lists, limiting the amount of shifting to “n” items while at the same time keeping the traversal to a manageable level

Data Structures

- Are there other alternatives?
  - How about an array of linked lists?
  - How about “marking” data in a relative list as “empty” to avoid shifting with an array?!
- Given the data structures discussed, which is best and why for:
  - absolute ordered list
  - relative ordered list

Next Time...

- Now that we have applied data structures to an ordered list
- We will move on to examine stack and queue abstractions
- Again, we will examine them from the standpoint of arrays, linked list, circular linked list, linked list of linked lists, doubly linked lists, etc. beginning next time!

Data Structures

Programming Assignment Discussion