CSE 373: Lists, Stacks, Queues

book Chapter 3

Pete Morcos
University of Washington
4/3/00

http://www.cs.washington.edu/education/courses/cse373/00sp

What's a List?

- A collection of elements
- Elements are ordered, no gaps
  - Sometimes you don't really care about the ordering. A list would still be suitable, but there are other data structures to consider
- Elements are of arbitrary type, but all are the same
  - C++ templates make it easier to define multiple list types

List ADT Operations

- *Note: slightly different from book*
- List MakeEmpty(List L) / void DeleteList(List L)
  - DeleteList actually deallocates each list element
  - MakeEmpty just initializes list when newly created
- int IsEmpty(List L)
- void Insert(List L, ElementType E, Position P)
- void Remove(List L, Position P)
  - void FindAndRemove(List L, ElementType E)
- Position Find(List L, ElementType E)
- Position GetNext/GetPrev(List L, Position P)
- Position First/Kth/Last(List L)
- int Length(List L)

In C++, the first List parameter is implicit; it's the "this" pointer.

Two Implementations

- You've seen this stuff before, so fast overview
- Array-based
  - pre-allocate big array
  - keep track of first free slot
  - shift elements around on insert/remove
- Pointer-based
  - each entry carries pointer to next entry (more memory)
  - last entry points to NULL
  - main program only stores pointer to first element
  - messing with first element requires special-casing

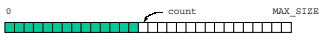
Array Implementation

```
typedef struct _ListInfo {
    ElementType *theArray; // = malloc(MAX_SIZE * sizeof(ElementType))
    int count; // = 0
    int maxSize; // = MAX_SIZE
} ListInfo;
typedef ListInfo *List;
typedef int      Position;

void Insert(List L, ElementType E, Position P) {
    if (P > count || count == MAX_SIZE) Error("insert out of range!");
    while (P <= count) {
        ElementType curEl = L->theArray[P];
        L->theArray[P++] = E;
        E = curEl;
    }
    count++;
}
```

New ListInfo structs need to be initialized as shown.

An empty list has a fully allocated array, and count = 0.



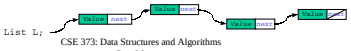
Pointer Implementation

```
typedef struct _node {
    ElementType Value;
    struct _node *next;
} node;
typedef node *List;
typedef node *Position;

// Insert() adds new node after the one pointed to by P
// (if P is NULL, or list is empty, insert at beginning)
void Insert(List *pL, ElementType E, Position P) {
    Position newCell = malloc(sizeof(node));
    newCell->Value = E;
    if (pL == NULL || P == NULL) { newCell->next = pL; pL = newCell; }
    else { newCell->next = P->next; P->next = newCell; }
}
```

An empty list has a NULL List pointer.

Note special case for inserts at head of list.



Gotchas

- When you write a line of pointer code that breaks the list, an alarm should go off in your head
  - As soon as possible, your code should fix the list up
  - Draw pictures to help see what must be done
- Boundary cases require special attention
  - Empty list
  - Single item – same item is both first and last
  - Two items – first item, last item, no others
  - Three or more items – first/last/middle items

Hassle with the Pointer Version

- Because our List points directly to the first entry in the list, any change to the first entry has to be reflected in the List variable itself.
- This means we have to change the parameter list of some functions to take a List pointer, so we can change it.
- Also need special checks in case List pointer is NULL, since L->next is invalid in that case.

A Solution

- If we add a *header node* at the beginning of all lists (even empty ones), problems go away.
- It's now always valid to reference L->next, since that refers to the header node. Thus we can use the same code for all positions in the list.
- When we start iterating through list, need to "prime the pump" by marching our pointer to current node one step forward.

List Analysis

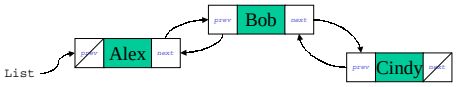
operation	*array impl.*	*pointer impl.*
MakeEmpty	O(1), ***O(max N) space***	O(1)
DeleteList	O(1)	**O(N)**
isEmpty	O(1)	O(1)
Insert	**O(N)**	O(1)
Remove	O(N)	O(N)
Find	O(N)	O(N)
GetNext	O(1)	O(1)
GetPrev	O(1)	**O(N)**
First	O(1)	O(1)
Kth	O(1)	**O(N)**
Last	O(1)	**O(N)**
Length	O(1)	**O(N)**

Tweaking the ADT

- When we look at an analysis such as the previous slide, some improvements suggest themselves.
- Two types of modification are typical
  - Enhance the ADT implementation with more information or a different organization
  - Change the ADT definition, often by restricting the semantics
- Both have costs, so the choice between basic or fancy versions is an engineering decision

Doubly Linked Lists

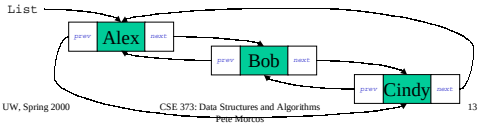
- GetPrev (and therefore Remove) is slow (O(N))in the pointer implementation
- We can't go from a node to the previous one
- Add a back-pointer to all nodes



- Costs: increase in space used (+50% if data is small), extra bookkeeping needed in list code

Circularly Linked Lists

- Make last element point to first instead of NULL
- Useful if you want to iterate through whole list starting from any element
  - Avoids need for special code to wrap around at end
- Can be combined with double linking, in which case the Last() operation gets faster



Stacks

- Array implementation is nice, but Insert and Remove require wasteful work
- What if we change the definition of the ADT as follows?
  - You can only Insert or Remove the *last* list item
- Now both ops become constant time!

Why Stacks?

- At first, looks silly – too weak of an ADT
- But, in practice this is often all we need
  - Want to remember a lot of items, but only deal with the most recent one
- Mental model is a stack of paper. You can add sheets to the top, or remove from the top.
- "LIFO" = "Last in, First out"
- Appears in many places in computer science
  - Including every time you run a program!

Stack Details

- Since this is such a restricted list, only need:
  - void push(Stack S, ElementType E)
  - ElementType pop(Stack S)
  - ElementType top(Stack S) // doesn't remove item
  - int isEmpty(Stack S)
  - Stack MakeEmpty(Stack S) / void DeleteStack(Stack S)
- Although array implementation seems natural, can use pointers as well
  - If pointer-based, probably want doubly-linked. Why?

Queues

- Having seen stacks, consider a list ADT that only inserts at one end, and removes at the *other* end
- "FIFO" = "First in, First out"
- Like standing in line at the store
- Instead of Push and Pop, we talk about Enqueue and Dequeue

Why Queues?

- Items can get "buried" in a stack and not surface for a long time
- Sometimes, we are concerned with "fairness"
  - Jobs sent to a printer
  - Applications for a contest
  - Input to a computer; mouse, keyboard, etc.

Queue Details

- Again, we can use our knowledge of lists to implement a queue
- Mixed sequences of enqueue / dequeue
- Pointer-based lists seem natural
  - What info needs to be available for a fast implementation?
- Array-based has a problem
  - Recall that enqueue/dequeue are basically same as old insert/remove
  - How to fix?