Lists

CSE 373 - Data Structures
April 5, 2002

Readings and References

• Reading
  › Sections 3.1 - 3.2.8, *Data Structures and Algorithm Analysis in C*, Weiss

• Other References

Review: Pointers and Memory

• Recall that memory is a one-dimensional array of bytes, each with an address
• Pointer variables contain an address

```c
int y, *aP, *bP; // pointer vars use * in declaration
y = 3;
aP = &y;
*aP = 17;
printf("aP: %p\n", aP);
printf("*aP = %d\n", y); // prints out what?
printf("bP: %p\n", bP);
*bP = 1; // what happens? (hint: DOOM)
```

Example result

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  int y, *aP, *bP; // pointer vars use * in declaration
  y = 3;
aP = &y;
  *aP = 17;
  printf("aP: %p", aP);
  printf("*aP = %d\n", y); // prints out what?
  printf("bP: %p", bP);
  *bP = 1; // what happens? (hint: DOOM)
  return 0;
}
```

```
Segmentation fault (core dumped)
```
Review: Memory Management

- Use “malloc” to allocate a specified number of bytes for new variables
  \[ \text{aP} = (\text{int} *) \text{malloc}(\text{sizeof(int)}); \]
  - Use the sizeof operator to compute the number of bytes needed for the data type
  - malloc does not initialize the memory
- To deallocate memory, use “free” and pass a pointer to an object allocated with malloc
  \[ \text{free(aP)}; \]

List ADT

- What is a List?
  - Ordered sequence of elements \( A_1, A_2, \ldots, A_N \)
- Elements may be of arbitrary type, but all are the same type
- Common List operations are
  - Insert, Find, Delete, IsEmpty, IsLast, FindPrevious, First, Kth, Last

List Implementations

- Two types of implementation:
  - Array-Based
  - Pointer-Based

List: Array Implementation

- Basic Idea:
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable \( \text{count} \)
  - Shift elements when you have to insert or delete

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…</th>
<th>count-1</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>( A_2 )</td>
<td>( A_3 )</td>
<td>…</td>
<td>( A_N )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
List: Array Implementation

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

//Empty list has allocated array and count = 0

Need to define: void Insert(List L, ElementType E, Position P)

// Example: Insert E at position P = 2

Array List Insert Operation

• Basic Idea: Insert new item and shift old items to the right.

void Insert(List L, ElementType e, Position p) {
    Position current;
    if (p > L->count || L->count == MAX_SIZE) exit(1);
    current = L->count;
    while (current != p) {
        L->a[current] = L->a[current-1];
        current--;
    }
    L->a[current] = e;
    L->count++;
}

Array List Insert Running Time

• Running time for N elements?
• On average, must move half the elements to make room
• Worst case is insert at position 0. Must move all N items down one position before the insert
• This is O(N) running time.

List: Pointer Implementation

• Basic Idea:
  › Allocate little blocks of memory (nodes) as elements are added to the list
  › Keep track of list by linking the nodes together
  › Change links when you want to insert or delete
List: A Pointer Implementation

typedef struct Node {
    ElementType Value;
    struct Node *next;
};
typedef struct Node *List;
typedef struct Node *Position;

// Pointer to an empty list = NULL

void Insert(List *pL, ElementType E, Position P)

// Insert adds new node after the one pointed to by P
// if P is NULL or list is empty (pL=NULL), insert at
// beginning of list

Position newItem;
newItem = (struct Node *)malloc(sizeof(struct Node));
FatalErrorMemory(newItem);
newItem->Value = E;
if (pL == NULL || P == NULL) { // insert at head of list
    newItem->next = pL;
pL = newItem;
} else { // insert newItem after the node pointed to by P
    newItem->next = P->next;
P->next = newItem;
}
Using a Header Node

- If the List pointer points to first item, then
  - any change in first item changes List itself
  - need special checks if List pointer is NULL
  - \( L->\text{next} \) is invalid (\( L \) is not a Node struct)
- Solution: Use “header node” at beginning of all lists (see text)
  - List pointer always points to header node, which points to first actual list item
  - Simplifies the code, but you need to remember that there is an "empty" node at the start of the list

Linked List with Header Node

Value Next
\[ pL \]

header node
first actual list node

Value
Next

NULL

Pointer Implementation Issues

- Whenever you break a list, your code should fix the list up as soon as possible
  - Draw pictures of the list to visualize what needs to be done
- Pay special attention to boundary conditions:
  - Empty list
  - Single item – same item is both first and last
  - Two items – first, last, but no middle items
  - Three or more items – first, last, and middle items

Pointer List Insert Running Time

- Running time for \( N \) elements?
- Insert takes constant time (\( O(1) \))
- Does not depend on input size
- Compare to array bases list which is \( O(N) \)
**Pointer-Based Linked List Delete**

To delete the node pointed to by P, need a pointer to the previous node.

---

**Doubly Linked Lists**

- FindPrev (and hence Delete) is slow because we cannot go directly to previous node.
- Solution: Keep a "previous" pointer at each node.

---

**Double Link Pros and Cons**

- Advantage
  - Delete and FindPrev are fast like Insert is
- Disadvantages:
  - More space used up (double the number of pointers at each node)
  - More book-keeping for updating the two pointers at each node

---

**Circularly Linked Lists**

- Set the pointer of the last node to first node instead of NULL
- Useful when you want to iterate through whole list starting from any node
  - No need to write special code to wrap around at the end
- Circular doubly linked lists speed up both the Delete and Last operations
Polynomial ADT

- Store and manipulate single variable polynomials with non-negative exponents
  - \(10x^3 + 4x^2 + 7\) (\(= 10x^3 + 4x^2 + 0x^1 + 7x^0\))
  - Store coefficients \(C_i\) and exponents \(i\)
- ADT operations

Polynomial Implementation

- Array Implementation: \(C[i] = C_i\)
- Problem with Array implementation
  - High-order sparse polynomials require large sparse arrays
  - E.g. \(10X^{3000} + 4X^2 + 7\) \(\rightarrow\) Waste of space and time (\(C_i\) are mostly 0s)
- Instead, use singly linked lists, sorted in decreasing order of exponents

Bucket Sort: Sorting integers

- Bucket sort: \(N\) integers in the range 0 to \(B-1\)
  - Array Count has \(B\) elements ("buckets"), initialized to 0
  - Given input integer \(i\), \(\text{Count}[i]++\)
  - After reading all \(N\) numbers go through the \(B\) buckets and read out the resulting sorted list
  - \(N\) operations to read and record the numbers plus \(B\) operations to recover the sorted numbers

Radix Sort: Sorting integers

- Radix sort = multi-pass bucket sort of integers in the range 0 to \(B^P-1\)
  - Bucket-sort from least significant to most significant "digit" (base \(B\))
  - Use linked list to store numbers that are in same bucket
  - Requires \(P*(B+N)\) operations where \(P\) is the number of passes (the number of base \(B\) digits in the largest possible input number)