CSE 326 Lecture 4: Lists and Stacks

- We will review:
  - Analysis: Searching a sorted array (from last time)
  - List ADT: Insert, Delete, Find, First, Kth, etc.
  - Array versus Linked List implementations
  - Stacks

- Focus on running time (big-oh analysis)
- Covered in Chapter 3 of the text

Analysis of a Search Algorithm

- Problem: Search for an item $X$ in a sorted array $A$. Return index of item if found, otherwise return $-1$.
- Brainstorming: What is an efficient way of doing this?

```
A  -4  -3  5   7  12  35  56  98  101  124
```

$X=101$
Binary Search

Problem: Search for an item X in a sorted array A. Return index of item if found, otherwise return –1.

Idea: Compare X with middle item A[mid], go to left half if \( X < A[mid] \) and right half if \( X > A[mid] \). Repeat.

A

| -4 | -3 | 5  | 7  | 12 | 35 | 56 | 98 | 101 | 124 |

\( A[mid] = X \)

Found!

Return Mid = 8

---

Binary Search

```java
public static int BinarySearch( int[] A, int X, int N )
{
    int Low = 0, Mid, High = N - 1;
    while( Low <= High ) {
        Mid = ( Low + High ) / 2; // Find middle of array
        if ( X > A[ Mid ] ) // Search second half of array
            Low = Mid + 1;
        else if ( X < A[ Mid ] ) // Search first half
            High = Mid - 1;
        else return Mid; // Found X!
    }
    return NOT_FOUND;
}
```
Running Time of Binary Search

Given an array A with N elements, what is the worst case running time of BinarySearch?

What is the worst case?

```c
int Low = 0, Mid, High = N - 1;
while( Low <= High ) {
    Mid = ( Low + High ) / 2; // Find middle of array
    if ( X > A[ Mid ] ) // Search second half of array
        Low = Mid + 1;
    else if ( X < A[ Mid ] ) // Search first half
        High = Mid - 1;
    else return Mid; // Found X!
}
```

Worst case is when item X is not found.

How many iterations are executed before Low > High?

```c
int Low = 0, Mid, High = N - 1;
while( Low <= High ) {
    Mid = ( Low + High ) / 2; // Find middle of array
    if ( X > A[ Mid ] ) // Search second half of array
        Low = Mid + 1;
    else if ( X < A[ Mid ] ) // Search first half
        High = Mid - 1;
    else return Mid; // Found X!
}
```
Running Time of Binary Search

- Worst case is when item X is not found.
- How many iterations are executed before Low > High?
- After first iteration: N/2 items remaining
- 2\textsuperscript{nd} iteration: (N/2)/2 = N/4 remaining
- K\textsuperscript{th} iteration: ?

Worst case: Last iteration occurs when \( N/2^K \geq 1 \) and \( N/2^{K+1} \) < 1 item remaining
\[
2^K \leq N \quad \text{and} \quad 2^{K+1} > N \quad \text{[take log of both sides]}
\]
- Number of iterations is \( K \leq \log N \) and \( K > \log N - 1 \)
- Worst case running time = \( \Theta(\log N) \)
Lists

- What is a list?
  - An ordered sequence of elements $A_1, A_2, \ldots, A_N$
- Elements may be of arbitrary type, but all are the same type
- List ADT: Common operations are:
  - Insert, Find, Delete, IsEmpty, IsLast, FindPrevious, First, Kth, Last
- Two types of implementation:
  - Array-Based
  - Linked List
- We will compare worst case running time of ADT operations

Lists: Array-Based Implementation

- Basic Idea:
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of first free slot using a variable $N$
  - Empty list has $N = 0$
  - 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>N-1</th>
<th>MAX_SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>A_2</td>
<td>A_3</td>
<td>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

- Example: Insert(List L, ElementType E, Position P)
public void insert(List L, ElementType X, Position P)
// Example: Insert X after position P = 1

Basic Idea: Shift existing elements to the right by one slot and insert new item

Running time for insert into N element array-based list = ?

Basic Idea: Shift existing elements to the right by one slot and insert new item

Running time for N elements = O(N)
⇒ Worst case is when you insert at the beginning of list – must shift all N items
Insert the value X after P:

1. Create new node containing X
2. Update Next pointers

(Go through the Java/C++ code in Chap. 3 of your text)
Lists: Linked List Implementation of Insert

Insert the value X after P:
1. Create new node containing X
2. Update Next pointers

Running Time = ?

R. Rao, CSE326

Running Time = \( \Theta(1) \)

- Insert takes constant time \( \rightarrow \) does not depend on input size \( N \)
- Comparison: Array implementation takes \( O(N) \) time

R. Rao, CSE326
Caveats with Linked List Implementation

- 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

Header Node in Linked List Implementation

- Why use a header node?
  - If List points to first item, any change in first item changes List itself
  - Need special checks if List pointer is NULL (e.g. Next is invalid)
  - Solution:
    - Use “header node” at beginning of all lists (see text)
    - List always points to header node, which points to first item
Other List Operations: Run time analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array-Based List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>FindPrev</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Delete</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
### Other List Operations: Run time analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array-Based List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>FindPrev</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Find</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>FindNext</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>First</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Kth</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Last</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Length</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Delete Operation using a Linked List

Problem: To delete the node pointed to by P, need a pointer to the previous node (= O(N))

Doubly Linked Lists

- FindPrev (and hence Delete) is O(N) because we cannot go to previous node
- Solution: Keep a back-pointer at each node
  - Doubly Linked List

- Advantages: Delete and FindPrev become O(1) operations
- Disadvantages:
  - More space (double the number of pointers at each node)
  - More book-keeping for updating 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
- A circular and doubly linked list speeds up both the Delete and Last operations (first and last nodes point to each other)
  - O(1) time for both instead of O(N)

Applications of Lists

- Polynomial ADT: store and manipulate single variable polynomials with non-negative exponents
  - E.g. $10X^3 + 4X^2 + 7 = 10X^3 + 4X^2 + 0X^1 + 7X^0$
  - Data structure: stores coefficients $C_i$ and exponents $i$
- Array Implementation: $C[i] = C_i$
- ADT operations: Input polynomials in arrays A and B
  - Addition: $C[i] = ?$
  - Multiplication: ?
Applications of Lists: Polynomials

- Polynomial ADT: store and manipulate single variable polynomials with non-negative exponents
  - E.g. $10X^3 + 4X^2 + 7 = 10X^3 + 4X^2 + 0X + 7X^0$
  - Array Implementation: $C[i] = C_i$

- ADT operations: Input polynomials in arrays $A$ and $B$
  - Multiplication: initialize $C[i] = 0$ for all $i$
    for each $i, j$ pair:

- Problem with Array implementation: Sparse polynomials
  - E.g. $10X^{3000} + 4X^2 + 7$
  - Waste of space and time ($C_i$ are mostly 0s)
  - Use **singly linked list**, sorted in decreasing order of exponents
Applications of Lists: Radix Sort

- **Bucket sort**: Sort $N$ integers $A_1, \ldots, A_N$ which are in the range 0 through $B-1$
  - Initialize array Count with $B$ slots (“buckets”) to 0's
  - Given an input integer $A_i$, Count[$A_i$]++
  - Time: $O(B+N) = O(N)$ if $B$ is $\Theta(N)$
- **Radix sort** = Bucket sort on digits of integers
  - Each digit in the range 0 through 9
  - Bucket-sort from least significant to most significant digit
  - Use linked list to store numbers that are in same bucket
  - Takes $O(P(B+N))$ time where $P =$ number of digits

Stacks

- In Array implementation of Lists
  - Insert and Delete took $O(N)$ time (need to shift elements)
- What if we avoid shifting by inserting and deleting only at the beginning of the list?
  - Both operations take $O(1)$ time!
- **Stack**: Same as list except that Insert/Delete allowed only at the beginning of the list (the top).
- “LIFO” – Last in, First out
- **Push**: Insert element at top
- **Pop**: Delete and Return top element
Next class: Queues and Trees

To do this week:
Homework no. 2 on the Web (due next Monday)
Read Chapters 3 and 4