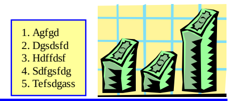
CSE 326 Lecture 4: Lists and Stacks



- 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

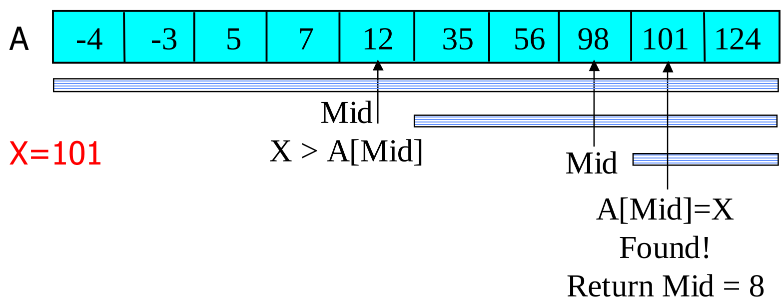
- Problem: Search for an item X in a sorted array A. Return index of item if found, otherwise return –1.
- Brainstorming: What is an efficient way of doing this?

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.



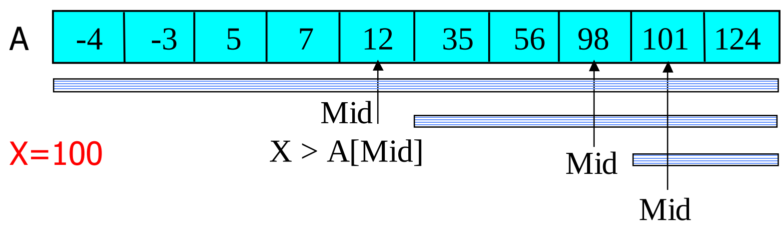
Binary Search

A										
	-4	-1	5	7	12	35	56	98	101	124

```
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?



Running Time of Binary Search

- Worst case is when item X is not found.
- How many iterations are executed before Low > High?

```
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^{nd} iteration: $(N/2)/2 = N/4$ remaining
- Kth iteration: ?

Running Time of Binary Search

- How many iterations are executed before Low > High?
- After first iteration: N/2 items remaining
- 2^{nd} iteration: $(N/2)/2 = N/4$ remaining
- Kth iteration: $N/2^K$ remaining
- Worst case: Last iteration occurs when $N/2^K \geq 1$ and $N/2^{K+1} < 1$ item remaining
  - ➩ $2^K \leq N$ and $2^{K+1} > N$ [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 A1, A2, …, AN
- 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

0	1	2	3	…	N-1		MAX_SIZE
A_1	A_2	A _ 3	A _ 4	…	A _ N		

- Example: Insert(List L, ElementType E, Position P)

Lists: Array-Based Implementation

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

0	1	2	3	...	N-1		...	MAX_SIZE
A_1	A_2	A_3	A_4	…	A_N		…	

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

0	1	2	3	...	N-1	N	...	MAX_SIZE
A_1	A_2	X	A_3	…	A_N-1	A_N	…	

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

Lists: Array-Based Insert Operation

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

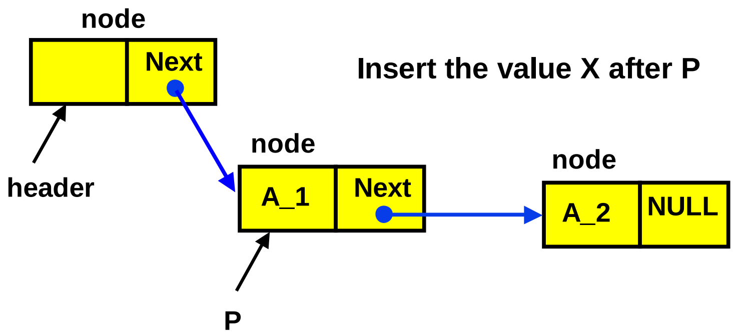
0	1	2	3	...	N-1		...	MAX_SIZE
A_1	A_2	A_3	A_4	…	A_N		…	

✦ Running time for N elements = O(N)

- ➪ Worst case is when you insert at the beginning of list – must shift all N items

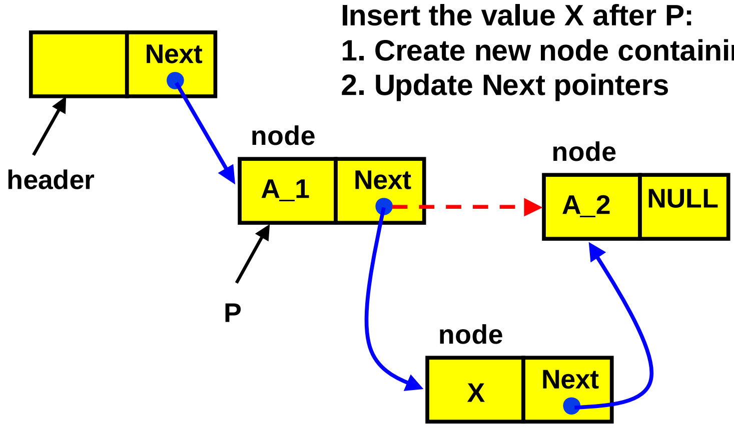
Lists: Linked List Implementation

Insert the value X after P



Lists: Linked List Implementation

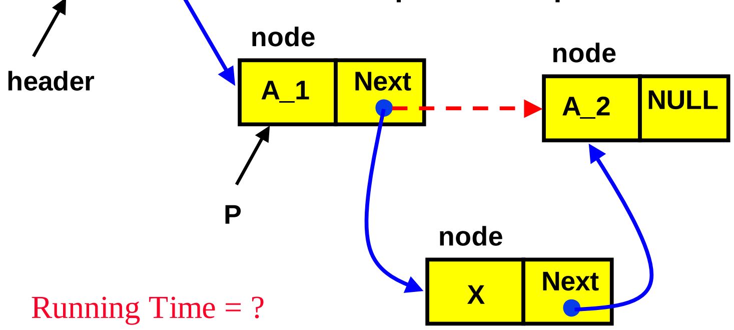
Insert the value X after P:
1. Create new node containing X
2. Update Next pointers



(Go through the Java/C++ code in Chap. 3 of your text)

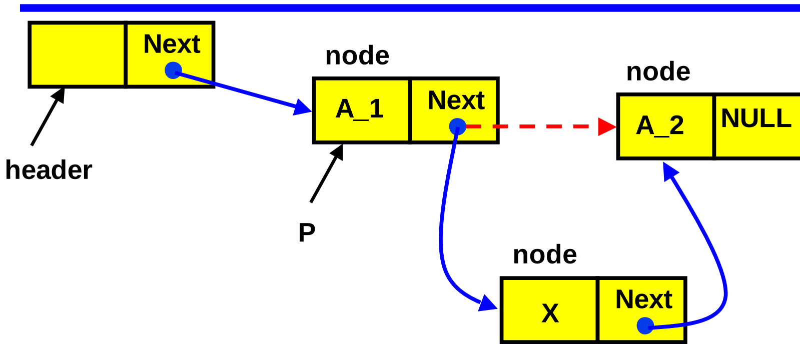
Lists: Linked List Implementation of Insert

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



Running Time = ?

Lists: Linked List Implementation of Insert



Running Time = Θ(1)

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

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

Operation	Array-Based List	Linked List
isEmpty	O(1)	O(1)
Insert	O(N)	O(1)
FindPrev	?	?
Delete	?	?

Other List Operations: Run time analysis

Operation	Array-Based List	Linked List
isEmpty	O(1)	O(1)
Insert	O(N)	O(1)
FindPrev	O(1)	O(N)
Delete	O(N)	O(N)
Find	?	?
FindNext	?	?

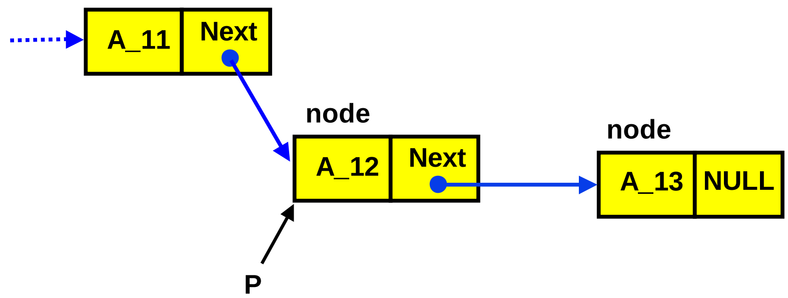
Other List Operations: Run time analysis

Operation	Array-Based List	Linked List
isEmpty	O(1)	O(1)
Insert	O(N)	O(1)
FindPrev	O(1)	O(N)
Delete	O(N)	O(N)
Find	O(N)	O(N)
FindNext	O(1)	O(1)
First	?	?
Kth	?	?
Last	?	?
Length	?	?

Other List Operations: Run time analysis

Operation	Array-Based List	Linked List
isEmpty	O(1)	O(1)
Insert	O(N)	O(1)
FindPrev	O(1)	O(N)
Delete	O(N)	O(N)
Find	O(N)	O(N)
FindNext	O(1)	O(1)
First	O(1)	O(1)
Kth	O(1)	O(N)
Last	O(1)	O(N)
Length	O(1)	O(N)

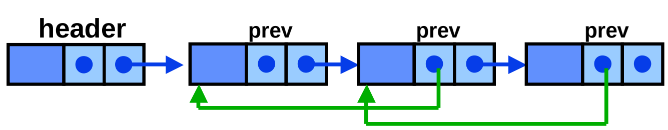
Delete Operation using a Linked List



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

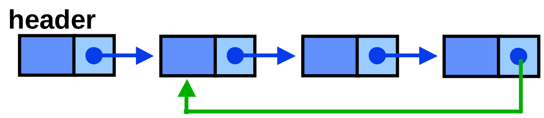
Doubly Linked Lists

- 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 <u>and</u> 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: $C[i] = C_i$
  - E.g. C[3] = 10, C[2] = 4, C[1] = 0, C[0] = 7
- 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. $10X^3 + 4X^2 + 7 = 10X^3 + 4X^2 + 0X^1 + 7X^0$
  - Array Implementation: $C[i] = C_i$
  - E.g. C[3] = 10, C[2] = 4, C[1] = 0, C[0] = 7
- ADT operations: Input polynomials in arrays A and B
  - Addition: `C[i] = A[i] + B[i];`
  - Multiplication: initialize `C[i] = 0` for all `i`

    for each i, j pair:

    `C[i+j] = C[i+j] + A[i]*B[j];`

Applications of Lists: Polynomials

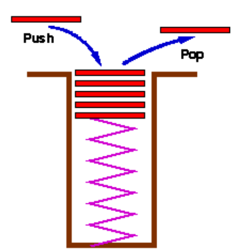
- Polynomial ADT: store and manipulate single variable polynomials with non-negative exponents
  - E.g. $10X^3 + 4X^2 + 7 = 10X^3 + 4X^2 + 0X^1 + 7X^0$
  - Array Implementation: $C[i] = C_i$
  - E.g. C[3] = 10, C[2] = 4, C[1] = 0, C[0] = 7
- **Problem with Array implementation**: Sparse polynomials
  - 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 Θ(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