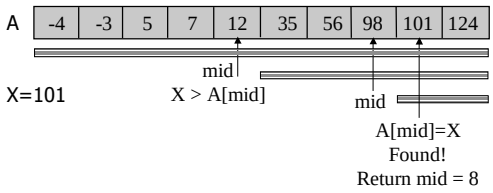
CSE 373 Lecture 4: Lists

- We will review:
  - Analysis: Binary search of a sorted array (from last time)
  - C review: Pointers and memory
  - Linked List ADT: Insert, Delete, Find, First, Kth, etc.
  - Array versus Pointer-based implementations
- Focus on running time (big-oh analysis)
- Covered in Chapter 3 of the text

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

```
int BinarySearch( const ElementType A[ ], ElementType X, int N )
{  int Low, Mid, High;
        Low = 0; High = N - 1;
        while( Low <= High )
        {   Mid = ( Low + High ) / 2; // Find middle of array index
                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 -1; }
```


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?

```
Low = 0; High = N - 1;
while( Low <= High )
    {
        Mid = ( Low + High ) / 2; // Find middle index
        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
  - Pointer-Based
- We will compare worst case running time of ADT operations

C Review: Pointers and Memory

- Recall that memory is a one-dimensional array of bytes, each with an address
- Pointer variables contain an address, instead of int/char etc.
- Examples:

```
int *pint, y, *pint1;  // pointer vars need * in declaration

y = 3;
pint = &y;
*pint = 17;
printf("%d",y);       // prints out what?

*pint1 = 1;           // what happens?
```


C Review: Pointers and Memory

- Recall that memory is a one-dimensional array of bytes, each with an address
- Pointer variables contain an address, instead of int/char etc.
- Examples:

```
int *pint, y, *pint1;  // pointer vars need * in declaration
y = 3;
pint = &y;             // assign address of y to pint
*pint = 17;            // *pint mean "contents of the address pint"
                       // puts 17 in the location pointed to by pint
printf("%d",y);        // prints out 17

*pint1 = 1;            //Error! pint1 not initialized
```


C Review: Memory Management

- Use "malloc" to allocate a specified number of bytes for new variables (use "new" in C++)
- Example: pint1 = (int *) malloc(sizeof(int));
- Use the sizeof operator to compute the number of bytes
- malloc returns the generic pointer type "void *"
  - Use the cast operation to convert to right type e.g. (int *)
- To deallocate memory, use the "free" operator ("delete" in C++) and pass a pointer to an object that was allocated with malloc
  - free(pint1);

Lists: Array-Based Implementation

- Basic Idea:
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of first free slot using a variable count
  - Shift elements when you have to insert or delete

0	1	2	3	…	count-1		MAX_SIZE
A1	A2	A3	A4	…	AN		

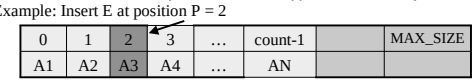
- Example: Insert(List L, ElementType E, Position P)

Lists: Array-Based Implementation

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

//Empty list has fully allocated array and count = 0

Need to define: void Insert(List L, ElementType E, Position P)
// Example: Insert E at position P = 2



0	1	2	3	…	count-1		MAX_SIZE
A1	A2	A3	A4	…	AN		

Lists: Array-Based Insert Operation

```
void Insert (List L, ElementType E, Position P) {
    ElementType PrevE;
    if (P > count || count == MAX_SIZE) Error("out of range"):
    while (P <= count) {
        PrevE = L->theArray[P]; // save prev element
        L->theArray[P++] = E; // insert E at P
        E = PrevE; // save prevE for insertion at P+1
    }
    count++; //increment location of next free slot
}
```

- Basic Idea: Insert new item and shift old items to the right.
- Running time for N elements = ?

Lists: Array-Based Insert Operation

```
void Insert (List L, ElementType E, Position P) {
    ElementType PrevE;
    if (P > count || count == MAX_SIZE) Error("out of range"):
    while (P <= count) {
        PrevE = L->theArray[P]; // save prev element
        L->theArray[P++] = E; // insert E at P
        E = PrevE; // save prevE for insertion at P+1
    }
    count++; //increment location of next free slot
}
```

- Basic Idea: Insert new item and shift old items to the right.
- Running time for N elements = O(N)
  - Worst case is when you insert at the beginning of list – must shift all N items.

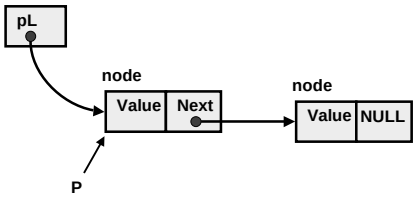
Lists: Pointer Implementation

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

// Pointer to an empty list = NULL
```

Need to define: 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

List: Pointer-Based Insert Operation



Insert the value E after P

Lists: Pointer Implementation

```
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, insert at beginning of list
Position newItem = (node *) malloc(sizeof(node));
newItem->Value = E;
If (pL ==NULL || P == NULL) { //special case: 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; }
```

Running Time = ?

Lists: Pointer Implementation

```
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, insert at beginning of list
Position newItem = (node *) malloc(sizeof(node));
newItem->Value = E;
If (pL ==NULL || P == NULL) { //special case: 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; }
```

Running Time = $\Theta(1)$

→ Insert takes constant time: does not depend on input size

→ Comparison: array implementation takes O(N) time

Caveats with Pointer 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
- Using 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: `L->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	Pointer-Based
isEmpty	O(1)	O(1)
Insert	O(N)	O(1)
FindPrev	?	?
Delete	?	?

Other List Operations: Run time analysis

Operation	Array-Based	Pointer-Based
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	Pointer-Based
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	Pointer-Based
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)

Next class:

1. Improving the performance of pointer-based lists
2. Stacks and Queues

To do this week:

Homework no. 1 (due Friday)

Read Chapters 3 and 4