Basic Types and Arrays

- **Basic Types**
  - integer, real (floating point), boolean (0, 1), character

- **Arrays**
  - \( A[0..99] \) : integer array

Records and Pointers

- **Record (also called a struct)**
  - Group data together that are related
    - \( X : \) complex pointer
      - \( \text{real}_\text{part} : \text{real} \)
      - \( \text{imaginary}_\text{part} : \text{real} \)
  - To access the fields we use “dot” notation.
    - \( X.\text{real}_\text{part} \)
    - \( X.\text{imaginary}_\text{part} \)
**Pointer**

- A pointer is a reference to a variable or record (or object in Java world).

  ```
  X : blob pointer
  \*X blob
  ```

- In C, if X is of type pointer to Y then \*X is of type Y

**Creating a Record**

- We use the “new” operator to create a record.

  ```
  P : pointer to blob;
  P := new blob;
  ```

**Simple Linked List**

- A linked list
  - Group data together in a flexible, dynamic way.
  - We’ll describe several list ADTs later.

  ```
  L : node pointer
  4 \(\rightarrow\) 9 \(\rightarrow\) 13 \(\rightarrow\) 20
  ```

**Application**

**Sparse Polynomials**

- \(10 + 4x^2 + 20x^{40} + 8x^{86}\)

  ```
  P \(\rightarrow\) 0 \(\rightarrow\) 2 \(\rightarrow\) 40 \(\rightarrow\) 86
  ```

  ```
  record poly : {
    exp : integer,
    coef : integer,
    next : poly pointer
  }
  ```

Exponents in Increasing order
**Identically Zero Polynomial**

P □ null pointer

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Addition of Polynomials**

\[10 + 4x^2 + 20x^{40} + 8x^{86}\]

<table>
<thead>
<tr>
<th>P</th>
<th>0</th>
<th>2</th>
<th>40</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[7x + 10x^2 - 8x^{86}\]

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>2</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
<td>-8</td>
<td></td>
</tr>
</tbody>
</table>

**Recursive Addition**

Add(P, Q : poly pointer): poly pointer{
R : poly pointer
    case {
        P = null : R := Q ;
        Q = null : R := P ;
        P.exp < Q.exp : R := P ;
            R.next := Add(P.next,Q);
        P.exp > Q.exp : R := Q ;
            R.next := Add(P,Q.next);
        P.exp = Q.exp : R := P ;
            R.coef := P.coef + Q.coef ;
            R.next := Add(P.next,Q.next);
    }
return R
}

**Example**

Add

<table>
<thead>
<tr>
<th>P</th>
<th>0</th>
<th>2</th>
<th>40</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>2</th>
<th>86</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
<td>-8</td>
<td></td>
</tr>
</tbody>
</table>
Example (first call)

The Recursive Call

During the Recursive Call

After the Recursive Call
Notes on Addition

- Addition is destructive, that is, the original polynomials are gone after the operation.
- We don’t salvage “garbage” nodes. Let’s talk about this.
- We don’t consider the case when the coefficients cancel. Let’s talk about this.

Unneeded nodes to Garbage

- How would you force the unneeded node to be garbage in the code on slide 11?
- Suggestions?

Memory Management – Global Allocator

- Global Allocator’s store – always get and return blocks to global allocator – an area in the memory from which we can dynamically allocate memory.
- The user (the program) must ‘free’ the memory when done.
Memory Management – Garbage Collection

- Garbage collection – run time system recovers inaccessible blocks from time-to-time. Used in Lisp, Smalltalk, Java.
  + No need to return blocks to an allocator.
  - Care must be taken to make unneeded blocks inaccessible.
  - When garbage collection kicks in there may be undesirable response time.

Solution for Polyn. Addition

```
P.exp = Q.exp : R := P ;
    R.coef := P.coef + Q.coef ;
    if R.coef = 0 then
        R := Add(P.next,Q.next);
    // The terms with coef = 0 have been removed from the // result
    else
        R.next := Add(P.next,Q.next);
    }
```

Use of Global Allocator

```
P.exp = Q.exp : R := P ;
    R.coef := P.coef + Q.coef ;
    if R.coef = 0 then
        R := Add(P.next,Q.next);
        Free(P); Free(Q);
    else
        R.next := Add(P.next,Q.next);
        Free(Q);
    }
```

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 of the same type
- Common List operations are:
  › Insert, Find, Delete, IsEmpty, IsLast, FindPrevious, First, Kth, Last, Print, etc.
Simple Examples of List Use

• Polynomials
  › 25 + 4x² + 75x⁸⁵
• Unbounded Integers
  › 4576809099383658390187457649494578
• Text
  › “This is an example of text”

Unbounded Integers Base 10

• -4572
  \[
  \begin{array}{c|c|c|c|c|c}
  10^3 & 10^2 & 10^1 & 10^0 & \text{sign} \\
  \hline
  -4 & 5 & 7 & 2 & -1 \\
  \end{array}
  \]
  \( X : \text{node pointer} \)

• 348
  \[
  \begin{array}{c|c|c|c|c|c}
  10^2 & 10^1 & 10^0 & \text{sign} \\
  \hline
  3 & 4 & 8 & 1 \\
  \end{array}
  \]
  \( Y : \text{node pointer} \)

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 `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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

---

**Insert Z in 3rd position**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>MAX_SIZE-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Z</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

---

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>

---

**Array List Insert Running Time**

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

**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: Array Implementation**

- **Basic Idea:**
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable `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>A_4</td>
<td>...</td>
<td>A_N</td>
<td></td>
</tr>
</tbody>
</table>
**Insertion After**

InsertAfter(p : node pointer, v : value_type): {
    x : node pointer;
    x := new node;
    x.value := v;
    x.next := p.next;
    p.next := x;
}

Note: cannot swap two last lines (why?)

**Linked List with Header Node**

Insertion After

- 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

Advantage: “insert after” and “delete after” can be done at the beginning of the list.
**Pointer List Insert Running Time**

- Running time for a list with N elements?
- Insert takes constant time (O(1))
- Does not depend on list size
- Compare to array based list which is O(N)

**Delete After**

```java
DeleteAfter(p : node pointer): {
    temp : node pointer;
    temp = p.next;
    p.next = temp.next; // p.next.next
    free(temp);
}
```

Note: p points to the node that comes before the deleted node!

- temp – the node to be removed.

**Linked List Delete**

To delete the node pointed to by Q, need a pointer to the previous node;
See book for findPrevious method
Doubly Linked Lists

- `findPrevious` (and hence `Delete`) is slow \([O(N)]\) because we cannot go directly to previous node
- Solution: Keep a "previous" pointer at each node

![Doubly Linked List Diagram]

Double Link Pros and Cons

- **Advantage**
  - Delete (not DeleteAfter) and `FindPrev` are faster
- **Disadvantages:**
  - More space used up (double the number of pointers at each node)
  - More book-keeping for updating the two pointers at each node (pretty negligible overhead)

Reverse a linked list

```c
Reverse(t : node pointer): node pointer {
    rev : node pointer;
    temp: node pointer;
    rev = NULL;
    while(t != NULL){
        temp = t.next;
        t.next = rev;
        rev = t;
        t = temp;
    }
    return (rev);
}
```

Implementing Pointers in Arrays

- "Cursor Implementation"
  - This is needed in languages like Fortran, Basic, and assembly language
  - Easiest when number of records is known ahead of time.
  - Each record field of a basic type is associated with an array.
  - A pointer field is an unsigned integer indicating an array index.
Idea

Pointer World

1 2 3 4 5
n nodes
data next

Nonpointer World

D  N

• D[ ] : basic type array
• N[ ] : integer array
• Pointer is an integer
• null is 0
• p.data is D[p]
• p.next is N[p]
• Free list needed for node allocation

Example of Use

n = 8
L = 4
Free = 7

Initialization

Free = n

D  N

1 2 3 4 5

means

Free

Try DeleteFront

• Define the cursor implementation of DeleteFront which removes the first member of the list when there is one.
  › Remember to add garbage to free list.

InsertFront(L : integer, x : basic type) {
  q : integer;
  if not(Free = 0) then q := Free
    else return "overflow";
  Free := N[Free];
  D[q] := x;
  N[q] := L;
  L := q;
}

DeleteFront(L : integer) {
  ????
}
DeleteFront Solution

DeleteFront(L : integer) {
    q : integer;
    if L = 0 then return "underflow"
    else {
        q := L;
        L := N[L];
        N[q] := Free;
        Free := q;
    }
}