CS 126 Lecture P3: Data Structures

Outline
(This is a hard lecture--study these slides after class.)

• Introduction
• Array
• Structure
• Linked list
• Implementation: C pointer
Why Data Structures?

<table>
<thead>
<tr>
<th>Users’ views: students, bank records, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>C basics: int, float, char, ...</td>
</tr>
<tr>
<td>Memory elements</td>
</tr>
</tbody>
</table>

- Users’ needs
  - What to do when we have a large amount of data to deal with?
  - Want to organize it in ways that are easy-to-understand
  - Want to be space-efficient
  - Want to be time-efficient

- What hardware gives us
  - Just a bunch of uniform, individually addressable storage elements

- Want to bridge the gap between the abstractions

Data Type and Data Structure

- Data TYPE
  - set of possible values for variables
  - operations on those values
  - Ex: int, float, char, ...

- Data STRUCTURE
  - collection of related values
  - mechanism for organizing information
  - Examples [stay tuned]:
    - built-in: array, struct
    - linked: linked list, binary tree
    - compound: array of structs, list of trees

- READ SEDGEWICK, SECTIONS 3.1, 3.2, 3.3
Outline

- Introduction
- **Array**
- Structure
- Linked list
- Implementation: C pointer

Array

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Index</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>h</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>l</td>
</tr>
<tr>
<td>103</td>
<td>3</td>
<td>l</td>
</tr>
<tr>
<td>104</td>
<td>4</td>
<td>o</td>
</tr>
</tbody>
</table>

array name: `word`
3rd letter: `word[2]`

(analogy: seats and students)

**Fundamental data structure**
- **HOMOGENEOUS** collection of values (all the same type)
- store values sequentially in memory
- associate INDEX with each value
- use array name and index to quickly access kth for any k
Array (cont.)

Concise and efficient method for working with large collections of data values

- Most important limitation: need to know size ahead of time

- Natural applications
  - vector, matrix
  - spreadsheet
  - string of characters
  ...

Computer memory is a huge array (array abstraction is easily implemented)

Example of array use

Symbolic manipulation of polynomials

C representation of $x^9 + 3x^5 + 7$:

```c
int a[10];
for (i = 0; i < 10; i++)
a[i] = 0;
a[0] = 7; a[5] = 3; a[9] = 1;
```

Possible memory representation of $x^9 + 3x^5 + 7$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>108</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>109</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(assumes array stored in locations 100-109)

Advantages of array use for this application:
can get to each item quickly
index carries implicit info, takes no space

Disadvantage: Uses up space for unused items

Use exponents as array indices
Store coefficients in the array
Memory address
Initialize all the histo-bins to 0.

```
i = val / 10;
h[i]++;
```

Calculate which bin; Increment that bin;

for all bins
   print right # of stars for each
Demo 1

Outline

• Introduction
• Array
• Structure
• Linked list
• Implementation: C pointer
**Structures**

**Fundamental data structure**
- **HETEROGENEOUS** collection of values (possibly different types)
- store values in **FIELDS**
- associate NAME with each field
- use struct name and field name to access value

**Built-in C mechanism**: **struct**
- Basis for building "user-defined types"

**Applications**
- database records
- linked list nodes (stay tuned)

**Ex**: C representation of C students

```c
struct student
{  char name [20]; float grade; }
struct student t, x, y;
x.name = "Bill Gates"; x.grade = 60.0;
y.name = "Steve Jobs"; y.grade = 70.0;
...
if (x.grade > y.grade) t = x; else t = y;
printf("Better student: %s ", t.name);
```

**typedef**

**User definition of type names**

- **Main use**: put type descriptions in one place (makes code more portable)

**Ex**:
```c
typedef float Grade;
typedef char name [20] Name;
struct student
{  Name name; Grade grade; }
struct student t, x, y;
```

- **Common use**: avoid typing "struct" (makes code more concise)

**Ex**:
```c
struct student
{  char name [20]; float grade; }
typedef struct student Student;
Student t, x, y;
```

**Ex**:
```c
typedef struct { int p; int q; } Rational;
float x; Rational t;
x = (1.0)*t.p/t.q
```
Outline

- Introduction
- Array
- Structure
- **Linked list**
- Implementation: C pointer

Linked List

- “Dynamic allocation”: allocate houses on demand
**Possible memory representation of** \(x^9 + 3x^5 + 7\)

- 100: 9
- 101: 1
- 102: 240
- ...
- 198: 0
- 199: 7
- 200: 000
- ...
- 240: 5
- 241: 3
- 242: 198

**Advantage:** space proportional to amount of info
**Disadvantage:** can only get to *next* item quickly

**C specification of** \(x^9 + 3x^5 + 7\): ???

**Need to know:**
- how to associate pieces of information
- how to specify links
- how to reserve memory to be used
- how to use links to access information

---

**Linked vs. Sequential allocation**

- Polynomial example illustrates tradeoffs
  - SPARSE polynomial: few terms, large exponent
    - ex: \(x^{1000000} + 5x^50000 + 1\)
  - DENSE polynomial: few nonzero coefficients
    - ex: \(x^7 + x^6 + 3x^4 + 2x^3 + 1\)

- **Huge sparse polynomial**
  - SPACE
    - huge
  - TIME
    - count terms
    - huge
    - find coefficient of \(x^k\)
    - tiny

- **Huge dense polynomial**
  - SPACE
    - huge
  - TIME
    - count terms
    - huge
    - find coefficient of \(x^k\)
    - instant

**Digression:** a few programming axioms
- know space and time costs
- there is never enough time or space
- it is easy to write programs that waste both
- you will not notice until it matters

**More examples of linked vs. sequential:**
Programs 3.5 and 3.9 in Sedgewick
Outline

- Introduction
- Array
- Structure
- Linked list

**Implementation: C pointer**
- pointers and simple variables
- pointers and arrays
- pointers and linked lists
- for each of these, understand how to
  + declare the variables involved
  + how to initialize them
  + how to use them

---

**Pointer**

<table>
<thead>
<tr>
<th>House address: pointer to the variable: &amp;x, p</th>
<th>House: name of the variable: x</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x; build a house of type int and name x</td>
<td>You: the value contained in the variable x, like 58</td>
</tr>
<tr>
<td>int *p; p can contain an address to any int-type house (decl)</td>
<td></td>
</tr>
<tr>
<td>p = &amp;x; p is now the address of house x (init)</td>
<td></td>
</tr>
<tr>
<td>x = 58; the person 58 moves into house x</td>
<td></td>
</tr>
<tr>
<td>*p = 58; the person 58 moves into the house at address p (use)</td>
<td></td>
</tr>
</tbody>
</table>

- \&x and p are equivalent (\& returns address of house)
- x and *p are equivalent (* gets to house at address)
**Pointer and Array**

- `&x[i]` and `p+i` are equivalent
- `x[i]` and `*(p+i)` are equivalent

```c
int a[100]; int *p; p = &a[0]; *p = 58;...
```

**Pointer and Linked List**

How to define a linked list type (recipe!!):

- To associate information, use structs
- To specify links, use `*` (pointers)

```c
typedef struct node
{ int coef; int exp; link next; }
struct node
{ int p; link p;
```
To reserve memory for a structure, use `malloc`

```c
p = malloc(sizeof *p);
```

`malloc` is a library function in `stdlib.h`

`sizeof` gives the number of memory words

needed for a node (`*p` is a node)

`malloc` reserves that much memory somewhere

and returns a pointer to it

To use a pointer to access information, use `(-)`

```c
p->coef = 1;
```

---

**Build the list for \(x^9 + 3x^5 + 7\)**

```c
p = malloc(sizeof *p)
p->coef = 1; p->exp = 9;
q = malloc(sizeof *p)
q->coef = 3; q->exp = 5;
r = malloc(sizeof *p)
r->coef = 7; r->exp = 0;
p->next = q; q->next = r; r->next = NULL;
```

[NULL is a special "no link" indicator]

---

**STUDY THIS CODE: Tip of the iceberg!**

---

**Demo 2**

![Diagram](image-url)
Closing

- Whew!
- Lots of material in this lecture.
- Pointers are confusing.
- Study these lecture slides.