CS 126 Lecture P4:
An Example Program

Outline

- Introduction
- Program
  - Data structures
  - Code
- Conclusions
Goals

• Gain insight of how to put together a “large” program
• Learn how to read a “large” program
• Appreciate the central role played by data structures
• Master the manipulation of linked lists (pointers)

Central Role of Data Structures

• How to choose data structure
  - Ease of programming
  - Time efficient
  - Space efficient
• Design of algorithms is largely design of data structures
  - Data structures largely determine the algorithms
Outline

- Introduction
- **Program**
  - Data structures
  - Code
- Conclusions

Represent A Single Card

Use integers 0-51 for the cards

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>H</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>26</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>27</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>28</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>29</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>17</td>
<td>30</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>31</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>32</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td>33</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>34</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>22</td>
<td>35</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>36</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>24</td>
<td>37</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>38</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>

card % 13: face value

card / 13: kind
Represent the Decks

- Why linked lists?
  - We want you to learn linked lists :)  
  - Little need for fast random access of the deck, mostly at the top and bottom of the stack
Outline

- Introduction
- **Program**
  - Data structures
  - **Code**
- Conclusions

**main()**

```c
main()
{
    int cnt;
    cnt = play(shuffle());
    if (Btop == NULL)
        printf("A wins in %d steps ", cnt);
    if (Atop == NULL)
        printf("B wins in %d steps ", cnt);
}
```

- Revisiting the concept of top-down design
- Revisit how to read code
- All your functions should be this short and readable (although the lecture notes don’t always practice this)
Goal: create a linked list of random cards

Create and shuffle the deck (algorithm)

- Hard to do efficiently without an array (!)
- Fill an array with integers in order
- Make a pass through to shuffle
  - pick up a new card
  - pick a random position among cards in hand
  - exchange new card with card at that position

Start with sorted cards

Ex: 10 cards

Start the deck with the first card

Build linked list

Create and shuffle the deck (code)

```
int randI(int i)
{
    return rand() / (RAND_MAX/i + 1);
}
link shuffle(int N)
{
    int j, k, t;
    int a[N];
    link x, deck = malloc(sizeof *deck);
    for (k = 0; k < N; k++) a[k] = k;
    for (k = 1; k < N; k++)
    {
      j = randI(k);
      t = a[k]; a[k] = a[j]; a[j] = t;
    }
    x = deck; x->card = a[0];
    for (k = 1; k < N; k++)
    {
      x->next = malloc(sizeof *x);
      x = x->next; x->card = a[k];
    }
    x->next = NULL;
    return deck;
}
```

Shuffle a linked list directly??
- put ith card in random position?
- works, but too slow for huge lists
Demo Part of `shuffle()`

Outline

- **Introduction**
- **Program**
  - Data structures
  - Code
- **Conclusions**
Deal the cards

- Function with a linked list as argument
- Makes two new linked lists for players A and B
- Sets global variables
  - Atop, Abot: links to first, last nodes of A
  - Btop, Bbot: links to first, last nodes of B

**Does not create any new nodes**

```
def deal(link d):
    Atop = d; Abot = d; d = d->next
    Btop = d; Bbot = d; d = d->next;
    while (d != NULL):
        Abot->next = d; Abot = d; d = d->next;
        if (d == NULL) break;
        Bbot->next = d; Bbot = d; d = d->next;
    Abot->next = NULL; Bbot->next = NULL;
```

- "move" one card from deck to A pile
- "move" one card from deck to B pile
- As long as the deck is not empty
- move one more from deck to A
- stop if the deck is empty
- move one more from deck to B
- end of piles are marked

---

Demo `deal()`

---
Take one card from each of the A, B piles and form a 2-card stack (Ttop, Tbot).

Put the 2-card stack at the bottom of the A pile.

Demo play()
Add the code for war

Add the following code before the test in “peace” code

```c
while (Av == Bv)
{
    for (i = 0; i < WAR; i++)
    {
        if (Atop == NULL) return cnt;
        Tbot->next = Atop; Tbot = Atop;
        Atop = Atop->next;
    }
    Av = Tbot->card % 13;
    for (i = 0; i < WAR; i++)
    {
        if (Btop == NULL) return cnt;
        Btop = Btop->next;
        Tbot->next = Btop; Tbot = Btop;
    }
    Bv = Tbot->card % 13;
    Tbot->next = NULL;
}
```

• Move a number of cards from A pile to T pile
• Peek at top of A pile

• Game STILL *never* ends:
  thousands of moves, or more

Why?

One bit of uncertainty

• Assume two cards in battles are randomly exchanged when picked up

```c
if (randI(2))
    { Ttop = Atop; Tbot = Btop; }
else { Ttop = Btop; Tbot = Atop; }
```

• Typical of simulation applications:
  proper use of randomness is vital!

• Ten typical games

B wins in 60 steps
A wins in 101 steps
B wins in 268 steps
B wins in 218 steps
B wins in 253 steps
A wins in 202 steps
A wins in 229 steps
B wins in 78 steps
B wins in 84 steps
B wins in 656 steps
Outline

• Introduction
• Program
  • Data structures
  • Code
• Conclusions

Q: "So, how long does it take?"
A: "About 10 times through the deck (254 battles)."

Q: "How do you know?"
A: "I played a million games..."

Q: "That sounds like fun!"
A: "Let's try having bigger battles..."

[change value of WAR]

100000 trials

<table>
<thead>
<tr>
<th>0</th>
<th>583</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>448</td>
</tr>
<tr>
<td>2</td>
<td>337</td>
</tr>
<tr>
<td>3</td>
<td>254</td>
</tr>
<tr>
<td>4</td>
<td>197</td>
</tr>
<tr>
<td>5</td>
<td>155</td>
</tr>
<tr>
<td>6</td>
<td>126</td>
</tr>
<tr>
<td>7</td>
<td>103</td>
</tr>
<tr>
<td>8</td>
<td>87</td>
</tr>
<tr>
<td>9</td>
<td>75</td>
</tr>
</tbody>
</table>
Stuff We Have Learned in This Lecture

- The process of constructing a “complex” program in a top-down fashion
- Reading a “complex” program to trace its top-down structure
- Judicious algorithm design starts with judicious choice of data structures
- Good examples of linked list (and pointer) manipulation
  - Draw pictures to read and write pointer codes

Problems with Simulation

- Doesn’t precisely mirror real game
- People pick up cards differently
- Separate hand, pile
  - requires much more code to handle
  - example: could have war as pile runs out
  - no real reason to simulate that part (?)
  - sort-of-shuffle pile after war?
- Tradeoff
  - convenience for implementation
  - fidelity to real game

Such tradeoffs typical in simulation
- try to identify which details matter