SEQUENCES, ARRAYS, AND LINKED LISTS

(with remarks on recursion and values vs. references)

University of Waterloo
List of Slides

1
2 List and Stack ADTs
3 Linked List Data Structure
4 An Array Version
5 List of What?
6 Implementation of Operations
7 Adding Elements to a Linked List
8 Linked List in an Array
9 Queue ADT
10 Variations on Linked Lists
11 Sparse Arrays
12 Variations on Linked Lists II
13 X-linked List
14 Converting Recursion to Iteration
15 Values vs. References
16 Summary
List and Stack ADTs

- Ordered sequences of elements

- Operations:

<table>
<thead>
<tr>
<th></th>
<th>List</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize</td>
<td>make-list</td>
<td>make-stack</td>
</tr>
<tr>
<td>test for emptiness</td>
<td>is-empty(l)</td>
<td>is-empty(s)</td>
</tr>
<tr>
<td>lookup 1st element</td>
<td>get(l,1)</td>
<td>top(s)</td>
</tr>
<tr>
<td>add 1st element $e$</td>
<td>insert(l,e,1)</td>
<td>push(s,e)</td>
</tr>
<tr>
<td>remove 1st element</td>
<td>delete(l,1)</td>
<td>pop(s)</td>
</tr>
</tbody>
</table>

⇒ a list ADT often allows looking up, adding, and removing $i$-th element adding elements “at the end”, etc.

⇒ Stack is just a special case of a list.
Linked List Data Structure

• An *inductive* data type

\[ \text{list}(\tau) = [\ ] + (\tau \times \text{ref to list}(\tau)) \]

⇒ instances of lists often written as \([a, b, c, d]\)

• A corresponding record type:

\[ \text{list} = \text{record} \]
\[ \quad \text{value}: \text{element}; \]
\[ \quad \text{next}: \ \overset{\text{list}}{\rightarrow}; \]
\[ \quad \text{end}; \]

⇒ assuming [ ] is represented by \(\bot\)

• Pictorially:

\[ \bullet \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow \bot \]
An Array Version

- An *inductive* data type

\[
\text{list}(\tau) = [\ ] + (\tau \times \text{list}(\tau))
\]

⇒ that’s just an array of \(\tau\)s!

\[
\begin{array}{cccc}
  a & b & c & d & \perp \\
\end{array}
\]

how many elements does it contain?

special “end of list” marker
additional “count” field

**Examples:**

⇒ strings (in C)
⇒ hardware stacks
  stored backwards with “extra space” for growth
List of What?

What can we store in lists/stacks?

- homogeneous lists: single type of elements
  ⇒ references to elements
  ⇒ in-lined elements
  common:

  \[
  t = \text{record} \\
  \quad \text{<many fields here>;} \\
  \quad \text{next:}^t; \\
  \quad \text{end;}
  \]

- heterogeneous lists
  ⇒ with variant records
  ⇒ with casts (wrong!)
Implementation of Operations

**IDEA:** induction on list structure

**Example:** get $n$-th element:

- using the list ADT operations

- from linked list:

  ```
  function getn(l:list,n:integer):element;
  if (l=nil) then `no n-th element’;
  else
    if (n=0) then getn:=l.value
    else getn:=getn(l.next,n-1);
  \Rightarrow O(\min(n, |l|))
  ```

- from array:

  ```
  function getn(l:list,n:integer):element;
  if (l.size>n) then getn := l.values[n];
  else `no n-th element’;
  \Rightarrow O(1)
  ```
Adding Elements to a Linked List

- to the beginning:

- to the end:

- in the middle of the list:
Linked List in an Array

**IDEA:** remember that the “memory” is just a big array
⇒ we can use an array explicitly

<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>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>value:</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next:</td>
<td>2</td>
<td>4</td>
<td>-1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

⇒ we need to remember where it starts (0)
⇒ insertions/deletions:
   what about the empty spaces?
⇒ memory management (ala `new/dispose`)
Queue ADT

An ADT that supports the following operations:

- new-queue : e
  ⇒ creates a new empty queue

- enqueue(q, e)
  ⇒ adds an element e on the end of q

- dequeue(q) : e
  ⇒ removes the first element from q and returns it

Implementations

⇒ based on an array (circular buffer)
⇒ based on a list
  here we have a problem: “enqueue” ∈ Θ(n)
Variations on Linked Lists

- **GOAL**: improve “appending to the end”
  \[ \Rightarrow \text{trade a little bit of space for lots of time} \]

- **SOLUTION**: additional \( O(1) \) space for a reference to the end of the list:

  ![Linked List Diagram]

  \[ \Rightarrow \text{enqueue drops from } \Theta(n) \text{ to } O(1) \text{ time} \]
Sparse Arrays

• how to store array (or a matrix) contains mostly 0’s?

\[ \begin{array}{c|c|c|c|c|c} & 10 & a & \bullet & 45 & b & \bullet & 70 & c & \perp \\ \end{array} \]

⇒ return 0 for “skipped” elements.

• cost of accessing element $i$?
  ⇒ linear in the size of the array
  
  we don’t know how many elements are before us
Variations on Linked Lists II

• what about computing $\sum_{i=1}^{n} a_i$?

\[
\text{sum} := 0;
\text{for } i:=1 \text{ to } n \text{ do }
\quad \text{sum} := \text{sum} + \text{getn}(a,i);
\Rightarrow O(n^2) \quad \text{— BAD!}
\]

• add extra $O(1)$ space for a reference to the “last accessed” item:

\[
\begin{array}{cccccc}
10 & a & \rightarrow & 45 & b & \rightarrow & 70 & c & \rightarrow & \downarrow \\
\end{array}
\]

\Rightarrow and modify $\text{getn}(l,i)$ as follows

start from the last accessed element if it is $\leq i$
start from the beginning otherwise

\Rightarrow improves the time complexity of the loop to $O(n)$
X-linked List

- doubly-linked list

\[ \bullet \rightarrow \quad \bot \rightarrow a \rightarrow b \rightarrow c \rightarrow \bot \]

⇒ with $O(1)$ space we can have queues with enqueue/dequeue at both ends is in $O(1)$ time

- circular list

\[ \bullet \rightarrow a \rightarrow b \rightarrow c \rightarrow \bullet \]

- circular doubly-linked list

- etc.
Converting Recursion to Iteration

Standard pattern of algorithms for lists:
⇒ recursion on the structure of the list

\[ F(l) = \begin{cases} 
  l = [] & \mapsto B \\
  l = [h|t] & \mapsto I(h, F(t)) 
\end{cases} \]

where \( B \) is what to do in the base case
\( I \) is what to do in the recursive case

Can we convert recursion into a \texttt{while} loop(s)?
⇒ use the stack ADT!
⇒ TRO optimization
Values vs. References

Consider the following code:

\[
p := \text{newstack}; \text{push}(p,1); \text{push}(p,2);
\]

\[
p1 := p; \text{push}(p1,3); \text{push}(p1,4);
p2 := p; \text{push}(p2,3); \text{push}(p2,4);
\]

Question is \( p1 \) the same as \( p2 \)?

⇒ NO!
⇒ why? \( \text{getn}(p1, i) = \text{getn}(p2, i) \)!
⇒ but changing, e.g., the 2nd element in \( p1 \)
    will make them different!

(partial) Solution: header node
⇒ makes sure everyone is “sharing” the same stack/list
Summary

- (X-)Linked Lists are probably the most common dynamic data structure in existence
  - while quite simple, it is surprising how many errors occur in processing simplest lists

- Lists/Arrays are commonly used to build other ADTs
  - Stacks
  - Queues
  - Sparse Arrays
  - ...

- Often possible to trade space for time
  - depends on (the most frequent) “access pattern”(s)