Today

- Quiz 7
- Amortized Analysis
- Intro to Binary Trees

Assignments

- HW 6 due
- HW 7 out
Resizable Arrays

Q: What does ArraySeq insert() cost when adding items to the end?

- Adding anywhere else is $O(n)$
- We’ll call this special version of insert insert-end

1. Note that a call to resize() is $\Theta(n)$

2. Most insert-end calls are $\Theta(1)$, but periodically we have a $\Theta(n)$ resize call

3. To help determine the full cost, we use the idea of “Amortized Analysis”
   - Instead of looking at just one call to insert-end
   - Average the cost of a sequence of calls
   - Gives the amortized cost, which averages out less-frequent expensive calls

4. For $m$ insert-end calls, $a_i$ the “amortized” and $t_i$ the actual costs, we have:
   \[ \sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i \]
   Since the amortized cost is the same for each call, we use it to determine cost
Example of using amortization for `insert-end`

- Assume a non-resize call to `insert-end` has unit cost
- Also assume array starts with 8 empty slots
- First resize call copies 8 values and then inserts at end

\[
\begin{array}{c}
\text{1st 8 calls} \\
1 + 1 + \ldots + 1 + 8 + 1 \\
\end{array}
\]

- For the first 9 items, we can use an amortized `insert-end` cost of 2

\[
\sum_{i=1}^{9} 2 \geq 17
\]

- Left-hand (amortized) side is 18, actual is 17
- To find the full amortized cost, need to consider for any \( n \)
(6) To simplify, assume `insert-end` starts as a 1-item array …

- If the array becomes full on an insert, then resize (not on the next insert)

- Assume resize cost (of copying) for $n$ insert calls is given by $f(n)$:

  \[
  f(1) = 1 \quad \text{1st resize (copy one elem)}
  \]
  \[
  f(2) = 1 + 2 \quad \text{2nd resize (two copy steps)}
  \]
  \[
  f(4) = 1 + 2 + 4 \quad \text{3rd resize (three copy steps)}
  \]
  \[\ldots\]
  \[
  f(n) = 1 + 2 + 4 + \cdots + \frac{n}{2} + n = 2n - 1 \quad \text{for } n = 2^k \text{ and } k \geq 0 \]

- Full cost includes the cost of adding $n$ elements to the array (not just copying):

  \[
  \sum_{i=1}^{n} t_i = f(n) + n = 3n - 1
  \]

  - Note this is an overestimate since we don’t always have to resize:
    e.g., for $n = 5$, cost is $f(4) + 5 = 12$ and not $3 \cdot 5 - 1 = 14$

- Using the definition of amortization with amortization cost $a = 3$, we get:

  \[
  \sum_{i=1}^{n} 3 \geq 3n - 1
  \]

- And so, each call to `insert-end` has (an amortized) cost of $O(1)$

  - Thus, it is (amortized) constant time even with resize!
  - We’ll use a similar idea for our hash table implementation as well

\[\text{1Note that } \sum_{i=0}^{k} 2^i = 2^{k+1} - 1\]
**Binary (Search) Trees**

**Basic Idea:**
- a tree-like structure (as opposed to list like)
- mixes linked lists and binary search

**A tree is a “Non-Linear” Data Structure**

**A list is a “linear” data structure**

```
a b c d e f g
d
b
a c
f
e g
```

**A tree is a “non-linear” data structure**

```
d
 b
/ \
 a  c
|   |
 f  e
  |  |
 g  
```
In general, a tree forms a **heirarchy** with

- zero or more “**nodes**”
- a distinguished “**root**” node (no parents)

Nodes are arranged in “parent-child” relationships

- each node has zero or more “**children**” (child nodes)
- each node has at most one “**parent**” (parent node)
- a node *without* children is a “**leaf**” node
- a node *with* children is an “**internal**” node

A collection of trees is called a “**forest**”
Parent-child relationships induce “ancestor-descendant” relationships

- The **ancestor** of a node is its parent and its parent’s ancestors
- The **descendent** of a node is its children and its children’s descendents
- The descendents of a node \( n \) lie on paths from \( n \) to leaves
- The ancestors of a node \( n \) lie on paths from the root to \( n \)

Each node can also have zero or more “**siblings**”

- Two nodes are siblings if they have the same parent