CSE 326: Data Structures

James Fogarty

Spring 2009
Logistics

- Closed Notes
- Closed Book
- We will provide scratch paper
Material Covered

• Everything we’ve read or talked about in class up to and including splay trees
Material Not Covered

• We won’t make you write syntactically correct Java code (think pseudocode)

• We won’t make you do a super hard proof (proofs okay, but no creative leaps)

• We won’t test you on the details of generics, interfaces, etc. in Java
  › But you should know the basic ideas
On Strategy

• Any time you’re taking a test, first get the points that are easiest for you

• Instructors may or may not organize questions in order of increasing difficulty, and their notion may not correspond to your own
Terminology

- **Algorithm**
  - A high level, language independent, description of a step-by-step process
- **Abstract Data Type (ADT)**
  - Mathematical description of an object with set of operations on the object. Useful building block.
- **Data structure**
  - A specific family of algorithms for implementing an abstract data type.
- **Implementation of data structure**
  - A specific implementation in a specific language
Algorithms vs Programs

• Proving correctness of an algorithm is very important
  › a well designed algorithm is guaranteed to work correctly and its performance can be estimated

• Proving correctness of a program (an implementation) is fraught with weird bugs
  › Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs
Queue ADT

- FIFO: First In First Out
- Queue operations
  - enqueue
  - dequeue
  - is_empty
Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty

E D C B A
List ADT

• Insert, Query, and Delete by *Position*

```
E   A   C   B   D
   F
```
Priority Queue ADT

1. **PQueue data**: collection of data with priority

2. **PQueue operations**
   - insert
   - deleteMin

3. **PQueue property**: for two elements in the queue, x and y, if x has a **lower** priority value than y, x will be deleted before y
The Dictionary ADT

• Data:
  › a set of (key, value) pairs

• Operations:
  › Insert (key, value)
  › Find (key)
  › Remove (key)

Dictionary ADT is also called the “Map ADT”

We will tend to emphasize the keys, don’t forget about the stored values
Proof by Induction

• **Basis Step:** The algorithm is correct for a base case or two by inspection.

• **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases.

• **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly.
Recursive algorithm for \textit{sum}

• Write a \textit{recursive} function to find the sum of the first \textbf{n} integers stored in array \textbf{v}.

\begin{verbatim}
sum(integer array v, integer n) returns integer
if n = 0 then
    sum = 0
else
    sum = nth number + sum of first n-1 numbers
return sum
\end{verbatim}
Program Correctness by Induction

• **Basis Step:**
  \[
  \text{sum}(v, 0) = 0. \checkmark
  \]

• **Inductive Hypothesis (n=k):**
  Assume \text{sum}(v, k) correctly returns sum of first k elements of v, i.e. \( v[0]+v[1]+\ldots+v[k-1]+v[k] \)

• **Inductive Step (n=k+1):**
  \text{sum}(v, n) returns \( v[k]+\text{sum}(v, k-1) = (\text{by inductive hyp.}) \)
  \[
  v[k]+(v[0]+v[1]+\ldots+v[k-1])=
  v[0]+v[1]+\ldots+v[k-1]+v[k] \checkmark
  \]
Asymptotic Analysis

Eliminate low order terms

\[ 4n + 5 \Rightarrow \]
\[ 0.5 n \log n + 2n + 7 \Rightarrow \]
\[ n^3 + 3 \ 2^n + 8n \Rightarrow \]

Eliminate coefficients

\[ 4n \Rightarrow \]
\[ 0.5 n \log n \Rightarrow \]
\[ 3 \ 2^n \Rightarrow \]
Solving Recurrence Relations

1. Determine the recurrence relation. What is/are the base case(s)?

2. “Expand” the original relation to find an equivalent general expression in terms of the number of expansions.

3. Find a closed-form expression by setting the number of expansions to a value which reduces the problem to a base case.
Asymptotic Analysis

• Asymptotic analysis looks at the *order* of the running time of the algorithm
  › A valuable tool when the input gets “large”
  › Ignores the *effects of different machines* or *different implementations* of the same algorithm
  › Intuitively, to find the asymptotic runtime, throw away constants and low-order terms
  › Linear search is $T(n) = 3n + 2 \in O(n)$
  › Binary search is $T(n) = 4 \log_2 n + 4 \in O(\log n)$
Definition of Order Notation

- **Upper bound:** \( T(n) = O(f(n)) \)
  
  Exist positive constants \( c \) and \( n' \) such that
  \[ T(n) \leq c f(n) \quad \text{for all} \quad n \geq n' \]

- **Lower bound:** \( T(n) = \Omega(g(n)) \)
  
  Exist positive constants \( c \) and \( n' \) such that
  \[ T(n) \geq c g(n) \quad \text{for all} \quad n \geq n' \]

- **Tight bound:** \( T(n) = \Theta(f(n)) \)
  
  When both hold:
  \[ T(n) = O(f(n)) \]
  \[ T(n) = \Omega(f(n)) \]
Meet the Family

- \( O(f(n)) \) is the set of all functions asymptotically less than or equal to \( f(n) \)
  - \( o(f(n)) \) is the set of all functions asymptotically strictly less than \( f(n) \)

- \( \Omega(f(n)) \) is the set of all functions asymptotically greater than or equal to \( f(n) \)
  - \( \omega(f(n)) \) is the set of all functions asymptotically strictly greater than \( f(n) \)

- \( \Theta(f(n)) \) is the set of all functions asymptotically equal to \( f(n) \)
### Big-O: Common Names

- **constant:** \( O(1) \)
- **logarithmic:** \( O(\log n) \) (\( \log_k n, \log n^2 \in O(\log n) \))
- **polylogarithmic:** \( O(\log^2 n) \)
- **linear:** \( O(n) \)
- **log-linear:** \( O(n \log n) \)
- **quadratic:** \( O(n^2) \)
- **cubic:** \( O(n^3) \)
- **polynomial:** \( O(n^k) \)  \( (k \text{ is a constant}) \)
- **exponential:** \( O(c^n) \)  \( (c \text{ is a constant } > 1) \)
Perspective: Kinds of Analysis

• Running time may depend on actual data input, not just length of input
• Distinguish
  › Worst Case
    • Your worst enemy is choosing input
  › Best Case
  › Average Case
    • Assumes some probabilistic distribution of inputs
  › Amortized
    • Average time over many operations
Circular Array Queue Data Structure

 enqueue(Object x) {
    Q[back] = x;
    back = (back + 1) % size
}

dequeue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}

How test for empty list?
How to find K-th element in the queue?
What is complexity of these operations?
Limitations of this structure?
Linked List Queue Data Structure

void enqueue(Object x) {
    if (is_empty())
        front = back = new Node(x)
    else
        back->next = new Node(x)
        back = back->next
}

bool is_empty() {
    return front == null
}

Object dequeue() {
    assert(!is_empty)
    return_data = front->data
    temp = front
    front = front->next
    delete temp
    return return_data
}
Array vs. Linked List

- Too much space, or not enough space
- Not as complex
- Could make array more robust

- Kth element accessed “easily”
- Insert requires shift

- Can grow as needed
- More memory per item in the queue
- Linked list code more complex

- Kth element access requires linear scan
Some Definitions:

A *Perfect* binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

- Height: $h$
- $2^{h+1} - 1$ nodes
- $2^h - 1$ non-leaves
- $2^h$ leaves
Heap Structure Property

- A binary heap is a complete binary tree.

**Complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

**Examples:**

![Binary Heap Examples](image-url)
Representing Complete Binary Trees in an Array

From node $i$:
- left child: $2 \times i$
- right child: $(2 \times i) + 1$
- parent: $i / 2$

Implicit (array) implementation:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Heap Order Property

**Heap order property**: For every non-root node $X$, the value in the parent of $X$ is less than (or equal to) the value in $X$.

```
not a heap
```

```
10

20

30 15

20

80

10

20

40

50

80

85

99

10

60

700
```
Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.
Insert: percolate up
DeleteMin: percolate down
BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree. Pretend it’s a heap and fix the heap-order property!
private void buildHeap() {
    for ( int i = currentSize/2; i > 0; i-- ) {
        percolateDown( i );
    }
}
Cycles to access:

- CPU
- Cache
- Memory
- Disk
A Solution: $d$-Heaps

- Each node has $d$ children
- Still representable by array
- Good choices for $d$:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block
Heap Merge Operation

- Useful operation for priority queues
  - Combining two sets of priorities
    (perhaps to balance them or because of a failure)

- Also useful to simplify heap implementation
  - Implement Insert, DeleteMin in terms of Merge
    - Insert – Create one-element heap, merge with existing
    - DeleteMin – Remove root, merge children
Heap Merge Operation

- first attempt:
  insert each element from smaller heap into larger

  runtime: $O(n \log n)$ worst, $O(n)$ average

- second attempt:
  concatenate heap arrays and run buildHeap

  runtime: $O(n)$ worst

- another approach:
  retain the existing information in the heaps

  array shifting keeps us at $O(n)$
Leftist Heaps

Idea:
Focus all heap maintenance work in one small part of the heap

Leftist heaps:
1. Most nodes are on the left
2. All the merging work is done on the right
Definition: Null Path Length

null path length (npl) of a node $x$ = the number of nodes between $x$ and a null in its subtree

OR

$npl(x) = \min$ distance to a descendant with 0 or 1 children

- $npl(\text{null}) = -1$
- $npl(\text{leaf}) = 0$
- $npl(\text{single-child node}) = 0$

Equivalent definitions:

1. $npl(x)$ is the height of largest complete subtree rooted at $x$
2. $npl(x) = 1 + \min\{npl(\text{left}(x)), npl(\text{right}(x))\}$
Leftist Heap Properties

• Heap-order property
  › parent’s priority value is $\leq$ to childrens’ priority values
  › result: minimum element is at the root

• Leftist property
  › For every node $x$, $npl(\text{left}(x)) \geq npl(\text{right}(x))$
  › result: tree is at least as “heavy” on the left as the right

Are leftist trees…
  complete? no
  balanced? no
Are These Leftist?

Every subtree of a leftist tree is leftist!
Leftist Heap Merge

• Merge two heaps by:
  › Choose smaller root as the merged root
  › Retain its left subtree (don’t touch it)
  › Merge its right subtree with the other root

• Done?
  › Need to preserve leftist property
Merge Continued

If $npl(R') > npl(L_1)$

$R' = \text{Merge}(R_1, T_2)$

runtime: $O(\log n)$
Leftest Merge Example

(special case)
Sewing Up the Example
Finally…
Skew Heaps

• “Blindly” adjusting version of leftist heaps

• Always switch left and right children of the root selected to be the merged root

• Don’t keep track of anything, don’t check anything, just always switch them
Skew Heap Merge

Only one step per iteration, with children always switched
Skew Heap Merge Example

merge

5

merge

3

merge

3

merge

5

merge

3

merge

5

merge

3
Amortized Complexity

Suppose you run $M$ times and average the running times

**Amortized complexity:**

$\max$ total # steps algorithm takes, in the worst case, for $M$ consecutive operations on inputs of size $N$, divided by $M$ (i.e., divide the max total by $M$).

If $M$ operations take total $O(M \log N)$ time in the worst case, *amortized* time per operation is $O(\log N)$.

Skew heaps have amortized complexity $O(\log N)$. 
Binomial Queue with \( n \) elements

Binomial Q with \( n \) elements has a unique structural representation in terms of binomial trees!

Write \( n \) in binary: \( n = 1101 \) (base 2) = 13 (base 10)

<table>
<thead>
<tr>
<th>height ((h))</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of elements</td>
<td>(2^3 = 8)</td>
<td>(2^2 = 4)</td>
<td>(2^1 = 2)</td>
<td>(2^0 = 1)</td>
</tr>
</tbody>
</table>
Properties of Binomial Queue

- At most one binomial tree of any height.

- \( n \) nodes
  \[ \Rightarrow \] # of bits in binary representation:
  \[ \Rightarrow \] number of trees:
  \[ \Rightarrow \] deepest tree has height:

- Is each subtree a binomial queue?

  Yes, as are the children of every node in the queue.
Merge Algorithm

• Just like binary addition

• Assume binomial queues X and Y are composed of forests of binomial trees $X_0,\ldots,X_k$ and $Y_0,\ldots,Y_k$
  › $X_i$ and $Y_i$ are of type $B_i$ or null

```plaintext
C_0 := null; //initial carry is null/
for i = 0 to k do
  combine $X_i,Y_i$, and $C_i$ to form $Z_i$ and new $C_{i+1}$
Z_{k+1} := C_{k+1}
```
Example 3.
Correct Result
Binary Trees

- Binary tree is
  - a root
  - left subtree (*maybe empty*)
  - right subtree (*maybe empty*)

- Representation:

<table>
<thead>
<tr>
<th>Data</th>
<th>left pointer</th>
<th>right pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>
Binary Tree: Representation

Keys required, but also store the value here
Binary Tree: Special Cases

Complete Tree

Perfect Tree

Full Tree
More Recursive Tree Calculations: Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

Three types:

- **Pre-order:** Root, left subtree, right subtree
- **In-order:** Left subtree, root, right subtree
- **Post-order:** Left subtree, right subtree, root

(an expression tree)
Binary Search Tree Data Structure

• Structural property
  › each node has \( \leq 2 \) children
  › result:
    • storage is small
    • operations are simple
    • average depth is small

• Order property
  › all keys in left subtree smaller than root’s key
  › all keys in right subtree larger than root’s key
  › result: easy to find any given key

• What must I know about what I store?
  Comparison, equality testing
Find in BST, Recursive

Node Find(Object key, Node root) {
    if (root == NULL) {
        return NULL;
    }
    if (key < root.key) {
        return Find(key, root.left);
    } else if (key > root.key) {
        return Find(key, root.right);
    } else {
        return root;
    }
}
Insert in BST

- Insert(13)
- Insert(8)
- Insert(31)

Insertions happen only at the leaves – easy!
Deletion

• Removing an item disrupts the tree structure.
• Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
• Three cases:
  › node has no children (leaf node)
  › node has one child
  › node has two children
Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:
• \( succ \) from right subtree: \( \text{findMin}(t.\text{right}) \)
• \( pred \) from left subtree: \( \text{findMax}(t.\text{left}) \)

Now delete the original node containing \( succ \) or \( pred \)
• Leaf or one child case – easy!
Balanced BST

Observation

• BST: the shallower the better!
• For a BST with \( n \) nodes
  › Average height is \( O(\log n) \)
  › Worst case height is \( O(n) \)
• Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a Balance Condition that
1. ensures depth is \( O(\log n) \) – strong enough!
2. is easy to maintain – not too strong!
The AVL Tree Data Structure

Structural properties

1. Binary tree property
   (0, 1, or 2 children)

2. Heights of left and right subtrees of every node differ by at most 1

Result:

Worst case depth of any node is: $O(\log n)$

Ordering property

- Same as for BST
An AVL Tree

Track height at all times.
AVL trees: find, insert

• **AVL find:**
  › same as BST find.

• **AVL insert:**
  › same as BST insert, except may need to “fix” the AVL tree after inserting new value.
Single rotation in general

$X < b < Y < a < Z$

Height of tree before?  Height of tree after?  Effect on Ancestors?
Double rotation in general

\[ h \geq 0 \]

\[ a \]

\[ b \]

\[ c \]

\[ Z \]

\[ W \]

\[ X \]

\[ Y \]

\[ h \]

\[ h - 1 \]

\[ W < b < X < c < Y < a < Z \]

Height of tree before?  Height of tree after?  Effect on Ancestors?
Let $x$ be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of $x$.
2. right subtree of the left child of $x$.
3. left subtree of the right child of $x$.
4. right subtree of the right child of $x$.

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.
Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:
   case #1: Perform single rotation and exit
   case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!
AVL complexity

What is the worst case complexity of a find?

\[ O(\log n) \]

What is the worst case complexity of an insert?

\[ O(\log n) \]

What is the worst case complexity of `buildTree`?

\[ O(n \log n) \]
Splay Trees

• Blind adjusting version of AVL trees
  › Why worry about balances? Just rotate like crazy!
  › Don’t track anything, store anything, just do it!

• *Amortized* time per operations is $O(\log n)$

• Worst case time per operation is $O(n)$
  › But guaranteed to happen rarely

• Splay Trees : AVL Trees :: Skew Heaps : Leftist Heaps
Recall: Amortized Complexity

If a sequence of M operations takes $O(M \cdot f(n))$ time, we say the amortized runtime is $O(f(n))$.

- Worst case time per operation can still be large, say $O(n)$

- Worst case time for any sequence of M operations is $O(M \cdot f(n))$

Average time per operation for any sequence is $O(f(n))$

Amortized complexity is worst-case guarantee over sequences of operations.
The Splay Tree Idea

If you’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!

All the way to the root!
Find/Insert in Splay Trees

1. **Find** or **insert** a node $k$

2. **Splay** $k$ to the root using three operations:
   - zig-zag rotation
   - zig-zig rotation
   - plain old zig rotation

   Depending on path from current location to the root

Why could this be good??

1. Helps the new root, $k$
   - Great if $k$ is accessed again

2. And helps many others!
   - Great if many others on the path are accessed
Splay: Zig-Zag*

Just like an…

AVL double rotation

Which nodes improve depth?

helps those in blue

Hurts those in red

k and its original children
Splay: Zig-Zig*  

Is this just two AVL single rotations in a row?

Not quite – we rotate g and p, then p and k

Why does this help?

Same number of nodes helped as hurt. k and its children benefit.
Special Case for Root: Zig

Relative depth of $p$, $Y$, $Z$?
Down 1 level

Relative depth of everyone else?
Nodes under $X$ have been repeatedly raised

Why not drop zig-zig and zig all the way?
Zig only helps one child!
Splay Operations: Remove

Everything else splayed, so we’d better do that for remove

Now what?
Join

Join(L, R):
given two trees such that (stuff in L) < (stuff in R), merge them:

Splay on the maximum element in L, then attach R

Similar to BST delete – find max = find element with no right child

Does this work to join any two trees? No, need L < R
Delete Example

Delete(4)

Delete(4)

find(4)

find(max)

find(max)