Logistics

• Closed Notes
• Closed Book
• Open Mind
• Four Function Calculator Allowed
Material Covered

• Everything we’ve talked/read in class up to and including B-trees
Material Not Covered

• We won’t make you write syntactically correct Java code (pseudocode okay)
• We won’t make you do a super hard proof
• We won’t test you on the details of generics, interfaces, etc. in Java
  › But you should know the basic ideas
Terminology

• Abstract Data Type (ADT)
  › Mathematical description of an object with set of operations on the object. Useful building block.

• Algorithm
  › A high level, language independent, description of a step-by-step process

• 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
First Example: Queue ADT

- FIFO: First In First Out
- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty

G → enqueue → F E D C B → dequeue → A
Second Example: Stack ADT

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

The Dictionary ADT is also called the “Map ADT”
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 \texttt{n} integers stored in array \texttt{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
  \[ v[k] + \text{sum}(v, k-1) = (\text{by inductive hyp.}) \]
  \[ v[k] + (v[0] + v[1] + \ldots + v[k-1]) = \]
  \[ v[0] + v[1] + \ldots + v[k-1] + v[k] \ \checkmark \]
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 $T(n) = 3n + 2 \in \mathcal{O}(n)$
  - Binary search is $T(n) = 4 \log_2 n + 4 \in \mathcal{O}(\log n)$
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 \( f(n) \)
Definition of Order Notation

• Upper bound: \( T(n) = O(f(n)) \) Big-O
  Exist positive constants \( c \) and \( n' \) such that
  \( T(n) \leq c f(n) \) for all \( n \geq n' \)

• Lower bound: \( T(n) = \Omega(g(n)) \) Omega
  Exist positive constants \( c \) and \( n' \) such that
  \( T(n) \geq c g(n) \) for all \( n \geq n' \)

• Tight bound: \( T(n) = \Theta(f(n)) \) Theta
  When both hold:
  \( T(n) = O(f(n)) \)
  \( T(n) = \Omega(f(n)) \)
Big-O: Common Names

› constant: \( O(1) \)
› logarithmic: \( O(\log n) \) \((\log_k n, \log n^2 \in O(\log n))\)
› 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

```java
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
}
```
Brief interlude: 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

![Diagram of a perfect binary tree with labels](image)
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:**
Representing Complete Binary Trees in an Array

From node $i$:

left child:
right child:
parent:

implicit (array) implementation:

<table>
<thead>
<tr>
<th></th>
<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></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.

[Diagram of a heap and a non-heap tree]
Heap Operations

- findMin:
- insert(val): 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!
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
New Heap Operation: Merge

Given two heaps, merge them into one heap

› first attempt: insert each element of the smaller heap into the larger.

   \textit{runtime}: \\

› second attempt: concatenate binary heaps’ arrays and run \texttt{buildHeap}.

   \textit{runtime}: 
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

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

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

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?
  balanced?
Operations on Leftist Heaps

- **merge** with two trees of total size \( n \): \( O(\log n) \)
- **insert** with heap size \( n \): \( O(\log n) \)
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node

- **deleteMin** with heap size \( n \): \( O(\log n) \)
  - remove and return root
  - merge left and right subtrees
Skew Heaps

Problems with leftist heaps

› extra storage for npl
› extra complexity/logic to maintain and check npl
› right side is “often” heavy and requires a switch

Solution: skew heaps

› “blindly” adjusting version of leftist heaps
› merge *always* switches children when fixing right path

› amortized time for: merge, insert, deleteMin = $O(\log n)$
› however, worst case time for all three = $O(n)$
Runtime Analysis: Worst-case and Amortized

- No worst case guarantee on right path length!
- All operations rely on merge

\[ \Rightarrow \text{worst case complexity of all ops} = \]

- Probably won’t get to amortized analysis in this course, but see Chapter 11 if curious.
- Result: $M$ merges take time $M \log n$

\[ \Rightarrow \text{amortized complexity of all ops} = \]
Binomial Queue with \( n \) elements

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

Write \( n \) in binary:  
\[
\begin{align*}  
1 & \quad \text{B}_3 \\
1 & \quad \text{B}_2 \\
\text{No} & \quad \text{B}_1 \\
1 & \quad \text{B}_0 
\end{align*}
\]

\[ n = 1101 \quad \text{(base 2)} = 13 \quad \text{(base 10)} \]
Properties of Binomial Queue

• At most one binomial tree of any height

• \( n \) nodes \( \Rightarrow \) binary representation is of size \( ? \)
  \( \Rightarrow \) deepest tree has height \( ? \)
  \( \Rightarrow \) number of trees is \( ? \)

Define: \( \text{height(forest } F \text{)} = \max_{\text{tree } T \text{ in } F} \{ \text{height}(T) \} \)

Binomial Q with \( n \) nodes has height \( \Theta(\log n) \)
Merging Two Binomial Queues

Essentially like adding two binary numbers!

1. Combine the two forests
2. For $k$ from 0 to maxheight {
   a. $m \leftarrow$ total number of $B_k$'s in the two BQs
   b. if $m=0$: continue;
   c. if $m=1$: continue;
   d. if $m=2$: combine the two $B_k$'s to form $B_{k+1}$
   e. if $m=3$: retain one $B_k$ and combine the other two to form a $B_{k+1}$
}

Claim: When this process ends, the forest has at most one tree of any height
deleteMin: Example

find and delete smallest root

merge BQ (without the shaded part) and BQ'
Binary Trees

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

- Representation:

<table>
<thead>
<tr>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>right</td>
</tr>
<tr>
<td>pointer</td>
<td>pointer</td>
</tr>
</tbody>
</table>
Binary Tree: Representation

```
        A
       / \  
      B   C
     / \  / \  
    D   E F   
```

Diagram:
```
A
/  
B   C
   /   
  D     F
```

```
D
   /  
  E     
      /  
     F
```
Binary Tree: Special Cases

- **Complete Tree**
- **Perfect Tree**
- **Full Tree**
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

(an expression tree)
Binary Tree: Some Numbers!

For binary tree of height $h$:

- max # of leaves: $2^h$, for perfect tree
- max # of nodes: $2^{h+1} - 1$, for perfect tree
- min # of leaves: 1, for “list” tree
- min # of nodes: $h+1$, for “list” tree

Average Depth for N nodes?
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

```java
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;
}
```

Runtime:
Insert in BST

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

Insertions happen only at the leaves – easy!

Runtime:
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: findMin(t.right)
- **pred** from left subtree: findMax(t.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: $O(\log n)$

This is an AVL tree
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

$h \geq 0$

$W < b < X < c < Y < a < Z$

Height of tree before? Height of tree after? Effect on Ancestors?
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!
Splay Trees

- Blind adjusting version of AVL trees
  › Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is $O(\log n)$
- Worst case time per operation is $O(n)$
  › But guaranteed to happen rarely

*Insert/Find always rotate node to the root!*

*SAT/GRE Analogy question:*
AVL is to Splay trees as ___________ is to ___________

Leftish heap : Skew heap
Recall: Amortized Complexity
If a sequence of \( M \) operations takes \( O(M f(n)) \) time, we say the amortized runtime is \( O(f(n)) \).

- Worst case time \textit{per operation} can still be large, say \( O(n) \)
- Worst case time for \textit{any sequence} of \( M \) operations is \( O(M f(n)) \)

Average time \textit{per operation} for \textit{any sequence} is \( O(f(n)) \)

Amortized complexity is \textit{worst-case} guarantee over \textit{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!
Find/Insert in Splay Trees

1. **Find** or **insert** a node $k$
2. **Splay** $k$ **to the root using:**
   - zig-zag, zig-zig, or plain old zig rotation

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

Helps those in blue
Hurts those in red

Which nodes improve depth?

\[ k \text{ and its original children} \]
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. But later rotations help the whole subtree.
Special Case for Root: Zig

Relative depth of $p$, $Y$, $Z$? Relative depth of everyone else?

- Down 1 level
- Much better

Why not drop zig-zig and just zig all the way?

- Zig only helps one child!
CPU (has registers)

Cache

Time to access (conservative)
1 ns per instruction

Cache
2-10 ns

Main Memory
40-100 ns

Disk
a few milliseconds
(5-10 Million ns)

SRAM
8KB - 4MB

DRAM
up to 10GB

many GB
Solution: B-Trees

- specialized $M$-ary search trees

- Each **node** has (up to) $M$-1 keys:

  - subtree between two keys $x$ and $y$ contains leaves with values $v$ such that $x \leq v < y$

- Pick branching factor $M$ such that each node takes one full {page, block} of memory
B-Tree: Example

B-Tree with $M = 4$ (# pointers in internal node) and $L = 4$ (# data items in leaf)

Note: All leaves at the same depth!

Data objects, that I’ll ignore in slides
B-Tree Properties ‡

› Data is stored at the leaves
› All leaves are at the same depth and contains between $\left\lfloor \frac{L}{2} \right\rfloor$ and $L$ data items
› Internal nodes store up to $M-1$ keys
› Internal nodes have between $\left\lfloor \frac{M}{2} \right\rfloor$ and $M$ children
› Root (special case) has between 2 and $M$ children (or root could be a leaf)

‡These are technically B⁺-Trees
Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with \( L+1 \) items, overflow!
   › Split the leaf into two nodes:
     • original with \( \lceil (L+1)/2 \rceil \) items
     • new one with \( \lfloor (L+1)/2 \rfloor \) items
   › Add the new child to the parent
   › If the parent ends up with \( M+1 \) items, overflow!

3. If an internal node ends up with \( M+1 \) items, overflow!
   › Split the node into two nodes:
     • original with \( \lceil (M+1)/2 \rceil \) items
     • new one with \( \lfloor (M+1)/2 \rfloor \) items
   › Add the new child to the parent
   › If the parent ends up with \( M+1 \) items, overflow!

4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!
Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\left\lceil L/2 \right\rceil$ items, underflow!
   › Adopt data from a sibling; update the parent
   › If adopting won’t work, delete node and merge with neighbor
   › If the parent ends up with fewer than $\left\lceil M/2 \right\rceil$ items, underflow!
3. If an internal node ends up with fewer than \( \lceil \frac{M}{2} \rceil \) items, **underflow**!
   - Adopt from a neighbor; update the parent
   - If adoption won’t work, merge with neighbor
   - If the parent ends up with fewer than \( \lceil \frac{M}{2} \rceil \) items, **underflow**!

4. If the root ends up with only one child, make the child the new root of the tree

This reduces the height of the tree!