GETTING YOUR DREAM TECH INTERNSHIP

MARCH 29TH 6PM-7:30PM
TROTTIER 0070

COME LEARN HOW TO APPLY TO COMPANIES, PREPARE FOR INTERVIEWS, AND WHAT COURSES TO TAKE! PRESENTED BY SUCCESSFUL INTERNS:

Lucille Hua
CS U3
Airbnb,
Google, Sony Ericsson

Michael Ho
Soft. Eng. U3
Facebook,
Apple,
Microsoft, Yahoo

Kevin Luk
CS/Bio U3
Knewton, SAP, CIHR

Pizza and swag provided by

Google Canada

CSUS
LIFE AFTER UNDERGRAD

ACADEMIA AND CAREERS IN SCIENCE

WEDNESDAY MARCH 23
7 - 9 PM AT LEACOCK 14

Unsure of your future with your BSc? Advice, anecdotes and guidance from professors, graduate students, and industry professionals who’ve “been there, done that.”

SPEAKERS

Dr. Kenneth J. Ragan
Professor
McGill Department of Physics

Dr. Daniel Bernard
Professor
McGill Department of Pharmacology

Bogdan Istrate
Full Stack Java Developer
TickSmith

Victoria Mallet
Product Manager
Ananda Microfluidics

Arjuna Rajakumar
Graduate Student
Abouheif Lab
2) **Write** *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case. **Explain** why this is its running time. I don’t care what it does. I only care about its running time…

```plaintext
WhatEver(int n)

FOR i=1 TO n
    FOR j=1 TO n
        x=n; WHILE x>1 DO { x=x/2; y=n;
            WHILE y>1 DO y=y/2 }
```

Comment about input size…
2) Write *any* algorithm that runs in time $\Theta(n^2 \log^2 n)$ in worse case. Explain why this is its running time. I don’t care what it does. I only care about its running time…

```java
WhatEver(int[] A)

n = A.length;
FOR i=1 TO n
    FOR j=1 TO n
        x=n; WHILE x>1 DO { x=x/2; y=n;
            WHILE y>1 DO y=y/2 }
```
16 configurations with 0 neighbours

HI    IQ
R
16 configurations with 38 neighbours
Mercury Course Evaluations

Course evaluations matter. Evaluate your courses and instructors!

Default period:
March 21 - May 1

Condensed period:
March 21 - April 17

Click HERE to complete your course evaluations.
Winter 2016
COMP-250: Introduction to Computer Science
Lecture 19, March 22, 2016
**SEARCHING**

- the dictionary ADT
- binary search trees
The Dictionary ADT

- a dictionary is an abstract model of a database
- like a priority queue, a dictionary stores key-element pairs
- the main operation supported by a dictionary is searching by key
- simple container methods:
  - `size()`
  - `isEmpty()`
  - `elements()`
The Dictionary ADT

- query methods:
  - findElement($k$)
  - findAllElements($k$)

- update methods:
  - insertItem($k, e$)
  - removeElement($k$)
  - removeAllElements($k$)

- special element
  - NO_SUCH_KEY, returned by an unsuccessful search
Implementing a Dictionary with a Sequence

• *unordered sequence*

![unordered sequence diagram]

- searching and removing takes $O(n)$ time
- inserting takes $O(1)$ time
- applications to log files (frequent insertions, rare searches and removals)

• *array-based ordered sequence* (assumes keys can be ordered)

![array-based ordered sequence diagram]

- searching takes $O(\log n)$ time (*binary search*)
- inserting and removing takes $O(n)$ time
- application to look-up tables (frequent searches, rare insertions and removals)
A binary search tree is a binary tree $T$ such that
- each internal node stores an item $(k, e)$ of a dictionary.
- keys stored at nodes in the left subtree of $v$ are less than or equal to $k$.
- keys stored at nodes in the right subtree of $v$ are greater than or equal to $k$.
- external nodes do not hold elements but serve as place holders.
Search

• A binary search tree $T$ is a \textit{decision tree}, where the question asked at an internal node $v$ is whether the search key $k$ is less than, equal to, or greater than the key stored at $v$.

\textbf{Algorithm TreeSearch}(k, v):

\textbf{Input}: A search key $k$ and a node $v$ of a binary search tree $T$.
\textbf{Output}: A node $w$ of the subtree $T(v)$ of $T$ rooted at $v$.

\begin{itemize}
  \item \textbf{if} $v$ is an external node \textbf{then}
    \begin{itemize}
      \item return $v$
    \end{itemize}
  \item \textbf{if} $k = \text{key}(v)$ \textbf{then}
    \begin{itemize}
      \item return $v$
    \end{itemize}
  \item \textbf{else if} $k < \text{key}(v)$ \textbf{then}
    \begin{itemize}
      \item return TreeSearch($k$, $T$.leftChild($v$))
    \end{itemize}
  \item \textbf{else}
    \begin{itemize}
      \item $\{ k > \text{key}(v) \}$
    \end{itemize}
    \begin{itemize}
      \item return TreeSearch($k$, $T$.rightChild($v$))
    \end{itemize}
\end{itemize}
Search Example I

• Successful `findElement(76)`

• A successful search traverses a path starting at the root and ending at an internal node
Search Example II

• Unsuccessful `findElement(25)`

• An unsuccessful search traverses a path starting at the root and ending at an external node
Insertion

- To perform \texttt{insertItem}(k, e), let \( w \) be the node returned by \texttt{TreeSearch}(k, T\.root())

- If \( w \) is external, we know that \( k \) is not stored in \( T \). We call \texttt{expandExternal}(w) on \( T \) and store \((k, e)\) in \( w \)
Insertion II

- If \( w \) is internal, we know another item with key \( k \) is stored at \( w \). We call the algorithm recursively starting at \( T.\text{rightChild}(w) \) or \( T.\text{leftChild}(w) \).
Removal I

- We locate the node $w$ where the key is stored with algorithm TreeSearch
- If $w$ has an external child $z$, we remove $w$ and $z$ with removeAboveExternal($z$)
Removal II

- If $w$ has no external children:
  - find the internal node $y$ following $w$ in inorder
  - move the item at $y$ into $w$
  - perform `removeAboveExternal(x)`, where $x$ is the left child of $y$ (guaranteed to be external)
Time Complexity

- A search, insertion, or removal, visits the nodes along a root-to-leaf path, plus possibly the siblings of such nodes
- Time $O(1)$ is spent at each node
- The running time of each operation is $O(h)$, where $h$ is the height of the tree
- The height of binary search tree is in $n$ in the worst case, where a binary search tree looks like a sorted sequence

![Binary Search Tree Diagram]

- To achieve good running time, we need to keep the tree balanced, i.e., with $O(\log n)$ height
HEAPS I

• Heaps
• Properties
• Insertion and Deletion
Heaps

- A *heap* is a binary tree $T$ that stores a collection of keys (or key-element pairs) at its internal nodes and that satisfies two additional properties:
  - **Order Property**: $\text{key(parent)} \leq \text{key(child)}$
  - **Structural Property**: all levels are full, except the last one, which is left-filled (*complete binary tree*)
Not Heaps

- bottom level is not left-filled
Not Heaps

- key(parent) > key(child)
A heap $T$ storing $n$ keys has height $h = \lceil \log(n + 1) \rceil$, which is $O(\log n)$.

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$

\[
\begin{align*}
0 & \quad 1 \\
1 & \quad 5 \quad 6 \\
h - 2 & \quad 15 \quad 9 \quad 7 \quad 20 \\
h - 1 & \quad 16 \\
h & \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square
\end{align*}
\]
The height of a heap $T$ storing $n$ keys has height $h = \lceil \log(n+1) \rceil$, which is $O(\log n)$.

- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$
- $2^{h-1} \leq n \leq 2^h - 1$
- Taking logs, we get $\log(n+1) \leq h \leq \log n + 1$
- Which implies $h = \lceil \log(n+1) \rceil$
Heap Insertion

So here we go ...

The key to insert is 6
Heap Insertion

Add the key in the *next available position* in the heap.

Now begin *Upheap*. 
Upheap

• *Swap parent-child keys out of order*
Upheap

• Swap parent-child keys out of order
Upheap Continues
Upheap Continues
• *Upheap* terminates when new key is greater than the key of its parent or the top of the heap is reached

• (total #swaps) \( \leq (h - 1) \), which is \( O(\log n) \)
Removal From a Heap

RemoveMin()

- The removal of the top key leaves a hole
- We need to fix the heap
- First, replace the hole with the last key in the heap
- Then, begin Downheap
• The removal of the top key leaves a hole
• We need to fix the heap
• First, replace the hole with the last key in the heap
• Then, begin *Downheap*
Downheap

Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.
Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.
Downheap Continues
Downheap Continues

```
4
   / \
20   10
   /   /
21  22 28
   /     /  
13 19   25
   /     /    
25 25   25
```

Heap I

Downheap Continues

```
6
   / \
4   7
   /   /
20  21 8
   /     /
22 28 13
   /     /  
19 19 19
   /     /    
25 25 25
```

Heap I

Downheap Continues
Downheap Continues
Downheap Continues
End of Downheap

• *Downheap* terminates when the key is greater than the keys of both its children or the bottom of the heap is reached.

• (total #swaps) ≤ (h – 1), which is O(log n)
HEAPS II

- Implementation
- HeapSort
- Bottom-Up Heap Construction
- Locators
Implementation of a Heap

public class HeapPriorityQueue implements PriorityQueue
{
    BinaryTree T;
    Position last;
    Comparator comparator;
    ...
}

Heap diagram: (4,C) -> (5,A) -> (15,K) -> (16,X) 
(6,Z) -> (9,F) -> (14,E) 
(7,Q) -> (12,H) 
(20,B) -> (8,W) 

Heaps II 6.2
Implementation of a Heap (cont.)

- Two ways to find the insertion position $z$ in a heap:
Vector Based Implementation

- Updates in the underlying tree occur only at the “last element”

- A heap can be represented by a vector, where the node at rank $i$ has
  - left child at rank $2i$ and
  - right child at rank $2i + 1$

- The leaves do no need to be explicitly stored

- Insertion and removals into/from the heap correspond to `insertLast` and `removeLast` on the vector, respectively
Heap Sort

- All heap methods run in logarithmic time or better

- If we implement PriorityQueueSort using a heap for our priority queue, `insertItem` and `removeMin` each take $O(\log k)$, $k$ being the number of elements in the heap at a given time.

- We always have at most $n$ elements in the heap, so the worst case time complexity of these methods is $O(\log n)$.

- Thus each phase takes $O(n \log n)$ time, so the algorithm runs in $O(n \log n)$ time also.

- This sort is known as **heap-sort**.

- The $O(n \log n)$ run time of heap-sort is much better than the $O(n^2)$ run time of selection and insertion sort.

**In-Place Heap-Sort**

- Do not use an external heap

- Embed the heap into the sequence, using the vector representation
Bottom-Up Heap Construction

• build \((n + 1)/2\) trivial one-element heaps

• now build three-element heaps on top of them
Bottom-Up Heap Construction

- *downheap* to preserve the order property

- now form seven-element heaps
Bottom-Up Heap Construction (cont.)
Bottom-Up Heap Construction (cont.)

The End
Analysis of Bottom-Up Heap Construction

- **Proposition**: Bottom-up heap construction with \( n \) keys takes \( O(n) \) time.
  - Insert \((n + 1)/2\) nodes
  - Insert \((n + 1)/4\) nodes and downheap them
  - Insert \((n + 1)/8\) nodes and downheap them
  - ...
  - visual analysis:

\[ \begin{array}{c}
4 \\
6 \\
15 \\
9 \\
5 \\
16 \\
25 \\
14 \\
12 \\
11 \\
7 \\
8 \\
23 \\
20 \\
27 \\
\end{array} \]

- \( n \) inserts, \( n/2 \) upheaps with total \( O(n) \) running time
Locators

- Locators can be used to keep track of elements as they are moved around inside a container.

- A **locator** sticks with a specific element, even if that element changes positions in the container.

- The locator ADT supports the following fundamental methods:
  - `element()`: return the element of the item associated with the locator.
  - `key()`: return the key of the item associated with the locator.

- Using locators, we define additional methods for the priority queue ADT
  - `insert(k,e)`: insert \((k,e)\) into \(P\) and return its locator
  - `min()`: return the locator of an element with the smallest key
  - `remove(l)`: remove the element with locator \(l\)

- In the stock trading application, we return a locator when an order is placed. The locator allows to specify unambiguously an order when a cancellation is requested.
Positions and Locators

• At this point, you may be wondering what the difference is between locators and positions, and why we need to distinguish between them.

• It’s true that they have very similar methods

• The difference is in their primary usage

• Positions abstract the specific implementation of accessors to elements (indices vs. nodes).

• Positions are defined relatively to each other (e.g., previous-next, parent-child)

• Locators keep track of where elements are stored. In the implementation of an ADT with locators, a locator typically holds the current position of the element.

• Locators associate elements with their keys
Locators and Positions at Work

- For example, consider the CS16 Valet Parking Service (started by the TA staff because they had too much free time on their hands).

- When they began their business, Andy and Devin decided to create a data structure to keep track of where exactly the cars were.

- Andy suggested having a position represent what parking space the car was in.

- However, Devin knew that the TAs were driving the customers’ cars around campus and would not always park them back into the same spot.

- So they decided to install a locator (a wireless tracking device) in each car. Each locator had a unique code, which was written on the claim check.

- When a customer demanded her car, the HTAs activated the locator. The horn of the car would honk and the lights would flash.

- If the car was parked, Andy and Devin would know where to retrieve it in the lot.

- Otherwise, the TA driving the car knew it was time to bring it back.
Winter 2016
COMP-250: Introduction to Computer Science
Lecture 19, March 22, 2016