Heap: (The data structure, not the memory area)
- A binary tree with properties:
  - 1. Parent nodes are greater than child nodes.
     o “the heap order”
  - 2. Nodes are inserted so that tree is complete.
     o “Complete” tree:
       ▪ every level, except possibly the last, is completely filled, and all nodes are as far left as possible

Insert:
- 1. Item is inserted at next location to maintain shape
   o Tree must be complete
- 2. Inserted leaf is “re-heaped” up:
   o Parent and child are compared and swapped if necessary
     ▪ Must maintain heap order
   o Repeated as go back up the tree

Remove:
- Performed at the root only.
  o 1. “final leaf” node replaces root
    ▪ Final leaf is the last node inserted
  o 2. Replaced root is “re-heaped” down:
    ▪ Parent and smallest child are compared swapped if necessary
    ▪ Repeats until heap order is established
Common use: priority queue
- Most important item is at the root

Heaps can be implemented with arrays

<table>
<thead>
<tr>
<th>Index</th>
<th>Root</th>
<th>Left child of root</th>
<th>Right child of root</th>
<th>Left child of node at 1</th>
<th>Right child of node at 1</th>
<th>Left child of node at 2</th>
<th>Right child of node at 2</th>
<th>Left child of node at 3</th>
<th>Right child of node at 3</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></td>
</tr>
</tbody>
</table>

Index of parent: integer result of: (index - 1) / 2

Index of left child: (index * 2) + 1
Index of right child: (index * 2) + 2
Templates and Generics:

What: C++ and Java language features to implement
_______________________________________.

How: By use of ___________________ in declaration,
you create a _____________ container where item to store in the
container is known at ________________.

___________ container: All items in the container are _____.
a design enforced at ________________.

Ex: SymTab<UCSDStudent> ST; // C++
or
    SymTab<UCSDStudent> ST =
        new SymTab<UCSDStudent>(); // Java

In C++: The instantiation argument can be ______________
In Java: The instantiation argument must be
______________________.

Why: C++: Use of template generates separate object code for
each type of instantiation argument used per template container.
This means:
- a Tree of UCSDStudent executes different object
code than a Tree of Variables.
- Polymorphism is implemented at compile time
- No unknown objects at run-time or a virtual table
Java: Generics is compile time enforcement only where generation of byte code is the same for all types used in generic containers.
This means:

- Polymorphism is still implemented at run-time via unknown object derived from an abstract parent using function pointers with a virtual table behind the scenes.

http://en.wikipedia.org/wiki/Virtual_method_table

<table>
<thead>
<tr>
<th></th>
<th>Generics</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polymorphic Generic Containers</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Store anything in the container</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size of item to store is known at compile time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiate an unknown item in container</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality of virtual constructor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality of virtual friend</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functionality of virtual static methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use of virtual table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polymorphism can begin before container instantiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polymorphism limited to member methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters of member methods of items to store are normal and expected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Casting needed when retrieving items</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More C/C++

mutable: Language feature of C++.
What: a prefix to a data field that can change even in a constant object.
Use: count number of times an operation is done.

union:
What: Language feature in C and in C++.
- C++ allows for anonymous unions.
  o Unnamed union:
    ▪ Use any fields of the union without a prefix.
  - Like a “struct”, except you only get one of the data fields declared with any one object.
    o sizeof (union object) is the size of its largest data field.
  - Same memory that changes its meaning depending upon use.
  - Uses are rare, but fits certain situations well.

Better example:
// read in ASCII codes from a user and convert to a floating point value.
// like "decin" but for real numbers.
float floatin() {

typedef union myFloat {
    long bits; // values that can shift and mask
    float result; // the floating point number
} myFloat;

MyFloat working; // 4 bytes
long mantissa;
long exponent;
long sign_bit;

...
// code below is okay because bits declared long
working.bits = mantissa | exponent | sign_bit;
...

return working.result;
// avoid implicit conversion by compiler to take the
    // integer value that is result and convert
    // that into a floating point number - float.
}

// float: 1 bit for sign, 8 bits for exponent, rest for mantissa.
// bit pattern, just like "setupword".
Struct in C:
   All data fields, all public, no methods, no access rights.

Struct in C++:
   - data fields
   - method fields
   - constructors
   - destructors
   - access rights: public, private, protected
     ○ … just like a class in C++, except:
       ▪ default access is public
         • class default is private
       ▪ default derivation is a public derivation
         • class default derivation is private
Efficiency:
What: A concept describing how good or bad a particular algorithm is.
Why: We want to save time and resources.
  - Time example: How long does it take for a computer to produce a result?
    o Real time systems have real-time requirements.
    o Users don’t want to wait too long for results.
  - Resource example: Customers billed by CPU cycle usage.

How do we measure:
  - Select an aspect of our algorithm to identify as a unit of work.
  - Then count it.
    o hw9: count _______________________

How do we compare efficiencies: Big-O
  - Measure of efficiency.
  - Not precise.
    o Constants are ignored
    o Lower order polynomial terms are ignored.

\[ O(3n^3 + 5n^2 + 45n + 2) = \text{___________} \]

n = number of items in your data set

Best Efficiency: _________
What: Amount of work performed is _______________________
_______________________________.

Array based Stack:
- We kept an index of the next available/last used/top of the stack.
- All operations used that pointer to locate the position in the array.

Operations:
- Push: ___
- Pop: ___
- Top: ___

Circular-Linked List:
- We kept a pointer to the front of the list.
- Insert to Front, Insert to End: ___
- Remove from Front or End: ___
- Viewing from Front or End: ___
- Insert anywhere: ___

Linked-list based Stack:
- Push: Insert at the END: ___
- Pop: Remove at the END: ___
- Top: View at the END: ___

Linked-list based Queue:
- Insert: Insert at the END: ___
- Remove: Remove from FRONT: ___
HashTable : Double Hashing, Ordered Hashing

- Insert, Lookup at the location that is the natural location for that object in your array.

  Insert on an empty table, few items:  
  Insert on a table with one slot left:  

  Lookup with empty, few items:  
  Lookup on a table with no slot left:  

Experiments to run:
  Fill the table to 85% full, 90% full, 95% full
  - Run 1000 lookups:
    o Measure the cost
    o Recommend an acceptable level of fullness for the table based upon analysis of results.
Binary Tree:

- Insert, Lookup, Remove on sparse Tree:
  - Each TNode has only one child
  - Just like a ________________:
<br>

- Insert, Lookup, Remove on a ideal Tree:
  - All Nodes have 0 or 2 children
  - All leaf Nodes are at the same level:
<br>

<table>
<thead>
<tr>
<th># of items in the Tree</th>
<th># of items in search path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
</tr>
</tbody>
</table>

- k-ary tree: Each Node could have k-1 data elements with k children:
  - _________________

Heap: Insert at final leaf and reheap up: __________
Remove: swap final leaf and reheap down: __________