Tables and Priority Queues
Tables

- Previously: each node stored one item
- Now: Groups of related information in records
  - Records indexed by key
  - Key is just one of the pieces of information – the one you want to be able to search on
- Support fast searching by key – $O(\log n)$
- Also support scanning for other information in records
  - But not as fast – $O(n)$ instead of $O(\log n)$
- Examples
  - Student records
  - List of pirated MP3s
Table ADT – Operations

1. Create an empty table
2. Determine whether a table is empty
3. Determine the number of items in a table
4. Insert a new item into the table
5. Delete the item with a given search key from the table
6. Retrieve the item with a given search key from the table
7. Traverse the items in a table in sorted search-key order
public abstract class KeyedItem {
    private Comparable searchKey;

    public KeyedItem(Comparable key) {
        searchKey = key;
    }

    public Comparable getKey() {
        return searchKey;
    }
}
public class City extends KeyedItem {
    private String country;
    private int population;
    public City(String theCity, String theCountry, int pop) {
        super(theCity);
        country = theCountry;
        population = pop;
    }
    public String toString() {
        return getKey() + ", " + country + " " + population;
    }
    public void setPopulation(int pop) { population = pop; }
    public int getPopulation() { return population; }
    public String getCountry() { return Country; }
}
TableInterface

import searchkeys.*;
public interface TableInterface {
    public boolean tableIsEmpty();
    public int tableLength();
    public void tableInsert(KeyedItem newItem)
        throws TableException;
    public boolean tableDelete(Comparable searchKey);
    public KeyedItem tableRetrieve(Comparable searchKey);
}

Implementing the Table

- Linear implementations
  - Unsorted array
  - Unsorted linked list
  - Sorted (by key) array
  - Sorted (by key) linked list

- Non-linear implementations
  - Binary search tree

- Criteria
  - What operations are needed?
  - How often will different operations be used?
  - How important are the different operations?
Examples

- Small, unsorted data, fixed size
  - Array
- Small, unsorted data, variable size
  - Linked list
- Small, sorted data, variable size
  - Linked list
- Large, unsorted data, variable size
  - Linked list
- Large, sorted data, variable size
  - Binary search tree
## Comparison of Implementation Alternatives

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted array</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Unsorted linked list</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(logn)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted linked list</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Binary search tree</strong></td>
<td>O(logn)</td>
<td>O(logn)</td>
<td>O(logn)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
See Table Code
Priority Queue

- It is often useful to assign a **priority** to different data items
  - so that urgent items can be processed first

- Examples
  - **Time** in a todo list
  - **Importance** in a list of phone calls to make
  - ...

It is often useful to assign a **priority** to different data items so that urgent items can be processed first. Examples include **Time** in a todo list, **Importance** in a list of phone calls to make, and...
Priority Queue ADT

1. Create an empty priority queue
2. Determine whether a priority queue is empty
3. Insert a new item into a priority queue
4. Retrieve and then delete the item in a priority queue with the highest priority
Implementing Priority Queues: Heaps

- A complete binary tree that is
  - Empty, or
  - Whose root has heaps as its subtrees, and
  - whose root contains a key greater than or equal to the key of each of its children
- Heaps are always balanced
- There is no order on the values of the keys of the two children of a node
  - Unlike binary search trees
Array-based Implementation of a Heap

- Breadth-first order in the array
- Complete tree $\Rightarrow$ no gaps in array
Inserting into the Heap

- Breadth-first order in the array
- Complete tree ⇒ no gaps in array
Deleting from the Heap (1)

- Always delete the root
  - It has the highest priority
Deleting from the Heap (2)

- Oops, now we have two heaps
Deleting from the Heap (3)

- Repairing the heap:
- Move the last element to the top
Deleting from the Heap (4)

- Repairing the heap:
- Now “trickle down” by comparing and swapping until heap is restored
Deleting from the Heap (5)

- Repairing the heap:
- Now “trickle down” by comparing and swapping until heap is restored
Deleting from the Heap (6)

Calculating indices:
- leftchild = 2*parent + 1
- Rightchild = 2*parent + 2
Deleting from the Heap (7)

- Now it’s a heap again
- Total time $O(\log n)$
Inserting to a heap

- The opposite of deleting
- Insert at the bottom, then “trickle up”
Heapsort

- **One way**
  - Insert everything into the heap, then
  - Take everything back out

- **Faster way**
  1. Make it a heap
     ```java
     for(int index = n/2; n >= 0; n--)
         heapRebuild(array, index, n);
     ```
  2. Swap the first item (largest) with the last (last--)
  3. heapRebuild(array, index, last);
  4. Repeat until all items are in the right place
Heapsort

- Make it a heap by doing `heapRebuild` to each node
  - Starting with leaf nodes

```
<table>
<thead>
<tr>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
```
Heapsort

- Swap first and "last"

Swap first and "last"
Heapsort

- last = last – 1
- heapRebuild()
Heapsort

- Swap first and “last”

```
2
5
3
1
6
9
```
Heapsort

- last = last - 1
- heapRebuild()
Heapsort

- Swap first and “last”

```
1
2
3
5
6
9
```
Heapsort

- $\text{last} = \text{last} - 1$
- heapRebuild()
Heapsort

- Swap first and “last”

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

LAST
Heapsort

- `last = last - 1`
- `heapRebuild()`

```
2
1
3
5
6
9
```

LAST
**Heapsort**

- Swap first and “last”
- Done!

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

LAST