CS171 Introduction to Computer Science II

Priority Queues and Binary Heap
Review

• Binary Search Trees (BST)
• Balanced search trees
• Hash tables
### ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>lg N *</td>
<td>lg N *</td>
<td>lg N *</td>
<td>3-5 *</td>
</tr>
<tr>
<td>linear probing</td>
<td>lg N *</td>
<td>lg N *</td>
<td>lg N *</td>
<td>3-5 *</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
Hashing vs. balanced search trees

**Hashing.**
- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

**Balanced search trees.**
- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

**Java system includes both.**
Priority Queues

• Need to process/search an item with largest (smallest) key, but not necessarily full sorted order

• Support two operations
  – Remove maximum (or minimum)
  – Insert

• Similar to
  – Stacks (remove newest)
  – Queues (remove oldest)
<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td>P</td>
</tr>
</tbody>
</table>
Applications

• Job scheduling
  – Keys corresponds to priorities of the tasks
• Sorting algorithm
  – Heapsort
• Graph algorithms
  – Shortest path
• Statistics
  – Maintain largest M values in a sequence
Priority queue API

Requirement. Generic items are comparable.

```java
public class MaxPQ<Key extends Comparable<Key>> {
    MaxPQ() create a priority queue
    MaxPQ(maxN) create a priority queue of initial capacity maxN
    void insert(Key v) insert a key into the priority queue
    Key max() return the largest key
    Key delMax() return and remove the largest key
    boolean isEmpty() is the priority queue empty?
    int size() number of entries in the priority queue
}
```

API for a generic priority queue
Priority queue client example

Challenge. Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).
- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store $N$ items.

```
% more tinyBatch.txt
Turing  6/17/1990   644.08
vonNeumann  3/26/2002  4121.85
Dijkstra  8/22/2007  2678.40
vonNeumann  1/11/1999  4409.74
Dijkstra  11/18/1995  837.42
Hoare  5/10/1993  3229.27
vonNeumann  2/12/1994  4732.35
Hoare  8/18/1992  4381.21
Turing  1/11/2002  66.10
Thompson  2/27/2000  4747.08
Turing  2/11/1991  2156.86
Hoare  8/12/2003  1025.70
vonNeumann  10/13/1993  2520.97
Dijkstra  9/10/2000  708.95
Turing  10/12/1993  3532.36
Hoare  2/10/2005  4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson  2/27/2000  4747.08
vonNeumann  2/12/1994  4732.35
vonNeumann  1/11/1999  4409.74
Hoare  8/18/1992  4381.21
vonNeumann  3/26/2002  4121.85
```

sort key
Possible implementations

• Sorting N items
  – Time: NlogN
  – Space: N

• Elementary PQ - Compare each new key against M largest seen so far
  – Time: NM
  – Space: M

• Using an efficient MaxPQ Implementation
**Priority queue client example**

**Challenge.** Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

```java
MinPQ<Transaction> pq = new MinPQ<Transaction>();

while (StdIn.hasNextLine())
{
    String line = StdIn.readLine();
    Transaction item = new Transaction(line);
    pq.insert(item);
    if (pq.size() > M)
        pq.delMin();
}
```

Transaction data type is Comparable

pq contains largest $M$ items

**Order of growth of finding the largest $M$ in a stream of $N$ items**

<table>
<thead>
<tr>
<th>implementation</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>$MN$</td>
<td>$M$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$N \log M$</td>
<td>$M$</td>
</tr>
<tr>
<td>best in theory</td>
<td>$N$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
Implementations

• Elementary representations
  – Unordered array (lazy approach)
  – ordered array (eager approach)

• Efficient implementation
  – Binary heap structure

• Can we implement priority queue using Binary Search Trees?
## Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td>1</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td>2</td>
<td>P E</td>
<td>E P</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>4</td>
<td>P E X A</td>
<td>A E P X</td>
<td>A E M P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td>5</td>
<td>P E X A M</td>
<td>A E M P X</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>4</td>
<td>P E M A</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>6</td>
<td>P E M A P P L</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>7</td>
<td>P E M A P P L E</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>6</td>
<td>E M A P P L E</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue
**Sequence-based Priority Queue**

- **Implementation with an unsorted list**
  
  - **Performance:**
    - `insert` takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
    - `removeMin` and `min` take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- **Implementation with a sorted list**
  
  - **Performance:**
    - `insert` takes $O(n)$ time since we have to find the place where to insert the item
    - `removeMin` and `min` take $O(1)$ time, since the smallest key is at the beginning
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;  // pq[i] = ith element on pq
    private int N;     // number of elements on pq

    public UnorderedMaxPQ(int capacity)
    {  pq = (Key[]) new Comparable[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void insert(Key x)
    {  pq[N++] = x;  }

    public Key delMax()
    {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max, i)) max = i;
        exch(max, N-1);
        return pq[--N];
    }
}
Priority queue elementary implementations

**Challenge.** Implement all operations efficiently.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Del Max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>goal</td>
<td>log (N)</td>
<td>log (N)</td>
<td>log (N)</td>
</tr>
</tbody>
</table>
A heap is a binary tree storing keys at its nodes and satisfying two properties:

- **Heap-Order:** for every internal node \( v \) other than the root, \( key(v) \geq key(parent(v)) \)

- **Complete Binary Tree:** let \( h \) be the height of the heap
  - for \( i = 0, \ldots, h - 1 \), there are \( 2^i \) nodes of depth \( i \)
  - at depth \( h - 1 \), the internal nodes are to the left of the external nodes
  - The last node of a heap is the rightmost node of depth \( h-1 \)
Binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.

Property. Height of complete tree with $N$ nodes is $\lceil \lg N \rceil$.

Pf. Height only increases when $N$ is a power of 2.
Height of a Heap

• **Theorem**: A heap storing $n$ keys has height $O(\log n)$

Proof: (we apply the complete binary tree property)

- Let $h$ be the height of a heap storing $n$ keys
- Since there are $2^i$ keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth $h$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$
A complete binary tree in nature

Hyphaene Compressa - Doum Palm
Binary heap representations

**Binary heap.** Array representation of a heap-ordered complete binary tree.

**Heap-ordered binary tree.**
- Keys in nodes.
- No smaller than children’s keys.

**Array representation.**
- Take nodes in **level** order.
- No explicit links needed!
Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.
- Parent of node at $k$ is at $k/2$.
- Children of node at $k$ are at $2k$ and $2k+1$. 
Insert/Remove and Maintaining Heap order

• When a node’s key is larger than its parent key
  – Upheap (promote, swim)

• When a node’s key becomes smaller than its children’s keys
  – Downheap (demote, sink)
Promotion in a heap

Scenario. Node's key becomes larger key than its parent's key.

To eliminate the violation:
- Exchange key in node with key in parent.
- Repeat until heap order restored.

```java
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```
Insertion in a heap

**Insert.** Add node at end, then swim it up.

**Cost.** At most \( 1 + \lg N \) compares.

```java
public void insert(Key x) {
    pq[++N] = x;
    swim(N);
}
```
Demotion in a heap

**Scenario.** Node's key becomes smaller than one (or both) of its children's keys.

**To eliminate the violation:**
- Exchange key in node with key in larger child.
- Repeat until heap order restored.

```java
private void sink(int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

Power struggle. Better subordinate promoted.
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most \(2 \log N\) compares.

```java
public Key delMax()
{
    Key max = pq[1];
exch(1, N--);
sink(1);
pq[N+1] = null;
return max;
}
```
Demo
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int N;

    public MaxPQ(int capacity)
    {   pq = (Key[]) new Comparable[capacity+1];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void insert(Key key)
    {   /* see previous code */   }

    public Key delMax()
    {   /* see previous code */   }

    private void swim(int k)
    {   /* see previous code */   }

    private void sink(int k)
    {   /* see previous code */   }

    private boolean less(int i, int j)
    {   return pq[i].compareTo(pq[j] < 0;   }

    private void exch(int i, int j)
    {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;   }
}
## Priority queues implementation cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>log N</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>log&lt;sub&gt;d&lt;/sub&gt; N</td>
<td>d log&lt;sub&gt;d&lt;/sub&gt; N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N †</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† amortized