06 B: Hashing and Priority Queues

CS1102S: Data Structures and Algorithms

Martin Henz

February 26, 2010

Generated on Friday 26th February, 2010, 09:23

1. Hashing
2. Priority Queues
3. Puzzlers

1 Hashing
- Collision Resolution Strategies
- Double Hashing
- A Detail: Removal from Hash Table
- Hash Tables in the Java API

2 Priority Queues

3 Puzzlers

Recap: Main Ideas

Implement set as array

Store values in array; compute index using a *hash function*.

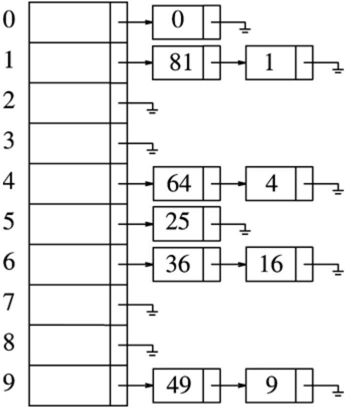
Spread

The hash function should "spread" the hash keys evenly over the available hash values

Collision

Hash table implementations differ in their strategies of collision resolution: Two hash keys mapping to the same hash value

Separate Chaining



Hash Tables without Linked Lists

Idea

Store items directly into array; use alternative cells if a collision occurs

More formally

Try cells $h_0(x), h_1(x), h_2(x), \ldots$ until an empty cell is found.

How to define h_i?

$$h_i(x) = (hash(x) + f(i)) \mod TableSize$$

where $f(0) = 0$.

Load factor, λ

Ratio of number of elements in hash table to table size.

Linear Probing

Conflict resolution

$$f(i) = i$$

Clustering

As the load factor λ increases, occupied areas in the array tend to occur in clusters, leading to frequent unsuccessful insertion tries.

Quadratic Probing

Conflict resolution

$$f(i) = i^2$$

Theorem

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Double Hashing

Idea

Use a second hash function to find the jump distance

Formally

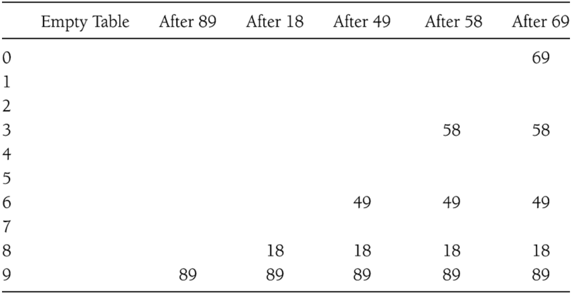
$$f(i) = i \cdot hash_2(x)$$

Attention

The function $hash_2$ must never return 0. Why?

Double Hashing: Example

$$\begin{aligned} hash_1(x) &= x \mod 10 \\ hash_2(x) &= 7 - (x \mod 7) \\ h_i(x) &= hash_i(x) + i \cdot hash_2(x) \end{aligned}$$



	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

A Detail: Removal from Hash Table

Removal from separate chaining hash table

Straightforward: remove item from respective linked list (if it is there)

Removal from Probing Hash Table: First idea

Set the respective table entry back to **null**

Problem

This operation interrupts probing chains; elements can be “lost”

Solution

```
private static class HashEntry<AnyType> {
  public AnyType  element;
  public boolean isActive;
  public HashEntry(AnyType e) {
    this(e,true); }
  public HashEntry(AnyType e, boolean i) {
    element  = e; isActive = i; }
}
public void remove( AnyType x ) {
  int currentPos = findPos( x );
  if( isActive( currentPos ) )
    array[ currentPos ].isActive = false;
}
```

Remember Sets?

Idea

A Set (interface) is a Collection (interface) that does not allow duplicate entries.

HashSet

A HashSet is a hash table implementation of Set.

```
class HashSet<E> implements Set<E>
```

1 Hashing

2 Priority Queues
- Motivation
- Binary Heaps
- Basic Heap Operations
- Priority Queues in Standard Library

3 Puzzlers

Motivation

Operations on queues

add(e): enter new element into queue
remove(): remove the element that has been entered first

A slight variation

Priority should not be *implicit*, using the time of entry, but *explicit*, using an ordering

Operations on priority queues

insert(e): enter new element into queue
deleteMin(): remove the *smallest* element

Application Examples

- Printer queue: use number of pages as “priority”
- Discrete event simulation: use simulation time as “priority”
- Network routing: give priority to packets with strictest quality-of-service requirements

Simple Implementations

- Unordered list: insert(e): $O(1)$, deleteMin(): $O(N)$
- Ordered list: insert(e): $O(N)$, deleteMin(): $O(1)$
- Search tree: insert(e): $O(\log N)$, deleteMin(): $O(\log N)$

Binary Heaps

Rough Idea

Keep a binary tree whose root contains the smallest element
insert(e) and deleteMin() need to restore this property

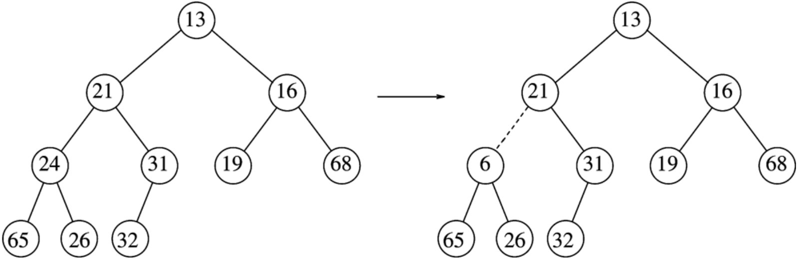
Completeness

Keep binary tree *complete*, which means completely filled, with the possible exception of the bottom level, which is filled from left to right.

Heap-order

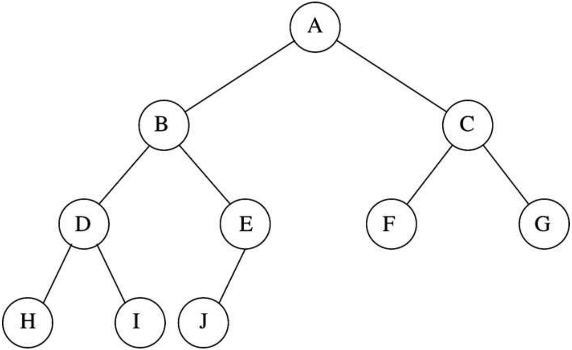
For every node X, the key in the parent of X is smaller than or equal to the key in X, with the exception of the root

Order in Binary Heap



Tree on the left is a binary heap; tree on the right is not!

Representation as Array



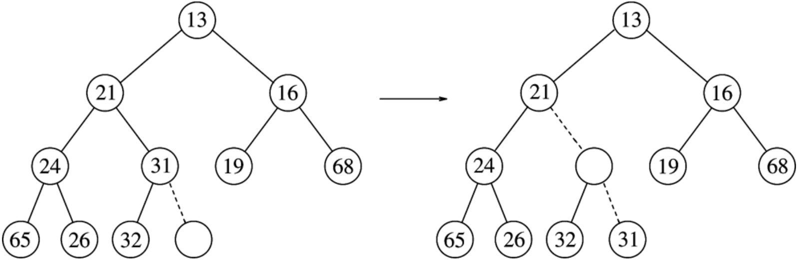
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F	G	H	I	J			

insert

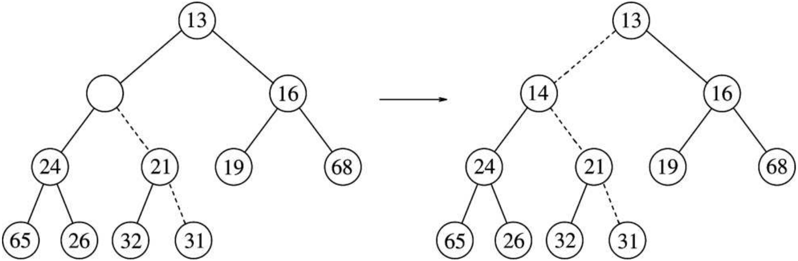
Idea

Add “hole” at bottom and “percolate” the hole up to the right place for insertion

Example: insert(14)



Example: insert(14), continued



Analysis

Worst case

$$O(\log N)$$

Average

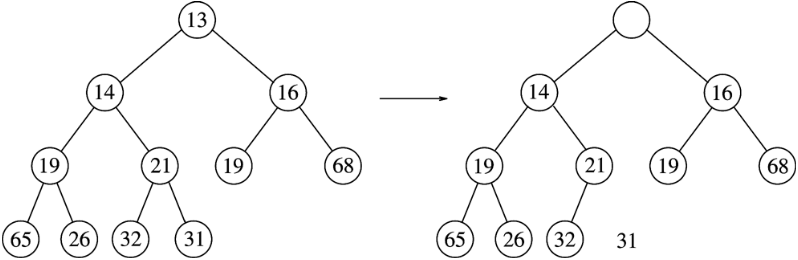
2.607 comparisons

deleteMin

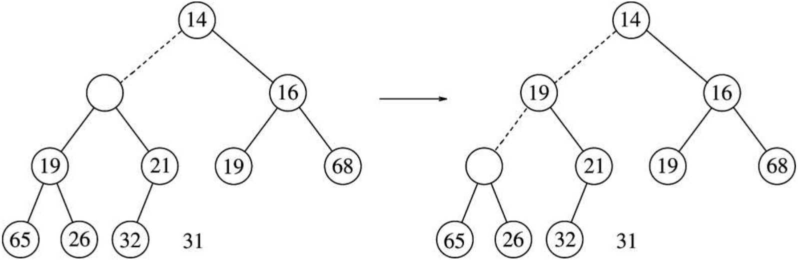
Idea

Remove root, leaving “hole” at top. “Percolate” the hole down to a correct place for insertion of bottom element

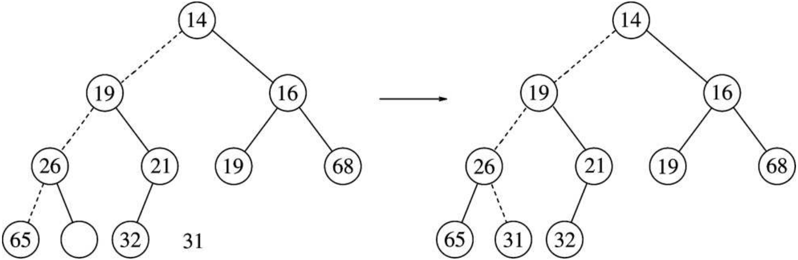
Example: deleteMin()



Example: deleteMin(), continued



Example: deleteMin(), continued



Analysis

Worst case

$$O(\log N)$$

Average

$$\log N$$

buildHeap

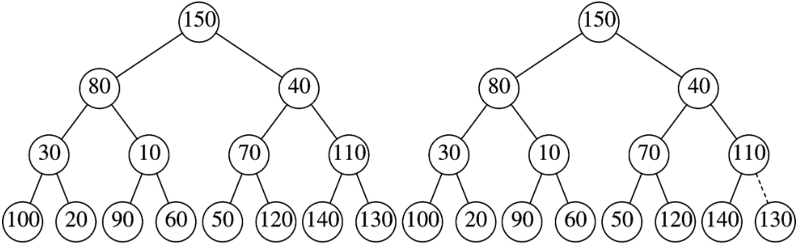
Initial setup

Build a heap from a given (unordered) collection of elements

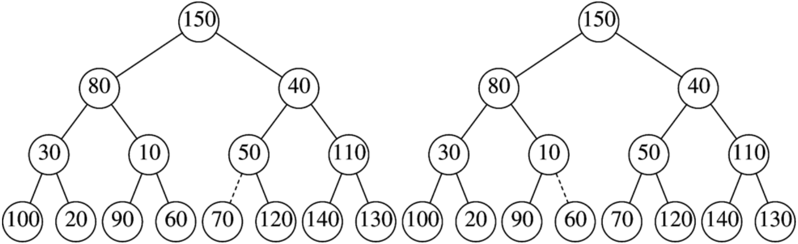
Idea

"Percolate" every inner node down the tree

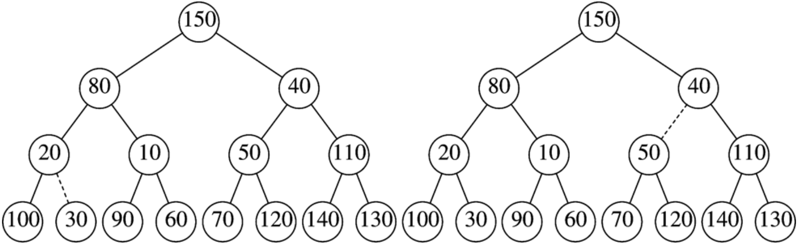
Example: buildHeap, percolateDown(7)



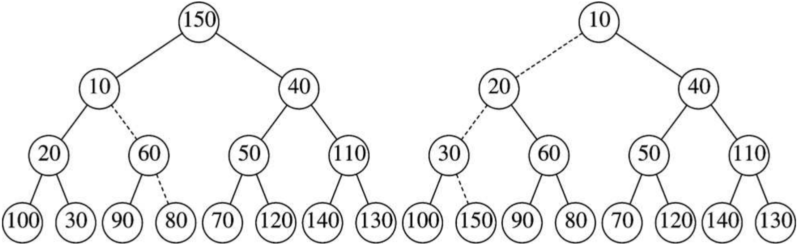
Example: buildHeap, percolateDown(6), ..(5)



Example: buildHeap, percolateDown(4), ..(3)



Example: buildHeap, percolateDown(2), ..(1)



Analysis

Bound

The runtime is bounded by the sum of all heights of all nodes

Theorem

For perfect binary tree of height h, containing $2^{h+1} - 1$ nodes, sum of heights of nodes is $2^{h+1} - 1 - (h + 1)$.

Worst case

$$O(N)$$

Other Heap Operations

decreaseKey(p,Δ)

Lowers the value of item at position p by a positive amount Δ.
Implementation: Percolate up

increaseKey(p,Δ)

Increases the value of item at position p by a positive amount Δ.
Implementation: Percolate down

delete(p)

Remove value at position p
Implementation: decreaseKey(p,∞), then deleteMin()

Priority Queues in Standard Library

```
class PriorityQueue<E> {
   boolean add(E e) {...} // add element
   E poll() {...}   // remove smallest
}
```

1. Hashing
2. Priority Queues
3. Puzzlers
    - Last Puzzler: “It’s Elementary”
    - New Puzzler: The Last Laugh

Last Puzzler: It's Elementary

What does the following program print?

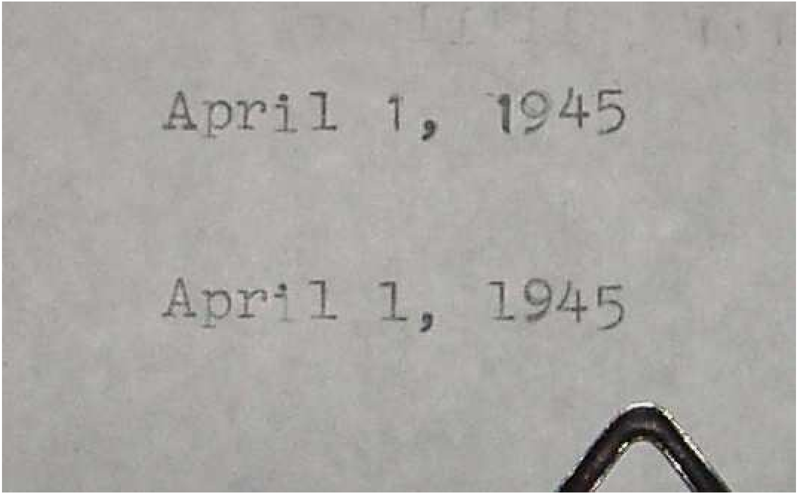
```
public class Elementary {
   public static void main(String[] args) {
      System.out.println(12345 + 5432l);
   }
}
```

Scary...

```
I’m so scared when try running these codes on
Eclipse. When I run the file downloaded from
the module homepage, the result is 17777.

But when I type it myself , the result is 66666.
Maybe the number in teacher’s file is not the
normal number, right ?
```

The "Fine" Print



April 1, 1945

April 1, 1945

Constant Numbers in Java

(see Java Language Specification)

- `12345`: **int** constant in decimal notation
- `0xff`: **int** constant in hexadecimal notation
- `077`: **int** constant in octal notation
- `45.23`: **double** constant
- `5432l`: **long** constant in decimal notation
- `0xffL`: **long** constant in hexadecimal notation
- `077L`: **long** constant in octal notation

Useful Habit

Use “L” (and not “l”) to indicate **long** literals:

```
public class Elementary {
   public static void main(String[] args) {
      System.out.println(12345 + 5432L);
   }
}
```

New Puzzler: The Last Laugh

What does the following program print?

```
public class LastLaugh {
   public static void main(String[] args) {
      System.out.println("H" + "a");
      System.out.println('H' + 'a');
   }
}
```

Next Week

- Sorting, sorting, and more sorting!