Heaps (7)

1 DYNAMIC PROBLEMS

1.1 Dynamic Problems

The applications we've talked about so far have the basic form: input, output, thank you computer.
Many applications have a different flavor: computer maintains some data in a data structure, various operations are performed on the data:

- adding new information,
- removing outdated information,
- querying the current set of data.

Examples: elevator simulation, customer lists.

1.2 Elevator Simulation

We are trying to design an algorithm to control a bank of 10 elevators in a 100-floor office building. At any moment in time, each elevator is on some floor (or between floors), and contains some number of passengers with some set of buttons pushed.
The elevators need to make a decision each time a button is pressed and/or an elevator stops and opens its doors.
The simulator needs to keep track of all this and correctly simulate all events in their proper order.
This is a kind of *discrete event simulator*.

1.3 Dynamic Operations

Some operations our elevator-simulator data structure needs to support:

- schedule an event for a time in the future,
- find next event,
- remove current event,

- merge two simulations together.

And these will need to happen fast because there will be a lot of events (button pushes and elevator arrivals) over the course of the simulation.

1.4 Customer Lists

Earlier, we discussed sorting as an important operation in telephone-book publishing. This application is kind of special in that the telephone book is only published once or twice a year, so all the work can be done right before the publishing date.
Consider, on the other hand, the database kept by American Express for tracking "card-members." This needs to be much more dynamic.
Each customer has a record: name, customer id number, address, perhaps some billing information (!).

1.5 Dynamic Operations

We need to be able to quickly:

- add new customers to the database,
- look up a customer's record by name,
- delete a customer,
- list all customers sorted by name,
- list all customers in a range of names,
- add a group of customers,
- split an entire group of infinitives (oops).

1.6 Ordered Trees

It turns out that binary trees are a wonderful way to flexibly store dynamic information.
Here's a tree T. Some terminology:

- root: top of the tree
- node: some place in the tree
- key: the value stored in a node
- left child, right child: the nodes immediately beneath a node
- subtree: tree rooted at some node
- parent: the node immediately above a node

- leaf: childless node
- depth: distance from node to root (depth of a tree is the depth of the deepest leaf)
- height: distance to furthest leaf (height of a tree is the height of the root)

What do we know about the height and depth of a tree?

1.7 Representing Trees

There are a few ways to represent a tree T with n nodes.
One particularly flexible way is in terms of three arrays. For $0 \leq i \leq n-1$:

- $T_v[i]$: The key for node i.
- $T_p[i]$: The index of the parent of node i (-1 for root).
- $T_l[i]$: The index of the left child of node i (-1 if there is none).
- $T_r[i]$: The index of the right child of node i (-1 if there is none).

How can we test if node i is a leaf?

2 HEAPS

2.1 Priority Queue

Rule #3 of Good Algorithm Design: Keep complex information organized in a good data structure.
A data structure that supports insert and delete-min is called a "priority queue."

2.2 Priority Queue for Elevator

These are the core operations we need for the elevator-simulator application. In the elevator simulator:

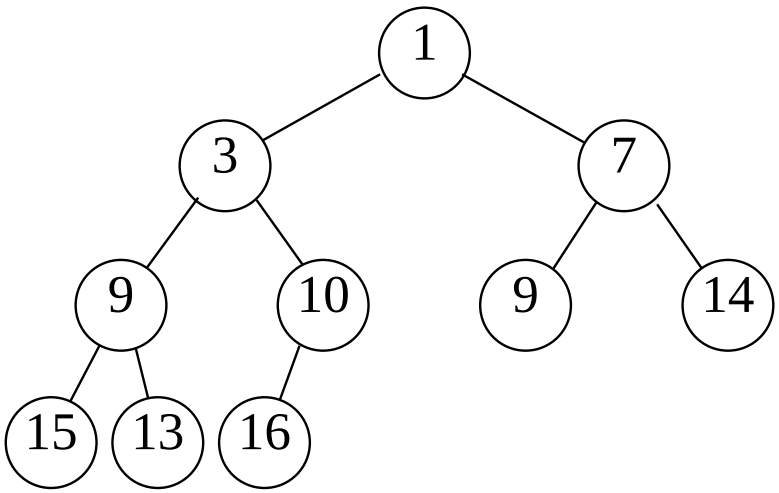
- The records are events.
- The keys are the time at which those events will occur.
- We always want to execute the event with the "smallest time" (closest to the present).
- Executing an event may result in other future events (elevator arriving at lobby and picking up passengers results in elevator planning to arrive at 5th floor at time $t + 20$ seconds).

One nice implementation of a priority queue is the *heap*.

2.3 Heap Definition

Complete n-node binary tree.
All nodes (other than root) satisfy the "heap property":

$$T_v[i] \geq T_v[T_p[i]].$$



2.4 Some Properties of Heaps

Height is $\Theta(\log(n))$.
Smallest item easy to identify.
Where's the largest item? Second smallest? Third smallest?
Like a bunch of sorted lists knitted together.
How many nodes have one child? Just a left child? Just a right child?

2.5 Representing a Heap

Can use the multiple-array representation.
Because binary tree is "complete" (packed to the left), there is an easier way.
Read off the keys by depth, left to right.
Where are the two children of a node i? How about the parent of i?
How recognize leaf? Parent?

2.6 Maintaining the Heap Property

A heap is a dynamic data structure, so we're going to have to contend with changes.
The most important changes can be handled by two basic operations:

- Up-heapify: The entire heap is valid, except for item i, which may violate the heap property by being smaller than its parent.
- Down-heapify: The entire heap is valid, except for the root, which might be too big. (Who violates the heap property, then?)

These don't change the shape of the tree, just which records are at which nodes.

2.7 Down-Heapify

```
void Down_Heapify( vector<int> & data, unsigned int i )
{
  unsigned int left = left_child( i );
  unsigned int right = right_child( i );
  bool hasleft = left < data.size();
  bool hasright = right < data.size();

  // data[i] is a leaf
  if( !hasleft && !hasright )
    return;

  // heap property is satisfied
  // left child is greater than parent and right child is greater than
  // parent if it exists
  if( hasleft && data[i] <= data[ left ]
      && ( ( hasright && data[i] <= data[ right ] )
   || !hasright ) )
    return;

  // Otherwise, swap data[i] and the smaller of its children.
  unsigned int smaller;
  if( hasright && data[ right ] <= data[ left ] )
    smaller = right;
  else
    smaller = left;

  swap( data, i, smaller );

  // Start again on the child that changed (i.e., move data[i] down).
  Down_Heapify( data, smaller );
}
```

Look at node i (initially the root).

- If key[i] $\leq$ key[left[i]] and key[i] $\leq$ key[right[i]], all is well. (Why?)
- Similarly if i is a leaf. (Why?)
- Otherwise, swap i and the smaller of its children.
- Start again on the child that changed (i.e., move i down).

We know that the only node that may violate the heap property is a child of the new i. (Why?)
Kind of like bubblesort or insertion sort in a tree. The root value trickles down until it stops.
Running Time? (Maximum number of comparisons and swaps.)

2.8 Up-Heapify

A bit simpler.
Look at node i (initially $i = n$).

- If key[i] $\geq$ key[parent[i]], all is well. (Why?)
- Similarly if i is the root. (Why?)
- Otherwise, swap i and its parent.
- Start again on the parent that changed (i.e., move i up).

We know that the only node that may violate the heap property is the new i. (Why?)
Kind of like bubblesort or insertion sort in a tree. The last leaf value trickles up until it stops.
Running Time? (Maximum number of comparisons and swaps.)

2.9 Building a Heap from a List

Let A be an n element list. We will turn A into a heap (in place).

```
void Build_Heap( vector<int> & data, unsigned int i )
{
  if( i >= data.size() )
    return;

  Build_Heap( data, left_child( i ) );
  Build_Heap( data, right_child( i ) );
  Down_Heapify( data, i );
}
```

Recall that `leftchild(i)` is $2i$ and `rightchild(i)` is $2i + 1$.
Also, `Down_Heapify(A, i)` performs the down-heapify action starting at node i.
Correctness argument? What does down-heapify do?

2.10 Build-Heap Analysis

How would you analyze this? What is the running time?
Simple analysis and tighter analysis.

2.11 Extract Min

Now that we can create a heap, how can we remove things from it?
Clearly the minimum element is at the top.
Now the number of elements in the heap has decreased by one.
Which node should we move to the root?
How do we restore the heap property?
Running Time?

2.12 Insert

Now we'll look at putting a new element into the heap.
Where should it go?
How do we restore the heap property?
Running time?
We can use insert repeatedly to build a heap from a set of n elements. What recurrence do we get for the running time? How does the running time of this compare to that of build-heap?

2.13 Heap Sort

We can use heaps to sort.

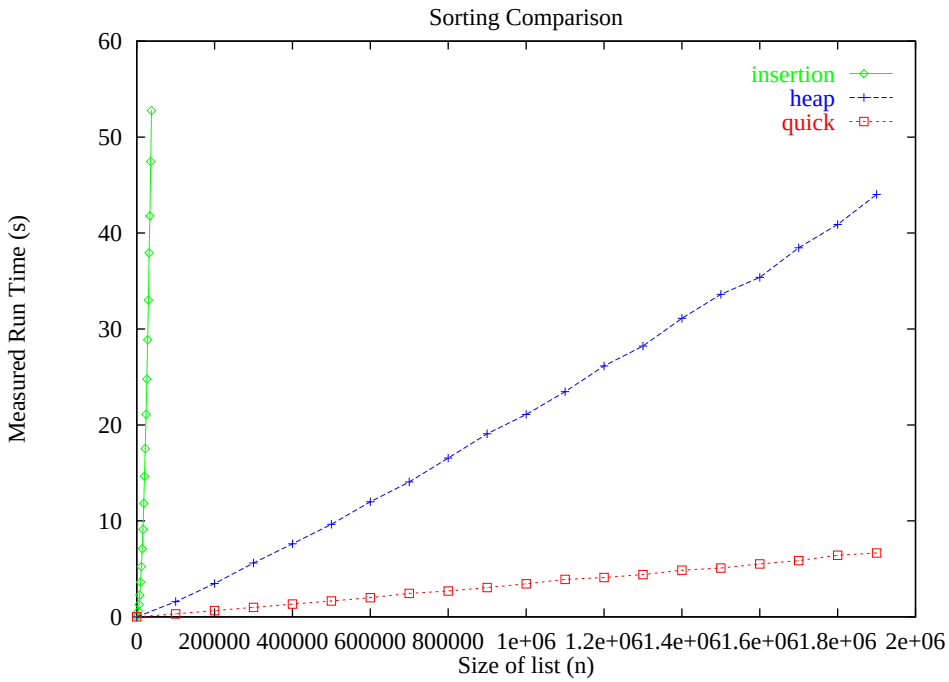
```
void heapsort( vector<int> & data )
{
  unsigned int N = data.size()-1; // Can't use zero slot.

  Build_Heap( data , 1 );

  for( unsigned int i = 0; i < N; i++ )
    data[N-i+1] = Extract_Min( data );

}
```

Running Time? How is this nicer than merge sort? Quicksort?



3 LISTING LARGEST ELEMENTS

Let's think about solving the following problem. A web search engine matches queries against all the documents in a database and computes a score for each. It then needs to present the best documents to the user, in order.

Step 0 : Formalize problem.

Step 1 : Propose algorithms.

Step 2 : Prove correctness.

Step 3 : Prove running-time bounds.