Basic Data Structures and Heaps

David Kauchak

- Sorting demo (http://math.hws.edu/TMCM/java/xSortLab/)
- Data structures

  What is a data structure?

  Way of storing data that facilitates particular operations.

  We'll talk about mostly about operations on a dymanic set:

  Operations:

  For a set S

  - SEARCH(S, x) - Does x exist in S?
  - INSERT(S, x) - Add x to S.
  - DELETE(S, x) - Given a pointer/reference to an element x, delete it from S
  - MINIMUM(S) - Return the smallest element from S
  - MAXIMUM(S) - Return the largest element from S

- Array

  Sequential locations in memory in linear order. Element are accessed via their index.

  Read/write application of particular indices occur in $\Theta(1)$ time. Insert, delete, and most set operations (e.g. MAXIMUM) occur in $\Theta(n)$ time.

  Representation

  - SEARCH - Iterate over elements until element is found - $O(n)$

- INSERT - Create new array of size n+1. Copy elements and insert new element - $\Theta(n)$
- INSERTINDEX - Insert element and shift over all remaining elements - $\Theta(n)$
- DELETE - Delete element and shift down remaining elements - $\Theta(n)$
- MINIMUM - Search every element for the minimum - $\Theta(n)$
- MAXIMUM - Search every element for the maximum - $\Theta(n)$

Uses?

- Double Linked list

  Objects are arranged linear. An element in the list points to the next and previous element in the list.

  Deletion of particular element occurs in $\Theta(1)$ time. Most other operations occur in $\Theta(n)$ time.

  Representation

  - SEARCH - Iterate over elements by following links until element is found - $\Theta(n)$
  - INSERT - Add element to the beginning of the linked list and update the head - $\Theta(1)$
  - INSERTINDEX - Iterate over elements until index is found and insert element by updating links - $\Theta(n)$
  - DELETE - Delete element and update pointers of previous and next element - $\Theta(1)$
  - MINIMUM - Search every element for the minimum - $\Theta(n)$
  - MAXIMUM - Search every element for the maximum - $\Theta(n)$

  Uses?

- Stack

  LIFO (last in first out)

  Picture the stack of plates at a buffet

  Can be implemented using an array or a linked list.

- EMPTY(S) - determines if stack is empty of not

  EMPTY(S)
  1 **if** TOP(S) = null
  2 **return** true
  3 **else**
  4 **return** false

- PUSH(S, x) - adds an element to the top of the stack

  PUSH(S, x)
  1 $temp \leftarrow$ TOP(S)
  2 TOP(S) $\leftarrow x$
  3 NEXT(x) $\leftarrow temp$

- POP(S) - removes and returns the top element of the stack

  POP(S)
  1 **if** EMPTY(S)
  2 **error**
  3 **else**
  4 $temp \leftarrow$ TOP(S)
  5 $top \leftarrow$ NEXT(TOP(S))
  6 **return** *temp*

Uses:

- runtime "stack"
- graph search algorithms (depth first search)
- syntactic parsing (i.e. compilers)

- Queue

  FIFO (first in first out)

  Picture a line at the grocery store

  Can be implemented with an array or a linked list

  - EMPTY(S) - determines if the queue is empty of not

EMPTY(S)

```
1  if HEAD(S) = null
2          return true
3  else
4          return false
```

– ENQUEUE(S, x) - adds an element to the end of the queue

ENQUEUE(S, x)

```
1  if EMPTY(S)
2          HEAD(S) ← x
3          TAIL(S) ← x
4  else
5          NEXT(TAIL(S)) ← x
6          TAIL(S) ← x
```

– DEQUEUE(S) - removes and returns the first element in the queue

DEQUEUE(S)

```
1  if EMPTY(S)
2          error
3  else
4          temp ← HEAD(S)
5          if HEAD(S) = TAIL(S)
6                  HEAD(S) ← null
7          else
8                  HEAD(S) ← NEXT(HEAD(S))
9          return temp
```

Uses:

– scheduling

– graph search (breadth first search)

- Binary Heap

  A binary tree where the value of a parent is greater than or equal to the value of it's children.

  We will also impose an additional restriction that all levels of the tree are completely filled except the last.

Given this restriction, for a heap with n elements, what is the largest number of nodes one of the sub-trees (i.e. children) can have? $2n/3$

Max vs. min heap

Operations:

- MAXIMUM(S) - return the largest element in the set - $\Theta(1)$
- EXTRACT-MAX(S) - return and remove the largest element in the set - $O(\log n)$
- INSERT(S, val) - insert an element with val into the set - $O(\log n)$
- INCREASE-ELEMENT(S, x, val) - increase the element x to val - $O(\log n)$
- BUILD-HEAP(A) - build a heap from an array of elements - $O(n)$

Can implement with pointers:

picture

or with an array, which we'll use since it simplifies error checking:

picture

What is the difference between these two representations?

PARENT(i)
    return $\lfloor i/2 \rfloor$

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Given the above definition, what can we say about each sub-tree in a binary heap? Each sub-tree in a binary tree (including the leaves) is

itself a binary heap.

Key function for maintaining the heap property: HEAPIFY. Assume left and right children are heaps, but current node/tree may or may not be. Turn current tree into a valid heap.

```
HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  largest ← i
 4  if l ≤ heap-size[A] and A[l] > A[i]
 5          largest ← l
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7          largest ← r
 8  if largest ≠ i
 9          swap A[i] and A[largest]
10          HEAPIFY(A, largest)
```

Example run

- Is it correct?

  The key is the knowledge that both children are heaps. Three cases:

  1. $A[i]$ is larger than $A[\text{LEFT}(i)]$ and $A[\text{RIGHT}(i)]$ - In this case, the tree at index i is a heap already and the call to HEAPIFY doesn't change anything.
  2. $A[\text{LEFT}(i)]$ is larger than $A[i]$ and $A[\text{RIGHT}(i)]$. $A[\text{LEFT}(i)]$ is swapped with $A[i]$ and HEAPIFY is called on $\text{LEFT}(i)$. When this call completes. $\text{LEFT}(i)$ will be a heap and $\text{RIGHT}(i)$ is a heap (since nothing changed). Finally, the tree rooted at i is a heap since both children are heaps and we know that $A[\text{LEFT}(i)]$ is the largest element, since both original subtrees were heaps and we selected the largest element from the roots.
  3. $A[\text{RIGHT}(i)]$ is larger than $A[i]$ and $A[\text{LEFT}(i)]$. Same as above for $\text{LEFT}(i)$.

- Running time?

How much work is done for each call? $\Theta(1)$

How many recursive calls are made? O(height of the tree).

Recall, that we impose the additional restriction that the tree be complete except for the last level:
n elements $= O(\lfloor \log n \rfloor) = O(\log n)$

Operations:

MAXIMUM(A)
return A[1]

Running time? $\Theta(1)$

EXTRACT-MAX(A)
1 **if** *heap-size*$[A] < 1$
2 **error**
3 $max \leftarrow A[1]$
4 $A[1] \leftarrow A[\text{HEAP-SIZE}[A]] - 1$
5 *heap-size*$[A] \leftarrow$ *heap-size*$[A] - 1$
6 HEAPIFY($A, 1$)
7 **return** max

Running time? Constant amount of work plus 1 call to HEAPIFY - $O(\log n)$

What about deleting an arbitrary element, e.g. DELETE(A, i)?

INCREASE-ELEMENT(A, i, val)
1 **if** $val < A[i]$
2 **error**
3 $A[i] \leftarrow val$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5 swap $A[i]$ and $A[\text{PARENT}(i)]$
6 $i \leftarrow \text{PARENT}(i)$

Running time? Follows a path from a node to the root - $O(\text{height of the tree}) = O(\log n)$

INSERT(A, val)

1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $A[heap\text{-}size[A]] \leftarrow -\infty$
3 INCREASE-ELEMENT$(A, heap\text{-}size[A], val)$

Running time? Constant amount of work plus 1 call to INCREASE-ELEMENT - $O(\log n)$

Building a heap

BUILD-HEAP1(A)

1 copy A to B
2 $heap\text{-}size[A] \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** $length[B]$
4 INSERT$(A, B[i])$

- Is it correct?
- Running time? n calls to INSERT - $O(n \log n)$. Can we get a tighter bound? Let's hold that thought and look at a second method of building heaps.

BUILD-HEAP2(A)

1 $heap\text{-}size[A] \leftarrow (length)[A]$
2 **for** $i \leftarrow \lfloor (length)[A]/2 \rfloor$**to**1
3 HEAPIFY(A, i)

What is this algorithm doing? Start with $\lfloor n/2 \rfloor$ simple heaps (i.e. heaps of just one element). Gradually build up a single heap by adding in each of the nodes $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, ..., 1$ one at a time as the parent of two previously created heaps.

Show example

– Is it correct?
  Invariant: Each node $i+1, i+2, ..., n$ is the root of a heap

  Base case: $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, ..., n$ is a leaf and therefore a trivial heap.

  Inductive case: We know $i+1, i+2, i+3, ..., n$ are all heaps. Therefore, the call to HEAPIFY(A, i) generates a heap rooted at node i.

  Termination: $i = 0$, so $1, 2, ..., n$ are heaps, including node 1.
– Running time? n calls to HEAPIFY - $O(n \log n)$

  Can we get a tighter bound?

  n element heap has height $h = \lfloor \log n \rfloor$ and $\lceil n/2^{h+1} \rceil$ nodes at any height

$$\begin{aligned} \sum_{h=0}^{\log n} \lceil \frac{n}{2^{h+1}} \rceil O(h) &= O(n \sum_{h=0}^{\log n} \frac{h}{2^h}) \\ &= O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\ &= O(n) \end{aligned}$$

  because

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 \end{aligned}$$

Uses:

– HEAPSORT - $O(n \log n)$
– Priority queue

– job/process/network traffic scheduling
– A* search alrgorithm

Satellite data

What about UNION(A_1, A_2)? Concatenate arrays and call BUILD-HEAP - $O(n)$

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

These notes are adapted from material found in chapters 6,10 of [1].

References

[1] Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest and Clifford Stein. 2007. Introduction to Algorithms, 2nd ed. MIT Press.