Tree→Heap→Priority Queue

Rivers

171860553@smail.nju.edu.cn

May 23, 2018

Section 1

Contents

Contents

Contents

- Pseudocode
- C source code

Contents

- Pseudocode
- C source code

Contents

- Pseudocode
- C source code

Contents

- Pseudocode
- C source code

Contents

- Pseudocode
- C source code

Contents

- Pseudocode
- C source code

1. Why is array preferred?
2. Those elementary things
3. How to implement
4. Easter egg

Section 2

Why is array preferred?

$\textsc{Parent}(i)$

return $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$

return $2i$

$\textsc{Right}(i)$

return $2i + 1$

$A[1] = A[\textit{heap-size}]$
$\textit{heap-size} = \textit{heap-size} - 1$

$A[i] = \textit{key}$

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

All of them is $\Theta(1)$

$A[1] = A[\mathit{heap\text{-}size}]$
$\mathit{heap\text{-}size} = \mathit{heap\text{-}size} - 1$

$A[i] = \mathit{key}$

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

1 $A[1] = A[heap\text{-}size]$
2 $heap\text{-}size = heap\text{-}size - 1$

1 $A[i] = key$

How can we implement these operations with tree form?
Can we maintain these amazing time costs?

Section 3

Those elementary things

Structure of Tree Node

key
parent
left
right

Structure of Tree Node

key
parent
left
right

PARENT(T)
1 **return** $T.parent$

LEFT(T)
1 **return** $T.left$

RIGHT(T)
1 **return** $T.right$

$\textsc{Parent}(i)$

return $\lfloor i/2 \rfloor$

$\textsc{Left}(i)$

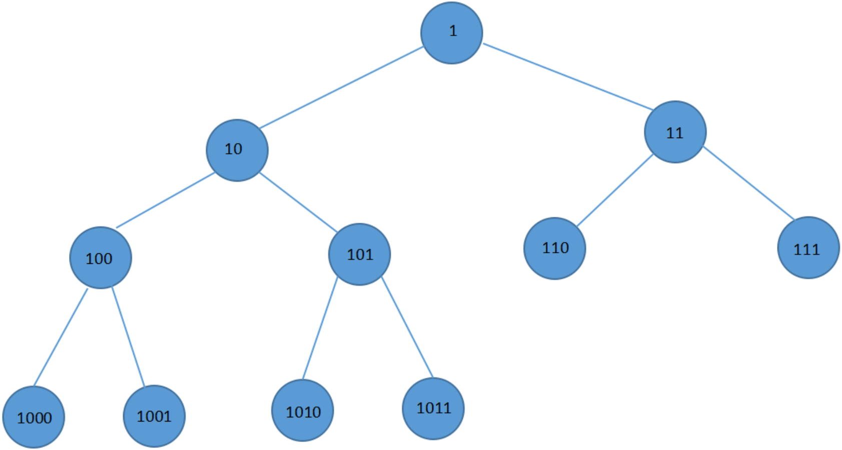
return $2i$

$\textsc{Right}(i)$

return $2i + 1$

$A[1] = A[\mathit{heap\text{-}size}]$
$\mathit{heap\text{-}size} = \mathit{heap\text{-}size} - 1$

$A[i] = \mathit{key}$


1
10
11
100
101
110
111
1000
1001
1010
1011

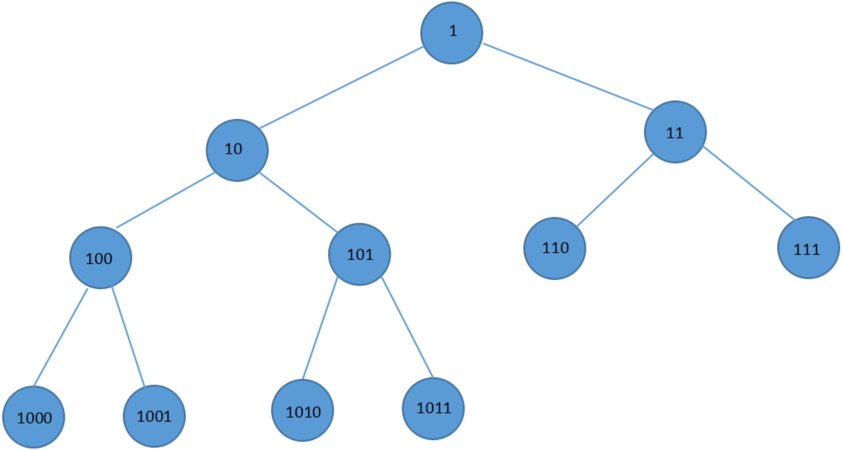


Figure: Heap Node Number and Direction of Path

$$H\begin{cases}\text{HeapTree}\\\text{Total}\\\text{Last}\end{cases}$$

$$H\begin{cases}\text{HeapTree}\\ \text{Total}\\ \text{Last}\end{cases}$$

FINDLAST(H)

```
H.Last = H.HeapTree
index = 0
while (1 << index) ≤ H.total
        Direction[index] =
        ((H.Total) >> index)&1
        index = index + 1
index = index − 2
while index ≥ 0
        if Direction[index] == 1
                Last = Last.right
        else
                Last = Last.left
        index = index − 1
```

FIND-NODE(H, N)

```
result = H.HeapTree
index = 0
while (1 << index) ≤ N
    Direction[index] = (N >> index)&1
    index = index + 1
index = index − 2
while index ≥ 0
    if Direction[index] == 1
        result = result.right
    else
        result = result.left
    index = index − 1
```

```
FIND-NODE(H, N)
result = H.HeapTree
index = 0
while (1 << index) ≤ N
    Direction[index] =
    (N >> index)&1
    index = index + 1
index = index − 2
while index ≥ 0
    if Direction[index] == 1
        result = result.right
    else
        result = result.left
    index = index − 1
```

One way to get random access to the element in a complete binary tree with time cost $\Theta(\log N)$

Section 4

Tree 2 Heap

Subsection 1

MaxHeapify

MaxHeapify

```
MaxHeapify(H)
  Left = H.left
  Right = H.right
  if Left ≠ NIL and Left.key > H.key
      largest = Left
  else
      largest = H
  if Right ≠ NIL and Right.key > largest.key
      largest = Right
  if largest ≠ H
      exchange largest.key and H.key
      MaxHeapify(largest)
```

MAXHEAPIFY

```
MAXHEAPIFY(H)
Left = H.left
Right = H.right
if Left ≠ NIL and Left.key > H.key
    largest = Left
else
    largest = H
if Right ≠ NIL and Right.key > largest.key
    largest = Right
if largest ≠ H
    exchange largest.key and H.key
    MAXHEAPIFY(largest)
```

RIGHT(H)=$H.right$
LEFT(H)=$H.left$

Subsection 2

Build-MaxHeap-FromTree

BUILD-MAXHEAP-FROMTREE

BUILD-MAXHEAP-FROMTREE(T)

if $T ==$ NIL
    return
BUILD-MAXHEAP-FROMTREE($T.left$)
BUILD-MAXHEAP-FROMTREE($T.right$)
MAXHEAPIFY(T)

Recurrence

$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(\log n)$ with $1/3 \le \alpha \le 2/3$

BUILD-MAXHEAP-FROMTREE

BUILD-MAXHEAP-FROMTREE(T)

```
1  if T == NIL
2      return
3  BUILD-MAXHEAP-FROMTREE(T.left)
4  BUILD-MAXHEAP-FROMTREE(T.right)
5  MAXHEAPIFY(T)
```

Recurrence

$T(n) = T(\alpha n) + T((1-\alpha)n) + \Theta(\log n)$ with $1/3 \leq \alpha \leq 2/3$

Solution

$T(n) = O(n)$?

Proof.

1. Claim that $T(n) \geq dn$

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + c\log n \\ &\geq d\alpha n + d(1-\alpha)n + c\log n \\ &= dn + c\log n \\ &\geq dn \end{aligned}$$

2. Claim that $T(n) \leq d_1 n - d_2 \log n$

$$\begin{aligned} T(n) &= T(\alpha n) + T((1-\alpha)n) + c\log n \\ &\leq d_1 n - d_2 \log \alpha n - d_2 \log(1-\alpha)n + c\log n \\ &= d_1 n - d_2 \log n \\ &\quad - (d_2 \log \alpha + d_2 \log(1-\alpha) + d_2 \log n - c\log n) \end{aligned}$$

We only needs d_2 to be big enough to overwhelm $c\log n$

Subsection 3

HEAPSORT

Heapsort

```
HEAPSORT(H)
BUILD-MAXHEAP-FROMTREE(H)
for i = H.total downto 1
    exchange H.HeapTree.key and Last.key
    A[i] = Last.key
    if H.Last.parent.right == NIL
        H.Last.parent.left = NIL
    else
        H.Last.parent.right = NIL
    H.total = H.total - 1
    FINDLAST(H)
    MAXHEAPIFY(H.HeapTree)
```

Section 5

Heap 2 Priority Queue

Subsection 1

HEAPMAX and EXTRACT-HEAPMAX

HEAPMAX(H)

return $H.HeapTree.key$

```
HeapMax(H)
    return H.HeapTree.key
```

```
Extract-HeapMax(H)
if H.HeapTree == NIL
    error "heap underflow"
max = H.HeapTree.key
H.HeapTree.key = H.Last.key
if H.Last.parent.right == NIL
    H.Last.parent.left = NIL
else
    H.Last.parent.right = NIL
H.Total = H.Total − 1
FindLast(H)
MaxHeapify(H.HeapTree)
```

Subsection 2

HEAP-INCREASE-KEY and MAXHEAP-INSERT

HEAP-INCREASE-KEY

```
HEAP-INCREASE-KEY(H, i, key)
target = FIND-NODE(H, i)
if key < target.key
        error "new key is smaller than current key"
while target ≠ H.HeapTree and
target.parent.key < target.key
        exchange target.parent.key and target.key
        target = target.parent
```

MaxHeap-Insert

MaxHeap-Insert(H, key)

```
H.Total = H.Total + 1
parent = Find-Node(H, ⌊H.Total/2⌋)
if parent.left == NIL
    parent.left = Create(−∞)
    H.Last = parent.left
else
    parent.right = Create(−∞)
    H.Last = parent.right
Heap-Increase-Key(H, H.Total, key)
```

Section 6

Easter Egg

Subsection 1

BUILD-COMPLETE-BINARY-TREE-FROM-ARRAY

BUILD-COMPLETE-BINARY-TREE-FROM-ARRAY(T, A, n)

```
if n == 0
    T = NIL
T = CREATETNODE(A[0])
ENQUEUE(Q, T)
index = 1
while index < n
    Parent = DEQUEUE(Q)
    Left = CREATETNODE(A[index])
    Parent.left = Left
    index = index + 1
    ENQUEUE(Q, Left)
if index < n
    Right = CREATETNODE(A[index])
    Parent.right = Right
    index = index + 1
    ENQUEUE(Q, Right)
```

BUILD-COMPLETE-BINARY-TREE-FROM-ARRAY(T, A, n)

```
if n == 0
    T = NIL
T = CREATETNODE(A[0])
ENQUEUE(Q, T)
index = 1
while index < n
    Parent = DEQUEUE(Q)
    Left = CREATETNODE(A[index])
    Parent.left = Left
    index = index + 1
    ENQUEUE(Q, Left)
if index < n
    Right = CREATETNODE(A[index])
    Parent.right = Right
    index = index + 1
    ENQUEUE(Q, Right)
```

Loop invariant?

Subsection 2

BUILD-MAXHEAP-FROM-ARRAY

Build-MaxHeap-From-Array(H, A, n)

1 Build-Complete-Binary-Tree-From-Array ($H.HeapTree, A, n$)
2 Build-MaxHeap-FromTree($H.HeapTree$)

$$\Theta(n) + \Theta(n)$$

A more intuitive BUILD-MAXHEAP-FROM-ARRAY

BUILD-MAXHEAP-FROM-ARRAY(H, A, n)

```
1  for i = 1 to n
2      HEAP-INSERT(H, A[i])
```

Analysis of time cost

A more intuitive Build-MaxHeap-From-Array

```
Build-MaxHeap-From-Array(H, A, n)
1  for i = 1 to n
2      Heap-Insert(H, A[i])
```

Analysis of time cost

$$\sum_{i=1}^{n} \log i = \log n! = \Theta(n \log n)$$


Thank you!!