BinomialHeap Class Reference

Public Member Functions

BinomialHeap ()

Description

Default Constructor, used to create a new empty Binomial Heap H. This operation takes O(1) time

void **BinomialHeap_Insert** (**Node** *)

Description

The BinomialHeap_Insert is used to insert a node n into a binomial heap H This operation has O(lg n) cost More...

void **BinomialHeap_Union** (**BinomialHeap** *)

Description

BinomialHeap_Union is used to meld binomial heaps together and keeping the binomial heap priorities It does so by creting a new heap that is the result of using Binomial_Merge(h1,h2); after that it starts linking the trees depending on degree and keys to keep the min-heap property. The result is H= H1 U H2. This operation has O(lg n) cost More...

Node * **BinomialHeap_extractMin** ()

Description

BinomialHeap_extractMin is used to return the node n with the minimum key k; it deletes it from the heap H. It reverses the children of the extracted node and puts them inside a new heap H `We then call BinomialHeap_Union to meld H and H` together. This operation takes O(lg n) time More...

void **BinomialHeap_decreaseKey** (**Node** *, int)

Description

BinomialHeap_decreaseKey is used to decrese a key k of a node n; It checks if the new key k‘ is less than the actual key k of node n then it 'sifts’ it down; i.e it swaps the node n with its child if it has a child that has key k less than that of node n This operation has O(lg n) cost More...

void **BinomialHeap_deleteNode** (**Node** *)

Description

BinomialHeap_deleteNode deletes a node n from heap H; It does so by simply calling BinomialHeap_decreaseKey and make it the node with the minimum key, and then it calls BinomialHeap_extractMin to take it out This Operation takes O(lg n) time More...

Node * **BinomialHeap_findMin** ()

Description

BinomialHeap_findMin() is used to find the node n with the minimum key k; This Operation takes O(lg n) time More...

Node * **getRoot** ()

Description

getter for the root of the heap More...

Private Member Functions

void **Binomial_Link** (**Node** *, **Node** *)

Description

Binomial_Link function is used to link binomial trees of the same degree together; it does so by setting the node with greatest key as a child to the other node. This operation has O(1) runtime complexity More...

Node * **Binomial_Merge** (**BinomialHeap** *, **BinomialHeap** *)

Description

Binomial_Merge is used to merge the root lists of two binomial heaps h1 and h2; However it does it without linking the trees together, and it orderes the root list in ascending order, depending on the degree of each node. This operation is executed in O(lg n) time More...

Private Attributes

Node * **root**

Pointer to the Root of the heap

Node * **min**

Pointer to the Minimum **Node** in the heap; this is not necessary to define as we can use findMin

Member Function Documentation

◆ Binomial_Link()

```
void BinomialHeap::Binomial_Link ( Node * ,
                                   Node *
                                 )
```

private

Description

Binomial_Link function is used to link binomial trees of the same degree together; it does so by setting the node with greatest key as a child to the other node. This operation has O(1) runtime complexity

Parameters

1st Node	The node with greatest Key
2nd Node	The node with least Key

Pseudocode

```
BINOMIAL-LINK(y, z)
    p[y] ← z
    sibling[y] ← child[z]
    child[z] ← y
    degree[z] ← degree[z] + 1
```

◆ Binomial_Merge()

Node * BinomialHeap::Binomial_Merge (**BinomialHeap** * ,
BinomialHeap *
) private

Description

Binomial_Merge is used to merge the root lists of two binomial heaps h1 and h2; However it does it without linking the trees together, and it orderes the root list in ascending order, depending on the degree of each node. This operation is executed in O(lg n) time

Parameters

1st Heap	the first Binomial Heap
2nd Heap	the second Binomial Heap

Returns

Heap with root list that is the result of merging the root lists of two Binomial Heaps H1 and H2

Pseudocode

```
BINOMIAL-Merge(H, H`)
a = head[H1]
b = head[H2]
head[H1] = Min - Degree(a, b)
if head[H1] = NIL
    return
if head[H1] = b
    then b = a
a = head[H1]
while b <> NIL
    do if sibling[a] = NIL
        then sibling[a] = b
            return
    else if degree[sibling[a]] < degree[b]
            then a = sibling[a]
    else c = sibling[b]
            sibling[b] = sibling[a]
            sibling[a] = b
            a = sibling[a]
            b = c
```

◆ BinomialHeap_decreaseKey()

void BinomialHeap::BinomialHeap_decreaseKey (**Node** * ,
int
)

Description

BinomialHeap_decreaseKey is used to decrese a key k of a node n; It checks if the new key k‘ is less than the actual key k of node n then it 'sifts’ it down; i.e it swaps the node n with its child if it has a child that has key k less than that of node n This operation has O(lg n) cost

Parameters

- **n** the **Node** n, whose key we want to decrese
- **k** new key that we want to assign to node n

Pseudocode

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
if k > key[x]
   then error "new key is greater than current key"
key[x] ← k
y ← x
z ← p[y]
while z≠ NIL and key[y] < key[z]
    do exchange key[y] ↔ key[z]
       ▸ If y and z have satellite fields, exchange them, too.
       y ← z
       z ← p[y]
```

◆ BinomialHeap_deleteNode()

void BinomialHeap::BinomialHeap_deleteNode (**Node** *)

Description

BinomialHeap_deleteNode deletes a node n from heap H; It does so by simply calling BinomialHeap_decreaseKey and make it the node with the minimum key, and then it calls BinomialHeap_extractMin to take it out This Operation takes O(lg n) time

Parameters

- **n** **Node** to be deleted

Pseudocode

```
BINOMIAL-HEAP-DELETE(H, x)
BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
BINOMIAL-HEAP-EXTRACT-MIN(H)
```

◆ BinomialHeap_extractMin()

Node * BinomialHeap::BinomialHeap_extractMin ( )

Description

BinomialHeap_extractMin is used to return the node n with the minimum key k; it deletes it from the heap H. It reverses the children of the extracted node and puts them inside a new heap H

`We then call BinomialHeap_Union to meld H and H` together. This operation takes O(lg n) time

Returns

The **Node** n with the minimum Key k

Pseudocode

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
find the root x with the minimum key in the root list of H, and remove x from the
   root list of H @see BinomialHeap_findMin()
H' ← MAKE-BINOMIAL-HEAP()
reverse the order of the linked list of x's children, and set head[H'] to point
   to the head of the resulting list
H ← BINOMIAL-HEAP-UNION(H, H')
return x
```

◆ BinomialHeap_findMin()

Node * BinomialHeap::BinomialHeap_findMin ( )

Description

BinomialHeap_findMin() is used to find the node n with the minimum key k; This Operation takes O(lg n) time

Returns

The **Node** n with the minimum Key k

Pseudocode

```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
   do if key[x] < min
         then min ← key[x]
              y ← x
       x ← sibling[x]
return y
```

◆ BinomialHeap_Insert()

void BinomialHeap::BinomialHeap_Insert (**Node** *)

Description

The BinomialHeap_Insert is used to insert a node n into a binomial heap H This operation has O(lg n) cost

Parameters

n **Node** to be inserted

Pseudocode

```
BINOMIAL-HEAP-INSERT(H, x)
    H'← MAKE-BINOMIAL-HEAP()     //create a new Binomial Heap
    p[x] ← NIL                   //p[x] is the parent of x
    child[x] ← NIL               //child[x] is the child of x
    sibling[x] ← NIL
    degree[x] ← 0
    head[H'] ← x
    H← BINOMIAL-HEAP-UNION(H, H')
```

◆ BinomialHeap_Union()

void BinomialHeap::BinomialHeap_Union (**BinomialHeap** *)

Description

BinomialHeap_Union is used to meld binomial heaps together and keeping the binomial heap priorities It does so by creting a new heap that is the result of using Binomial_Merge(h1,h2); after that it starts linking the trees depending on degree and keys to keep the min-heap property. The result is H= H1 U H2. This operation has O(lg n) cost

Parameters

Heap A binomial Heap to meld with the current heap

Pseudocode

```
BINOMIAL-HEAP-UNION(H1, H2)
H ← MAKE-BINOMIAL-HEAP()
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)        @see Binomial_Merge(BinomialHeap*, BinomialHeap*)
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
   then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
    do if (degree[x] ≠ degree[next-x]) or (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
          then prev-x ← x
               x ← next-x
          else if key[x] ≤ key[next-x]
                  then sibling[x] ← sibling[next-x]
                       BINOMIAL-LINK(next-x, x)     @see Binomial_Link(Node*,Node*);
                  else if prev-x = NIL
                          then head[H] ←next-x▹ Case 4
                          else sibling[prev-x] ← next-x
                       BINOMIAL-LINK(x, next-x)
                       x ← next-x
       next-x ← sibling[x]
return H
```

◆ getRoot()

Node * BinomialHeap::getRoot ( )

Description

getter for the root of the heap

Returns

Root r of Heap H

The documentation for this class was generated from the following file:

- **BinomialHeap.hpp**