Binomial heaps

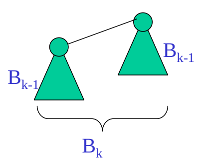
- Last time
  - Heaps
- Today
  - Binomial heaps

Motivation

- Heap
  - Merging two heaps takes time O(n)
  - Insertion takes time O(log n)
- Binomial heap
  - Merging two binomial heaps takes time O(log n)
  - Insertion takes amortized time O(1)

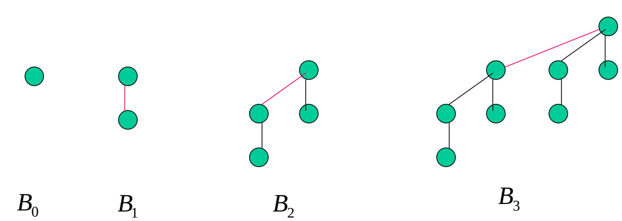
Binomial trees

- A binomial tree, B_k, is an ordered tree defined recursively
  - B_0 consists of a single node
  - B_k consists of two binomial trees, B_{k-1}, that are linked together
    - The root of one is the leftmost child of the other



Binomial tree: the other view/definition

- The Binomial tree, B_k, consists of a root node with *k* children, where the i-th (1 <= i <= k) child is in turn the root of a binomial tree B_{k-i}.



Properties of binomial trees

- The binomial tree B_k
  - Its height is k
  - It has 2^k nodes,
    - Proof by induction

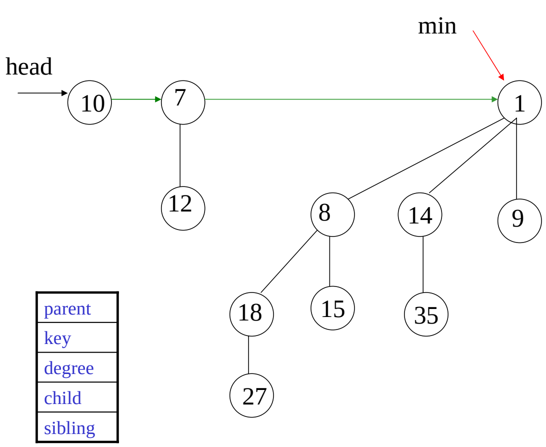
The name "binomial" comes from here

  - of which $\binom{k}{i}$ are at depth *i*, *0 <= i <= k*.
    - Proof by induction using $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$

Binomial heap

- A binomial heap is
  - a set of binomial trees of distinct sizes,
  - and each binomial tree satisfies the heap property:
    - Max_heap: the key of any node is no smaller than its children
    - Min_heap: the key of any node is no bigger than its children
- A binomial heap of size *n* has at most $\lfloor \lg n \rfloor + 1$ binomial trees
- A practical representation of the binomial heap
  - The trees are sorted by their sizes in ascending order.
  - The roots are organized as a linked list for easy insertion and deletion of a binomial tree.
  - Maintaining a pointer pointing to a root with the maximum/minimum key of the binomial heap.
    - This is not implemented in the textbook
  - Each binomial tree use left-child, right sibling representation

A binomial heap



Unite two equal size binomial trees

```
BinomialTree uniteBinomialTrees(B1, B2){
    // B1, B2 are the same size: B1.degree = B2.degree
    if (B1.root().key < B2.root().key) {
        B.copy(B1);
        B.setDegree(B1.degree()+1);
        B2.root().setParent(B1.root());
        B2.root().setSibling(B1.child());
        B.setChild(B2);
    } else {
        // link in the other way
        …
    }
}
```

It takes a time in O(1).

Unite two binomial heaps

```
binomialHeapsUnion(H1, H2)
{
   while (simultaneously following the links in H1 and H2) {
      if there are three degree i trees{ // one from the carry-on
         merge two of them and set it as carry-on;
         add the remainder to H;
      } else if there are two degree i trees {
         merge the two trees;
         set it as carry on;
      } else if there is one degree i tree{
         add it to H;
      }
   }
   add the carry-on if exists to H.
}
```

Assume the result binomial heap contains n nodes. The construction can be done in $\lfloor \lg n \rfloor + 1$ stages. Time in O(log n)

minimum()

- Return the node pointed by the *min* pointer.
  - Cost O(1)
- Without the *min* pointer
  - Traverse the link to find the min
  - Cost O(lg n)

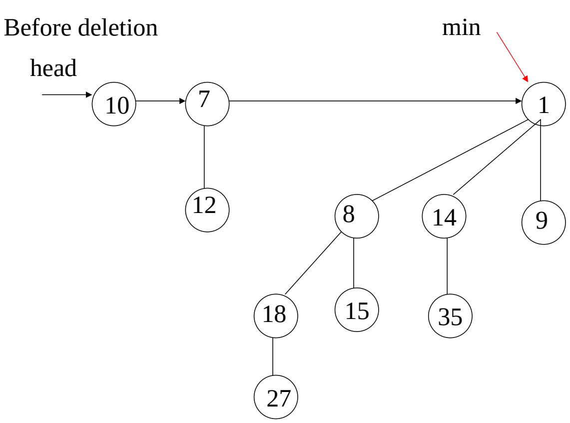
extractMin(): remove the minimum node

```
extractMin(H)
{
    take the min binomial tree B out (H/B);
    remove the root of B;
    join the subtrees of B into a new binomial heap H';
    unite H/B and H';
}
```

Cost: O(log n)

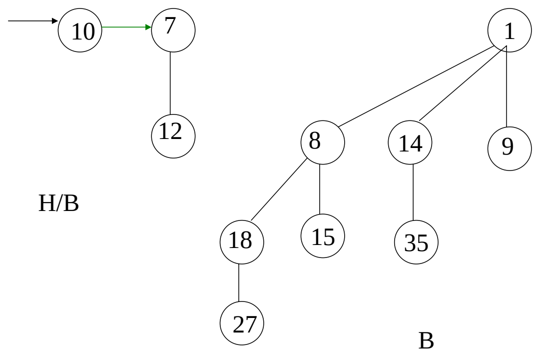
Delete Example: step 0

Before deletion



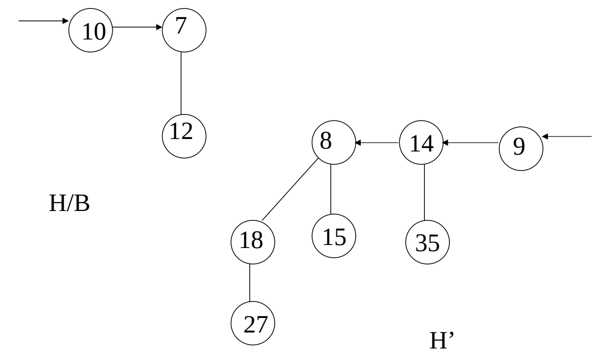
Delete Example: step 1

Take the tree with *min* out



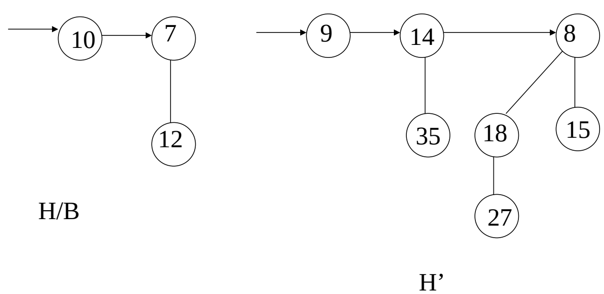
Delete Example: step 2

Join the sub-trees in B as a binomial heap, H’



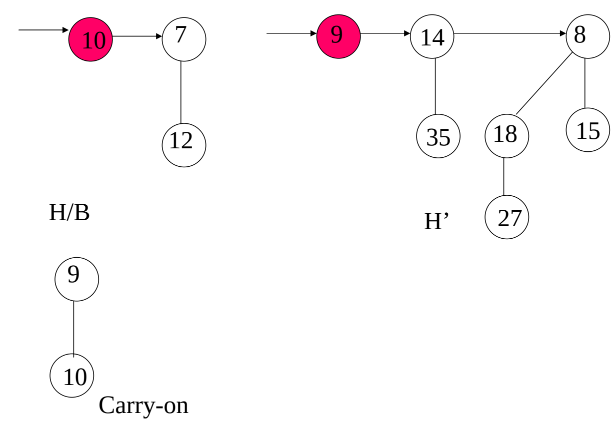
Delete Example: step 2

Join the sub-trees in B as a binomial heap, H’



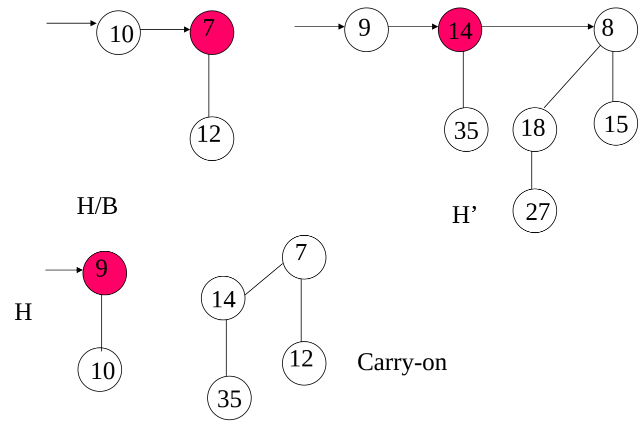
Delete Example: step 3.1

Merge H/B and H’: from left to right



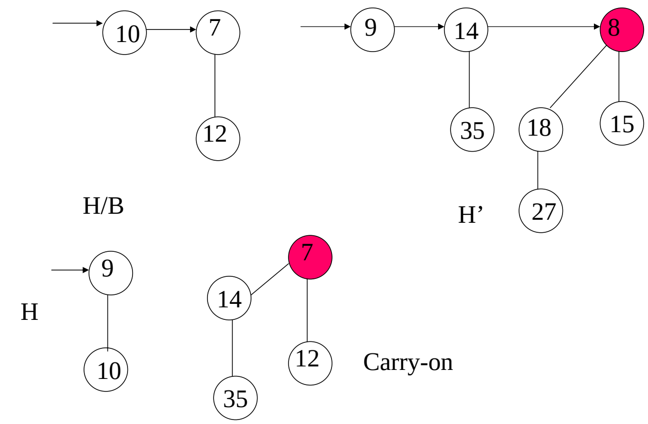
Delete Example: step 3.2

Merge H/B and H': from left to right

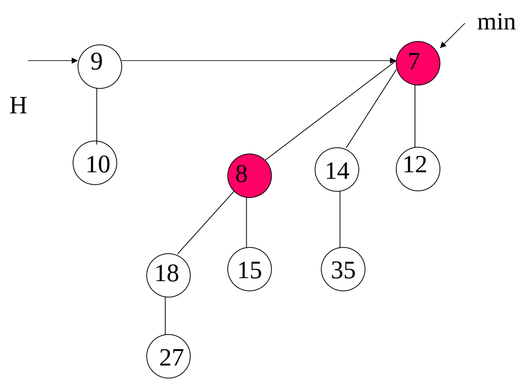


Delete Example: step 3.3

Merge H/B and H': from left to right



Delete Example: step 3.3



decreaseKey

```
public void decreaseKey(Node x, int key)
{
   Node  cur = x;
   Node  parent = x.parent;

   while (parent !=NULL && cur.key < parent.key){
        swap(cur.key, parent.key);
        cur = parent;
        parent = cur.parent;
   }
}
```

Cost?

deleteKey

```
public void deleteKey(Node x)
{
   decreaseKey(x, -∞);
   extractMin();
}
```

Cost?

insert

```
insert(v, H)
{
   make a 1 node binomial tree B0;
   Build a binomial heap H0 that contains B0;
   merge H0 and H;
}
```

insert

```
insert(v, H)
{
   1. make a 1 node binomial tree B_0*;
   2. i = 0;
   3. while (1) {
         if (H include a B_i) {
            remove B_i from H;
            merge B_i* and B_i into a binomial tree B_{i+1}*;
            i++;
         } else
             break;
      }
   4. insert B_i* into the list of roots of H.
}
```

Amortized cost

- What is the amortized cost of insertion when we build a binomial heap of size *n*?
- Accounting trick
  - Deposit a token when creating a tree
  - Withdraw one when deleting a tree
    - Note that we delete a tree at each loop iteration

Priority List

- Use a BinomialHeap H to implement the priority list
  - insert(S, x)
    - H.insert(x)
  - minimum(S)
    - H.minimum()
  - extractMin(L)
    - H.extractMin()
  - decreaseKey(S, x, k)
    - H.decreaseKey(x, k);