Binomial Heaps

- Our ordinary heaps are called *binary* heaps. They are inefficient for UNION.
- These binomial heaps are *mergeable.*
- In order to get compact slides, we will occasionally refer to a binomial heap as *binheap*.

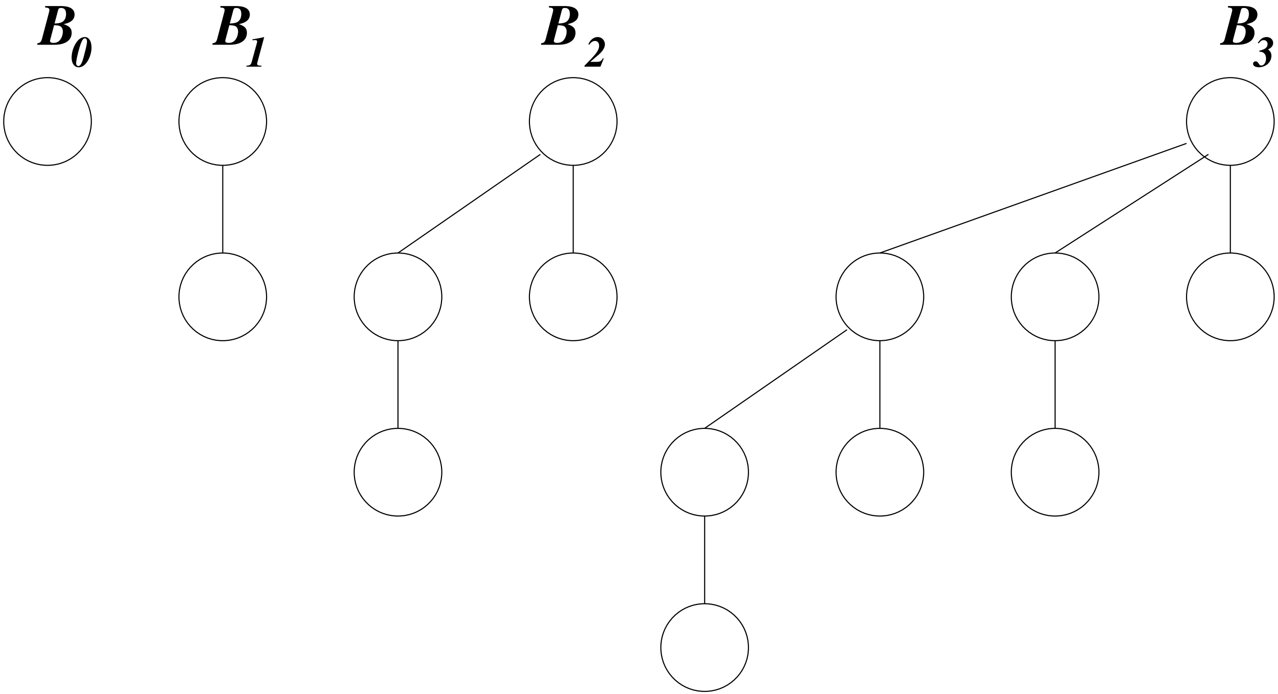
- These are *not* almost complete binary trees; so we cannot use arrays.

- Operations supported:
  - MAKEHEAP() creates an empty binheap;
  - INSERT(H, x) inserts node x into binheap H (x is pre-filled with key and other fields);
  - MINIMUM(H) returns pointer to node with minimum key value;
  - EXTRACTMIN(H) deletes the min-key-node and returns pointer to it (to free it);
  - UNION(H_1, H_2) returns new binheap containing all nodes of H_1, H_2.
    The old binheaps H_1, H_2 are destroyed.

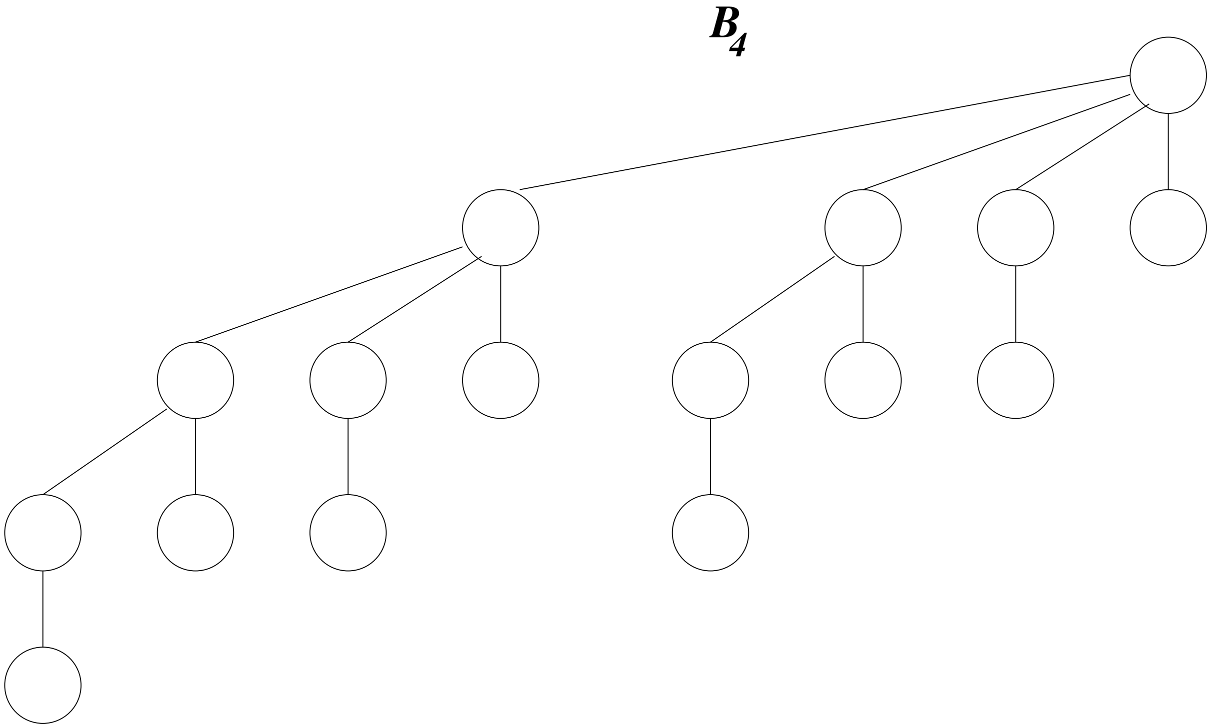
Binomial Tree of order k

- Binomial Tree B_k is an *ordered* tree:
  - B_0 is the tree with a single node;
  - B_k consists of two B_{k-1} binomial trees linked together with the root of one being the left-most child of the root of the other.

Binomial Tree of order k



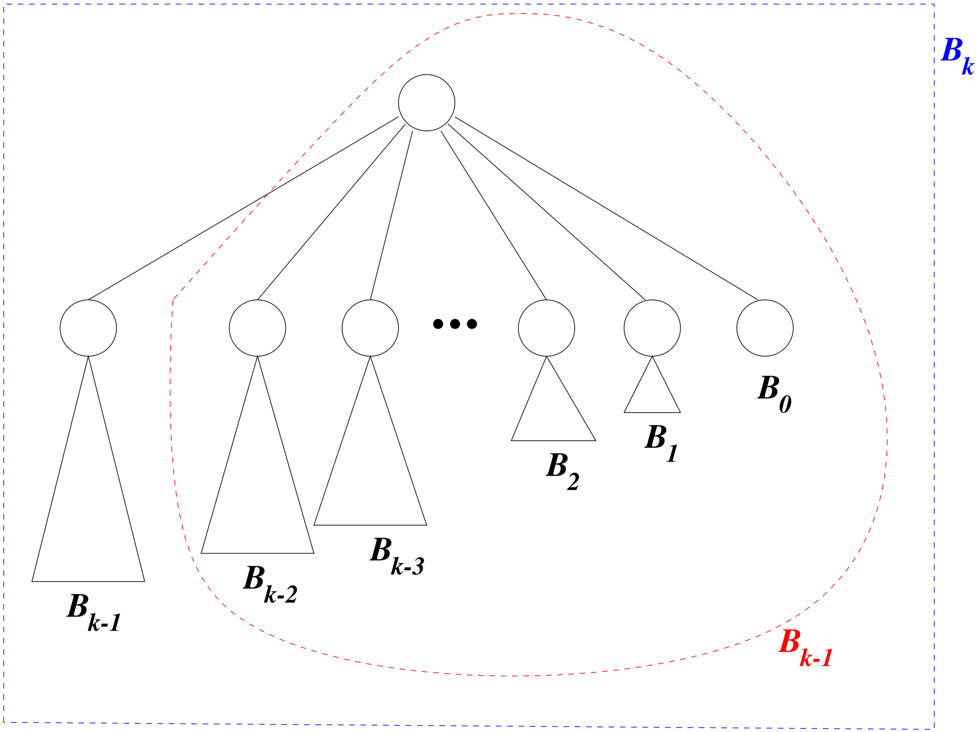
Binomial Tree of order k



Properties of Binomial Tree B_k

1. It has 2^k nodes
2. Its height is k
3. It has exactly $C(k, i) = \binom{k}{i}$ nodes at depth i for $i = 0, 1, .., k$.
4. Its root has degree k, the largest among all nodes; if the children are numbered $0, 1, ..., k-1$ from the right, then child i is the root of a subtree B_i.

Properties of Binomial Tree B_k



- *Proof*: By induction on k.
  Base Case: Consider B_0 ...
  Ind Step: Assume true for B_{k-1}

1. Num nodes in $B_k = 2^{k-1} + 2^{k-1}$ (why?) = ...
2. Height of $B_k = 1 + ...$ (why?) =
3. Let $D(k, i)$ = # nodes in B_k at depth i.

$$D(k, i) = D(k-1, i) + D(k-1, i-1) \text{ (why?)}$$
$$= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

4. By Ind. Hyp., the two highest degree nodes in the component B_{k-1} trees were ... the degree of one of them has increased ...

   Focus on the B_{k-1} component whose root is the root of B_k.
   By Ind. Hyp., its children $0, 1, ..., k-2$ are roots of subtrees $B_0, ..., B_{k-2}$.
   These are also children of the root of B_k with identical numbers.
   The only new child of that root is the leftmost one, numbered $k-1$, and by definition, it is ...

- *Corollary*: For an n-node binomial tree, the maximum degree of any node is $\lg n$.
- The name *binomial* heap comes from property 3 above (binomial coefficients).

- A (min) *Binomial Heap* H is a set of binomial trees such that:

  1. Each binomial tree in H is (min) heap-ordered, i.e., the key of any node is $\geq$ the key of its parent.
  2. There is at most one binomial tree in H whose root has a given degree.

- Hence the root of each binomial tree in H contains the minimum key in that tree.

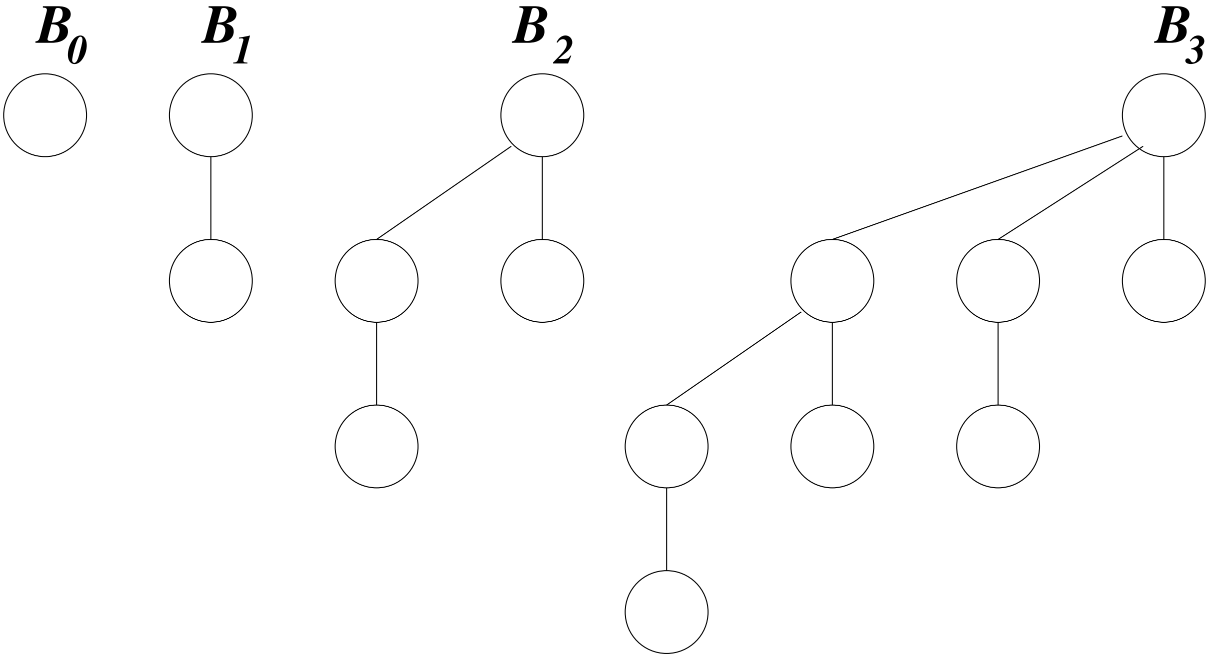
- An n-node binomial heap consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.
  Why?...

... because each component binomial tree contains 2^j nodes for some j and there is only one binary representation of the number n; that representation uses $\lfloor \lg n \rfloor + 1$ bits.

Representing a Binomial Heap

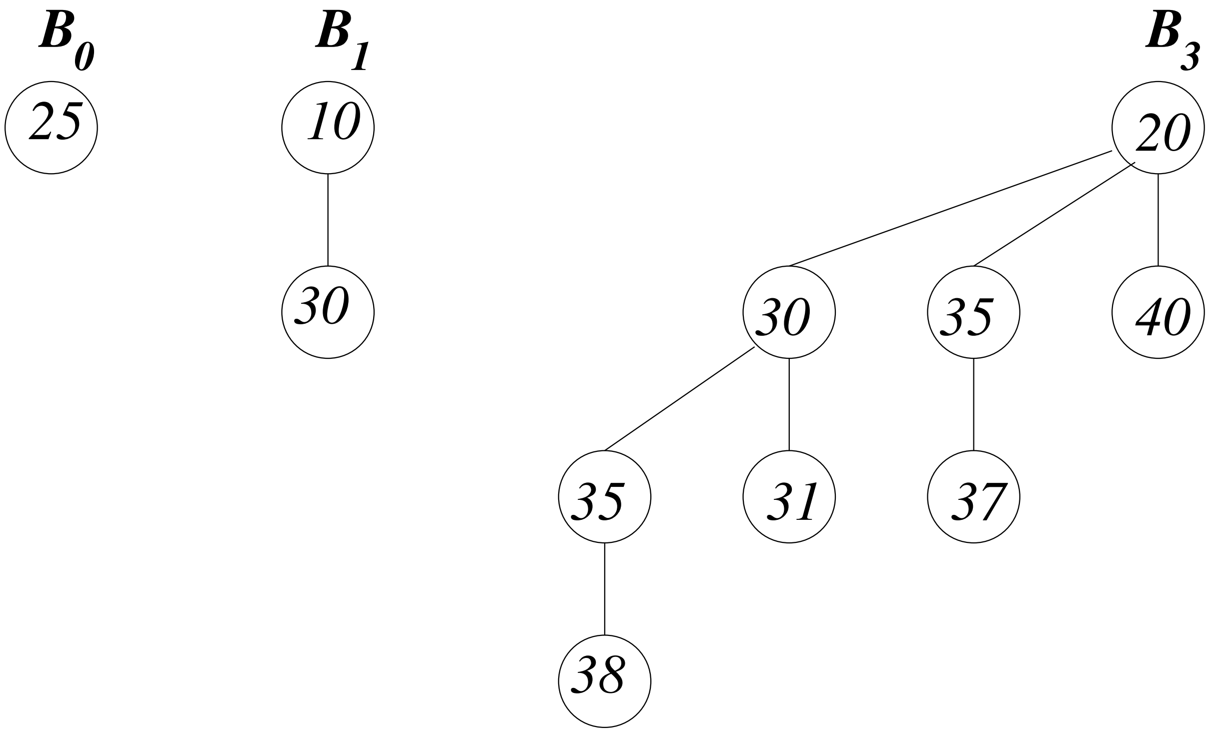
- Let there be 11 nodes, i.e., $n = 11$.
- In binary $11 = 1011$, i.e., $11 = 2^3 + 2^1 + 2^0$
- Then the structure must be...

Representing a Binomial Heap

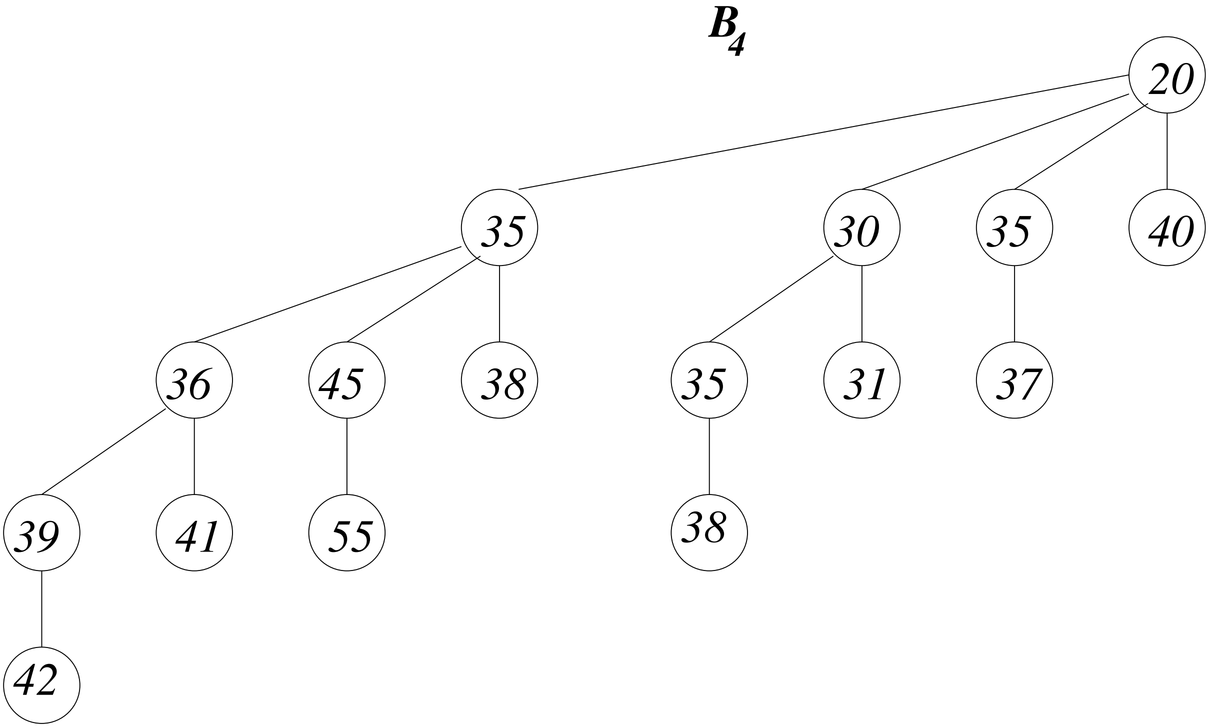


- And with some keys...

Representing a Binomial Heap



Representing a Binomial Heap



Representing a Binomial Heap

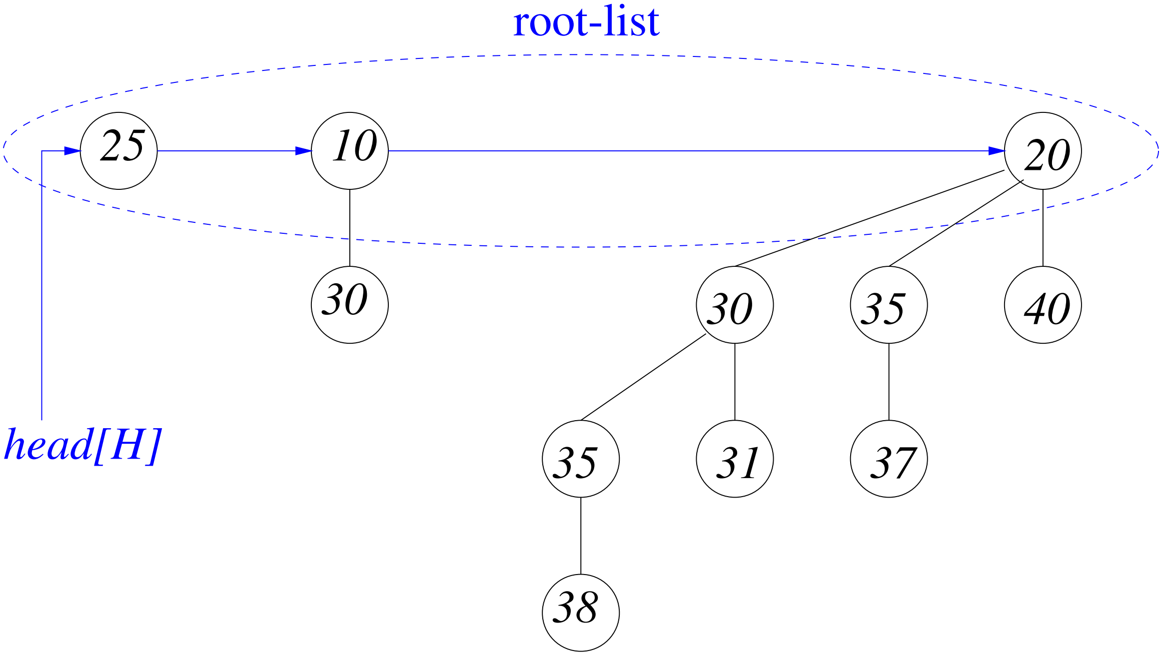
- The heap H is a *set* of binomial trees.
  In the implementation,

  we link the roots of the binomial trees in order of increasing degree.

  This linked list is called the *root list*.

  head[H] points to the head of the list.

Representing a Binomial Heap

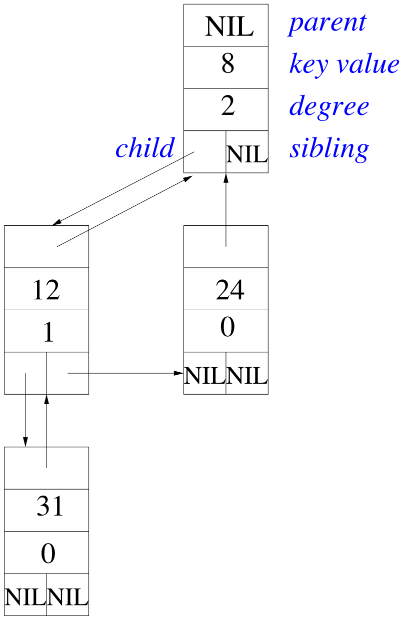


Note: the roots are *NOT* ordered by key.

Representing a Binomial Tree

- To represent a tree with variable number of children, use the *child+sibling* strategy.
- Each node has the fields:
  - parent pointer
  - key value
  - degree of node
  - child pointer
  - sibling pointer

Representing a Binomial Tree



Implementing the Operations

- MAKEBINOMIALHEAP (create an empty binomial heap) is easy.
  It returns an object H of the appropriate type such that *head[H]* = NIL.
  $\Theta(1)$

Implementing the Operations

- $\textsc{BinomialHeapMinimum}(H)$ searches the roots of the binomial trees in H.
  Since they are not ordered by key, all may have to be scanned.
  There at most $\lfloor \lg n \rfloor + 1$ roots.
  Thus, $O(\lg n)$

Implementing the Operations

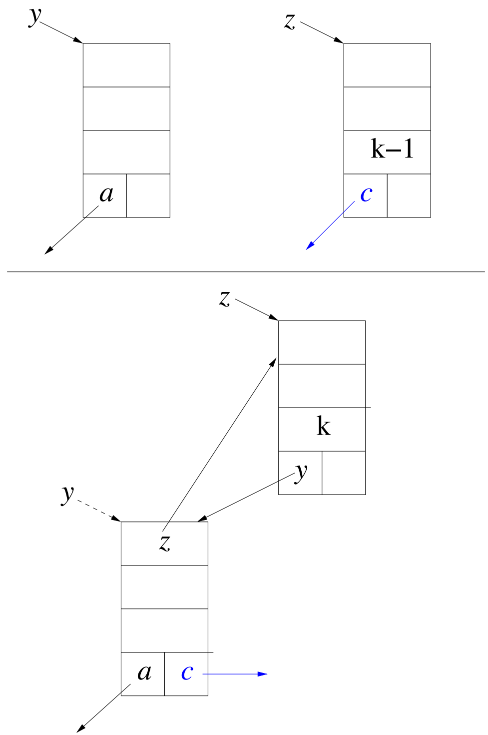

```
BinomialHeapMinimum(H)
    y ← NIL
    x ← head[H]
    min ← ∞ /* assume: no key with value ∞ */
    while x ≠ NIL do
        if key[x] < min then
            min ← key[x]
            y ← x
        x ← sib[x]
    return y
```

- BinomialLink(y, z)
  makes y (actually the node y points to) the left-most child of z (actually the node z points to).

  i.e., y (...) is the new head of the linked list of the children of z (...).

  If y, z were both ptrs to roots of B_k trees, this can result in z becoming a ptr to the root of a B_{k+1} tree.

Implementing the Operations



- BINOMIALHEAPMERGE(H_1, H_2) *merges* the root lists of binomial heaps H_1 and H_2 into one linked list *sorted by degree* in monotonically increasing order.

  This is basically the MERGE operation we saw in MERGESORT.

  The result may not be a binomial heap. (Why?)

- BINOMIALHEAPUNION uses BINOMIALLINK, BINOMIALHEAPMERGE ...

Implementing the Operations

BINOMIALHEAPUNION(H_1, H_2)

$H \leftarrow$ MAKEBINOMIALHEAP$()$

head[H] $\leftarrow$ BINOMIALHEAPMERGE(H_1, H_2)

$\triangleright$ free H_1, H_2

if *head[H]* = NIL **then return** H

prev-x $\leftarrow$ NIL

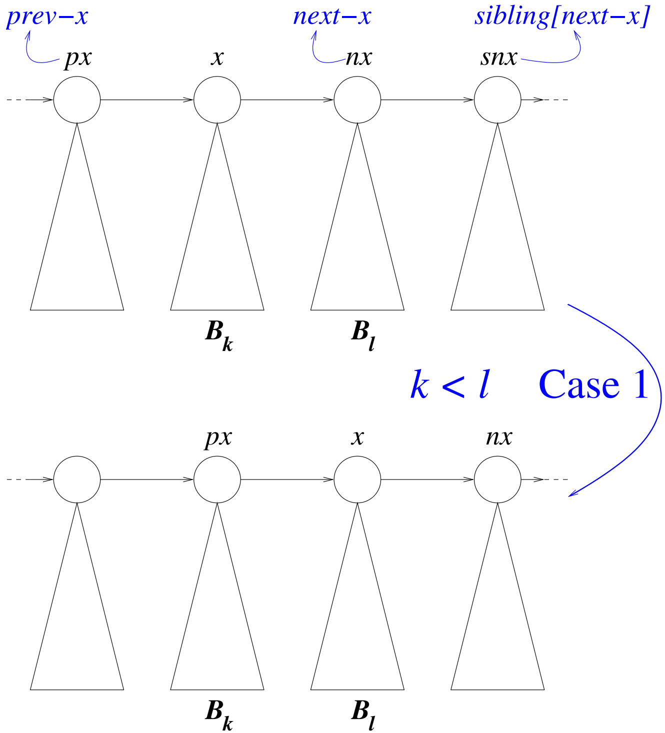
$x \leftarrow$ *head[H]*

next-x $\leftarrow$ *sibling[x]*

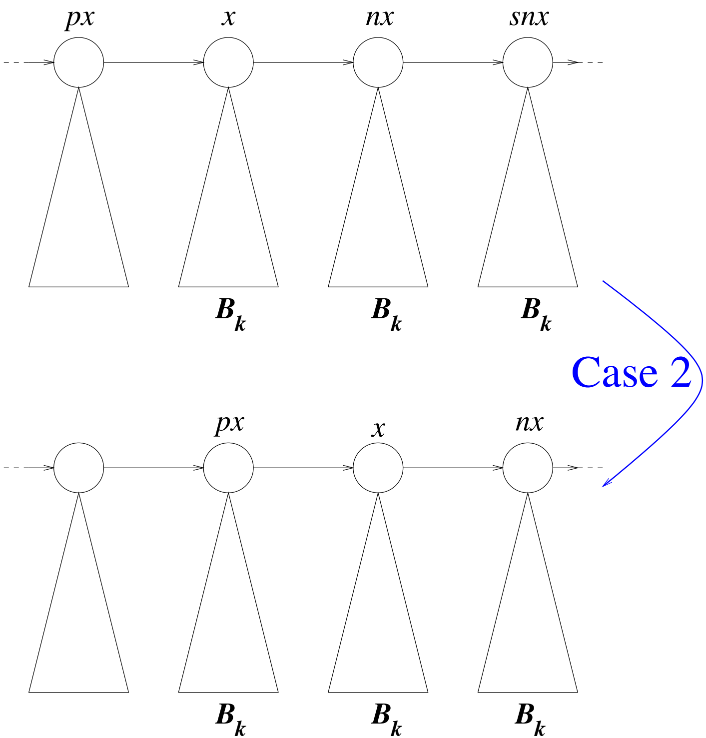
Implementing the Operations


```
while next-x ≠ NIL do
    if (d[x] ≠ d[next − x]) or
        (sibling[next-x] ≠ NIL and
            degree[sibling[next-x]] = degree[x]) ▷  *** Cases 1,2
    then
        prev-x ← x
        x ← next-x
    else if key[x] ≤ key[next-x] ▷ *** Case 3 else Case 4
        then
            sibling[x] ← sibling[next-x]
            BinomialLink(next-x,x)
        else if prev-x = NIL
                then head[H] ← next-x
                else sibling[prev-x] ← next-x
            BinomialLink(x,next-x)
            x ← next-x
    next-x ← sibling[x]
return H
```

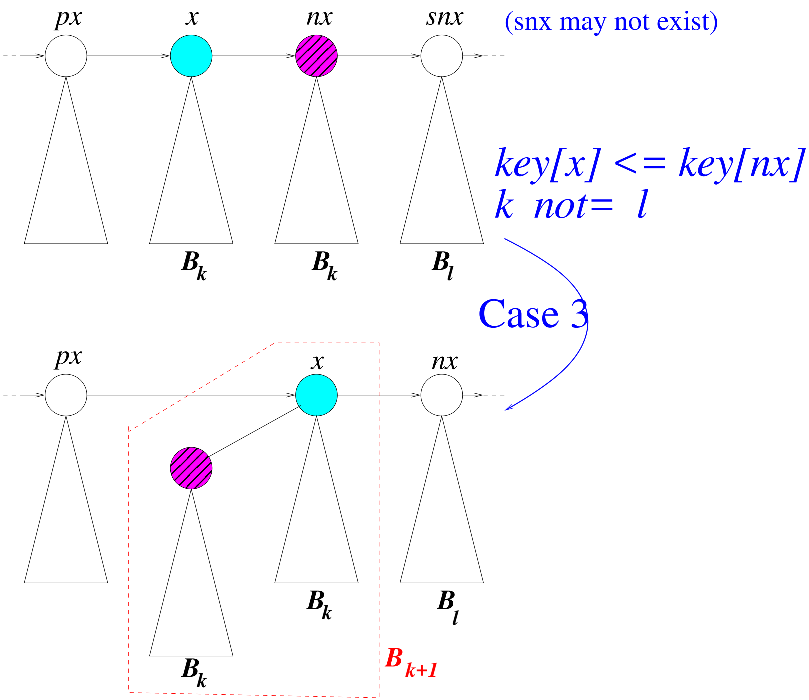
Implementing the Operations



Implementing the Operations

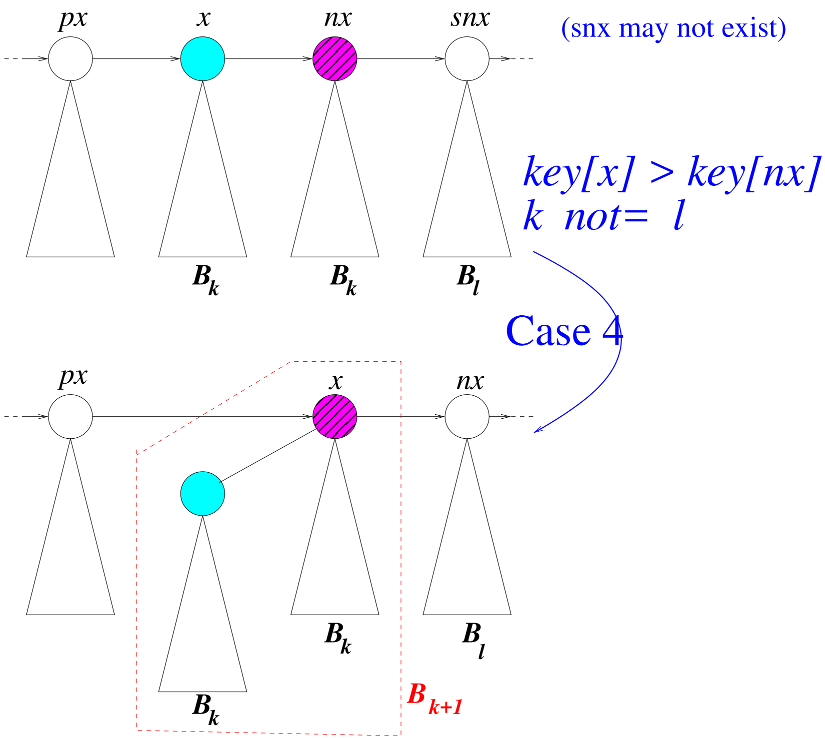


Implementing the Operations



Note: x does not change.

Implementing the Operations



Implementing the Operations

- Let H_1, H_2 have n_1, n_2 nodes respectively.
  Let $n = n_1 + n_2$.

  Number of roots in $H_1 \leq \lfloor \lg n_1 \rfloor + 1$.
  Number of roots in $H_2 \leq \lfloor \lg n_2 \rfloor + 1$.

  Right after the call of BINOMIALHEAPMERGE:
  Number of roots in $H \leq \lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$
  $\leq 2\lfloor \lg n \rfloor + 2$

  So, BINOMIALHEAPMERGE costs time $O(\lg n)$

Each while-loop iteration takes $O(1)$ time (why?)

Number of while-loop iterations

$$\leq \lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$$ (why?)

...

Thus, the running time of $\textsc{BinomialHeapUnion}(H_1, H_2)$ is $O(\lg n)$

Implementing the Operations

- BinomialHeapInsert(H, x)
  creates a one-node binomial heap H' containing x and then calls BinomialHeapUnion(H, H').
  $O(\lg n)$

- $\textsc{BinomialHeapExtractMin}(H)$:
  - finds the root x with the minimum key in the root-list of H and removes that subtree from H.
  - It next creates a new binomial heap whose root list is the *reverse of the children of x* (smaller trees need to come first in the root list).
  - These two binomial heaps are then unioned.

Each step takes $O(\lg n)$ time.
So $O(\lg n)$.

Implementing the Operations