COMP9024: Data Structures and Algorithms

Priority Queues and Disjoint Set Union-Find Data Structures

Contents

- Priority queue ADT
- Heap-based priority queues
- Binomial Heaps
- Disjoint set union-find data structures and algorithms

Priority Queue ADT

- A priority queue stores a collection of items.
- Each **item** is a pair (key, value), where key is the priority of the item.
- Main operations of the Priority Queue ADT:
  - Insert(k, x)
    Inserts an item with key k and value x.
  - RemoveMin() (RemoveMax())
    Removes and returns the item with smallest key (largest key).
    We consider RemoverMin() only.
    Implementation of RemoveMax() is similar.
- Additional operations
  - Min() (Max())
    returns, but does not remove, an entry with smallest key (largest key)
  - Size(), IsEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which a total order is defined.
- Two distinct entries in a priority queue can have the same key.
- Mathematical concept of total order relation $\leq$
  - Reflexive property:
    $x \leq x$
  - Antisymmetric property:
    $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property:
    $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements:
  1. Insert the elements one by one with a series of Insert operations.
  2. Remove the elements in sorted order with a series of RemoveMin operations.
- The running time of this sorting algorithm depends on the priority queue implementation.

```
Algorithm PQ-Sort(S)
 Input sequence S
 Output sequence S sorted  in
   non-decreasing order
    { Create an empty priority queue P;
      while (¬IsEmpty (S) )
        { e = RemoveFirst (S);
          Insert (P, e) ;
        }
      while ( ¬IsEmpty(P) )
        { e = RemoveMin(P);
          InsertLast(S, e);
        }
    }
```


List-based Priority Queue

- Implementation with an unsorted list:

- Performance:
  - Insert takes $O(1)$ time since we can insert the item at the beginning or end of the list.
  - RemoveMin and Min take $O(n)$ time since we have to traverse the entire list to find the smallest key.

- Implementation with a sorted list:

- Performance:
  - Insert takes $O(n)$ time since we have to find the place where to insert the item.
  - RemoveMin and Min take $O(1)$ time, since the smallest key is at the beginning.

Selection-Sort

- Selection-sort is a variation of PQ-sort where the priority queue is implemented with an unsorted list.
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time.
  2. Removing the elements in sorted order from the priority queue with n RemoveMin operations takes time proportional to
     $$1 + 2 + \ldots + n$$
- Selection-sort runs in $O(n^2)$ time.

7

Selection-Sort Example

	List *S*	***Priority Queue*** *P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
.	.	.
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

8

Insertion-Sort

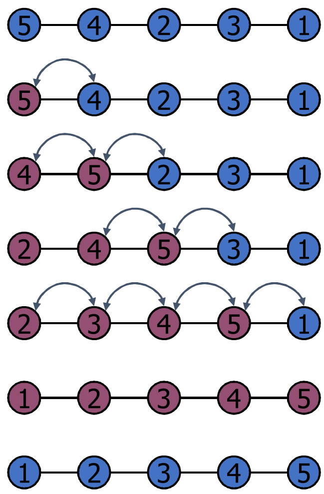
- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted list.
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n Insert operations takes time proportional to
     $1 + 2 + \ldots + n$
  2. Removing the elements in sorted order from the priority queue with a series of n RemoveMin operations takes $O(n)$ time.
- Insertion-sort runs in $O(n^2)$ time.

Insertion-Sort Example

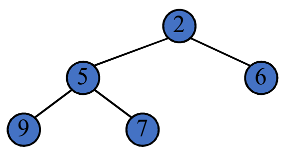
	***List** S*	***Priority queue** P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place.
- A portion of the input list itself serves as the priority queue.
- For in-place insertion-sort
  - We keep sorted the initial portion of the list.
  - We can use swaps instead of modifying the list.



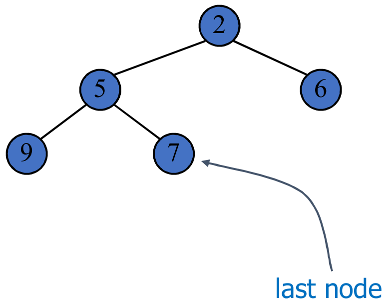
Heaps



Heaps

- A min-heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
    - at depth $h-1$, all the nodes are as far left as possible
- The last node of a heap is the rightmost node of depth h.

- A max-heap satisfies a different heap order property:
  - For every node v other than the root, $key(v) \leq key(parent(v))$
- We consider min-heap only

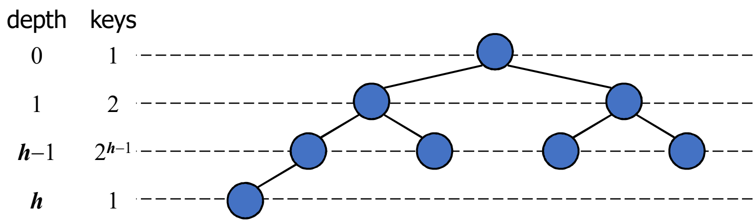


Height of a Heap

- Theorem: A heap storing n keys has height $O(\log n)$.

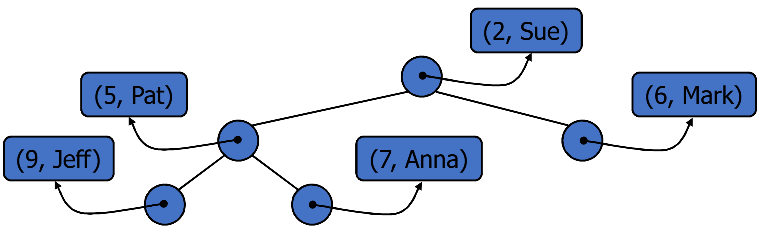
Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys.
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$.
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$.



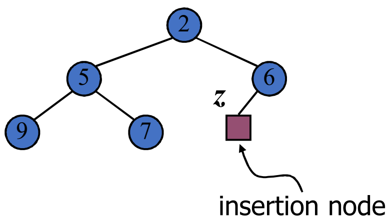
Heaps and Priority Queues

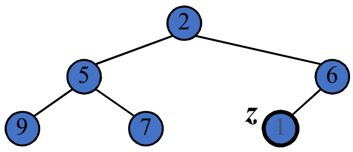
- We can use a heap to implement a priority queue.
- We store a (key, element) item at each node.
- We keep track of the position of the last node.
- For simplicity, we show only the keys in the pictures.



Insertion into a Heap

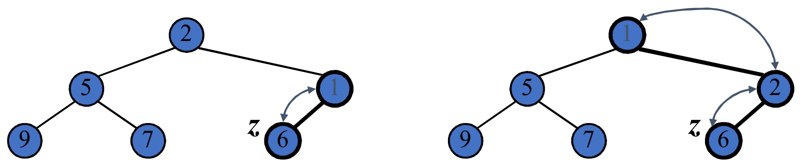
- Operation Insert of the priority queue ADT corresponds to the insertion of a key k to the heap.
- The insertion algorithm consists of three steps:
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)





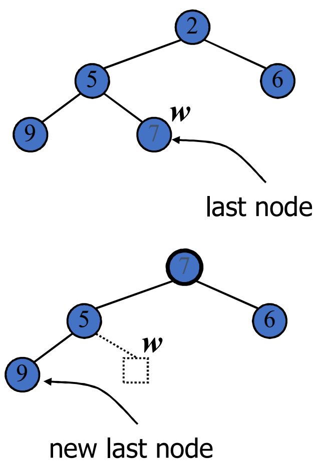
Upheap

- After the insertion of a new key $\boldsymbol{k}$, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping $\boldsymbol{k}$ along an upward path from the insertion node.
- Upheap terminates when the key $\boldsymbol{k}$ reaches the root or a node whose parent has a key smaller than or equal to $\boldsymbol{k}$.
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time.



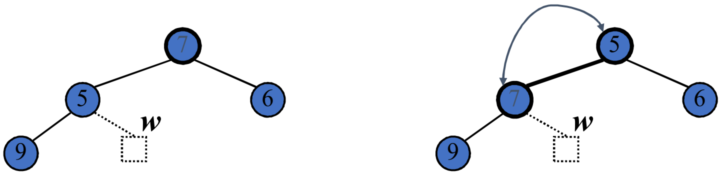
Removal from a Heap

- Method RemoveMin of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node $\boldsymbol{w}$
  - Remove $\boldsymbol{w}$
  - Restore the heap-order property (discussed next)



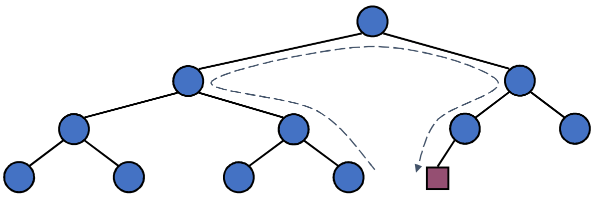
Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root.
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k.
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time.



Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes:
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until the next node is null.
- Similar algorithm for updating the last node after a removal.

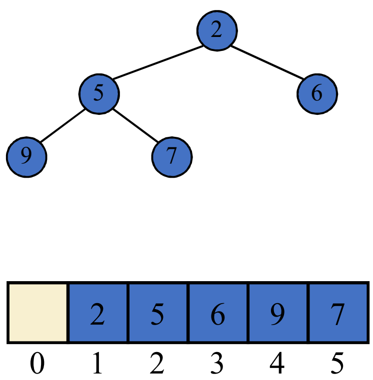


Heap-Sort

- Consider a priority queue with $\boldsymbol{n}$ items implemented by means of a heap
  - The space used is $O(n)$
  - Operations Insert and RemoveMin take $O(\log n)$ time
  - Operations Size, IsEmpty, and Min take $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of $\boldsymbol{n}$ items in $O(n \log n)$ time.
- The resulting algorithm is called heap-sort.
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.

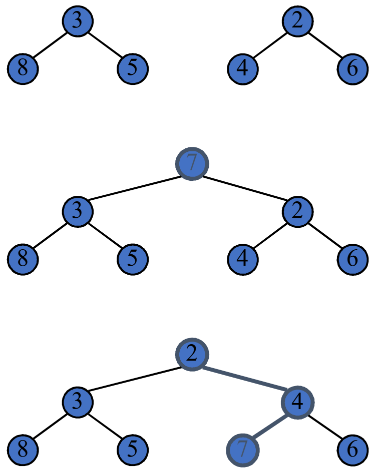
Array-based Heap Implementation

- We can represent a heap with $\boldsymbol{n}$ keys by means of an array of length $n + 1$.
- For the node at rank $\boldsymbol{i}$
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored.
- The cell of at rank 0 is not used.
- Operation Insert corresponds to inserting at rank $n + 1$.
- Operation RemoveMin corresponds to removing at rank $\boldsymbol{n}$.
- Yields in-place heap-sort.



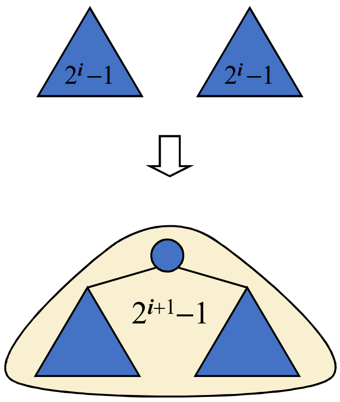
Merging Two Heaps

- We are given two two heaps and a key ***k***.
- We create a new heap with the root node. storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property.

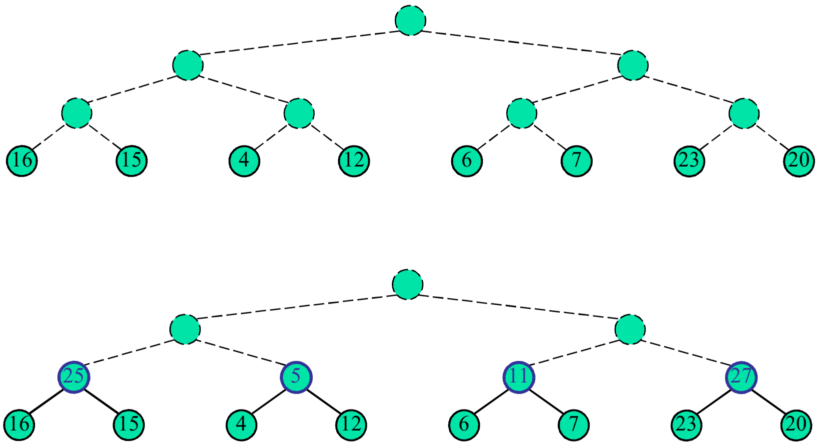


Bottom-up Heap Construction

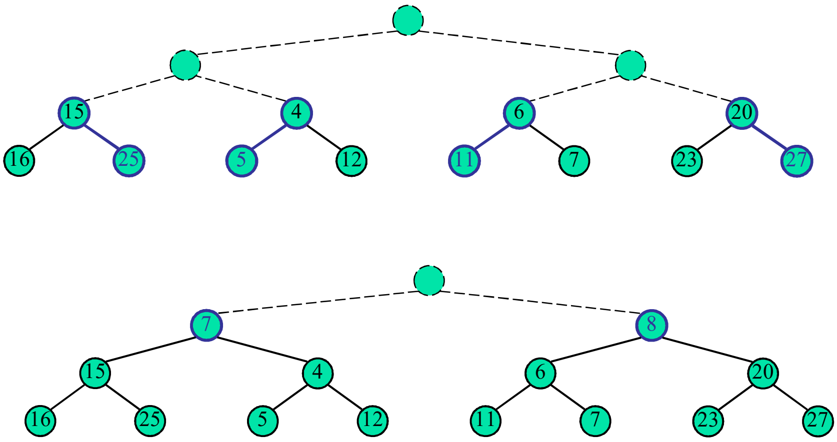
- We can construct a heap storing ***n*** given keys in using a bottom-up construction with $\log n$ phases.
- In phase ***i***, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys.



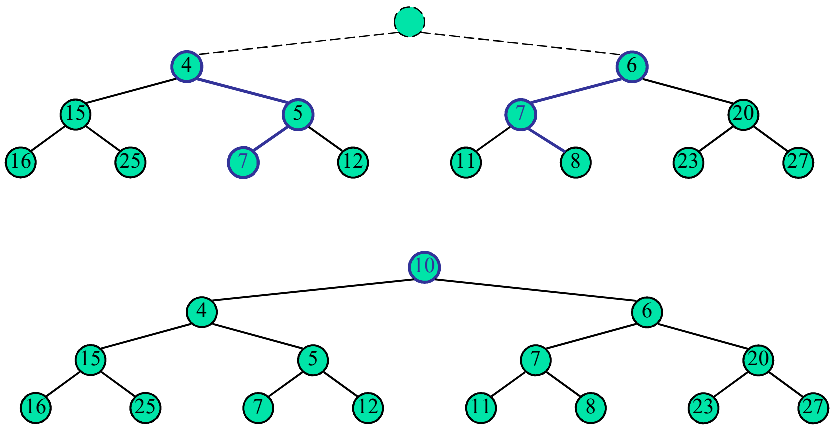
Example (1/4)



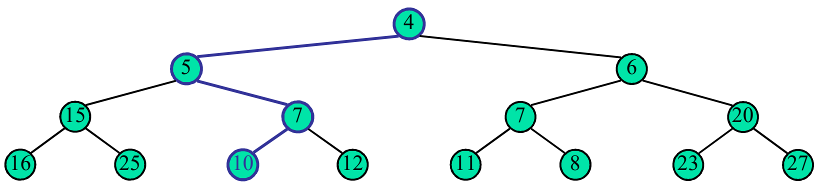
Example (2/4)



Example (3/4)

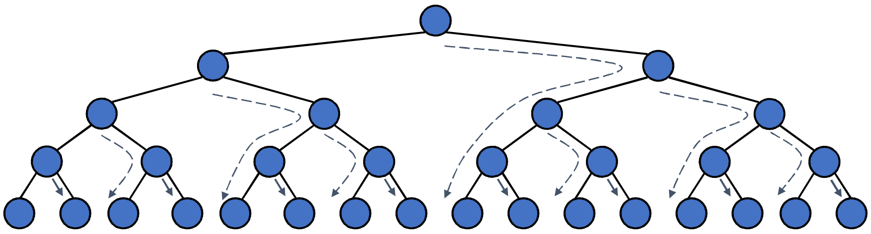


Example (4/4)

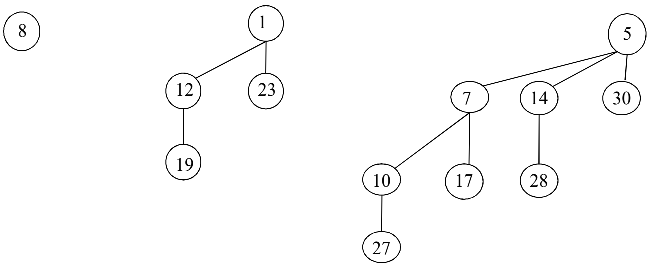


Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$.
- Thus, bottom-up heap construction runs in $O(n)$ time.
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort.



Binomial Heaps



Merge Two Heaps with Different Sizes

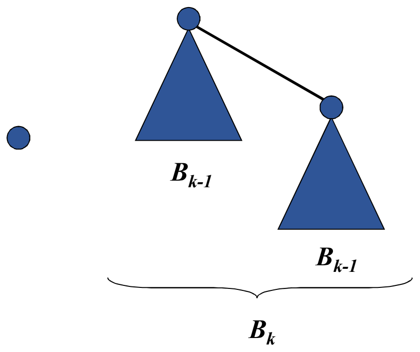
Merge(H1,H2): Merge two heaps H1 and H2 with sizes m and n.

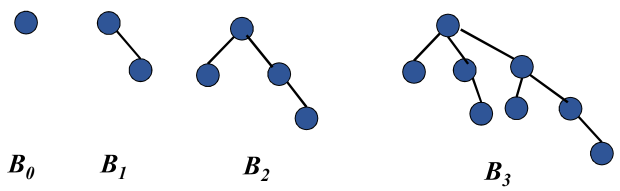
Algorithm 1: Insert each entry from H1 and H2 into a new heap:
Running time: O((m+n) log (m+n))
Algorithm 2: Use the bottom-up heap construction algorithm
Running Time: O(m+n)

Can we merge two heaps in O(log(m+n)) time?

Binomial Trees

- Recursive Definition of Binomial tree B_k of height k:
  - B_0 = single root node
  - B_k = Attach B_{k-1} to root of another B_{k-1}





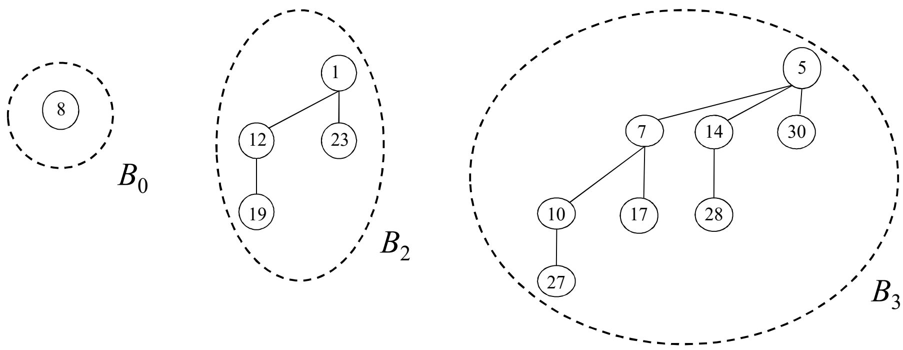
Properties of Binomial Trees

For the binomial tree B_k:

1. There are 2^k nodes
2. The height of B_k is k
3. There are exactly $\binom{k}{i}$ (binomial coefficient) nodes at depth i for i = 0, 1, …, k

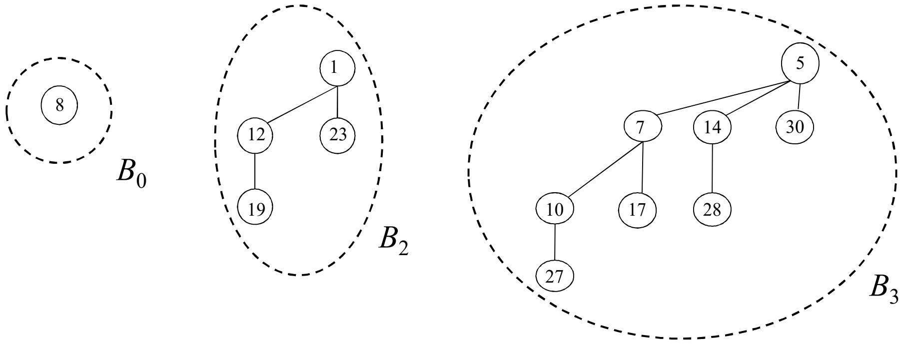
Binomial Heaps

- A binomial heap H_k is a set of binomial trees $B_0, B_1, \ldots, B_k$ where each binomial tree is heap-ordered:
  - ➢ The key of each node ≥ the key of the parent
- The root of each binomial tree in H_k contains the smallest key in that tree.



findMin()

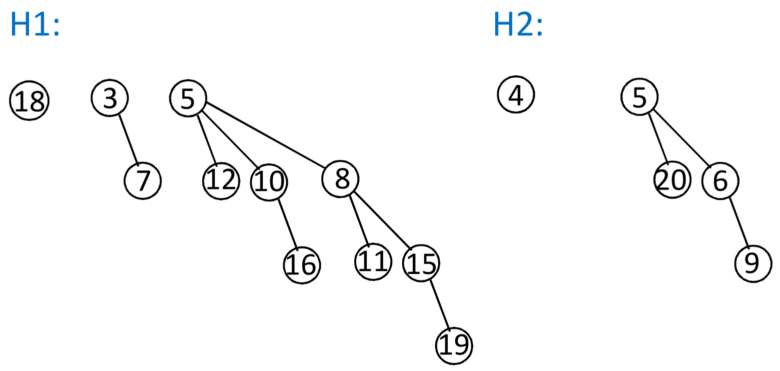
- Traverse all the roots, taking O(log n) time



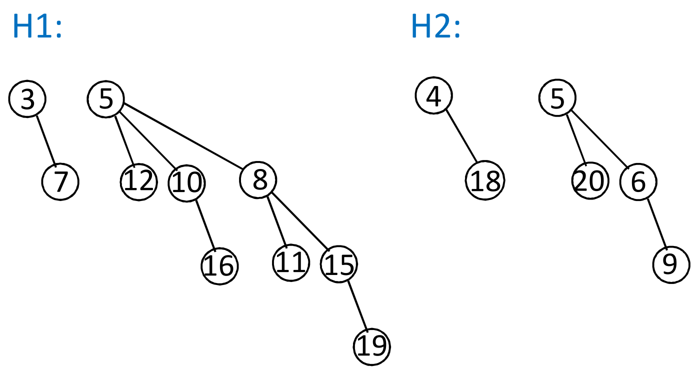
Merge Two Binomial Heaps (1/6)

- Key ideas: merge individual pairs of heaps with the same height
- Steps for merging two binomial heaps:
  1. Create a new empty binomial heap
  2. Start with B_k for the smallest k
  3. If there is only one B_k, add B_k to the new binomial heap and go to Step 3 with k = k + 1
  4. Merge two B_k's into a new B_{k+1} by making the root with a larger key the child of the other root. Go to Step 3 with k = k + 1
- Time complexity: O(log (m+n)), where m and n are the sizes of two heaps.

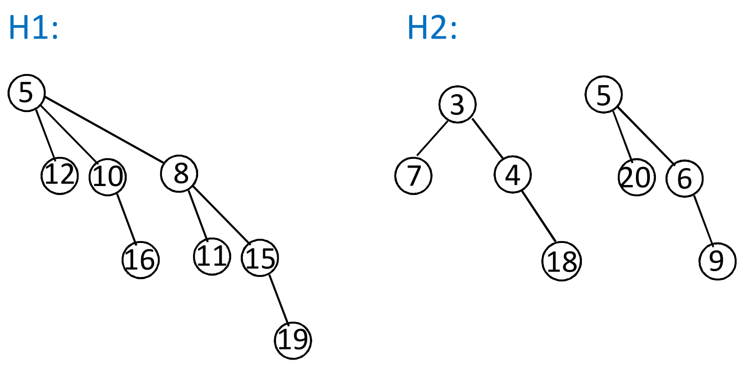
Merge Two Binomial Heaps (2/6)



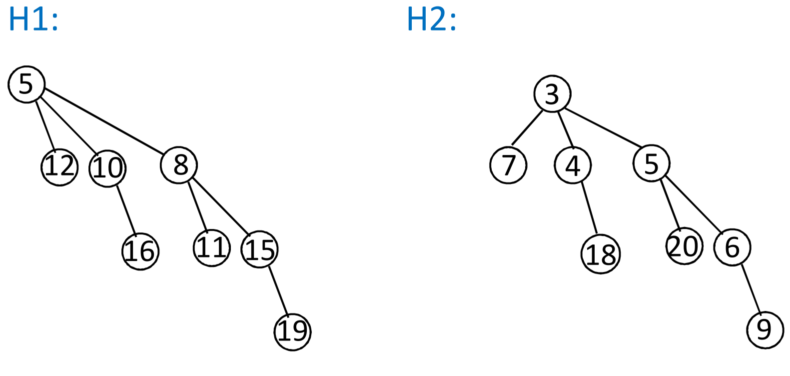
Merge Two Binomial Heaps (3/6)



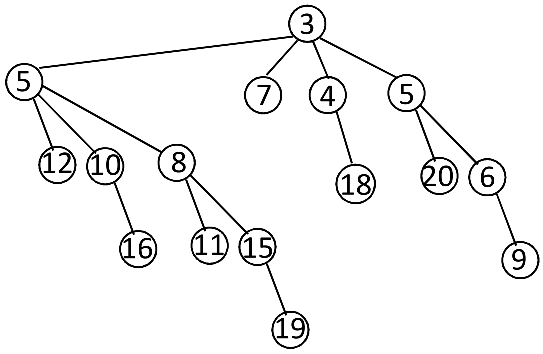
Merge Two Binomial Heaps (4/6)



Merge Two Binomial Heaps (5/6)



Merge Two Binomial Heaps (6/6)



Insertion

- Create a single node tree B_0 with the new item and merge with the existing heap
- Time complexity: O(log n)

How Many Binomial Trees in a Binomial Heap?

Consider a binomial heap with n nodes

- Convert n into a binary number b_k b_{k-1} … b_1 b_0
- If $b_i \neq 0$ (i=0, 1, …, k), the binomial tree B_i is not empty

Example: n=28.

- 28=16+8+4=11100. So k=4. The binomial heap consists of B_4 (16 nodes), B_3 (8 nodes) and B_2 (4 nodes).

removeMin() (1/4)

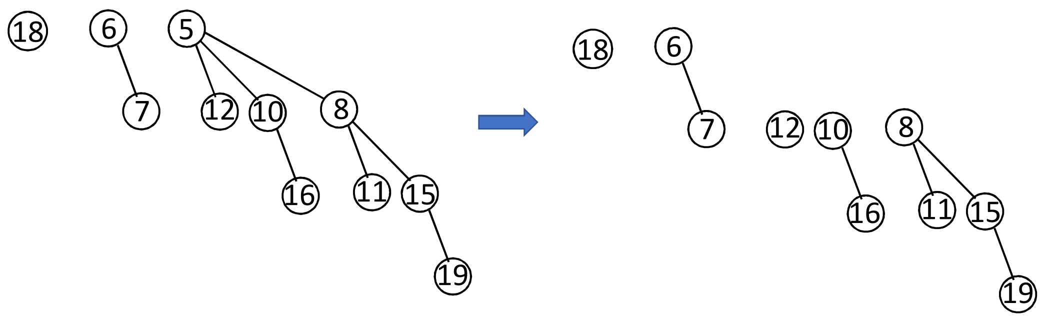
Steps:

1. Find the tree B_k with the smallest root
2. Remove B_k from the heap
3. Keep the entry stored at the root of B_k (return value) and remove the root of B_k (now we have a new forest B_0, B_1, …, B_{k-1})
4. Merge this new forest with remainder of the original
5. Return the entry with the min key

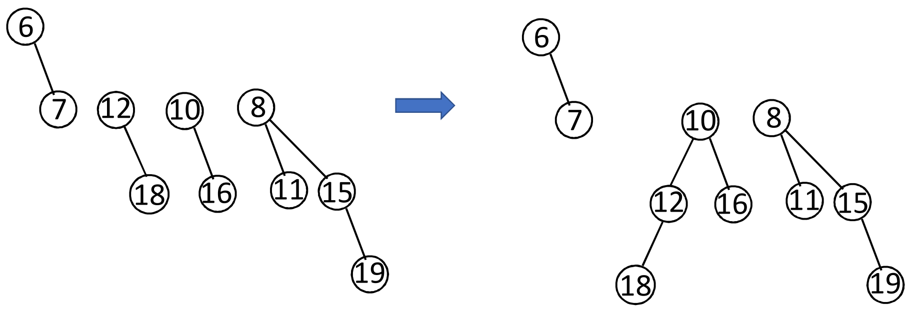
Run time analysis:

- Step 1 is O(log n), Step 2 and Step 3 are O(1), and Step 4 is O(log n)
- Total time complexity is O(log n)

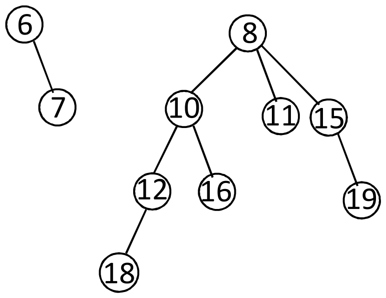
removeMin() (2/4)



removeMin() (3/4)



removeMin() (4/4)



Disjoint Set Union-Find Structures

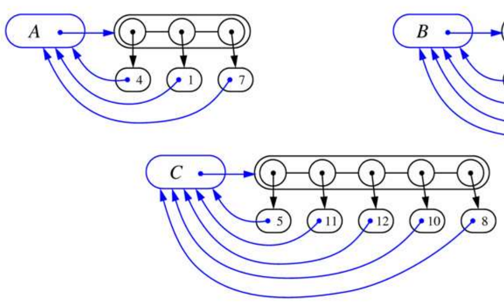
48

Disjoint Set Union-Find Operations

- MakeSet(x): Create a singleton set containing the element x and return the position storing x in this set.
- Union(A,B): Return the set A U B, destroying the old A and B.
- Find(e): Return the set containing the element e.

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name



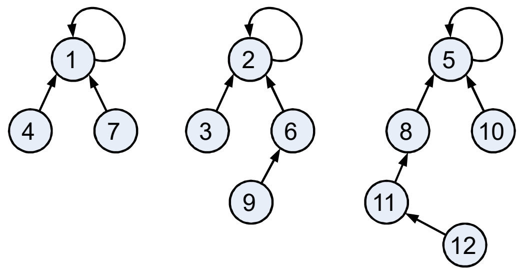
Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times
- Total time needed to do n unions and finds is O(n log n).

51

Tree-based Implementation

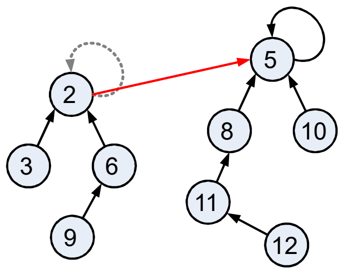
- Each element is stored in a node, which contains a pointer to a set name
- A node v whose set pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets “1”, “2”, and “5”:



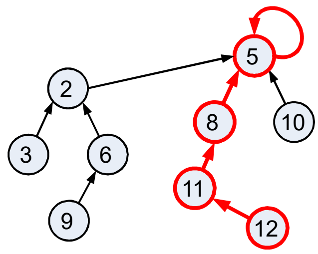
52

Union-Find Operations

- To do a Union, simply make the root of one tree point to the root of the other

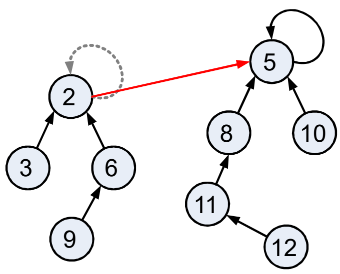


- To do a Find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself



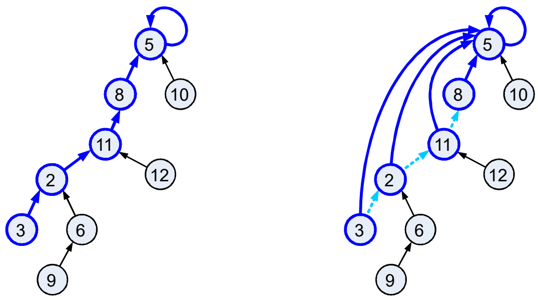
Union-Find Heuristic 1

- Union by size:
  - When performing a union, make the root of smaller tree point to the root of the larger
- Implies O(n log n) time for performing n union-find operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most O(log n) pointers for any find.



Union-Find Heuristic 2

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root



- Implies O(n $\log^* n$) time for performing n union-find operations:
  - Proof is somewhat involved.

Proof of log* n Amortized Time

- For each node *v* that is a root
  - define $n(v)$ to be the size of the subtree rooted at *v* (including *v*)
  - identified a set with the root of its associated tree.
- We update the size field of *v* each time a set is unioned into *v*. Thus, if *v* is not a root, then $n(v)$ is the largest the subtree rooted at *v* can be, which occurs just before we union *v* into some other node whose size is at least as large as *v* 's.
- For any node *v*, then, define the ***rank*** of *v*, which we denote as $r(v)$, as $r(v) = [\log n(v)]$:
- Thus, $n(v) \geq 2^{r(v)}$.
- Also, since there are at most *n* nodes in the tree of *v*, $r(v) = [\log n]$, for each node *v*.

Proof of log* n Amortized Time (2)

- For each node *v* with parent *w:*
  - $r(v) > r(w)$
- **Claim:** *There are at most n/* 2^s *nodes of rank s.*
- **Proof:**
  - Since $r(v) < r(w)$, for any node *v* with parent *w*, ranks are monotonically increasing as we follow parent pointers up any tree.
  - Thus, if $r(v) = r(w)$ for two nodes *v* and *w*, then the nodes counted in $n(v)$ must be separate and distinct from the nodes counted in $n(w)$.
  - If a node *v* is of rank *s*, then $n(v) \geq 2^s$.
  - Therefore, since there are at most *n* nodes total, there can be at most $n/2^s$ that are of rank *s*.

Proof of log* n Amortized Time (3)

- Definition: Tower of two's function:
  - $t(i) = 2^{t(i-1)}$
- Nodes *v* and *u* are in the same rank group *g* if
  - $g = \log^*(r(v)) = \log^*(r(u))$*:*
- Since the largest rank is log *n*, the largest rank group is
  - $\log^*(\log n) = (\log^* n)-1$

Proof of log* n Amortized Time (4)

- Charge 1 cyber-dollar per pointer hop during a find:
  - If *w* is the root or if *w* is in a different rank group than *v*, then charge the find operation one cyber-dollar.
  - Otherwise (*w* is not a root and *v* and *w* are in the same rank group), charge the node *v* one cyber-dollar.
- Since there are most (log* *n*)-1 rank groups, this rule guarantees that any find operation is charged at most log* *n* cyber-dollars.

59

Proof of log* n Amortized Time (5)

- After we charge a node *v* then *v* will get a new parent, which is a node higher up in *v*'s tree.
- The rank of *v*'s new parent will be greater than the rank of *v*'s old parent *w*.
- Thus, any node *v* can be charged at most the number of different ranks that are in *v*'s rank group.
- If *v* is in rank group *g* > 0, then *v* can be charged at most *t*(*g*)-*t*(*g*-1) times before *v* has a parent in a higher rank group (and from that point on, *v* will never be charged again). In other words, the total number, *C*, of cyber-dollars that can ever be charged to nodes can be bound as

$$C \leq \sum_{g=1}^{\log^* n - 1} n(g) \cdot (t(g) - t(g-1))$$

60

Proof of log* n Amortized Time (end)

- Bounding $n(g)$:

$$n(g) \le \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s}$$
$$= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s}$$
$$< \frac{n}{2^{t(g-1)+1}} \cdot 2$$
$$= \frac{n}{2^{t(g-1)}}$$
$$= \frac{n}{t(g)}$$

- Returning to C:

$$C < \sum_{g=1}^{\log^* n-1} \frac{n}{t(g)} \cdot (t(g) - t(g-1))$$
$$\le \sum_{g=1}^{\log^* n-1} \frac{n}{t(g)} \cdot t(g)$$
$$= \sum_{g=1}^{\log^* n-1} n$$
$$\le n \log^* n$$

61

Summary

- Priority queue ADT
- List-based priority queues
- Heap-based priority queues
- Bottom-up heap construction
- Binomial heaps
- Disjoint set union-find data structures and algorithms
- Suggested reading:
  - Sedgewick, Ch. 1.3, 9.