UNIT – 7 : PRIORITY QUEUES

Priority Queue:

Need for priority queue:

- In a multi user environment, the operating system scheduler must decide which of several processes to run only for a fixed period for time.
- For that we can use the algorithm of QUEUE, where Jobs are initially placed at the end of the queue.
- The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and placing it at the and of the queue if it doesn't finish.
- This strategy is generally not approximate, because very short jobs will soon to take a long time because of the wait involved to run.
- Generally, it is important that short jobs finish as fast as possible, so these jobs should have precedence over jobs that have already been running.
- Further more, some jobs that are not short are still very important and should also have precedence.
- This particular application seems to require a special kind of queue, known as a PRIORITY QUEUE.

Priority Queue:

It is a collection of ordered elements that provides fast access to the minimum or maximum element.

Basic Operations performed by priority queue are:

1. Insert operation
2. Deletemin operation

Insert operation is the equivalent of queue's ***Enqueue*** operation.

Deletemin operation is the priority queue equivalent of the queue's ***Dequeue*** operation.

Implementation:

There are three ways for implementing priority queue. They are:

1. Linked list
2. Binary Search tree
3. Binary Heap

7.1. Single- and Double-Ended Priority Queues:

A (single-ended) priority queue is a data type supporting the following operations on an ordered set of values:
1) find the maximum value (FindMax);
2) delete the maximum value (DeleteMax);
3) add a new value x (Insert(x)).
Obviously, the priority queue can be redefined by substituting operations 1) and 2) with FindMin and DeleteMin, respectively. Several structures, some implicitly stored in an array and some using more complex data structures, have been presented for implementing this data type, including max-heaps (or min-heaps)

Eduncloud.com

Conceptually, a max-heap is a binary tree having the following properties:
a) heap-shape: all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions; all other levels are complete.
b) max-ordering: the value stored at a node is greater than or equal to the values stored at its children. A max-heap of size n can be constructed in linear time and can be stored in an n-element array; hence it is referred to as an implicit data structure [g].

When a max-heap implements a priority queue, FindMax can be performed in constant time, while both DeleteMax and Insert(x) have logarithmic time. We shall consider a more powerful data type, the double-ended priority queue, which allows both FindMin and FindMax, as well as DeleteMin, DeleteMax, and Insert(x) operations. An important application of this data type is in external quicksort .
A traditional heap does not allow efficient implementation of all the above operations; for example, FindMin requires linear (instead of constant) time in a max-heap. One approach to overcoming this intrinsic limitation of heaps, is to place a max-heap ―back-to-back‖ with a min-heap.

Definition
A **double-ended priority queue (DEPQ)** is a collection of zero or more elements. Each element has a priority or value. The operations performed on a double-ended priority queue are:

1. isEmpty() ... return true iff the DEPQ is empty
2. size() ... return the number of elements in the DEPQ
3. getMin() ... return element with minimum priority
4. getMax() ... return element with maximum priority
5. put(x) ... insert the element x into the DEPQ
6. removeMin() ... remove an element with minimum priority and return this element
7. removeMax() ... remove an element with maximum priority and return this element

Application to External Sorting

The internal sorting method that has the best expected run time is quick sort (see Section 19.2.3). The basic idea in quick sort is to partition the elements to be sorted into three groups L, M, and R. The middle group M contains a single element called the **pivot**, all elements in the left group L are <= the pivot, and all elements in the right group R are >= the pivot. Following this partitioning, the left and right element groups are sorted recursively.

In an external sort, we have more elements than can be held in the memory of our computer. The elements to be sorted are initially on a disk and the sorted sequence is to be left on the disk. When the internal quick sort method outlined above is extended to an external quick sort, the middle group M is made as large as possible through the use of a DEPQ. The external quick sort strategy is:

1. Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group of elements.
2. Read in the remaining elements. If the next element is <= the smallest element in the DEPQ, output this next element as part of the left group. If the next element is >= the largest element in the DEPQ, output this next element as part of the right group. Otherwise, remove either the max or min element from the DEPQ (the choice may be made randomly or alternately); if the max element is removed, output it as part of the right group; otherwise, output the removed element as part of the left group; insert the newly input element into the DEPQ.
3. Output the elements in the DEPQ, in sorted order, as the middle group.
4. Sort the left and right groups recursively.

Generic Methods for DEPQs:
General methods exist to arrive at efficient DEPQ data structures from single-ended priority queue (PQ) data structures that also provide an efficient implementation of the remove(theNode) operation (this operation removes the node theNode from the PQ). The simplest of these methods, **dual structure method**, maintains both a min PQ and a max PQ of all the DEPQ elements together with **correspondence pointers** between the nodes of the min PQ and the max PQ that contain the same element. Figure 1 shows a dual heap structure for the elements 6, 7, 2, 5, 4. Correspondence pointers are shown as red arrows.

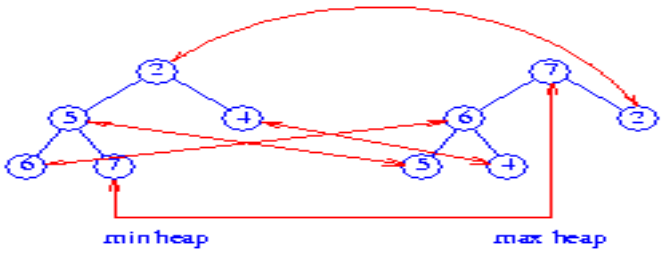


Figure 1 Dual heap

Although the figure shows each element stored in both the min and the max heap, it is necessary to store each element in only one of the two heaps.

The isEmpty and size operations are implemented by using a variable size that keeps track of the number of elements in the DEPQ. The minimum element is at the root of the min heap and the maximum element is at the root of the max heap. To insert an element x, we insert x into both the min and the max heaps and then set up correspondence pointers between the locations of x in the min and max heaps. To remove the minimum element, we do a removeMin from the min heap and a remove(theNode), where theNode is the corresponding node for the removed element, from the max heap. The maximum element is removed in an analogous way.

Total and leaf correspondence are more sophisticated correspondence methods. In both of these, half the elements are in the min PQ and the other half in the max PQ. When the number of elements is odd, one element is retained in a buffer. This buffered element is not in either PQ. In total correspondence, each element a in the min PQ is paired with a distinct element b of the max PQ. (a,b) is a corresponding pair of elements such that priority(a) <= priority(b). Figure 2 shows a total correspondence heap for the 11 elements 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 10. The element 9 is in the buffer. Corresponding pairs are shown by red arrows.

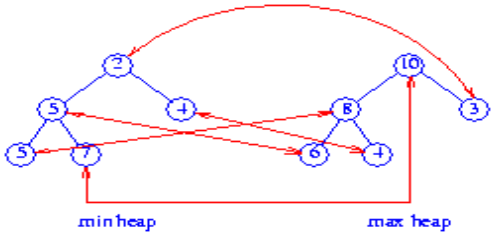


Figure 2 Total correspondence heap

In leaf correspondence, each leaf element of the min and max PQ is required to be part of a corresponding pair. Nonleaf elements need not be in any corresponding pair. Figure 3 shows a leaf correspondence heap.

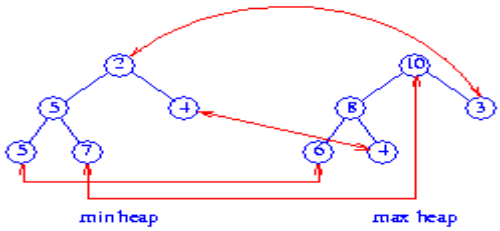


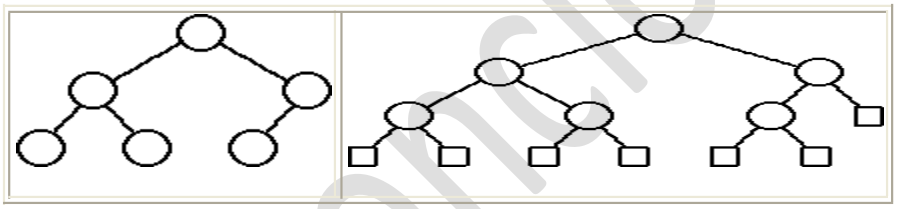
Figure 3 A leaf correspondence heap

Total and leaf correspondence structures require less space than do dual structures. However, the DEPQ algorithms for total and leaf correspondence structures are more complex than those for dual structures. Of the three correspondence types, leaf correspondence generally results in the fastest DEPQ correspondence structures.

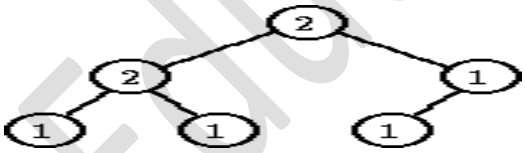
Using any of the described correspondence methods, we can arrive at DEPQ structures from heaps, height biased leftist trees, and pairing heaps. In these DEQP structures, the operations put(x), removeMin(), and removeMax() take O(log n) time (n is the number of elements in the DEPQ, for pairing heaps, this is an amortized complexity), and the remaining DEPQ operations take O(1) time.

7.2. Leftist tree:

Definitions:An **external node** is an imaginary node in a location of a missing child.



Notation. Let the **s-value** of a node be the shortest distance from the node to an external node.

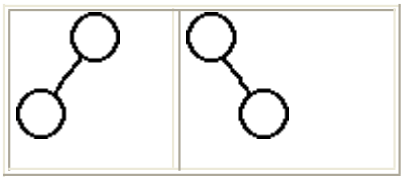


- An external node has the s-value of 0.
- An internal node has the s-value of 1 plus the minimum of the s-values of its internal and external children.

Height-Biased Leftist Trees

In a **height-biased leftist tree** the s-value of a left child of a node is not smaller than the s-value of the right child of the node.

height-biased	non height-biased

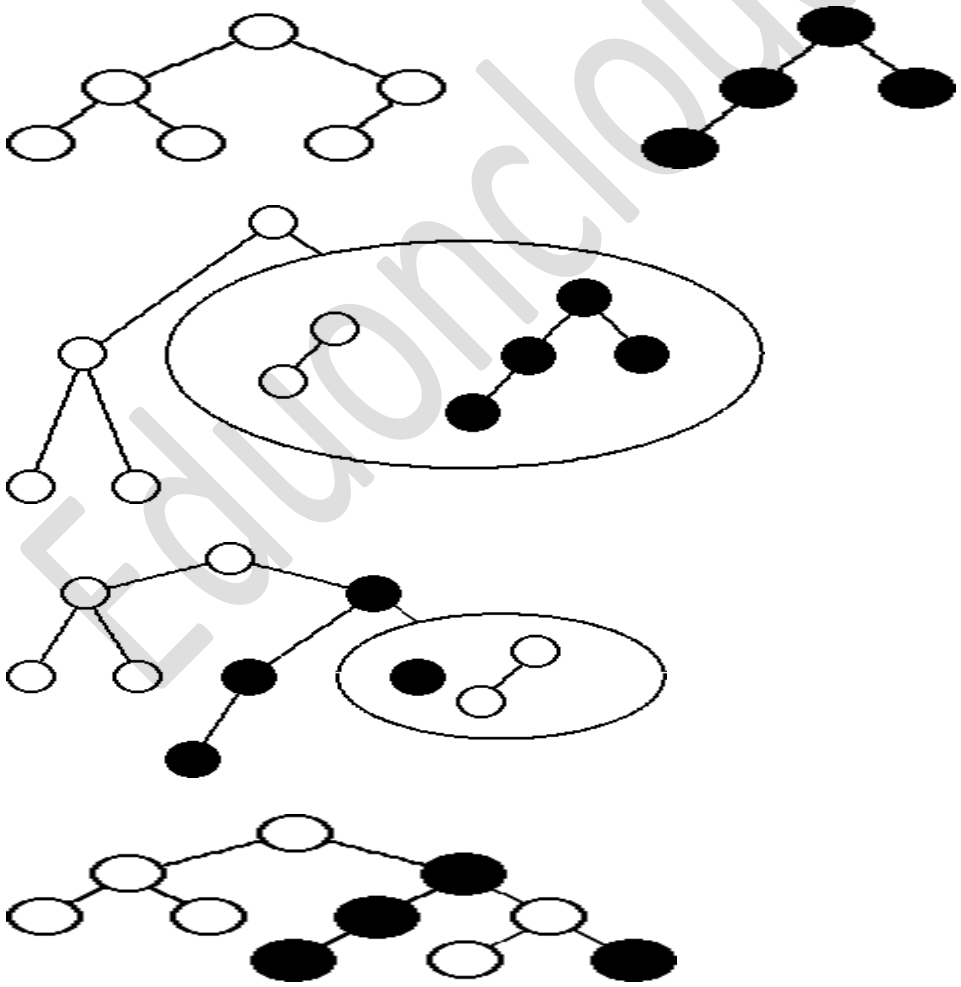


Merging Height-Biased Leftist Trees

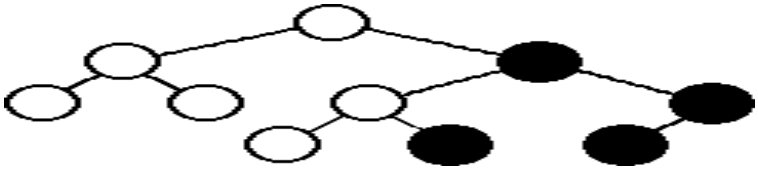
Recursive algorithm

- Consider two nonempty height-biased leftist trees A and B, and a relation (e.g., smaller than) on the values of the keys.
- Assume the key-value of A is not bigger than the key-value of B
- Let the root of the merged tree have the same left subtree as the root of A
- Let the root of the merged tree have the right subtree obtained by merging B with the right subtree of A.
- If in the merged tree the s-value of the left subtree is smaller than the s-value of the right subtree, interchange the subtrees.

For the following example, assume the key-value of each node equals its s-value.



Eduncloud.com



Time complexity

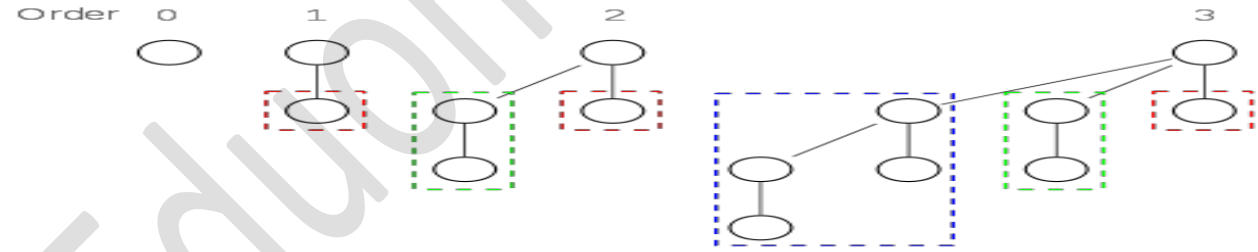
- Linear in the rightmost path of the outcome tree.
- The rightmost path of the the outcome tree is a shortest path
- A shortest path can‘t contain more than log n nodes.

Proof If the shortest path can contain more than log n nodes, then the first 1 + log n levels should include $2^0 + 2^1 + \ldots + 2^{\log n} = 2^{1+\log n} - 1 = 2n - 1$ nodes. In such a case, for $n > 1$ we end up with $n > 2n - 1$.

7.3. Binomial Heaps:

Binomial heap is a heap similar to a binary heap but also supports quickly merging two heaps. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation. A binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of a single binary tree). A **binomial tree** is defined recursively:

- A binomial tree of order 0 is a single node
- A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).



Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.

A binomial tree of order *k* has 2^k nodes, height *k*.

Because of its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*−1 trivially by attaching one of them as the leftmost child of root of the other one. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.The name comes from the shape: a binomial tree of order n has $\binom{n}{d}$ nodes at depth d. (See Binomial coefficient.)

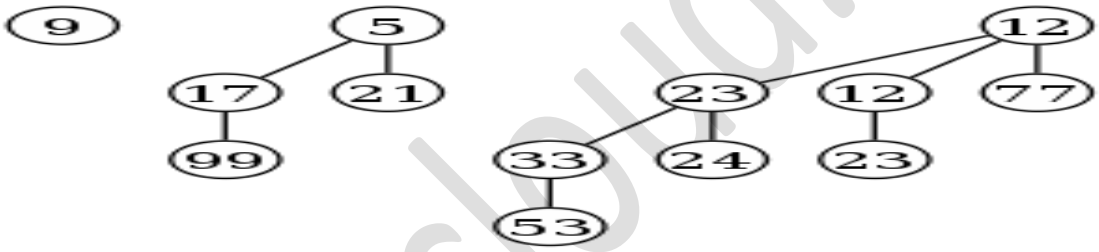
Structure of a binomial heap

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap.

The second property implies that a binomial heap with *n* nodes consists of at most log *n* + 1 binomial trees. In fact, the number and orders of these trees are uniquely determined by the number of nodes *n*: each binomial tree corresponds to one digit in the binary representation of number *n*. For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).



Example of a binomial heap containing 13 nodes with distinct keys. The heap consists of three binomial trees with orders 0, 2, and 3.

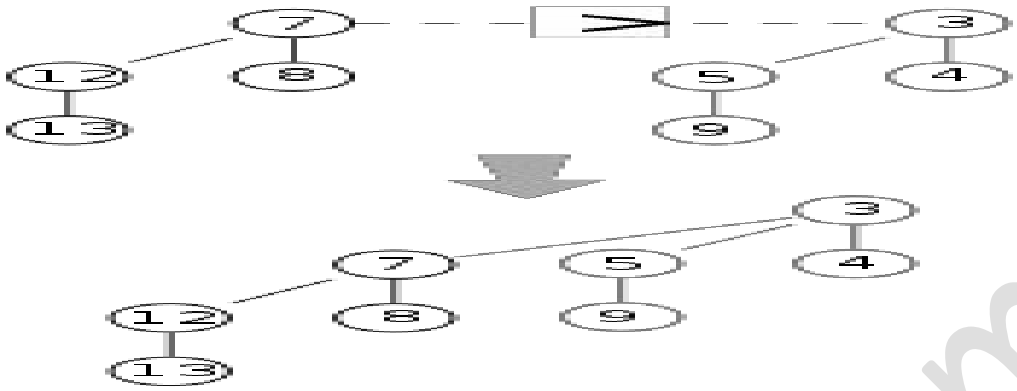
Implementation

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Merge

As mentioned above, the simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps. Due to the structure of binomial trees, they can be merged trivially. As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes the new root node. Then the other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps.

```
function mergeTree(p, q)
   if p.root.key <= q.root.key
      return p.addSubTree(q)
   else
      return q.addSubTree(p)
```



To merge two binomial trees of the same order, first compare the root key. Since 7>3, the black tree on the left(with root node 7) is attached to the grey tree on the right(with root node 3) as a subtree. The result is a tree of order 3.

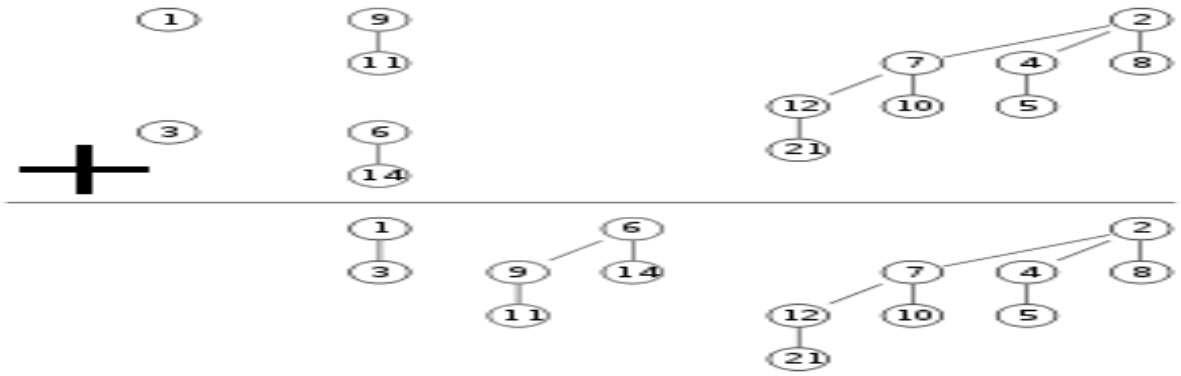
The operation of **merging** two heaps is perhaps the most interesting and can be used as a subroutine in most other operations. The lists of roots of both heaps are traversed simultaneously, similarly as in the merge algorithm.

If only one of the heaps contains a tree of order *j*, this tree is moved to the merged heap. If both heaps contain a tree of order *j*, the two trees are merged to one tree of order *j*+1 so that the minimum-heap property is satisfied. Note that it may later be necessary to merge this tree with some other tree of order *j*+1 present in one of the heaps. In the course of the algorithm, we need to examine at most three trees of any order (two from the two heaps we merge and one composed of two smaller trees).

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Each tree has order at most log *n* and therefore the running time is *O*(log *n*).

```
function merge(p, q)
   while not( p.end() and q.end() )
      tree = mergeTree(p.currentTree(), q.currentTree())
      if not heap.currentTree().empty()
         tree = mergeTree(tree, heap.currentTree())
         heap.addTree(tree)
      else
         heap.addTree(tree)
      heap.next() p.next() q.next()
```



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time,however it has an *amortized* time of O(1) (i.e. constant).

Find minimum

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can again be done easily in *O*(log *n*) time, as there are just *O*(log *n*) trees and hence roots to examine.By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to *O*(1). The pointer must be updated when performing any operation other than Find minimum. This can be done in *O*(log *n*) without raising the running time of any operation.

Delete minimum

To **delete the minimum element** from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees. Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each tree has at most log *n* children, creating this new heap is *O*(log *n*). Merging heaps is *O*(log *n*), so the entire delete minimum operation is *O*(log *n*).

```
function deleteMin(heap)
   min = heap.trees().first()
   for each current in heap.trees()
      if current.root < min then min = current
   for each tree in min.subTrees()
      tmp.addTree(tree)
   heap.removeTree(min)
   merge(heap, tmp)
```

Decrease key

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most log *n*, so this takes *O*(log *n*) time.

Delete

To **delete** an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

Performance

All of the following operations work in O(log *n*) time on a binomial heap with *n* elements:

- Insert a new element to the heap
- Find the element with minimum key
- Delete the element with minimum key from the heap
- Decrease key of a given element
- Delete given element from the heap
- Merge two given heaps to one heap

Finding the element with minimum key can also be done in *O*(1) by using an additional pointer to the minimum.

Applications Discrete event simulation, Priority queues

7.4. Fibonacci Heaps:

A **Fibonacci heap** is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap. Fibonacci heaps were developed by Michael L. Fredman and Robert E. Tarjan in 1984 and first published in a scientific journal in 1987. The name of Fibonacci heap comes from Fibonacci numbers which are used in the running time analysis.

Find-minimum is O(1) amortized time.Operations insert, decrease key, and merge (union) work in constant amortized time. Operations delete and delete minimum work in *O*(log *n*) amortized time. This means that starting from an empty data structure, any sequence of *a* operations from the first group and *b* operations from the second group would take $O(a + b \log n)$ time. In a binomial heap such a sequence of operations would take $O((a + b)\log (n))$ time. A Fibonacci heap is thus better than a binomial heap when *b* is asymptotically smaller than *a*.

Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

Structure

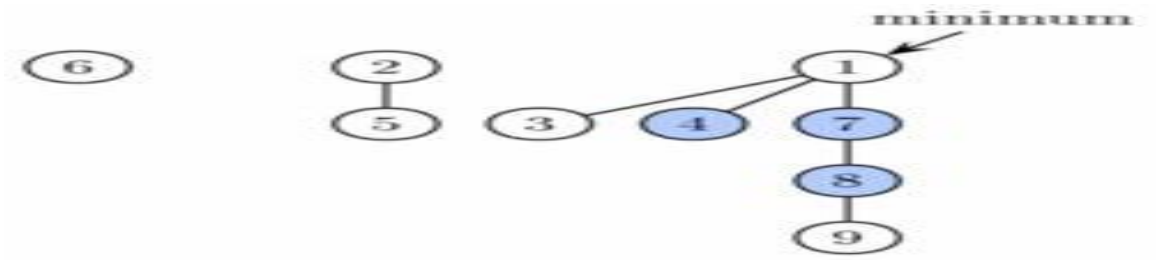


Figure 1. Example of a Fibonacci heap. It has three trees of degrees 0, 1 and 3. Three vertices are marked (shown in blue). Therefore the potential of the heap is 9.

A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations. For example merging heaps is done simply by concatenating the two lists of trees, and operation *decrease key* sometimes cuts a node from its parent and forms a new tree.

However at some point some order needs to be introduced to the heap to achieve the desired running time. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most $O(\log n)$ and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number. This is achieved by the rule that we can cut at most one child of each non-root node. When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree (see Proof of degree bounds, below). The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

As a result of a relaxed structure, some operations can take a long time while others are done very quickly. In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do. This additional time is then later subtracted from the actual running time of slow operations. The amount of time saved for later use is measured at any given moment by a potential function. The potential of a Fibonacci heap is given by

$$\text{Potential} = t + 2m$$

where t is the number of trees in the Fibonacci heap, and m is the number of marked nodes. A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

Thus, the root of each tree in a heap has one unit of time stored. This unit of time can be used later to link this tree with another tree at amortized time 0. Also, each marked node has two units of time stored. One can be used to cut the node from its parent. If this happens, the node becomes a root and the second unit of time will remain stored in it as in any other root.

Implementation of operations

To allow fast deletion and concatenation, the roots of all trees are linked using a circular, doubly linked list. The children of each node are also linked using such a list. For each node, we maintain its number of

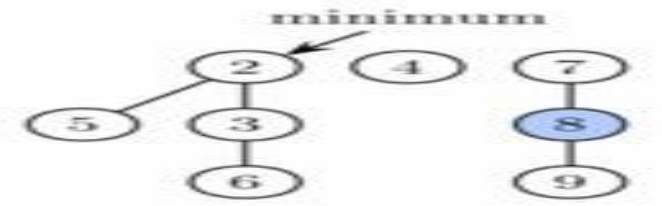
children and whether the node is marked. Moreover we maintain a pointer to the root containing the minimum key.

Operation **find minimum** is now trivial because we keep the pointer to the node containing it. It does not change the potential of the heap, therefore both actual and amortized cost is constant. As mentioned above, **merge** is implemented simply by concatenating the lists of tree roots of the two heaps. This can be done in constant time and the potential does not change, leading again to constant amortized time.

Operation **insert** works by creating a new heap with one element and doing merge. This takes constant time, and the potential increases by one, because the number of trees increases. The amortized cost is thus still constant.

Fibonacci heap from Figure 1 after first phase of extract minimum. Node with key 1 (the minimum) was deleted and its children were added as separate trees.

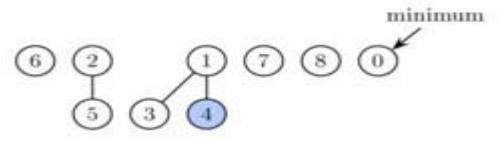
Operation **extract minimum** (same as *delete minimum*) operates in three phases. First we take the root containing the minimum element and remove it. Its children will become roots of new trees. If the number of children was d, it takes time $O(d)$ to process all new roots and the potential increases by $d-1$. Therefore the amortized running time of this phase is $O(d) = O(\log n)$.



Fibonacci heap from Figure 1 after extract minimum is completed. First, nodes 3 and 6 are linked together. Then the result is linked with tree rooted at node 2. Finally, the new minimum is found.

However to complete the extract minimum operation, we need to update the pointer to the root with minimum key. Unfortunately there may be up to n roots we need to check. In the second phase we therefore decrease the number of roots by successively linking together roots of the same degree. When two roots u and v have the same degree, we make one of them a child of the other so that the one with the smaller key remains the root. Its degree will increase by one. This is repeated until every root has a different degree. To find trees of the same degree efficiently we use an array of length $O(\log n)$ in which we keep a pointer to one root of each degree. When a second root is found of the same degree, the two are linked and the array is updated. The actual running time is $O(\log n + m)$ where m is the number of roots at the beginning of the second phase. At the end we will have at most $O(\log n)$ roots (because each has a different degree). Therefore the difference in the potential function from before this phase to after it is: $O(\log n) - m$, and the amortized running time is then at most $O(\log n + m) + O(\log n) - m = O(\log n)$. Since we can scale up the units of potential stored at insertion in each node by the constant factor in the $O(m)$ part of the actual cost for this phase.

In the third phase we check each of the remaining roots and find the minimum. This takes $O(\log n)$ time and the potential does not change. The overall amortized running time of extract minimum is therefore $O(\log n)$.



Fibonacci heap from Figure 1 after decreasing key of node 9 to 0. This node as well as its two marked ancestors are cut from the tree rooted at 1 and placed as new roots.

Operation **decrease key** will take the node, decrease the key and if the heap property becomes violated (the new key is smaller than the key of the parent), the node is cut from its parent. If the parent is not a root, it is marked. If it has been marked already, it is cut as well and its parent is marked. We continue upwards until we reach either the root or an unmarked node. In the process we create some number, say k, of new trees. Each of these new trees except possibly the first one was marked originally but as a root it will become unmarked. One node can become marked. Therefore the potential decreases by at least $k - 2$. The actual time to perform the cutting was $O(k)$, therefore the amortized running time is constant.

Finally, operation **delete** can be implemented simply by decreasing the key of the element to be deleted to minus infinity, thus turning it into the minimum of the whole heap. Then we call extract minimum to remove it. The amortized running time of this operation is $O(\log n)$.

Proof of degree bounds

The amortized performance of a Fibonacci heap depends on the degree (number of children) of any tree root being $O(\log n)$, where n is the size of the heap. Here we show that the size of the (sub)tree rooted at any node x of degree d in the heap must have size at least F_{d+2}, where F_k is the kth Fibonacci number. The degree bound follows from this and the fact (easily proved by induction) that $F_{d+2} \geq \varphi^d$ for all integers $d \geq 0$, where $\varphi = (1 + \sqrt{5})/2 \doteq 1.62$. (We then have $n \geq F_{d+2} \geq \varphi^d$, and $d \leq \log_\varphi n$ taking the log to base φ of both sides gives as required.)

Consider any node x somewhere in the heap (x need not be the root of one of the main trees). Define **size**(x) to be the size of the tree rooted at x (the number of descendants of x, including x itself). We prove by induction on the height of x (the length of a longest simple path from x to a descendant leaf), that **size**(x) $\geq F_{d+2}$, where d is the degree of x.

Base case: If x has height 0, then $d = 0$, and **size**(x) $= 1 = F_2$.

Inductive case: Suppose x has positive height and degree $d>0$. Let $y_1, y_2, ..., y_d$ be the children of x, indexed in order of the times they were most recently made children of x (y_1 being the earliest and y_d the latest), and let $c_1, c_2, ..., c_d$ be their respective degrees. We **claim** that $c_i \geq i-2$ for each i with $2 \leq i \leq d$: Just before y_i was made a child of x, $y_1,...,y_{i-1}$ were already children of x, and so x had degree at least $i-1$ at that time. Since trees are combined only when the degrees of their roots are equal, it must have been that y_i also had degree at least $i-1$ at the time it became a child of x. From that time to the present, y_i can only have lost at most one child (as guaranteed by the marking process), and so its current degree c_i is at least $i-2$. This proves the **claim**.

Since the heights of all the y_i are strictly less than that of x, we can apply the inductive hypothesis to them to get **size**(y_i) $\geq F_{ci+2} \geq F_{(i-2)+2} = F_i$. The nodes x and y_1 each contribute at least 1 to **size**(x), and so we have

$$\mathbf{size}(x) \geq 2 + \sum_{i=2}^{d} \mathbf{size}(y_i) \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i.$$

A routine induction proves that $1 + \sum_{i=0}^{d} F_i = F_{d+2}$ for any $d \geq 0$, which gives the desired lower bound on **size**(x).

Worst case

Although the total running time of a sequence of operations starting with an empty structure is bounded by the bounds given above, some (very few) operations in the sequence can take very long to complete (in particular delete and delete minimum have linear running time in the worst case). For this reason Fibonacci heaps and other amortized data structures may not be appropriate for real-time systems. It is possible to create a data structure which the same worst case performance as the Fibonacci heap has amortized performance.[3] However the resulting structure is very complicated, so it is not useful in most practical cases.

7.5. Pairing heaps

Pairing heaps are a type of heap data structure with relatively simple implementation and excellent practical amortized performance. However, it has proven very difficult to determine the precise asymptotic running time of pairing heaps.

Pairing heaps are heap ordered multiway trees. Describing the various heap operations is relatively simple (in the following we assume a min-heap):

- *find-min*: simply return the top element of the heap.
- *merge*: compare the two root elements, the smaller remains the root of the result, the larger element and its subtree is appended as a child of this root.
- *insert*: create a new heap for the inserted element and *merge* into the original heap.
- *decrease-key* (optional): remove the subtree rooted at the key to be decreased then *merge* it with the heap.
- *delete-min*: remove the root and *merge* its subtrees. Various strategies are employed.

- The amortized time per *delete-min* is $O(\log n)$.[1] The operations *find-min*, *merge*, and *insert* take $O(1)$ amortized time[2] and *decrease-key* takes $2^{O(\sqrt{\log \log n})}$ amortized time.[3] Fredman proved that the amortized time per *decrease-key* is at least $\Omega(\log \log n)$.[4] That is, they are less efficient than Fibonacci heaps, which perform *decrease-key* in $O(1)$ amortized time.

Implementation

A pairing heap is either an empty heap, or a pair consisting of a root element and a possibly empty list of pairing heaps. The heap ordering property requires that all the root elements of the subheaps in the list are not smaller that then root element of the heap. The following description assumes a purely functional heap that does not support the *decrease-key* operation.

type PairingHeap[Elem] = Empty | Heap(elem: Elem, subheaps: List[PairingHeap[Elem]])

Operations

find-min

The function *find-min* simply returns the root element of the heap:

```
function find-min(heap)
  if heap == Empty
  error
  else
    return heap.elem
```

merge:

Merging with an empty heap returns the other heap, otherwise a new heap is returned that has the minimum of the two root elements as its root element and just adds the heap with the larger root to the list of subheaps:

```
function merge(heap1, heap2)
  if heap1 == Empty
    return heap2
  elsif heap2 == Empty
    return heap1
  elseif heap1.elem < heap2.elem
    return Heap(heap1.elem, heap2 :: heap1.subheaps)
  else
    return Heap(heap2.elem, heap1 :: heap2.subheaps)
```

Insert:

The easiest way to insert an element into a heap is to merge the heap with a new heap containing just this element and an empty list of subheaps:

```
function insert(elem, heap)
  return merge(Heap(elem, []), heap)
```

delete-min:

The only non-trivial fundamental operation is the deletion of the minimum element from the heap. The standard strategy first merges the subheaps in pairs (this is the step that gave this datastructure its name) from left to right and then merges the resulting list of heaps from right to left:

```
fuction delete-min(heap)
  if heap == Empty
```

```
    error
  elsif length(heap.subheaps) == 0
    return Empty
  elsif length(heap.subheaps) == 1
    return heap.subheaps[0]
  else
    return merge-pairs(heap.subheaps)
```

This uses the auxiliary function *merge-pairs*:

```
function merge-pairs(l)
  if length(l) == 0
  return Empty
  elsif length(l) == 1
    return l[0]
  else
    return merge(merge(l[0], l[1]), merge-pairs(l[2.. ]))
```

That this does indeed implement the described two-pass left-to-right then right-to-left merging strategy can be seen from this reduction:

```
   merge-pairs([H1, H2, H3, H4, H5, H6, H7])
=> merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7]))
    # merge H1 and H2 to H12, then the rest of the list
=> merge(H12, merge(merge(H3, H4), merge-pairs([H5, H6, H7])))
    # merge H3 and H4 to H34, then the rest of the list
=> merge(H12, merge(H34, merge(merge(H5, H6), merge-pairs([H7]))))
    # merge H5 and H5 to H56, then the rest of the list
=> merge(H12, merge(H34, merge(H56, H7)))
    # switch direction, merge the last two resulting heaps, giving H567
=> merge(H12, merge(H34, H567))
    # merge the last two resulting heaps, giving H34567
=> merge(H12, H34567)
    # finally, merge the first merged pair with the result of merging the rest
=> H1234567
```

7.6. RECOMMENDED QUESTIONS

1. Define a priority queue

2. Define a Deque (Double-Ended Queue)

3. What is the need for Priority queue?

4. What are the applications of priority queues?

5. What are binomial heaps?

6. Explain the height-biased leftist trees.