Heaps

Binary Heaps

A binary heap is a complete binary tree in which evey node satisfies the heap property which states:

`If B is a child of A, then key(a)` $\geq$ `key(B)`

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a **max-heap**. Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a **min-heap**.

The property of binary heaps are:

• Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position $2i$ and its right child at position $2i + 1$. Conversely, an element at position i has its parent at position $\frac{i}{2}$.

• Being a compelete binary tree, all the levels of the tree except the last level are completely filled.

• The height of a binary tree is given as $\log_2 n$, where n is the number of elements.

• Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max head and lowest value in case of min heap. A binary tree is an effeicient data structre, but binary heap is more space efficient and simpler.

Inserting a New Element in a Binary Heap

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarly a heap.

2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repreated.

Deleting an Element from a Binary Heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.

2. Delete the last node.

3. Sink down the new new root node's value so that H satisfies the heap property. In this step interchange the root node's value wih its child node's value (whichever is largest among its children).

Applications of Binary Heaps

Binary heaps are mainly applied for

1. Sorting an array using **heapsort** algorithm.

2. Implementing priority queues.

Binomial Heaps

A binomial heap H is a set of binomial trees that satisfy the binomial heap properties. A binomial tree is an ordered tree that can be recursively defined as follows:

- A binomial tre of order 0 has a single node.

- A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order $i-1, i-2, \ldots, 2, 1$ and 0.

- A binomial tree B_1 has 2^i nodes.

- The height of a binomial tree B_i is i

A binomial heap H is a collection of binomial trees that satisfy the following properties:

- Every binomial tree in H satisfies the minimum heap property (i.e., the key of a node is either greater than or equal to they key of its parent).

- There can be one or zero binomial trees for each order including zero order.

According to the first property, the root of a heap-ordered tree contains the smallest key in the tree. The second property, on the other hand, implies that a binomial heap H having N nodes contains at most $\log(N+1)$ trees.

Linked Representation of Binomial Heaps

Each node in a binomial heap H has a `val` field that stores its value. In addition, each node N has following pointers:

- $P[N]$ that points to the parent of N

- `Child[N]` that points to the leftmost child

- `Sibling[N]` that points to the sibling of N which is immediately to its right

If N is the root node, then `P[N] = NULL`. If N has no children, then `Child[N] = NULL`, and if N is the rightmost child of its parent, then `Sibling[N] = NULL`.

In addition to this, every node N has a dregree field which stores the number of children of N.

Operations on Binomial Heaps

The different operations that can be performed on binomial heaps.

Creating a New Binomial Heap

The procedute `Create_Binomial-Heap()` allocates and returns an object H, where `Head[H]` is set to `NULL`. The running time of this procedure can be given as $O(1)$

Finding the Node with Minimum Key

The procedure `Min_Binomial-Heap()` returns a pointer to the node which has the minimum value in the binomial heap H

The node with the minimum value in a particular binomial tree will appear as a root node in the binomial heap. Thus, the `Min_Binomial-Heap()` procedure checks all roots. Since there are at most $\log(n+1)$ roots to check, the running time of this procedure is $O(\log n)$.

Uniting Two Binomial Heaps

The procedure of uniting two binomial heaps is used as a subroutine by other operations.

The `Union_Binomial-Heap()` procedure links together binomial trees whose roots have the same degree.

The `Link_Binomial-Tree()` procedure makes Y the new head of the linked list of node Z's children in $O(1)$ time.

The algorithm to unite two binomial heaps destorys the original representations of heaps H_1 and H_2. Apart from `Link_Binomial-Tree()`, it uses another procedure `Merge_Binomial-Heap()` which is used to merge the root lists of H_1 and H_2 into single linked list that is sorted by degree into a monotonically increasing order.

The running time of `Union_Binomial-Heap()` can be given as $O(\log n)$, where n is the total number of nodes in binomial heaps H_1 and H_2. If H_2 containts n_1 nodes and H_2 contains n_2 nodes, then H_1 contains at most $\log(n_1+1)$ roots and H_2 containts at most $\log(n_2+1)$ roots, so H contains at most $(\log n_2 + \log n_1 + 2) \leq (2\log n + 2) = O(\log n)$ roots when we call `Merge_Binomial-Heap()`. Since, $n = n_1 + n_2$, the `Merge_Binomial-Heap()` takes $O(\log n)$ to execute. Each iteration of the `while` loop takes $O(1)$ time, and because there are at most $(\log n_1 + \log n_2 + 2)$ iterations, the total time is thus $O(\log n)$

Inserting a New Node

The `Insert_Binomial-Heap()` procedure is used to insert a node x into the binomial heap H. The pre-condition of this procedure is that x has already been allocated space and `val[x]` has already been filled in. The algorithm makes a binomial heap H' in $O(1)$ time. H' contains just one node which is x. Finally, the algorithm united H' with the n-node binomial heap H in $O(\log n)$ time. Note that the memory occupied by H' is freed in the `Union_Binomial-Heap(H, H')` procedure.

Extracing the Node with Minimum Key The `Min-Extract_Binomial-Heap()` procedure accepts a heap H as a parameter and returns a pointer to the extracted node. In the first step, it finds a root node R with the minimum value and removes it from the root list of H. Then, the order of R's children is reversed and they are all added to the root list of H'. Finally, `Union_Binomial-Heap(H, H')` is called to unite the two heaps and R is returned. The algorithm `Min-Extract_Binomial-Heap()` runs in $O(\log n)$ time, where n is the number of node in H.

Decreasing the Value of a Node

In the algorithm to decrease the value of a node x in a binomial heap H, the value of the node is overwritten with a new value k, which is less than the current value of the node.

In the algorithm, we first ensure that the new value is not greater than the current value and then assign the new value to the node. We then go up the tree with `PTR` initially pointing to node x. in each iteration of the `while` loop, `val[PTR]` is compared with the value of its parent tt PAR. However, if either `PTR` is the root `key[PTR]` $\geq$ `key[PAR]`, then the binomial tree is heap-ordered. Otherwise, node `PTR` violates heap-ordering, so its key is exchanged with that of its parent. We set `PTR = PAR` and `PAR = Parent[PTR]` to move up one level in the tree and continue the process.

The `Binomial-Heap_Decrease_Val` procedure takes $O(\log n)$ time as the maximum depth of node x in $\log n$, so the `while` loop will iterate at most $\log n$ times.

Deleting a Node

To delete a Node, we set the value of x to $-\infty$. Assuming that there is no node in the heap that has a value less than $-\infty$.

The `Binomial-Heap_Delete-Node` procedure sets the value of x to $-\infty$, which is a unique minimum value in the entire binomial heap. The `Binomial-Heap_Decrease_Val` algorithm bubbles this key upto a root and then this root is removed from the heap by making a call to the `Min-Extract_Binomial-Heap()` procedure. The `Binomial-Heap_Delete-Node()` procedure takes $O(\log n)$ time.

Fibonacci Heaps

Theoretically, Fibonacci heaps are especially desirable when the number of extract-minimum and delete operations is small relative to the number of other operations performed. This situation arises in many applications, where algorithms for graph problems may call the decrease-value once per edge. However, the programming complexity of Fibonacci heaps makes them less desirable to use.

A Fibonacci heap is a collection of trees. It is loosely based on binomial heaps. If neither the decrease-value nor the delete operation is performed, each tree in the heap is like a binomial tree. Fibonacci heaps differ from binomial heaps as they have a more relaxed structure, allowing improved asymptotic time bounds.

Strucutre of Fibonacci Heaps

Although a Fibonacci heap is a collection of heap-ordered trees, the trees in a Fibonaci heap are not constrained to be binomial trees. That is, while the trees in a binomial heap are ordered, those within Fibonacci heaps are rooted but unordered.

Each node in a Fibonacci heap contains the following pointers

- a pointer to its parent, and
- a pointer to any one of its children.

Note that the children of each node are linked together in a circular doubly linked list which is known as the child list of that node. Each child x in a child list contains pointers to its left and right siblings. If node x is the only child of its parent, then `left[x] = right[x] = x`.

Circular doubly linked list provide an added advantage, as they allow a node to be removed in $O(1)$ time. Also, given two circular doubly linked lists, the list can be concatenated to form one list in $O(1)$ time. Apart from this information, every node will store two other fields. First the number of children in the child list of node x is stored in `degree[x]`. Second, a boolean value `mark[x]` indicates whether node x has lost a child since the last time x was made the child of another node. The newly create nodes are unmarked. Also, when the node x is made the child of another node, it becomes unmarked.

Fibonacci heap H is generally accessed by a pointer called `min[H]` which points to the root that has a minimum value. If the Fibonacci heap H is empty, then `min[H] = NULL`. Roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular doubly linked list called the root list of the Fibonacci heap. Also note that the order of the trees within a root list is arbitrary. In a Fibonacci heap H, the number of nodes in H is stored in $n[H]$ and the degree of nodes is stored in $D(n)$.

Operations on Fibonacci Heaps

If we perform operations such as `create-heap`, `insert`, find `extract-minimum`, and `union`, then each Fibonacci heap is simply a collection of unordered binomial trees. An unordered binomial tree U_0 consists of a single node, and an unordered binomial tree U_1 consists of two unordered binomial trees U_{i-1} for which the root of one is made into a child of the root of another. All the properties of a binomial tree also hold for unordered binomial trees but for an unordered binomial tree U_1, the root has degree i, which is greater than that of any other node. The children of the roots of sub-trees $U_0, U_1, \ldots, U_{i-1}$ in some order. Thus, if an n-node Fibonacci heap is a collection of unordered binomial trees, then $D(n) = \log n$ The underlying principle of operations on Fibonacci heaps is to delay the work as much as possible.

Creating a New Fibonacci Heap

To create an empty Fibonacci heap, the `Create_Fib-Heap` procedure allocates and returns the Fibonacci heap object H, where $n[H] = 0$ and `min[H] = NULL`. The amoritzed cost of `Create_Fib-Heap` is equal to $O(1)$

Inserting a New Node

In the algorithm to insert a new node in a Fibonacci heap,In Steps 1 and 2, we first initialize the structural fields of node x, making it its own circular doubly linked list. Step 3 adds x to the root list of H in $O(1)$ actual time. Now x becomes an unordered binomial tree in the Fibonacci heap. In Step 4, the pointer to the minimum node of the Fibonacci heap H is updated. Finally, we increment the number of nodes in H

to reflect the addition of the new node. Note that unlike the insert operation in the case of a binomial heap, when we insert a node in a Fibonacci heap, no attempt is made to consolidate the trees within the Fibonacci heap. So, even if k consecutive insert operations are performed, then k single-node trees are added to the root list.

Finding the Nodw with Minimum Key

Fibonacci heaps maintain a pointer `min[H]` that points to the root having the minimum value. Therefore, finding the minimum node is a straighforward task that can be performed in just $O(1)$ time.

Uniting Two Fibonacci Heaps

In the algorithm, we first concatenate the root list of H_1 and H_2 into a new root list H. Then, the minimum node in H is updated. Finally, the memory occupied by H_1 and H_2 is freed and the resultant heap H is returned.

Extracting the Node with Minimum Key

In the `Extract-Min_Fib-Heap` algorithm, we first make a root out of each of the minimum node's children and then remove the minimum node from the root list of H. Finally, the root list of the resultant Fibonacci heap H is consolidated by linking the roots of equal degree until at most one root remains of each degree. Note that in Step 1, we save a pointer x to the minimum node; this pointer is returned at the end. However, if `x = NULL`, then the heap is already empty. Otherwise, the node x is deleted from H by making all its children the roots of H and then removing x from the root list. If `x = right[x]`, then x is the only node on the root list, so now H is empty. However if `x` $\neq$ `right[x]`, then we set the pointer `min[H]` to the node whose address is stored in the right field of x.

Consolidating a Heap

A Fibonacci heap is consolidated to reduce the number of trees in the heap. While consolidating the root list of H, the following steps are repeatedly executed until every root in the root list has a distinct degree value.

- Find two roots x and y in the root list that has the same degree and where `Val[x]` $\leq$ `Val[y]`

- Link y to x. That is, remove y from the root list of H and make it a child of x. This operation us actually done in the `Link_Fib-Heap()` procedure. Finally, `degree[x]` is incremented and the mark on y, if any is cleared.

In the consolidate algorithm, we have use an auxiliary array $A[0 \ldots D(n[H])$, such that if $A[i] = x$, then x is currently a node in the root list of H and `degree[x] = i`

In Step 1, we set every entry in the array A to `NULL`. When Step 1 is over, we get a tree that is rooted at some node x. Initially, the array entry `A[degree[x]]` is set to point to x. In the `for` loop, each root node in H is examined. In each iteration of the `while` loop, $A[d]$ points to some root `TEMP` because `d = degree[PTR] = degree[TEMP]`, so these two nodes must be linked with each other. The node with the smallest key becomes the parent of the other as a result of the link operation and so if need arises, we exchange the pointers to `PTR` and `TEMP`.

Next we linke `Temp` to `PTR` using the `Link_Fib-Heap()` procedure. The `Link_Fib-Heap()` increments the degree of x but leaves the degree of y unchanged. Since node y is no longer a root, the pointer to it in array A is removed in Step 10. Note that the value of degree x is incremented in the `Link_Fib-Heap()` procedure, so Step 13 restores the value od `d = degree[x]`. The `while` loop is repeated until `A[d] = NULL`, that is until no other root with the same degree as x exists in the root list of H.

Decreasing the Value of a Node

In the `Decrease-Val_Fib-Heap`, we first ensure that the new value is not greater than the current value of the node and then assign the new value to `PTR`. If either the `PTR` points to a root node or if `Val[PTR]` $\geq$ `Val[PAR]`, where `PAR` is `PTR`'s parent, then no structural changes need to be done. The condition is checked in Step 4.

However, if the IF condition in Step 4 evaluates to a false value, then the heap order has been violated and series of changes may occur. First, we call the Cut procedure to disconnect (or cut) any link between PTR and its PAR, thereby making PTR a root.

If PTR is a node that has undergone the following history, then the importance of the mark field can be understood as follows:

- Case 1: PTR was a root node.
- Case 2: Then PTR was linked to another node.
- Case 3: The two children of PTR were removed by the Cut procedure.

Note that PTR will lose its second child, it will be cut from its parent to form a new root. mark[PTR] is set to TRUE when cases 1 and 2 occur and PTR has lost one of its child by the Cut operation. The Cut procedure, thereforem clears mark[PTR] in Step 4 of the Cut procedure.

However, if PTR is the second child cut from its parent PAR (since the time that PAR was linked to another node), then a Cascading-Cut operation is performed on PAR. If PAR is unmarked, then it is marked as it indicates that its first chld has just been cut, and the procedure returns. Otherwise, if PAR is marked, then it means that PAR has now lost its second child. Therefore PTR is cut and Cascading-Cut procedure is called recursively up the tree until either a root or an unmarked node is found.

Once we are done with the Cut operation and the Cascading-Cut operations, Step 5 of the Decrease-Val_Fib-Heap finishes up by updating min[H.

Note that the amortized cost of Decrease-Val_Fib-Heap is $O(1)$. The actual cost of cost of Decrease-Val_Fib-Heap is $O(1)$ time plus the time required to perform the cascading cuts. If Cascading-Cut procedure is recursively called c times, then each call of the Cascading-Cut takes $O(1)$ time exclusive of recursive calls. Therefore, the actual cost of Decrease-Val_Fib-Heap including all recursive calls is $O(c)$.

Deleting a Node

A node from a Fibonacci heap can be very easily deleted in $O(D(n))$ amortized time. Del_Fib-Heap assigns a minimum value x. The node x is then removed from the Fibonacci heap by making a call to the Extract-Min_Fib-Heap procedure. The amortized time of the delete procedure is the sum of the $O(1)$ amortized time of Decrease-Val_Fib-Heap and the $O(D(n))$ amoritized time of Extract-Min_Fib-Heap.

Applications of Heaps

Heaps are preferred for applications that include:

- **Heap sort** It is one of the best sorting methods that has no quadratic worst-case scenarios.
- **Selection algorithms** These algorithms are used to find the minimum and maximum values in linear or sub-linear time.
- **Graph algorithms** Heaps can be used as internal traversal data structures. This guarantess that runtime is reduced by an order of polynomial. Heaps are therefore used for implementing Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

Points to Remember

• A binary heap is defined as a complete binary tree in which every node satisfies the heap property. There are two types of binary heaps: max heap and min heap.

• In a min heap, elements at every node will either be less than or equal to the element at its left and right child. Similarly, in a max heap, elements at every node will either be greater than or equal to element at its left and right child.

• A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order $i-1, i-2, \ldots, 2, 1, 0$.

• A binomial tree B_1 of height i has 2^i nodes.

• A binomial heap H is a collection of binomial trees that satisfy the following properties:
  • Every binomial tree in H satisfies the minimum heap property.
  • There can be zero binomial trees for each order including zero order.

• A Fibonacci heap is a collection of trees. Fibonacci heaps differ from binomial heaps, as they have a more relaxed structure, allowing improved asymptotic time bounds.