# Project
# Shortest Path Algorithm with Heaps

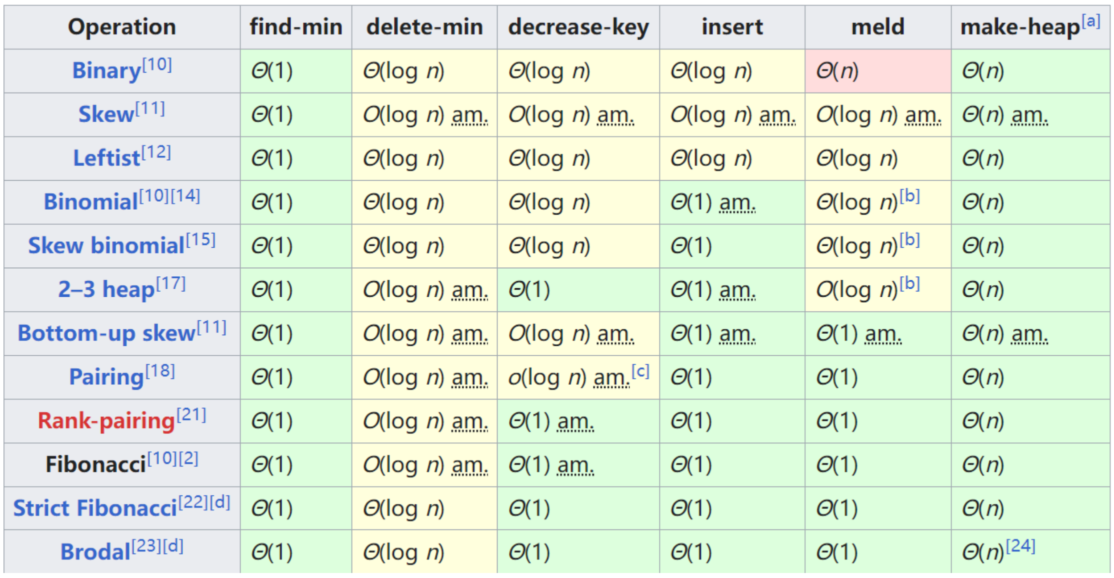
**Date: 2024-10-20**

# Chapter 1: Introduction

Dijkstra is the most well-known algorithm for finding shortest path. The key step of Dijkstra algorithm is to find the minimum element in array distance[]. If we just iterate through the array, it will take O(n) time, which is unacceptable. So the data structure heap is used to shorten this time. This goal of this project is to find the best data structure for Dijkstra's algorithm.

# Chapter 2: Algorithm Specification

Here are time complexities of various heap data structures in Figure 1.



| Operation | find-min | delete-min | decrease-key | insert | meld | make-heap[a] |
|---|---|---|---|---|---|---|
| Binary[10] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Skew[11] | $\Theta(1)$ | $O(\log n)$ am. | $O(\log n)$ am. | $O(\log n)$ am. | $O(\log n)$ am. | $\Theta(n)$ am. |
| Leftist[12] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(n)$ |
| Binomial[10][14] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ am. | $\Theta(\log n)$[b] | $\Theta(n)$ |
| Skew binomial[15] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(\log n)$[b] | $\Theta(n)$ |
| 2–3 heap[17] | $\Theta(1)$ | $O(\log n)$ am. | $\Theta(1)$ | $\Theta(1)$ am. | $O(\log n)$[b] | $\Theta(n)$ |
| Bottom-up skew[11] | $\Theta(1)$ | $O(\log n)$ am. | $O(\log n)$ am. | $\Theta(1)$ am. | $\Theta(1)$ am. | $\Theta(n)$ am. |
| Pairing[18] | $\Theta(1)$ | $O(\log n)$ am. | $o(\log n)$ am.[c] | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| Rank-pairing[21] | $\Theta(1)$ | $O(\log n)$ am. | $\Theta(1)$ am. | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| Fibonacci[10][2] | $\Theta(1)$ | $O(\log n)$ am. | $\Theta(1)$ am. | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| Strict Fibonacci[22][d] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| Brodal[23][d] | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$[24] |

Figure 1. Time complexities of various heap

In Dijkstra algorithm, operations make-heap, find-min, delete-min, decrease-key and insert are used. Through Figure 1, we can find Fibonacci heap is an ideal heap for Dijkstra, whose time complexities of each operation is the lowest. In This project, we will test Fibonacci heap, Binomial heap and skew heap by running them and recording their runtime, finally to find the best heap for Dijkstra.

## 2.1 Fibonacci Heap

Fibonacci heap is a type of mergeable heap that supports the following five operations:

**MAKE-HEAP()**: Create and return a new heap with no elements.

**INSERT(H, x)**: Insert an element with a given key x into heap H.

**MINIMUM(H)**: Return a pointer to the element with the minimum key in heap H.

**EXTRACT-MIN(H)**: Delete the element with the minimum key from heap H and return a pointer to it.

**UNION(H1, H2)**: Create and return a new heap that contains all elements of heaps H1 and H2. The heaps H1 and H2 are "destroyed" by this operation.

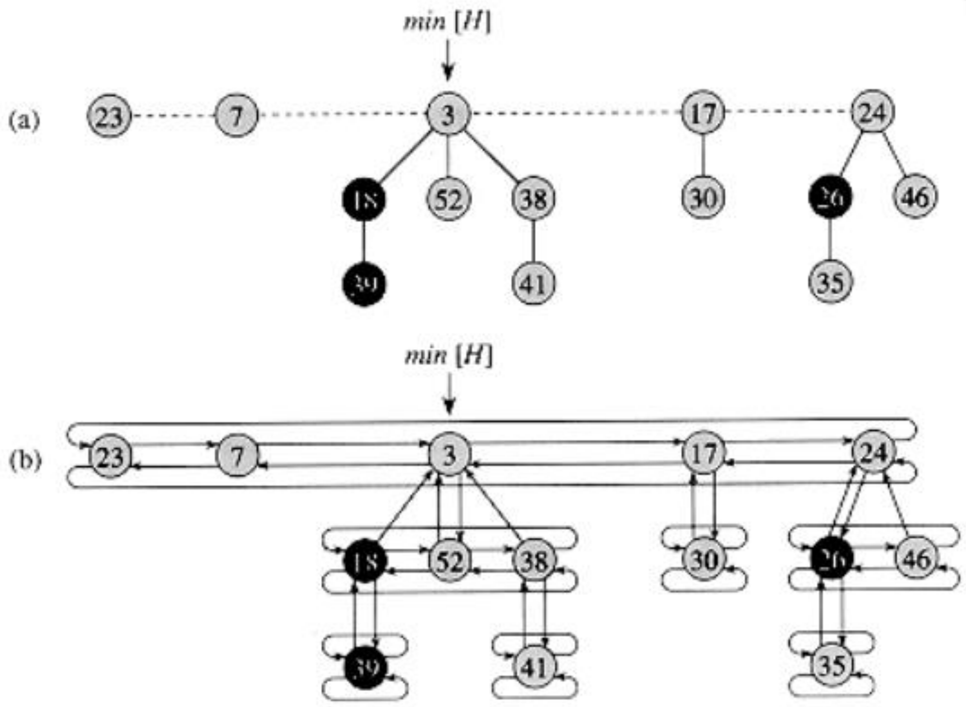
In addition to these operations, the Fibonacci heap also supports the following two operations:

**DECREASE-KEY(H, x, k)**: Change the key of element x in heap H to a new value k. Assume that the new value k is no greater than the current key.

**DELETE(H, x)**: Delete element x from heap H.

**2.1.1 Fibonacci Heap Structure:**

A Fibonacci heap is a collection of rooted trees with minimum heap properties. Each rooted tree has the property of a minimum heap.



The minimum of Fibonacci heap H can be accessed through H->min. This node points to the root of the tree with the minimum key.

In a Fibonacci heap, all tree roots are linked into a circular doubly linked list called the root list of the Fibonacci heap using their left and right pointers.

The child list of a node is also stored using a circular linked list, with H->min pointing to a random child, and the order between brothers is arbitrary. The degree of a node is the number of children it has.

**Potential Function:**

For future amortized analysis, we define the potential function of Fibonacci heap H as $\Phi(H) = t(H) + 2m(H)$

where t(H) represents the number of trees in the root list of H, and m(H) represents the number of marked nodes in H.

Maximum Degree

For amortized analysis, we first assume that for a Fibonacci heap with n nodes, the upper bound of the degree of any node is D(n), where $D(n) = O(\lg n)$.

## 2.1.2 Mergeable Heap Operations

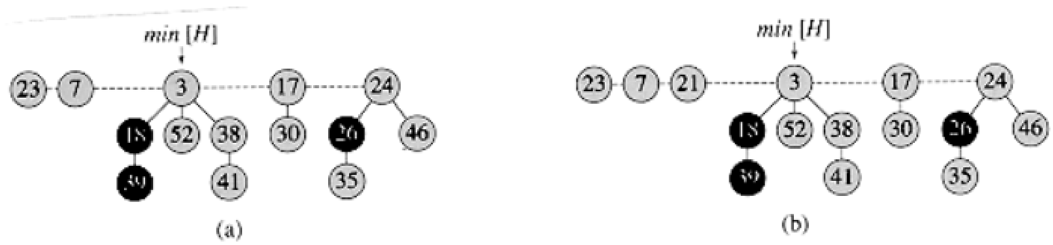
**Create a new heap**

The node structure in the heap is as follows:

```
struct FibNode {
    Position LeftSibling;
    Position RightSibling; // A circular linked list, with pointers to the left and right children.
    Position Parent;
    Position FirstChild; // Pointing to the parent and one of the children.
    bool Mark;
    int Element;// element:the distance to start
    int Degree; // Node degree
};
```

**Insert a node**

The insert operation is very convenient, it only needs to insert a node into the root list. As shown in the figure, simply insert 21 to the left of the 3.



**Amortized Analysis**

Let H’ be the heap after the operation, then we have t(H’) = t(H) + 1, M(H’) = M(H), and the increase in potential is: (t(H’) + 1 + 2m(H’)) - (t(H’) + 2m(H’)) = 1

The actual cost is O(1), so the amortized cost is O(1) + 1 = O(1).

**Fibonacci Heap Union**

Merge the root linked lists of X and Y.

X->RightSibling->LeftSibling = Y;

Y->RightSibling->LeftSibling = X;

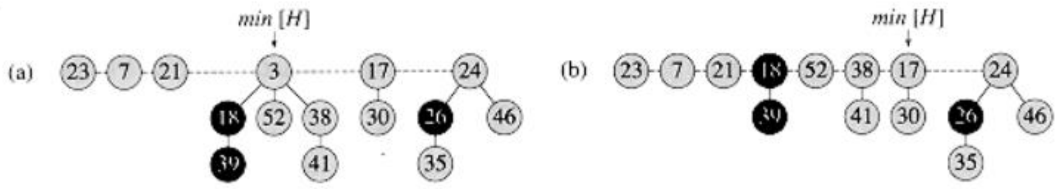
X->RightSibling = Y->RightSibling;

Y->RightSibling = X->RightSibling;

**Extract the minimum node**

First, promote all sons of the minimum node X to the root list, then remove X from the list.

First, arbitrarily point H->min to a root node, then perform CONSOLIDATE and merge root nodes to find H->min.



**Merge the root list**

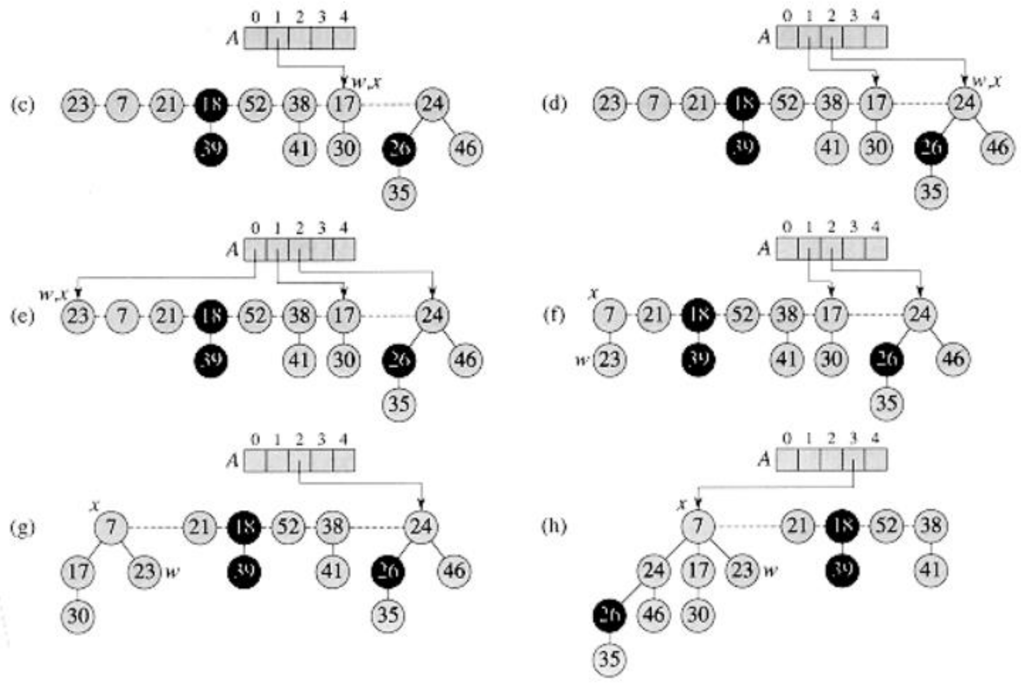
This is the most critical step of the Fibonacci heap, which reduces the time complexity through merging.

1. The process of merging the root list is to repeatedly execute the following steps until each root in the root list has a different degree.Find two roots with the same degree x and y in the root list, and assume x.key ⩽ y.key without loss of generality.

2. Link y to x: Remove y from the root list, call HEAP-LINK to make

y a child of x. Increase x.degree by 1 and clear y's mark (the role of y will be mentioned later).

Use an auxiliary array A[0, … D(n)]



**Amortized Analysis**

In the CONSOLIDATE step, the size of the root list is at most D(n) + t(H) - 1. The for loop traverses the root list, and the while loop's merge operations will merge the root nodes at most the number of times (each time merging nodes - 1), thus the actual workload of extracting the smallest node is O(D(n) + t(H)).

The potential before extraction is t(H) + 2m(H), because at most D(n) + 1 nodes remain and no nodes are marked. Therefore, the maximum potential after the operation is D(n) + 1 + 2m(H).

The change in potential is at most:

$$(t(H) + D(n) + 1 + 2m(H) - t(H) - 2m(H)) = O(D(n))$$

### 2.1.3 Key Decrease and Deletion

**Key Decrease**

Find X and break the link between X and its parent Y, then add X to the root list.

The key here is the cascading operation. After X is cut off, Y's mark may change, causing Y to be cut off as well to maintain the Fibonacci heap.

**Mark Property**

Use the mark property to record the state of nodes. Assume the following steps have already been performed on X:

1. At some point, X is a root.
2. Then X is linked to another node (becomes a child node).
3. Then X loses its second child and is cut off.

Once the second child is lost, cut off X's link to its parent, making it a new root. This is the meaning of CASCADING.

Therefore, every time X is moved to the root, its mark is false, returning to state one.

**Amortized Analysis**

Assume that a DECREASE-KEY call causes c CASCADING operations, then the actual cost is O(c).

Next, calculate the change in potential. Except for the last cascade, each time will make a node's mark become false, so at most m(H) + c - 2 nodes are marked (reduced by c - 1, and the last time may increase by one, the cut node may be false). At this time, the Fibonacci heap contains t(H) + c trees.

So the potential change is at most:

$$(t(H) + c + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$$

The amortized cost of the Fibonacci heap is at most O(1) + 4 - c = O(1).

**Deletion of a Node**

The cost of DELETE is the sum of the costs of DECREASE-KEY and EXTRACT-MIN, so the amortized time for DELETE is O(lg n).

## 2.2 Binomial Heap

A binomial heap is implemented as a set of binomial trees (compare with a binary heap, which has a shape of a single binary tree), which are defined recursively as follows:

1. A binomial tree of order 0 is a single node

2. A binomial tree of order k-1,k-2,...,2,1,0(in this order)

A binomial tree of order k has 2^k nodes,and height k. Because of its structure,a binomial tree of order k can be constructed from two trees of order k-1 by attaching one of them as the leftmost child of the root of the other tree. This feature is central to the merge operation of a binomial heap,which is its major advantage over other conventional heaps.

In order to use a Binomial Heap to implement Djikastar algorithm.

We need to create the following functions:

**biQP CreateQ(int n)**: Creates a new Binomial Heap with n elements.

**void AddBi(biQP Q, biQP tempQ, int v,int key)**: Adds a new element with value v and key key to the Binomial Heap Q.

**int* DeleteminBi(biQP Q,biQP tempQ)**: Pop the minimum element from the Binomial Heap Q and return its v and key.

**void DestroyQ_tempQ(biQP Q ,biQP tempQ)**: Destroys the Binomial Heap Q.

There will be more specific details about each functioin below.We also make some changes to the data structure of the Binomial Heap to make it more efficient for Djikastar algorithm.

### 2.2.1 Data Structure

```
typedef struct biQ* biQP;

typedef struct biQ {

    int n;

    int nodeNum;

    int minindex;

    biQnodeP* whereNode;

    biQnodeP* Trees;

}biQ;
```

n: The number of total vertices in the graph.(it tells the max size of the colleciton of trees)

nodeNum: The total number of nodes in the binomial heap.

minindex: The index of the minimum root node in the binomial heap.

whereNode: An array of pointers to the nodes (whose v is equal to the index of the array) in the binomial heap(This is uesed to quickly find the nodes if its key should be decreased)

Trees: An array of pointers to the root nodes of the binomial trees.(which is the actual structure of the binomial heap)

```
typedef struct biQnode* biQnodeP;
typedef struct biQnode{
    int key;
    int v;
    biQnodeP parent;
    biQnodeP firstchild;
    biQnodeP nextsibling;
}biQnode;
```

key: The distance from the source vertex to the current vertex.

v: The value of the current vertex.

parent: A pointer to the parent node.(This is used to percolate up the node in the heap efficiently)

firstchild: A pointer to the first child node.

nextsibling: A pointer to the next sibling node.

### 2.2.2 Function Details

#### CreateQ

The binomial heap can be seen as a binary number. However,the array's size is limited,so we need to maximize it which is the number of vertices in the graph . In this function,we initialize the binomial heap with n as the size of the `Trees` and `whereNode`. At first ,all the elemnts of the array are NULL.

#### AddBi

This function is uesd to add new elments into the heap. (Although what we add is the node we define above , here we only take its `v` and `key` as the element we add.)If `v` has already been added into the heap,we shouldn't add it again , Instead ,we should update its `key` if the new key is smaller than the old key(which satisfies the Dijkstra's algorithm).If `v` is not in the heap,we will add the `v` and `key` into the heap directely.

Therefore we put two other functions in this function:

**void DecreaseKeyBi(biQP Q, int v, int key)**: This function is used to decrease the old key of the node with value `v` to new `key`.

1. Check if v's key is bigger than the new key.

2. If it is,we update the key of the node with value `v` to the new key.

3. Do percolation upwards until has no parents or the parent's key is smaller than the new key.

**void MergeBi(biQP Q1, biQP Q2)**:

In this funtion ,we put in two binomial heaps `Q1` and `Q2` and merge Q2 into Q1. Because the structure of the binomial heap is a set of binomial trees, we can merge two binomial heaps by attaching the root of one tree to the root of the other tree , which is similar to add two binary numbers. Therefore we use `T1` `T2` `carry` to 'add' Q2 into Q1.(`biQnodeP MergeTrees(biQnodeP T1, biQnodeP T2)`is used to do the merging of two binomial trees which is easy to understand so we won't explain it here.)

You may notice that we also put a `tempQ` parameter in the function. This is because to fit the API of the merge algorithm, we need another heap to store the new elments,However if we malloc it in the function and free it after the merge, it will cause a waste of plenty time,which will bring down the efficiency of the algorithm. Therefore we should create two heap in the beginning one is uesd to store the real data and the other is uesed for merge operation.

With the help of these two functions, we can add new elements into the binomial heap efficiently.

1. check if `v` is already in the heap.

2. If it is,call `DecreaseKeyBi` to update its key.

3. If it is not,put the new element into `tempQ` and call `MergeBi`

4. clear the `tempQ` and return.

**DeleteminBi**

The core of the Djikstra's algorithm is to find the next vertex that is most close to the source vertex. Therefore, we need to pop the node with the minimum key from the heap. In order to do this in O(1) time , we need to have a 'pointer' to the min root in the `Trees`, so there is a `minindex` in the `biQ` structure.

After we pop the node with the minimum key, its children will become another 'binomial heap'(we ues `tempQ` to store its children),which means all we have to do is ues merge operation to merge it into the main heap.

You may wonder how can we calculate minindex without wasting time to traverse the whole heap. The answer is that I put it in the `MergeBi` function. When we merge two binomial heaps,its no way to avoid traverse both heaps, so we can calculate the `minindex` in this step.

1. Find the minimum root node in the `Trees` array through `minindex` and free it afterwards.

2. Put its children into `tempQ` and call `MergeBi` to merge it into the main heap.

3. Clear the `tempQ` and return the `v` and `key` of the minimum root node.

**DestroyQ_tempQ**

1. Free all the nodes in the `Trees` array.

2. Free the `biQ` structure.

3. Free the `tempQ` structure.

### 2.2.3 Time Complexity

CreateQ: O(n)

AddBi: 1. if `v` is in the heap: O(logn) because we need to update its key and percolate upwards.

2. if `v` is not in the heap: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. Because of the merge, a single insertion takes time O(logn). However, this can be sped up using a merge procedure that shortcuts the merge after it reaches a point where only one of the merged heaps has trees of larger order. With this speedup, across a series of k consecutive insertions, the total time for the insertions is O(k+logn).Another way of stating this is that each successive insert has an amortized time of O(1) per insertion.

DeleteminBi: To delete the minimum element from the heap, first find this element, remove it from the root of its binomial tree, and obtain a list of its child subtrees (which are each themselves binomial trees, of distinct orders). 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 root has at most logn children, creating this new heap takes time O(logn), Merging heaps takes time O(logn), so the entire delete minimum operation takes time O(logn).

### 2.2.4 Space Complexity

The total space we use in the binomial heap to solve the problem mostly the main heap and the temp heap which is 4*n, so the space complexity is O(n).

## 2.3 Skew Heap

A skew heap (or self-adjusting heap) is a heap data structure implemented as a binary tree. Skew heaps are advantageous because of their ability to merge more quickly than binary heaps. In contrast with binary heaps, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied:

The general heap order must be enforced. Every operation (add, remove_min, merge) on two skew heaps must be done using a special skew heap merge.

In order to use it to solve the shortest path problem,we design the following functions:

**SkewheapP CreateSkewheap(int n)**: creates a new empty skew heap with `n` vertices.

**void InsertSkew(SkewheapP heap, int v, int key)**: inserts a new vertex `v` with key `key` into the skew heap `heap`.

**int* DeleteminSkew(SkewheapP heap)**: removes the vertex with the minimum key from the skew heap `heap` and returns its value.

**void DestroySkew(SkewheapP heap)**: destroys the skew heap `heap`.

We will explain them later.

### 2.3.1 Data structure

```
typedef struct sNode* sNodeP;
typedef struct sNode{
    int key;
    int v;
    sNodeP left, right;
    sNodeP parent;
}sNode;
```

This is the structure of a node in the skew heap. Each node has a key, a vertex, and two pointers to its left and right children, as well as a pointer to its parent.

```
typedef struct Skewheap* SkewheapP ;
typedef struct Skewheap{
    sNodeP root;
    sNodeP* wherenode;
}Skewheap ;
```

This is the structure of the skew heap. It has a pointer to the root node, and a pointer to an array of pointers to the nodes where each vertex is stored. This is used to quickly find the node where a vertex is stored(which will help us do decresekey).

### 2.3.2 Function Details

**CreateSkewheap**

In this function ,we create a new empty skew heap with `n` vertices. We initialize the `root` pointer to `NULL` and the `wherenode` array to `NULL`.

**InsertSkew**

In this function , we should analyze whether we should insert the new vertex into the heap or decresekey it. So we check if the vertex is already in the heap or not. If it is not in the heap, we create a new node and insert it into the heap. If it is already in the heap, we decresekey it.

Here we use two other functions to do this:

**void DecreasekeySkew(SkewheapP heap, int v, int key)**: This is similar to the Decreasekey function in the binomial heap. It decreases the key of the vertex `v` to `key` in the skew heap `heap`.

**sNodeP insertSkew(sNodeP root, sNodeP* wherenode, int v ,int key)**: This function inserts a new node with vertex `v` and key `key` into the skew heap,which call another function `sNodeP MergeSkew(sNodeP root1, sNodeP root2)` to merge the new node with the root of the heap.

When two skew heaps are to be merged, we can use a similar process as the merge of two leftist heaps:

Compare roots of two heaps; let p be the heap with the smaller root, and q be the other heap. Let r be the name of the resulting new heap.

Let the root of r be the root of p (the smaller root), and let r's right subtree be p's left subtree.

Now, compute r's left subtree by recursively merging p's right subtree with q.

**DeleteminSkew**

In this function we use another function `int* deleteminSkew(sNodeP* root)` to remove the vertex with the minimum key from the skew heap. And then ajust the `wherenode` array.

In deleteminSkew, we can simply delete the root and then merge its two children as two separate heaps.

**DestroySkew**

This function is called to free the skew heap.

### 2.3.3 Time Complexity

By amortized analysis, the time complexity of merge two skew heap is O(log n)(so do insert), where n is the number of vertices in the heap. Because after each delete we should call merge to balance the skew heap.Therefore the time complexity of delete is  O(log n).

### 2.3.4 Space Complexity

The space complexity of a skew heap is O(n), where n is the number of vertices in the heap. This is because we need to store the nodes of the heap and the `wherenode` array. And there will be at most n node in the heap.

## 2.4 STL priority_queue

In the STL, the priority_queue is implemented using a standard binary heap, specifically an array based implementation that logically corresponds to a complete binary tree. Compared to other linked storage heaps, the time complexity for merging binary heaps is O(n), (Others are O(log n)).

**Insert**

Heapification from bottom to top (sifting up): To insert element X, we create a hole at the last available position. If placing this element does not violate the order, the insertion is complete. Otherwise, the parent node is moved down, and this node is sifted up. This process is repeated until the order is satisfied.

**Delete**

Heapification from top to bottom (sifting down): Since we are building a min-heap, deleting the minimum element means removing the root node, which disrupts the order. To maintain the structure, the last element X in the heap must be moved to the appropriate position. If it can be placed directly into the hole, the deletion is complete (which is generally not possible); otherwise, the smaller of the two children of the hole is moved into it, causing the hole to move down a level. This process continues until

X can be placed in the hole, thus satisfying both the structural and heap order properties.

# Chapter 3: Testing Results

Data source: http://www.diag.uniroma1.it/challenge9/download.shtml

Considering that the points and edges provided on the website are relatively sparse, with an average of about 2 edges per point, it may not be suitable for comparing the time complexities of the three different heap-optimized Dijkstra algorithms. Therefore, after completing the tests on 12 sets of data, we will add randomly generated complete graphs based on the testing conditions, resulting in the number of edges m=n(n−1). This will allow us to compare the time complexity differences between O((m + n)log n) and O(m + n log n), and we will create a runtime table for visual representation.

Since the provided data does not represent a completely connected graph, after completing the partitioning of connected components, we decided to eliminate the 1000 queries. Instead, we will treat each node in every connected component as a starting point to run a single-source shortest path algorithm. This will increase the actual number of queries and widen the time differences needed to complete the test data.

The test data and running times are as follows:

| | *n* | *m* | *STLHeap* | *BiHeap* | *FibHeap* |
|---|---|---|---|---|---|
| *testdata1* | 264346 | 733,846 | 0.121 | 0.091 | 0.135 |
| *testdata2* | 321270 | 800,172 | 0.137 | 0.12 | 0.185 |
| *testdata3* | 435666 | 1,057,066 | 0.184 | 0.16 | 0.225 |
| *testdata4* | 1070376 | 2,712,798 | 0.427 | 0.377 | 0.538 |
| *testdata5* | 1207945 | 2,840,208 | 0.522 | 0.473 | 0.69 |
| *testdata6* | 2758119 | 6885658 | 1.186 | 1.072 | 1.578 |
| | | | | | |
| *complete1* | 2000 | 3998000 | 0.039 | 0.022 | 0.023 |
| *complete2* | 3500 | 12246500 | 0.096 | 0.082 | 0.086 |
| *complete3* | 5000 | 24995000 | 0.189 | 0.174 | 0.176 |
| *complete4* | 8000 | 63992000 | 0.475 | 0.431 | 0.462 |
| *complete5* | 10000 | 99990000 | 0.733 | 0.687 | 0.714 |

Figure1: Evaluation of 6 Groups of U.S. Road Networks and 6 Groups of Complete Graphs

# Chapter 4: Analysis and Comments

By analyzing the above data, we can see that for sparse graphs using Dijkstra's algorithm to find the single-source shortest path, in some cases, the running time of using a Fibonacci heap exceeds that of using a binomial heap or a skew heap. This is mainly because the number of edges m in the provided data is approximately equal to 2n, which causes the time complexity of the binomial heap and skew heap to reduce to O(n log n), while the time complexity of the Fibonacci heap is O(2n + n log n). When n<300000, the difference between the two is not significant. Additionally, because the Fibonacci heap has many constant-time operations during the consolidate process, the actual time required to maintain the heap exceeds the theoretical time complexity, resulting in longer running times when using the Fibonacci heap compared to the binomial and skew heaps.

In randomly generated complete graphs, where m=n(n−1), the value of m*m* is sufficiently large. Therefore, when running Dijkstra's algorithm in a complete graph, the time complexity for the binomial heap and skew heap is O((m + n) log n, while the time complexity for the Fibonacci heap is O(m + n log n). There is a significant difference between the two. However, due to the randomness of the generated data, the n*n* points correspond to n−1edges, and not every edge necessarily requires a relaxation operation. As a result, the advantage of the Fibonacci heap's amortized time complexity of O(1) for the decrease-key operation compared to the binomial heap's O(log n) time complexity cannot be fully realized in this scenario. Therefore, it is necessary to specifically construct data such that every edge undergoes relaxation in order to bring the time complexities of the binomial heap and Fibonacci heap closer to their theoretical complexities. Otherwise, when using the Fibonacci heap for Dijkstra's algorithm, each time a deletemin operation is performed, it requires a consolidate operation that traverses the entire root list, resulting in a significant number of constant-time operations. This time cost is associated with the number of nodes n.

## Appendix: Source Code (in C)

## Declaration

***I hereby declare that all the work done in this project titled " Performance Measurement" is of my independent effort.***