UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

JOÃO HENRIQUE MOURO SUAIDEN
JOÃO PEDRO DE FREITAS ZANQUI
LUCAS YUKI NISHIMOTO
Orientador: João Paulo Papa

COMPLEXIDADES HEAP

Bauru, SP
2023

Sumário

1 Binário

Um heap binário é uma estrutura de dados especializada em armazenar elementos com uma relação de ordem parcial. É uma árvore binária completa em que cada nó possui um valor associado, chamado de chave.

Existem dois tipos de heaps binários: o heap máximo e o heap mínimo. No heap máximo, o valor associado a cada nó é maior ou igual aos valores associados aos seus filhos. Já no heap mínimo, o valor associado a cada nó é menor ou igual aos valores associados aos seus filhos.

A principal propriedade de um heap binário é a propriedade do heap, que garante que, para cada nó na árvore, o valor associado ao nó seja maior (ou menor) do que os valores associados aos seus filhos. Essa propriedade permite que o elemento de maior (ou menor) valor esteja sempre na raiz da árvore.

Um heap binário pode ser utilizado para implementar uma fila de prioridade, onde o elemento de maior (ou menor) prioridade pode ser rapidamente acessado e removido. Além disso, os heaps binários são frequentemente utilizados em algoritmos de ordenação, como o heapsort.

1.1 Inserção

Primeiro, o novo elemento é adicionado ao final do heap. Em seguida, o novo elemento é comparado com seu pai (nó pai na árvore) para verificar se a propriedade do heap é violada. Se a propriedade do heap não for violada (por exemplo, se o novo elemento for menor que seu pai em um heap máximo), a inserção é concluída. Caso a propriedade do heap seja violada, ocorre uma troca entre o novo elemento e seu pai. Isso coloca o elemento recém-inserido na posição correta no heap, preservando a propriedade do heap. O passo 3 é repetido até que o novo elemento esteja na posição correta e a propriedade do heap seja preservada em toda a árvore. O processo de inserção garante que o novo elemento suba na árvore, se necessário, até que ele esteja em uma posição onde sua chave seja maior (ou menor) do que as chaves de seus filhos. Dessa forma, a propriedade do heap é mantida.

A complexidade da inserção em um heap binário é O(log n), onde n é o número de elementos no heap. Isso ocorre porque a inserção pode envolver uma comparação e troca de posição com cada pai na árvore, que possui uma altura máxima de log n.

1.2 Extração

O elemento raiz do heap é removido. Esse é o elemento de maior valor em um heap máximo, ou de menor valor em um heap mínimo. Em seguida, o último elemento do heap (que está no nível mais baixo e mais à direita da árvore) é movido para a posição da raiz. Essa troca é feita para preservar a propriedade de uma árvore binária completa. Agora, o novo elemento na raiz é comparado com seus filhos para verificar qual filho tem o maior (ou menor) valor. Isso é necessário para manter a propriedade do heap. Se o novo elemento não estiver na posição correta em relação aos seus filhos, ocorre uma troca entre o elemento raiz e o filho adequado. Esse processo é repetido até que o novo elemento esteja na posição correta e a propriedade do heap seja preservada em toda a árvore. O processo de extração garante que o elemento substituto na raiz desça na árvore, trocando de posição com o filho apropriado para manter a propriedade do heap. Dessa forma, o próximo elemento de maior (ou menor) valor estará na raiz após a extração.

A complexidade da extração em um heap binário é O(log n), onde n é o número de elementos no heap. Isso ocorre porque a extração envolve uma comparação e uma possível troca de posição com cada filho na árvore, que possui uma altura máxima de log n.

1.3 Busca

O heap binário não é otimizado para busca direta, pois não possui uma estrutura que facilite a localização rápida de um elemento com base em seu valor.

No entanto, se você souber o valor exato do elemento que está procurando, é possível realizar uma busca sequencial percorrendo todos os elementos do heap, comparando-os com o valor desejado. Essa busca sequencial tem uma complexidade de tempo de O(n), onde n é o número de elementos no heap, já que pode ser necessário percorrer todos os elementos para encontrar o elemento desejado.

A principal vantagem do heap binário está na inserção e extração eficientes dos elementos com base em sua prioridade. Se a busca de elementos específicos for um requisito frequente, pode ser mais adequado utilizar uma estrutura de dados diferente, como uma árvore binária de busca ou uma tabela hash, que oferecem uma busca mais eficiente em relação ao valor dos elementos.

1.4 Código

```
// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
```

```
class MinHeap
{
        int *harr; // pointer to array of elements in heap
        int capacity; // maximum possible size of min heap
        int heap_size; // Current number of elements in min
           heap
public:
        // Constructor
        MinHeap(int capacity);

        // to heapify a subtree with the root at given index
        void MinHeapify(int );

        int parent(int i) { return (i-1)/2; }

        // to get index of left child of node at index i
        int left(int i) { return (2*i + 1); }

        // to get index of right child of node at index i
        int right(int i) { return (2*i + 2); }

        // to extract the root which is the minimum element
        int extractMin();

        // Decreases key value of key at index i to new_val
        void decreaseKey(int i, int new_val);

        // Returns the minimum key (key at root) from min
           heap
        int getMin() { return harr[0]; }

        // Deletes a key stored at index i
        void deleteKey(int i);

        // Inserts a new key 'k'
        void insertKey(int k);
};

```

```
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
        heap_size = 0;
        capacity = cap;
        harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
        if (heap_size == capacity)
        {
                cout << "\nOverflow: Could not insertKey\n";
                return;
        }

        // First insert the new key at the end
        heap_size++;
        int i = heap_size - 1;
        harr[i] = k;

        // Fix the min heap property if it is violated
        while (i != 0 && harr[parent(i)] > harr[i])
        {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
        }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
        harr[i] = new_val;
        while (i != 0 && harr[parent(i)] > harr[i])
        {
```

```
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
        }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
        if (heap_size <= 0)
                return INT_MAX;
        if (heap_size == 1)
        {
                heap_size--;
                return harr[0];
        }

        // Store the minimum value, and remove it from heap
        int root = harr[0];
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);

        return root;
}


// This function deletes key at index i. It first reduced
   value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
        decreaseKey(i, INT_MIN);
        extractMin();
}

// A recursive method to heapify a subtree with the root at
   given index
// This method assumes that the subtrees are already
   heapified
```

```
void MinHeap::MinHeapify(int i)
{
        int l = left(i);
        int r = right(i);
        int smallest = i;
        if (l < heap_size && harr[l] < harr[i])
                smallest = l;
        if (r < heap_size && harr[r] < harr[smallest])
                smallest = r;
        if (smallest != i)
        {
                swap(&harr[i], &harr[smallest]);
                MinHeapify(smallest);
        }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
```

2 Binominal

Um heap binomial é uma estrutura de dados formada por uma coleção de árvores binomiais, que é uma árvore binária com propriedades especiais:

- Cada árvore binomial é uma árvore binária completa, ou seja, todas as posições dos nós estão preenchidas, exceto possivelmente a última camada, que é preenchida da esquerda para a direita.
- Em cada árvore binomial, o valor do nó pai é estritamente menor (ou maior) que os valores dos nós em suas subárvores filhas.
- A altura de cada árvore binomial é igual ao número de seus nós.
- Um heap binomial permite operações eficientes, como a inserção, união e remoção de elementos.
- A inserção em um heap binomial envolve a criação de uma árvore binomial de ordem 0 (um único nó) e sua fusão com o heap binomial existente.
- A união de dois heaps binomiais envolve a fusão de suas árvores binomiais correspondentes.
- A remoção do elemento mínimo (ou máximo) em um heap binomial envolve a identificação e a fusão das árvores binomiais com o mesmo número de nós.

A principal vantagem dos heaps binomiais é que eles garantem que a operação de união de dois heaps binomiais possa ser realizada em tempo O(log n), onde n é o número total de elementos nos heaps. Isso torna os heaps binomiais eficientes em termos de tempo para operações como inserção e remoção em comparação com outras estruturas de dados como árvores binárias de busca.

2.1 Inserção

Cria-se uma heap binomial com uma única árvore binomial de ordem 0 contendo o elemento a ser inserido. É realizada a união do heap binomial existente com o heap binomial recém-criado, essa união envolve a fusão das árvores binomiais correspondentes nos dois heaps.

Para realizar a união:

- Compare as raízes das árvores binomiais nos dois heaps. A árvore com o elemento de menor valor (ou maior, dependendo do tipo de heap) se tornará a árvore raiz no heap resultante.

- Faça a árvore com a raiz menor (ou maior) se tornar a subárvore esquerda da outra árvore. Isso é feito através da atualização dos ponteiros de próxima e anterior das árvores binomiais envolvidas.
- Se as duas árvores tiverem o mesmo grau (ou seja, a mesma ordem), faça a árvore que estava no heap original se tornar a subárvore direita da outra árvore.
- Se as duas árvores tiverem graus diferentes, simplesmente faça a árvore com o menor grau se tornar a subárvore direita da outra árvore.
- Atualize o grau da árvore resultante.

O processo de inserção em um heap binomial é eficiente, com complexidade de tempo O(log n), onde n é o número total de elementos nos heaps. Isso ocorre porque a inserção envolve principalmente operações de fusão de árvores binomiais, que são operações eficientes.

2.2 Encontrar o mínimo

São percorridas todas as árvores binomiais no heap binomial e é encontrada a árvore binomial que contém o elemento mínimo. Essa árvore será a raiz do heap. Dentro da árvore binomial selecionada, encontra-se o nó com o menor valor. Esse nó será o elemento mínimo no heap binomial. Ao encontrar o mínimo elemento em um heap binomial, o tempo de execução é eficiente, com uma complexidade de tempo de O(log n), onde n é o número total de elementos nos heaps. Isso ocorre porque você precisa percorrer as árvores binomiais e comparar os valores dos nós para encontrar o elemento desejado.

2.3 Diminuir chave

É localizado o nó que contém o elemento cuja chave será diminuída e então percorre-se todas as árvores binomiais do heap binomial para encontrar o nó desejado. Isso pode exigir uma busca sequencial por meio das árvores binomiais e seus nós. A chave do nó selecionado é atualizada para o novo valor desejado, diminuindo sua chave. Se a chave diminuída violar a propriedade do heap binomial (ou seja, se a chave diminuída for menor que a chave do nó pai), é realizada uma troca entre o nó atual e seu pai. Continua fazendo essa troca até que a propriedade do heap binomial seja restaurada ou até que o nó atual se torne a raiz da árvore binomial. Ao diminuir a chave de um elemento em um heap binomial, é importante manter a propriedade do heap binomial, garantindo que a chave do nó seja sempre maior (ou menor) do que a chave de seus nós filhos.

A complexidade de diminuir a chave em um heap binomial é eficiente, com uma complexidade de tempo de O(log n), onde n é o número total de elementos nos heaps. Isso ocorre porque a diminuição da chave envolve principalmente a atualização da chave e uma possível troca com o pai, que não exige uma reestruturação completa do heap binomial.

2.4 Remover Mínimo

Encontra-se a árvore binomial com o elemento mínimo no heap binomial. Essa árvore será a raiz do heap. Após isso a árvore binomial com o elemento mínimo do heap é removida. A árvore removida é separada em várias árvores binomiais menores, uma para cada subárvore da raiz removida. Cada uma dessas árvores menores se tornará um heap binomial separado. Realiza-se a união dos heaps binomiais resultantes do passo anterior para formar um novo heap binomial.

O processo de remoção do mínimo em um heap binomial é eficiente, com uma complexidade de tempo de O(log n), onde n é o número total de elementos nos heaps. Isso ocorre porque a remoção envolve principalmente operações de fusão de árvores binomiais, que são operações eficientes.

Ao remover o elemento mínimo em um heap binomial, o heap continua a obedecer às propriedades do heap binomial, e o próximo elemento mínimo pode ser encontrado com eficiência.

2.5 Remoção

Localiza-se o nó que contém o elemento desejado, todas as árvores binomiais no heap binomial são percorridas para encontrá-lo. Isso pode exigir uma busca sequencial por meio das árvores binomiais e seus nós. Após encontrar o nó desejado, sua chave é diminuida para o valor mínimo possível. Em seguida, o elemento mínimo é extraído do heap binomial, removendo a árvore binomial que contém o elemento mínimo e realizando as operações necessárias para manter as propriedades do heap binomial.

A remoção de um elemento específico em um heap binomial pode exigir uma combinação de diminuição da chave e extração. Portanto, a complexidade de tempo da remoção depende da complexidade dessas duas operações. No geral, a complexidade da remoção em um heap binomial é O(log n), onde n é o número total de elementos nos heaps.

2.6 Código

```cpp
// C++ program to implement different operations
// on Binomial Heap
#include<bits/stdc++.h>
using namespace std;
```

```
// A Binomial Tree node.
struct Node
{
int data, degree;
Node *child, *sibling, *parent;
};

Node* newNode(int key)
{
Node *temp = new Node;
temp->data = key;
temp->degree = 0;
temp->child = temp->parent = temp->sibling = NULL;
return temp;
}

// This function merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
// Make sure b1 is smaller
if (b1->data > b2->data)
        swap(b1, b2);

// We basically make larger valued tree
// a child of smaller valued tree
b2->parent = b1;
b2->sibling = b1->child;
b1->child = b2;
b1->degree++;

return b1;
}

// This function perform union operation on two
// binomial heap i.e. l1 & l2
list<Node*> unionBionomialHeap(list<Node*> l1,
                        list<Node*> l2)
{
```

```
// _new to another binomial heap which contain
// new heap after merging l1 & l2
list<Node*> _new;
list<Node*>::iterator it = l1.begin();
list<Node*>::iterator ot = l2.begin();
while (it!=l1.end() && ot!=l2.end())
{
        // if D(l1) <= D(l2)
        if((*it)->degree <= (*ot)->degree)
        {
        _new.push_back(*it);
        it++;
        }
        // if D(l1) > D(l2)
        else
        {
        _new.push_back(*ot);
        ot++;
        }
}

// if there remains some elements in l1
// binomial heap
while (it != l1.end())
{
        _new.push_back(*it);
        it++;
}

// if there remains some elements in l2
// binomial heap
while (ot!=l2.end())
{
        _new.push_back(*ot);
        ot++;
}
return _new;
}

```

```
// adjust function rearranges the heap so that
// heap is in increasing order of degree and
// no two binomial trees have same degree in this heap
list<Node*> adjust(list<Node*> _heap)
{
if (_heap.size() <= 1)
        return _heap;
list<Node*> new_heap;
list<Node*>::iterator it1,it2,it3;
it1 = it2 = it3 = _heap.begin();

if (_heap.size() == 2)
{
        it2 = it1;
        it2++;
        it3 = _heap.end();
}
else
{
        it2++;
        it3=it2;
        it3++;
}
while (it1 != _heap.end())
{
        // if only one element remains to be processed
        if (it2 == _heap.end())
        it1++;

        // If D(it1) < D(it2) i.e. merging of Binomial
        // Tree pointed by it1 & it2 is not possible
        // then move next in heap
        else if ((*it1)->degree < (*it2)->degree)
        {
        it1++;
        it2++;
        if(it3!=_heap.end())
                it3++;
        }
```

```

        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are same
        // in heap
        else if (it3!=_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
        it1++;
        it2++;
        it3++;
        }

        // if degree of two Binomial Tree are same in heap
        else if ((*it1)->degree == (*it2)->degree)
        {
        Node *temp;
        *it1 = mergeBinomialTrees(*it1,*it2);
        it2 = _heap.erase(it2);
        if(it3 != _heap.end())
                it3++;
        }
}
return _heap;
}

// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
                        Node *tree)
{
// creating a new heap i.e temp
list<Node*> temp;

// inserting Binomial Tree into heap
temp.push_back(tree);

// perform union operation to finally insert
// Binomial Tree in original heap
temp = unionBionomialHeap(_heap,temp);
```

```

return adjust(temp);
}

// removing minimum key element from binomial heap
// this function take Binomial Tree as input and return
// binomial heap after
// removing head of that tree i.e. minimum element
list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
list<Node*> heap;
Node *temp = tree->child;
Node *lo;

// making a binomial heap from Binomial Tree
while (temp)
{
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
}
return heap;
}

// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
Node *temp = newNode(key);
return insertATreeInHeap(_head,temp);
}

// return pointer of minimum value Node
// present in the binomial heap
Node* getMin(list<Node*> _heap)
{
list<Node*>::iterator it = _heap.begin();
Node *temp = *it;
while (it != _heap.end())
```

```
{
        if ((*it)->data < temp->data)
        temp = *it;
        it++;
}
return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
list<Node*> new_heap,lo;
Node *temp;

// temp contains the pointer of minimum value
// element in heap
temp = getMin(_heap);
list<Node*>::iterator it;
it = _heap.begin();
while (it != _heap.end())
{
        if (*it != temp)
        {
        // inserting all Binomial Tree into new
        // binomial heap except the Binomial Tree
        // contains minimum element
        new_heap.push_back(*it);
        }
        it++;
}
lo = removeMinFromTreeReturnBHeap(temp);
new_heap = unionBionomialHeap(new_heap,lo);
new_heap = adjust(new_heap);
return new_heap;
}

// print function for Binomial Tree
void printTree(Node *h)
{
while (h)
```

```
{
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
}
}

// print function for binomial heap
void printHeap(list<Node*> _heap)
{
list<Node*> ::iterator it;
it = _heap.begin();
while (it != _heap.end())
{
        printTree(*it);
        it++;
}
}
```

3 Fibonacci

Um heap de Fibonacci é uma estrutura de dados que combina as propriedades de um heap binomial com a eficiência de certas operações de uma sequência de Fibonacci. Ele é usado para implementar uma fila de prioridade que suporta inserção, remoção e diminuição de chave em tempo amortizado O(1) para algumas operações.

Um heap de Fibonacci é composto por uma coleção de árvores de Fibonacci, que são árvores binomiais especiais. Cada árvore de Fibonacci é uma árvore com nós que possuem chaves e ponteiros para seus filhos, irmãos e pai. Além disso, cada árvore de Fibonacci atende a algumas propriedades específicas:

Cada nó possui um grau diferente, ou seja, nenhum par de nós possui o mesmo número de filhos diretos. Cada nó tem um ponteiro para seu pai, mas não possui um ponteiro para seu irmão direito. A raiz de cada árvore de Fibonacci é um nó com a chave mínima. Cada nó, exceto a raiz, possui um campo booleano "marcado"que indica se o nó perdeu um filho desde a última vez que se tornou filho de seu pai. As operações principais em um heap de Fibonacci são:

O heap de Fibonacci tem uma complexidade de tempo amortizado O(1) para inserção, remoção mínima e diminuição de chave, o que o torna eficiente em cenários onde essas operações são realizadas com frequência. No entanto, ele possui um custo de espaço um pouco maior em comparação com outras estruturas de dados de heap.

3.1 Inserção

- Criação de uma árvore de Fibonacci isolada contendo o elemento a ser inserido.
- Mescla da árvore de Fibonacci recém-criada com a coleção existente de árvores de Fibonacci no heap.
- Atualização do ponteiro para o mínimo (ou máximo) elemento, se necessário.

Ao realizar a inserção, os seguintes casos podem ocorrer:

- Se o heap de Fibonacci estiver vazio, a árvore de Fibonacci recém-criada se torna a única árvore no heap.
- Se o heap de Fibonacci não estiver vazio, a árvore de Fibonacci recém-criada é mesclada com a árvore de Fibonacci existente de mesma ordem (mesmo grau). Essa mescla envolve comparar as raízes das duas árvores e tornar a árvore com o elemento de menor valor a subárvore da outra árvore.

- Após a mescla, se houver mais de uma árvore de Fibonacci com a mesma ordem, elas são mescladas recursivamente até que não haja mais árvores com a mesma ordem. Isso garante que não haja duas árvores de Fibonacci com a mesma ordem no heap.

Durante a inserção, também é importante atualizar o ponteiro para o mínimo (ou máximo) elemento do heap, se necessário. Se o elemento recém-inserido for menor (ou maior) do que o mínimo (ou máximo) atualmente armazenado, o ponteiro para o mínimo (ou máximo) é atualizado para o novo elemento inserido.

A complexidade de tempo da inserção em um heap de Fibonacci é amortizada O(1), tornando-a uma operação eficiente. No entanto, o tempo de execução pode aumentar em casos raros, onde ocorre uma série de mesclas de árvores de Fibonacci de mesma ordem.

3.2 União

A operação de união em um heap de Fibonacci envolve a combinação de dois heaps de Fibonacci em um único heap. Isso é feito mesclando as coleções de árvores de Fibonacci de cada heap e atualizando o ponteiro para o mínimo elemento, se necessário.

- Verifique qual dos dois heaps de Fibonacci possui o menor elemento como sua raiz. Essa informação será usada para atualizar o ponteiro para o mínimo elemento.
- Anexe a coleção de árvores de Fibonacci do heap menor ao heap maior.
- Se houver árvores de Fibonacci com a mesma ordem em ambos os heaps, mescle-as para evitar duplicação de árvores com a mesma ordem.
- Atualize o ponteiro para o mínimo elemento, se necessário, com base na comparação entre as raízes dos dois heaps.

A operação de união em um heap de Fibonacci é uma operação eficiente com complexidade de tempo O(1). Isso ocorre porque a união envolve principalmente a combinação das coleções de árvores de Fibonacci e a atualização do ponteiro para o mínimo elemento, que pode ser feita em tempo constante.

A eficiência da operação de união é uma das vantagens dos heaps de Fibonacci em relação a outras estruturas de dados de heap. No entanto, vale ressaltar que outras operações, como inserção e remoção mínima, podem ter complexidades amortizadas mais altas em comparação com outras estruturas de heap.

3.3 Remoção mínima

A operação de remoção mínima em um heap de Fibonacci envolve a remoção do elemento mínimo do heap, seguida por algumas etapas de reestruturação para manter as propriedades do heap.

- Encontre o nó que contém o elemento mínimo no heap de Fibonacci. Esse nó será a raiz de uma das árvores de Fibonacci.
- Remova o nó mínimo do heap de Fibonacci.
- Separe os filhos do nó removido, tornando-os árvores independentes.
- Anexe as árvores independentes à coleção de árvores de Fibonacci no heap.
- Realize um passo de reestruturação chamado "passo de corte", que envolve a verificação e a atualização dos nós pais para garantir que cada nó pai tenha pelo menos uma criança marcada (indicando que perdeu um filho anteriormente).
- Faça uma segunda passagem de reestruturação chamada "passo de casamento", que envolve a verificação e a mescla de árvores com a mesma ordem (mesmo grau), garantindo que não haja duas árvores com a mesma ordem no heap.
- Atualize o ponteiro para o mínimo elemento, se necessário.

Durante a remoção mínima, as etapas de corte e casamento garantem que as propriedades do heap de Fibonacci sejam mantidas. O passo de corte ajuda a melhorar a eficiência do heap, reduzindo o número de nós que precisam ser considerados em operações futuras. O passo de casamento garante que não haja duas árvores de Fibonacci com a mesma ordem no heap, o que é essencial para manter a estrutura correta do heap.

A complexidade de tempo amortizado da remoção mínima em um heap de Fibonacci é O(log n), onde n é o número total de elementos no heap. Embora essa complexidade seja um pouco maior do que em outras estruturas de heap, a remoção mínima em um heap de Fibonacci ainda é uma operação eficiente em muitos casos.

3.4 Diminuição de chave

A operação de diminuição de chave em um heap de Fibonacci envolve a redução do valor de uma chave específica em um nó específico do heap. Essa operação é útil quando é necessário atualizar a prioridade de um elemento já existente no heap.

- Localize o nó que contém a chave a ser diminuída no heap de Fibonacci.

- Atualize o valor da chave para o novo valor desejado, diminuindo-a.
- Verifique se a nova chave viola a propriedade do heap de Fibonacci, que afirma que o valor de uma chave de um nó é sempre menor do que os valores das chaves de seus filhos.
- Se a propriedade do heap de Fibonacci for violada, corte o nó do seu pai e faça dele uma árvore independente. Marque o nó para indicar que perdeu um filho.
- Anexe a árvore resultante da etapa anterior à coleção de árvores de Fibonacci no heap.
- Verifique se a chave atualizada é menorS do que o valor mínimo atualmente armazenado no heap. Se sim, atualize o ponteiro para o mínimo elemento. Ao diminuir a chave de um nó, é possível que ocorra a necessidade de realizar operações de corte para manter as propriedades do heap. O corte envolve destacar um nó do seu pai e torná-lo uma árvore independente. Isso é necessário para manter o heap de Fibonacci balanceado.

A complexidade de tempo amortizado da diminuição de chave em um heap de Fibonacci é O(1), o que a torna uma operação extremamente eficiente. No entanto, é importante mencionar que, se uma sequência de diminuições de chave for realizada em um mesmo nó, o tempo de execução total dessas operações pode aumentar.

3.5 Código

```
// C++ program to demonstrate Extract min, Deletion()
// and Decrease key() operations in a fibonacci heap
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

// Creating a structure to represent a node in the heap
struct node {
        node* parent; // Parent pointer
        node* child; // Child pointer
        node* left; // Pointer to the node on the left
        node* right; // Pointer to the node on the right
        int key; // Value of the node
        int degree; // Degree of the node
        char mark; // Black or white mark of the node
        char c; // Flag for assisting in the Find node
           function
```

```
};

// Creating min pointer as "mini"
struct node* mini = NULL;

// Declare an integer for number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in heap
void insertion(int val)
{
        struct node* new_node = new node();
        new_node->key = val;
        new_node->degree = 0;
        new_node->mark = 'W';
        new_node->c = 'N';
        new_node->parent = NULL;
        new_node->child = NULL;
        new_node->left = new_node;
        new_node->right = new_node;
        if (mini != NULL) {
                (mini->left)->right = new_node;
                new_node->right = mini;
                new_node->left = mini->left;
                mini->left = new_node;
                if (new_node->key < mini->key)
                        mini = new_node;
        }
        else {
                mini = new_node;
        }
        no_of_nodes++;
}
// Linking the heap nodes in parent child relationship
void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
{
        (ptr2->left)->right = ptr2->right;
        (ptr2->right)->left = ptr2->left;
        if (ptr1->right == ptr1)
```

```
                mini = ptr1;
        ptr2->left = ptr2;
        ptr2->right = ptr2;
        ptr2->parent = ptr1;
        if (ptr1->child == NULL)
                ptr1->child = ptr2;
        ptr2->right = ptr1->child;
        ptr2->left = (ptr1->child)->left;
        ((ptr1->child)->left)->right = ptr2;
        (ptr1->child)->left = ptr2;
        if (ptr2->key < (ptr1->child)->key)
                ptr1->child = ptr2;
        ptr1->degree++;
}
// Consolidating the heap
void Consolidate()
{
        int temp1;
        float temp2 = (log(no_of_nodes)) / (log(2));
        int temp3 = temp2;
        struct node* arr[temp3+1];
        for (int i = 0; i <= temp3; i++)
                arr[i] = NULL;
        node* ptr1 = mini;
        node* ptr2;
        node* ptr3;
        node* ptr4 = ptr1;
        do {
                ptr4 = ptr4->right;
                temp1 = ptr1->degree;
                while (arr[temp1] != NULL) {
                        ptr2 = arr[temp1];
                        if (ptr1->key > ptr2->key) {
                                ptr3 = ptr1;
                                ptr1 = ptr2;
                                ptr2 = ptr3;
                        }
                        if (ptr2 == mini)
                                mini = ptr1;
```

```
                        Fibonnaci_link(ptr2, ptr1);
                        if (ptr1->right == ptr1)
                                mini = ptr1;
                        arr[temp1] = NULL;
                        temp1++;
                }
                arr[temp1] = ptr1;
                ptr1 = ptr1->right;
        } while (ptr1 != mini);
        mini = NULL;
        for (int j = 0; j <= temp3; j++) {
                if (arr[j] != NULL) {
                        arr[j]->left = arr[j];
                        arr[j]->right = arr[j];
                        if (mini != NULL) {
                                (mini->left)->right = arr[j];
                                arr[j]->right = mini;
                                arr[j]->left = mini->left;
                                mini->left = arr[j];
                                if (arr[j]->key < mini->key)
                                        mini = arr[j];
                        }
                        else {
                                mini = arr[j];
                        }
                        if (mini == NULL)
                                mini = arr[j];
                        else if (arr[j]->key < mini->key)
                                mini = arr[j];
                }
        }
}

// Function to extract minimum node in the heap
void Extract_min()
{
        if (mini == NULL)
                cout << "The heap is empty" << endl;
        else {
```

```
                node* temp = mini;
                node* pntr;
                pntr = temp;
                node* x = NULL;
                if (temp->child != NULL) {

                        x = temp->child;
                        do {
                                pntr = x->right;
                                (mini->left)->right = x;
                                x->right = mini;
                                x->left = mini->left;
                                mini->left = x;
                                if (x->key < mini->key)
                                        mini = x;
                                x->parent = NULL;
                                x = pntr;
                        } while (pntr != temp->child);
                }
                (temp->left)->right = temp->right;
                (temp->right)->left = temp->left;
                mini = temp->right;
                if (temp == temp->right && temp->child ==
                   NULL)
                        mini = NULL;
                else {
                        mini = temp->right;
                        Consolidate();
                }
                no_of_nodes--;
        }
}

// Cutting a node in the heap to be placed in the root list
void Cut(struct node* found, struct node* temp)
{
        if (found == found->right)
                temp->child = NULL;

```

```
        (found->left)->right = found->right;
        (found->right)->left = found->left;
        if (found == temp->child)
                temp->child = found->right;

        temp->degree = temp->degree - 1;
        found->right = found;
        found->left = found;
        (mini->left)->right = found;
        found->right = mini;
        found->left = mini->left;
        mini->left = found;
        found->parent = NULL;
        found->mark = 'B';
}

// Recursive cascade cutting function
void Cascase_cut(struct node* temp)
{
        node* ptr5 = temp->parent;
        if (ptr5 != NULL) {
                if (temp->mark == 'W') {
                        temp->mark = 'B';
                }
                else {
                        Cut(temp, ptr5);
                        Cascase_cut(ptr5);
                }
        }
}

// Function to decrease the value of a node in the heap
void Decrease_key(struct node* found, int val)
{
        if (mini == NULL)
                cout << "The Heap is Empty" << endl;

        if (found == NULL)
                cout << "Node not found in the Heap" << endl;
```

```

        found->key = val;

        struct node* temp = found->parent;
        if (temp != NULL && found->key < temp->key) {
                Cut(found, temp);
                Cascase_cut(temp);
        }
        if (found->key < mini->key)
                mini = found;
}

// Function to find the given node
void Find(struct node* mini, int old_val, int val)
{
        struct node* found = NULL;
        node* temp5 = mini;
        temp5->c = 'Y';
        node* found_ptr = NULL;
        if (temp5->key == old_val) {
                found_ptr = temp5;
                temp5->c = 'N';
                found = found_ptr;
                Decrease_key(found, val);
        }
        if (found_ptr == NULL) {
                if (temp5->child != NULL)
                        Find(temp5->child, old_val, val);
                if ((temp5->right)->c != 'Y')
                        Find(temp5->right, old_val, val);
        }
        temp5->c = 'N';
        found = found_ptr;
}

// Deleting a node from the heap
void Deletion(int val)
{
        if (mini == NULL)
```

```
                cout << "The heap is empty" << endl;
        else {

                // Decreasing the value of the node to 0
                Find(mini, val, 0);

                // Calling Extract_min function to
                // delete minimum value node, which is 0
                Extract_min();
                cout << "Key Deleted" << endl;
        }
}

// Function to display the heap
void display()
{
        node* ptr = mini;
        if (ptr == NULL)
                cout << "The Heap is Empty" << endl;

        else {
                cout << "The root nodes of Heap are: " <<
                   endl;
                do {
                        cout << ptr->key;
                        ptr = ptr->right;
                        if (ptr != mini) {
                                cout << "-->";
                        }
                } while (ptr != mini && ptr->right != NULL);
                cout << endl
                        << "The heap has " << no_of_nodes <<
                           " nodes" << endl
                        << endl;
        }
}
```

4 Resumo

Procedimento	Heap binário (pior caso)	Heap binomial (pior caso)	Heap de Fibonacci (amortizado)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Figura 4.1 – Tabela de complexidade das operações sobre heaps binários, binomiais e de Fibonacci.
Disponível em: <https://www.ic.unicamp.br/˜meidanis/courses/mo417/2003s1/aulas/2003-04-23.html: :text=3.-,Defini%C3%A7%C3%A3o%20de%20heap%20binomial,a%20chave%20de%20seu%20pai>

Referências

1. AGUIAR, Marilton Sanchotene de. Análise formal da complexidade de algoritmos genéticos. 1998.

2. BEDER, Delano M. Complexidade de Algoritmos.

3. KURITA, Takio. An efficient agglomerative clustering algorithm using a heap. Pattern Recognition, v. 24, n. 3, p. 205-209, 1991.

4. HINZE, Ralf. Explaining binomial heaps. Journal of functional programming, v. 9, n. 1, p. 93-104, 1999.

5. SADAGOPAN, N.; BHARTI, Nitin Vivek; HEAP, Binomial. 1 Binomial Tree.

6. BRODAL, Gerth Stølting; LAGOGIANNIS, George; TARJAN, Robert E. Strict fibonacci heaps. In: Proceedings of the forty-fourth annual ACM symposium on Theory of computing. 2012. p. 1177-1184.

7. BALDASSIN, Alexandro José. MO417 - Ata de Aula - Heaps binomiais e de Fibonacci. Disponível em: <https://www.ic.unicamp.br/ meidanis/courses/mo417/2003s1/aulas/2003-04-23.html: :text=3.-,Defini%C3%A7%C3%A3o%20de%20heap%20binomial,a%20chave%20de%20seu%20pai>. Acesso em: 01 jun. 2023.

8. Binary Heap. Disponível em: <https://www.geeksforgeeks.org/binary-heap/> Acesso em: 01 jun. 2023.

9. Implementation of Binomial Heap. Disponível em: <https://www.geeksforgeeks.org/implementation-binomial-heap/>. Acesso em: 01 jun. 2023.

10. Fibonacci Heap – Deletion, Extract min and Decrease key. Disponível em: <https://www.geeksforgeeks.org/fibonacci-heap-deletion-extract-min-and-decrease-key/> Acesso em: 01 jun. 2023.