1 Fibonacciho halda

Fibonacciho halda je implementace priority queue, která je navržená za účelem snížení časové složitosti pro jisté operace. Je to typ haldy, což je struktura, která splňuje **vlastnost haldy**: každý prvek má větší nebo rovný klíč než jeho potomek. Není tu však omezení na počet potomků, jako například u binární haldy, ve Fibonacciho haldě může mít jakýkoliv prvek jakýkoliv počet potomků.

1.1 Struktura haldy

Halda se skládá z obousměrně zřetězeného cyklického spojového seznamu (dále pouze **seznam**) ukazatelů na prvky. Každý prvek obsahuje seznam svých potomků. V haldě je ještě uložen ukazatel na prvek s nejmenším klíčem, který z definice haldy vždy leží v kořenovém seznamu.

Každá instance Prvku obsahuje data a klíč, dále seznam svých potomků a stupeň prvku. Stupeň prvku je počet přímých potomků, které Prvek má.

1.2 Operace

Fibonacciho halda má 4 hlavní operace: vložení prvku, nalezení minima, odstranění minima a snížení klíče prvku:

1.2.1 Vložení prvku

Probíhá vytvořením nového prvku, který je vložen na konec kořenového seznamu. Složitost operace je O(1).

1.2.2 Nalezení minima

Protože je vždy skladován ukazatel na prvek s nejmenším klíčem, je hodnota v něm přístupná se složitostí O(1).

1.2.3 Odstranění minima

Prvním krokem je uložení hodnoty nejmenšího prvku do proměnné, protože tento prvek bude v průběhu smazán a na konci metody k němu nebude přístup. Následně se spustí cyklus přes všechny prvky kořenového seznamu. V každé iteraci se nejdříve ověří, jestli je na daný prvek uložený ukazatel jakožto na minimální prvek. Pokud ne, prvek se [vloží do pole] a pokračuje se v seznamu. Pokud je na prvek uložen ukazatel, provede se stejná operace s [vkládáním do pole] s potomky tohoto minimálního prvku a ten se následně odstraní, po čemž cyklus pokračuje s dalším prvkem kořenového seznamu.

Vkládání do pole: Na začátku operace se vytvoří pole dostatečně veliké (stupeň jakéhokoliv prvku nikdy nepřekročí $\log_\varphi n$, kde $\varphi = 1.6180...$ a n je počet prvků v haldě) a do něho se ukládají prvky podle hodnoty jejich stupně, tj. prvek se stupněm 3 bude vložen na pozici s indexem 3 (za podmínky, že se pole indexuje od 0). Mohou nastat dvě situace:

1. Na této pozici v poli není žádný prvek. Nově příchozí prvek se proto jednoduše vloží na toto místo v poli.
2. Na této pozici již nějaký prvek je. V tom případě proběhne porovnání klíčů a prvek s větším prvkem se stane potomkem prvku s menším klíčem. Prvku s menším klíčem se takto ale zvýší stupeň a musí být přesunut na místo s o jedna větším indexem.

Po projetí celého seznamu zbývá znovu složit kořenový seznam haldy z prvků v poli, pod kterými nyní "vyrostly stromy". Složitost operace je $O(\log n)$.

1.2.4 Snížení klíče prvku

Mohou nastat dvě situace:

1. Po snížení klíče se neporuší vlastnost haldy, tento prvek bude mít tedy nejméně takový klíč jako jeho rodič.

2. Po snížení klíče bude porušena vlastnost haldy. V tom případě je prvek odříznut a je vložen na konec kořenového seznamu. V případě, že jeho rodič není kořenovým prvkem, je rodič označkován. Pokud již rodič označkován byl, je také odříznut a je označkován jeho rodič. Takto se rekurzivně postupuje dokud není nalezen prvek, který ještě nebyl označkován, nebo který je kořenovým prvkem.

Složitost operace je $O(\log n)$.

2 Moje implementace

2.1 Struktura haldy

Haldu jsem implementoval jako `template`, je tedy možné pomocí haldy řadit jakákoliv data, pokud jim je přiřazen validní celočíselný klíč. Pro seznam používám `std::list` ze standartní knihovny C++, v metodě "odstranění minima" používám jako pole `std::vector`. Místo raw pointerů používám `std::unique_ptr`.

Halda je implementovaná jako třída `template <typename T> class Fib_Heap;` a stejně tak třída implementující prvky `template <typename T> class Node;`. Třída `Node` má veškeré atributy i metody nastavené jako `public`, což není nejvhodnější nastavení, ale protože třída `Fib_Heap` v žádné veřejné metodě nebere jako argument `Node`, nemůže nastat poškození struktury zvenku.

V kódu používám zkratku `N_ptr<T>` definovanou pomocí `template <typename T> using N_ptr = std::unique_ptr<Node<T>>;`.

Zvláštní věcí je atribut `T null_data;` ve `Fib_Heap`. Slouží k tomu, aby program nespadl ani když se uživatel snaží například zjistit minimum prázdné haldy; v každé metodě, která takto k prvkům přistupuje, je podmínka `if (empty()) {...}`. Asi by bylo lepší, aby program spadl, ale nepodařilo se mi to kódově zprovoznit, takže jsem to vyřešil takto. Zřejmá chyba tohoto postupu je ta, že v haldě mohou být uložena stejná data jako tato `null_data`, takže by se muselo hlídat, aby tato možnost nenastala a to ubírá na použitenosti.

2.2 Třída Node

Třída má jednoduchý konstruktor, nejzajímavější metoda je `void add_child(args)`, která vloží prvek z argumentu do seznamu potomků.

2.3 Třída Fib_Heap

Metody `Fib_Heap(args)`, `bool empty()` a `T minimum()` jsou jednoduché, `insert(args)` dynamicky vytvoří novou instanci `Node<T>` pomocí `std::make_unique<Node<T>>(constructor args)`, vloží ji na konec kořenového seznamu `root_list`; pokud má nový prvek menší klíč než současné minimum, přepíše se ukazatel na nový prvek.

Metoda `T extract_min()` pracuje v následujících krocích:

- Příprava: do proměnné se uloží současné minimum, deklaruje se vektor, ve kterém se budou skládat stromy a zaplní se `nullptr` až do maximálního indexu pro daný počet prvků v haldě.
- Iterace přes `root_list`: První prvek v seznamu se přesune do pomocné proměnné a ze seznamu se odstraní. Pokud na prvek není uložen minimální ukazatek, zavolá se metoda [merge_node_return_new_min]. Pokud na prvek je uložen minimální ukazatel, proběhne separátní smyčka, ve které se volá stejná metoda na potomky minimálního prvku. Ten se odstraní a `extract_min()` pokračuje dalším prvkem v `root_list`.
- `root_list` je v tuto chvíli prázdný, stačí proto přepsat ne-`nullptr` složky vektoru do tohoto seznamu. Sníží se počet prvků v haldě, uloží se případný nový ukazatel na minimální prvek a vrátí se hodnota starého minimálního prvku.
- Metoda `merge_node_return_new_min(args)`: ověří, zda je nově přidávaný prvek kandidát na minimální prvek a zachová se podle toho. Zavolá metodu [merge_in_vector()].

- Metoda `merge_in_vector(args)`: Nově příchozí prvek má celočíselný stupeň n a snaží se uložit se do vektoru na index n. Pokud na tomto indexu žádný prvek není, uloží se tam a metoda skončí. Pokud tam prvek je, porovnají se klíče obou prvků, ten s větším klíčem se stane potomkem toho s menším klíčem. Vznikne tak nový prvek se stupněm o jedna větším. Ten se rekurzivně vloží do stejné metody, což se opakuje, dokud se ve vektoru nenajde prázdné místo.

Metoda `decrease_key()`: Tuto metodu jsem nenaimplementoval. Hlavním důvodem je, že jsem nevěděl, jak programu předat informaci, který prvek jak snížit. Z informací, které jsem našel, jsem usoudil, že metoda slouží k tomu, aby prvky, které dlouho nebyly použity, byly upřednostněny. Nenapadlo mě ale, jak bych měl čas od vložení řídit. Možné řešení by bylo prohledat celou haldu a najít prvky ke snížení přímo, ale to by velice zvýšilo složitost operace. Také by se dalo nechat uživatele ukládat si ukazatele na prvky a podle jeho vlastního rozhodovacího algoritmu ho nechat prvky měnit klíče. To ale vyžaduje pozornost a není bezpečné pro celistvost struktury.