(ADT) Priority Queue Damian Kamiński 21.03.2021

W ramach należało zaimplementować abstrakcyjny typ Priority Queue za pomocą kopca binarnego typu max.

InsertElement

Dodawanie elementu do kolejki odbywa się poprzez dodanie elementu do kopca. Każdy push() tworzy nową tablicę o pojemności aktualnej + 1. Elementy z aktualnej tablicy są kopiowane do nowej, aktualna tablica jest usuwana, a miano aktualnej przenosi się na nową tablice. Dodawany elementy zostanie dodany na koniec tablicy. Całość wyniesie n operacji, gdzie n to ilosc wszystkich elementow łącznie z nowym.

Teraz należy sprawdzić czy struktura kopca jest zachowana. Korzystam z informacji, że przed dodaniem nowego elementu struktura kopca była już zachowana. Wystarczy więc jeśli przejdę od liścia do korzenia sprawdzając czy dziecko jest mniejsze od rodzica, jeśli tak to nie idę dalej, jeśli jest to zamieniam i sprawdzam poziom wyżej. Wykona się to w czasie O(logn) .

Łącznie cała funkcja wykona się w czasie O(n). Głównym problemem w tej implementacji jest to, że przy każdym dodaniu elementu tworzę nową tablicę i kopiuje elementy. Mógłbym osiągnąć złożoność tej funkcji O(logn) poprzez np. ustawienie stałego rozmiaru tablicy, zwiększanie rozmiaru tablicy nie o 1, a pewną ilość w wypadku gdy brakuje miejsca (tak jak vector). Uniknąłbym w ten sposób niepotrzebnego tworzenia i kopiowania elementów, przy każdorazowym użyciu funkcji InsertElement.

GetMaxPriority

Najwyższy priorytet znajduje się w korzeniu kopca stąd złożoność wyniesie O(1), wystarczy pobrać wartosc z korzenia.

Pop

Przy usuwaniu elementu o najwyższym priorytecie usuwany jest korzeń, a więc należy teraz zadbać o prawidłową strukturę kopca przy przenoszeniu elementów.

Na samym początku zamieniam korzeń z ostatnim elementem kopca. Poprzednio ostatni element jest teraz pustym elementem.

Sprawdzam teraz pętlą czy struktura się zgadza, idąc od korzenia. Porównuję z dziećmi, jeśli któreś jest większe to zamieniam miejscami. W korzeniu powinien znaleźć się element o największym priorytecie. Przechodzę niżej, do miejsca z którego pochodzi nowy korzeń i ponownie sprawdzam stosunek dzieci do rodzica. Robię to dopóki dojdę do końca lub dzieci będę mniejsze od rodzica.

Złożoność wyniesie O(logn), ponieważ każda zamiana to stała ilość operacji. Z każdą iteracją liczba elementów do sprawdzenia zmniejsza się dwukrotnie.

IncreasePriority

Na samym początku przechodzę cały kopiec w poszukiwaniu elementów, którym należy podnieść priorytet. Wykona się to w czasie O(n).
Teraz jeśli podniosiono przynajmniej jednemu elementowi priorytet należy sprawdzić czy struktura kopca jest zachowana. Sprawdzenie wykona się w czasie O(n/2). Jeśli nie to dokonuje heapify w celu naprawy kopca. Naprawa kopca wykona się w czasie O(nlogn), ponieważ dla każdego elementu będę dokonował heapify, który wykonuje się w czasie O(logn).

Stąd cała funkcja wykona się w czasie O(nlogn) jeśli zajdzie potrzeba poprawy struktury i O(n) w przeciwnym wypadku.

Budowa kolejki na podstawie listy par

Mając listę będę przechodził od jej końca i używał heapify której złożonośc wynosi O(logn) do budowy kopca.

Heapify ma złożoność O(logn) ponieważ rekurencyjnie przechodzi po poziomach i przy każdym przejściu zmniejsza liczbę porównywanych elementów o połowę.

Z racji, że używam heapify tyle ile jest elementów na liście to całość wykona się w czasie O(nlogn).