Struktury danych i złożoność obliczeniowa

Patryk Wlazłyń

# Zaimplementowane struktury danych

W ramach projektu zostały zaimplementowane następujące struktury danych:

1.Tablica

2. Lista dwu kierunkowa

3. Kopiec binarny

4. Drzewo czerwono-czarne

Cały kod źródłowy został napisany w języku C++, standardzie 17. Do wypisywania na konsole i do pliku dodatkowo użyto biblioteki fmt. Podczas implementacji zostały przyjęte założenia:

* Podstawowym elementem struktur jest 4 bajtowa liczba całkowita typu int32\_t.
* Pomiarom poddane zostały procedury: wstawiania elementu, usuwania, oraz wyszukiwania.
* Wszystkie struktury są alokowane dynamicznie.
* Do zarządzania zasobami struktury wykorzystano model obiektowy.
* Tablica przy każdym dodaniu elementu i usunięciu dokonuje całkowitej realokacji.
* Dodawanie i usuwanie elementów z tablicy i listy możliwe jest na ich początku, końcu, oraz miejscu wskazanym przez indeks i takie też operacje zostały poddane testom szybkości działania. Pozostałe struktury ze względu na wewnętrzne zarządzanie położeniem elementów implementują jedynie możliwość prostego wstawiania i usuwania.
* Pomiary prędkości wykonania danych operacji wykonywane są dla różnych rozmiarów początkowych struktur danych.
* Wypełnianie struktur danymi do zadanej wielkości przed testem prędkości odbywało się poprzez wstawianie losowo wygenerowanych elementów.
* Dodatkowo został zaimplementowany interfejs konsolowy pozwalający na weryfikację działania poszczególnych operacji na strukturach danych.
* Operację możliwe do wykonania na strukturach z pozycji interfejsu to: budowanie z pliku, usuwanie elementu, dodawanie elementu, wyszukiwanie elementu, losowe generowanie elementów, wyświetlanie.
* Do pomiaru czasu wykonania operacji użyto biblioteki std::chrono.
* Skompilowany kod, poddany testom kompilowany był kompilatorem optymalizującym GNU GCC w wersji 8 z flagą kompilacji -O3, która powoduje faworyzacje kodu który jest najszybszy nawet pomimo zajmowania większej ilości miejsca.
* Pomiary prędkości wykonania zawsze poprzedzone były zestawem „trenującym”, aby zminimalizować różnice w czasie wykonania spowodowanymi przez „branch predicting” oraz „out of order execution”.
* Wszelkie implementacje wspomniane w całym sprawozdaniu wykonane są w ramach jednego programu, poza kodem poszczególnych testów które wywoływane były pojedynczo i nie znajdują się w kodzie źródłowym. Kod wspomagający testy jednakże jest w pełni dostępny i załączony z kodem źródłowym.
* Cały kod źródłowy pisany jest w języku angielskim, tj. komentarze i nazwy symboli.

# Złożoności obliczeniowe operacji na strukturach danych

Tablica

Struktura danych przechowująca elementy w ciągłej pamięci komputera. Posiada wskaźnik z pierwszy z elementów znajdujący się w strukturze, oraz wielkość sekwencji elementów. Podczas wstawiania i usuwania tablica była całkowicie przealokowywana, tak aby wykorzystywać jak najmniej pamięci. Wstawianie do tablicy polega na alokacji miejsca na nowa tablice, większą o 1, przekopiowanie elementów do miejsca wstawiania nowego, wstawienie nowego elementu do nowej tablicy oraz przekopiowania jej drugiej części za wstawiany element. Operacja ta ma złożoność O(n), ponieważ musi przekopiować każdy element jeden po drugim do nowej pamięci, dodatkowo wstawienie elementu O(1), oraz dealokacja poprzedniej tablicy, czyli O(1), co ostatecznie łącznie daje O(n). Usuwanie obywa się analogicznie, otrzymujemy O(n). Miejsce wykonania wcześniej wymienionych operacji nie ma znaczenia w tej konkretnej implementacji, ze względu na realokację przy każdej operacji. Szukanie w tablicy polega na przejściu po kolei przez wszystkie elementy i porównywanie ich klucza z szukanym kluczem, co odpowiada złożoności O(n).

Lista

Struktura danych przechowująca elementy w postaci węzłów z podwójnym dowiązaniem. Każdy węzeł posiada 3 pola, wskaźnik na węzeł poprzedni i następny oraz wartość klucza. Podczas wstawiania i usuwania z listy wystarczy wykonać prostą operację przepisania wskaźników sąsiadów usuwanego węzła, w taki sposób aby poprzednik wskazywał teraz na następnik i odwrotnie w usuwaniu, oraz ustawienie poprzednika i następnika aby wskazywały na nowy węzeł przy dodawaniu, po czym następuje zwolnienie pamięci w przypadku usuwania węzła. Gdy mamy już wskaźnik na węzeł, który chcemy usunąć lub dodać sama operacja ma złożoność O(1). Wyszukiwanie elementu w liście polega na liniowym przejściu po wszystkich elementach listy po kolei i porównywaniu klucza, co daje złożoność O(n). Należy jednak zaznaczyć, że mimo podobnej złożoności przeszukiwania co tablica, przechodzenie przez elementu jest dużo wolniejsze ze względu na liczne wyłuskania wskaźników które mogą wskazywać w oddalone od siebie miejsca w pamięci. Tak jak w poprzednim przypadku miejsce wstawiania i usuwania nie ma znaczenia.

Sterta

Sterta to struktura drzewiasta wykorzystująca do implementacji zwykłą tablice. Każdy element tej tablicy jest dodatkowo węzłem, który posiada dwoje potomków które od teraz będziemy oznaczać jako LEFT(x) oraz RIGHT(x) co odpowiednio oznacza lewe dziecko x oraz prawe dziecko x. Własność którą dodatkowo posiada kopiec to: x > LEFT(x) oraz x > RIGHT(x). Co implikuje, że wartość największa znajduje się na szczycie sterty. Podczas wstawiania elementu, lub jego usuwania może zostać zatracona ta właściwość co oznacza, że po każdej takiej operacji musi zostać ona przywrócona. Przywracanie polega na rekursywnym wywołaniu procedury HEAPIFY do momentu naprawienia właściwości sterty. Procedura ta powoduje „wędrowanie” źle ustawionego węzła w górę sterty, do momentu aż nie znajdzie się on na odpowiednim miejscu. Takie wędrowanie może następować jedynie poprzez zamianę jego miejsca z jego rodzicem, co oznacza że sama operacja jest O(log(n)). Wstawianie polega na dodaniu nowego elementu na końcu i przywróceniu w jego obrębie własności sterty. Usuwanie jest podobne z jedyna różnicą polegająca na tym, że najpierw usuwany element jest zmieniony z końcowym, a wielkość sterty zostanie zmniejszona o jeden przez co w pewnym sensie ostatni element (aktualnie usuwany) zostanie w pewnym sensie zapomniany. Następnie zostanie wywołana procedura przywrócenia własności sterty.

Drzewo poszukiwań binarnych (wstęp do drzewa czerwono czarnego)

Struktura drzewiasta która implementowana jest na węzłach z podwójnymi dowiązaniami. Każdy z węzłów ma wskaźnik na swojego rodzica, lewe dziecko, oraz prawe dziecko. Jeżeli któreś z tych elementów nie istnieją (są puste) to przyjmują one wartość NIL, w tej implementacji nullptr. Drzewo poszukiwań binarnych posiada właściwość która czyni je bardzo łatwym w przeszukiwaniu elementów. Każdy z jego węzłów spełnia własność: LEFT(x) < x <= RIGHT(x). Co powoduje że poszukując danego elementu zawsze wiemy w które poddrzewo powinniśmy się skierować by mieć szanse go znaleźć. Co oznacza ze będziemy przechodzić po kolejnych potomkach od korzenia (węzła nieposiadającego rodzica) w dół drzewa ale tylko po drodze jednego z jego potomków co oznacza ze najdłuższa droga takie poszukiwania to wysokość drzewa co w przeciętnym wypadku daje log(n), chociaż może się zdarzyć ze drzewo rozwinięte jest do postaci listy i wtedy takie przeszukanie potrwa O(n). Dodawanie elementu do drzewa polega na porównywaniu klucza wstawianego elementu węzłem i jeżeli klucz jest mniejszy schodzenie w jego lewe poddrzewo, w innym przypadku w prawe. Skutkuje to znalezieniem odpowiedniego miejsca dla wstawianego elementu, co podobnie do operacji wyszukiwania ma złożoność O(log(n)). Po znalezieniu odpowiedniego miejsca następuje przepisania wskaźników tak aby nowy węzeł wskazywał na nowego rodzica i rodzic wskazywał na swojego nowego syna, co zajmuje O(1). W efekcie otrzymujemy O(log(n)) dla całej operacji. Usuwanie elementu z drzewa również rozpoczyna się od jego znalezienia w drzewie co zajmie co najwyżej O(log(n)), a następnie odpowiednie przepisanie wskaźników rodzica i dzieci usuwanego węzła tak, aby własność drzewa została zachowana. W tym celu musimy rozpatrzeć trzy przypadki:

1. Usuwany węzeł nie ma dzieci
2. Usuwany węzeł ma jedno dziecko
3. Usuwany węzeł ma dwoje dzieci

W pierwszym przypadku, jedyne co trzeba zrobić to dla rodzica usuwanego węzła ustawić wskaźnik na dziecko na NIL (nullptr) i zwolnic pamięć zajmowana przez usuwany węzeł. Drugi przypadek jest tylko nieco bardziej skomplikowany. Wymaga on przypisania jako nowe dziecko dla rodzica usuwanego węzła, dziecko właśnie usuwanego węzła i skorygować jego wskaźnik na rodzica. Trzeci przypadek polega na znalezieniu następnika usuwanego węzła, jest to węzeł większy kluczem od aktualnie usuwanego ale jest on najmniejszy z większych od niego. Ta własność okaże się przydatna ze względu na to ze po wstawieniu takiego węzła w miejsce usuwanego jego prawe poddrzewo nadal będzie zawierało węzły większe od niego, gdyż jest on najmniejszy z tego poddrzewa, oraz jednocześnie lewe poddrzewo usuwanego węzła będzie na pewno mniejsze od niego gdyż jest ono mniejsze od usuwanego węzła a jego następnik jest większy od niego. W praktyce nie zamienia się samych węzłów a jedynie przepisuje ich klucze, co znacznie upraszcza całą operacje. Samo przepisanie wartości, lub podmiana wartości wskaźników zajmuje O(1). Co ostatecznie daje O(n) jeżeli drzewo listą, oraz O(log(n)) w większości przypadków. Jak widać problemem drzew BST jest tutaj fakt, że efektywne wykonywanie operacji możliwe jest tylko wtedy gdy drzewo jest zrównoważone. Takim drzewem binarnych jest drzewo czerwono-czarne, przedstawione poniżej.

Drzewo czerwono-czarne

Struktura bardzo podobna do drzewa poszukiwań binarnych (BST). Posiada ona jednak dodatkowe zasady, o które trzeba zadbać przy operacjach dodawania i usuwania z drzewa. Te zasady to:

1. Każdy węzeł jest czerwony albo czarny
2. Korzeń jest czarny
3. Każdy liść jest czarny (Można traktować *nil* jako liść)
4. Jeśli węzeł jest czerwony, to jego synowie muszą być czarni
5. Każda ścieżka z ustalonego węzła do liścia liczy tyle samo czarnych węzłów
6. Nowo wstawiany węzeł ma początkowo kolor czerwony

Na wstępie należy zauważyć, że tak dobrane zasady implikują pewną bardzo ważna cechę takich drzew. Mianowicie w najgorszym przypadku stosunek drogi najdłuższej do najkrótszej od korzenia do liścia ma się jak 1:2. W najgorszym przypadku każdy z węzłów jednego z poddrzew od korzenia do liścia będzie kolory czarnego a drugiego z poddrzew będzie na zmianę czerwony i czarny, dwa węzły czerwone nie mogą nie zdarzyć ze względu na zasadę 4. Nawet w takim przypadku drzewo nadal jest dostatecznie zbalansowane, żeby operacje na nim zajmowały O(log(n)). Jednak przez istnienie takich zasad podczas dodawania lub usuwania może się zdarzyć, że zasada 4. zostanie zaburzona i drzewo będzie wymagało korekty. Podsumowując wyszukanie elementu co najwyżej zajmie O(log(n)), a ewentualna korekta może wymagać rekurencyjnego wywoływania korekty aż do wierzchołka co również daje O(log(n)), co skutkuje złożonością dodawania i usuwania równą O(log(n)).

# Złożoność operacji podsumowanie

2.2. Złożoności operacji wykonywanych na tablicy

|  |  |  |
| --- | --- | --- |
| **Funkcja** | **Średnia** | **Pesymistyczna** |
| Dodanie elementu | O(n) | O(n) |
| Usunięcie elementu | O(n) | O(n) |
| Wyszukanie elementu | O(n) | O(n) |

2.3. Złożoność operacji wykonywanych na liście

|  |  |  |
| --- | --- | --- |
| **Funkcja** | **Średnia** | **Pesymistyczna** |
| Dodanie elementu | O(-) | O(n) |
| Usunięcie elementu | O(-) | O(n) |
| Wyszukanie elementu | O(n) | O(n) |

2.4. Złożoność operacji wykonywanych na kopcu

|  |  |  |
| --- | --- | --- |
| **Funkcja** | **Średnia** | **Pesymistyczna** |
| Dodanie elementu | O(log(n)) | O(log(n)) |
| Usunięcie elementu | O(log(n)) | O(log(n)) |
| Wyszukanie elementu | O(n) | O(n) |

2.5. Złożoność operacji wykonywanych na drzewie czerwono-czarnym

|  |  |  |
| --- | --- | --- |
| **Funkcja** | **Średnia** | **Pesymistyczna** |
| Dodanie elementu | O(log(n)) | O(log(n)) |
| Usunięcie elementu | O(log(n)) | O(log(n)) |
| Wyszukanie elementu | O(log(n)) | O(log(n)) |

# Pomiary rzeczywiste – przedstawienie graficzne

# Tablica

Poniższe trzy wykresy przedstawiają wyniki dla operacji wstawiania na odpowiednio początek, losowe miejsce i na koniec tablicy. Jak widać wykresu wraz ze wzrostem argumentów przypominają zależność liniowa, czyli teoretyczną.

Kolejne dane dotyczą usuwania z tablicy i znów widać zależność liniową.

Tak samo z przeszukiwaniem.

# Kopiec

Następną struktura danych jest kopiec z wykresem złożoności dodawania i usuwania elementu przypominający log(n). Oraz przeszukiwanie w czasie liniowym.

# Lista

Lista charakteryzuje się stałym czasem wstawiania i usuwania elementów jeżeli ich pozycja jest znana. Na wykresach warto zwrócić uwagę na skale na osi Y (czasu) wartości ledwie od siebie odbiegają. Przypomina to złożoność liniową.

Usuwanie z listy z początku jest szybkie, bo od razu znamy pozycje usuwanego węzła, na wykresie rysuje się złożoność liniowa.

Wyszukiwanie w liście – złożoność liniowa.

# Drzewo czerwono-czarne

Atutem drzewa czerwono-czarnego jest złożoność jego operacji równa nlog(n), co dobrze widać na wykresach.

# Wnioski

Otrzymane wyniki wiernie odtwarzają oczekiwane rezultaty złożoności operacji na zmierzonych strukturach danych. Warto jednak zauważyć, że nie są one idealne. Mimo zastosowania „zestawu uczącego” przed każdym testem, takie zjawiska jak *branch prediction*, out of order *execution*, oraz  
 *cache* *miss’es* nadal powodują, że teoretyczne złożoności nie odzwierciedlają w pełni rzeczywistych pomiarów. Mimo starania autora o jak najmniejsze zużywanie zasobów komputera przez inne procesy niż ten dokonujący pomiarów, to wpływ procesów zewnętrznych na pewno miał udział w ostatecznym wyniku pomiarów.

# Źródła informacji

Introduction to Algorithms Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest.

https://eduinf.waw.pl/inf/alg