PAŃSTWOWA WYŻSZA
SZKOŁA ZAWODOWA W NYSIE

Sprawozdanie

Apartamenty

Przedmiot **Zarządzanie Danymi Informacyjnymi - Laboratoria**

Prowadzący:	Autor:
dr inż. Damian Raczyński	Dávid Ali

2022/2023

Spis treści

1. Wybrane technologie

Język C# i platforma .NET Framework są głównymi technologiami użytymi do tworzenia aplikacji. Aplikacja została napisana w języku C#, co pozwoliło na wykorzystanie silnika języka oraz bogatej biblioteki klas .NET Framework do tworzenia interfejsu użytkownika, obsługi zdarzeń, pracy z danymi itp.

Do tworzenia interfejsu użytkownika została użyta biblioteka WPF (Windows Presentation Foundation), która dostarcza szereg narzędzi i kontrolek do tworzenia profesjonalnych aplikacji graficznych dla systemu Windows.

W aplikacji został również wykorzystany mechanizm wiazanie danych (ang. data binding), który umożliwia połączenie elementów interfejsu użytkownika z danymi, co ułatwia pracę z danymi i zapewnia ich automatyczną aktualizację w przypadku zmian.

Do zarządzania kolekcjami danych w aplikacji została użyta klasa `ObservableCollection`, która jest specjalną kolekcją implementującą interfejs `INotifyCollectionChanged`, co umożliwia automatyczne powiadomienie elementów interfejsu o zmianach w kolekcji.

W aplikacji zostały również wykorzystane narzędzia do obsługi plików, takie jak klasa `FileStream` i klasa `StreamReader`, które umożliwiają odczyt i zapis danych z plików.

Ogólnie rzecz biorąc, technologie i biblioteki użyte w aplikacji pozwalają na szybkie i efektywne tworzenie aplikacji graficznych dla systemu Windows oraz ułatwiają pracę z danymi i plikami.

2. Struktury indeksowe

Struktury indeksowe to specjalne struktury danych, które są stosowane do przechowywania informacji o pozycjach elementów w danym zbiorze danych. Celem użycia struktur indeksowych jest zwiększenie szybkości wyszukiwania elementów w zbiorze danych oraz zmniejszenie złożoności obliczeniowej operacji wyszukiwania.

Najczęściej stosowane struktury indeksowe to:

- Indeksy liniowe - polegają na tworzeniu tablicy indeksów, gdzie każdy indeks odpowiada pozycji elementu w oryginalnym zbiorze danych.
- Indeksy binarne - polegają na tworzeniu drzewa binarnego, gdzie każdy węzeł zawiera informację o pozycji elementu w oryginalnym zbiorze danych.
- Indeksy łańcuchowe - polegają na tworzeniu lancuchów indeksów, gdzie każdy indeks zawiera informację o pozycji elementu w oryginalnym zbiorze danych oraz odsyłacz do kolejnego indeksu.
- Indeksy haszujące - polegają na użyciu funkcji haszującej do obliczenia indeksu elementu w tablicy indeksów.

Użycie struktur indeksowych pozwala znacznie zwiększyć szybkość wyszukiwania elementów w zbiorze danych, szczególnie w przypadku dużych zbiorów danych. Jest to szczególnie istotne w przypadku aplikacji, które wymagają szybkiego dostępu do danych, takich jak bazy danych czy systemy informacyjne.

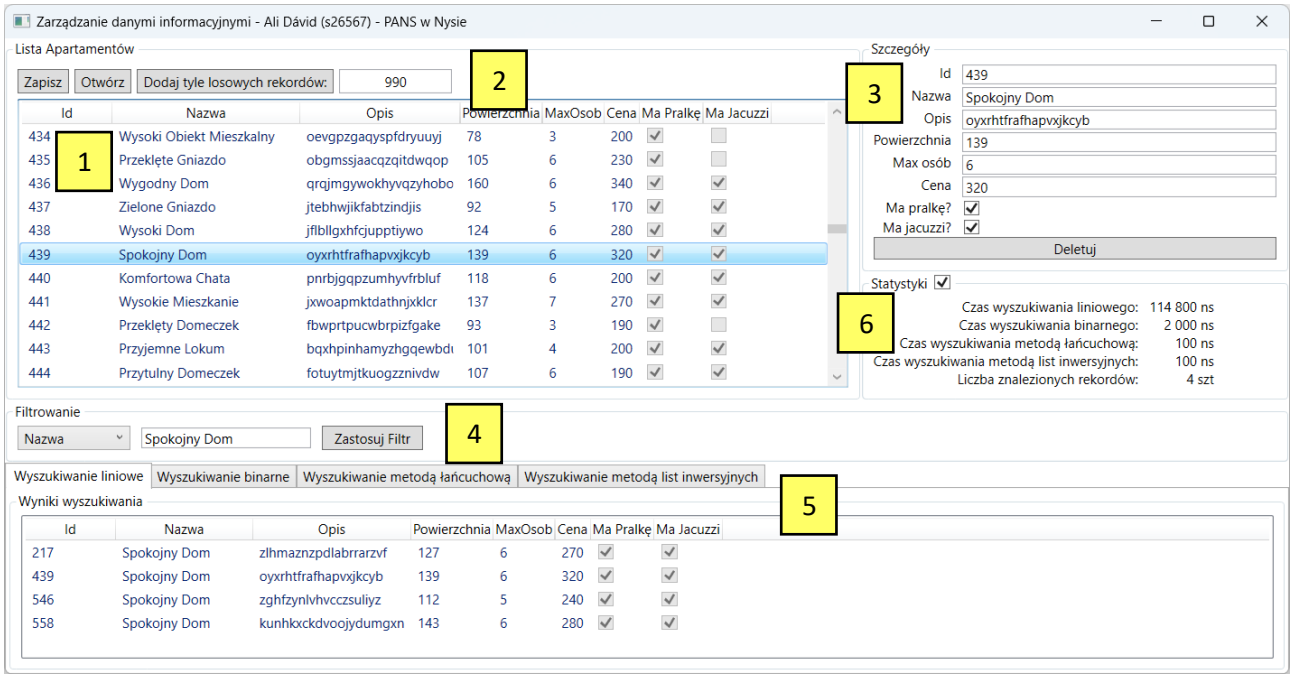
3. Opis projektu

Opis struktury bazy danych

Baza danych składa się z listy obiektów typu `Apartament`, które zawierają informacje o apartamentach, takie jak numer identyfikacyjny, nazwa, opis, powierzchnia, maksymalna liczba osób, cena oraz informacje o tym, czy posiadają pralkę i jacuzzi. Te obiekty są przechowywane w strukturze `ObservableCollection`, która pozwala na dynamiczne dodawanie i usuwanie elementów z kolekcji oraz poinformowanie o tym interfejsu graficznego.

Przedstawienie i opis interfejsu graficznego

Interfejsem graficznym programu jest okno aplikacji, które składa się z kilku różnych elementów.



[1] Głównym elementem jest lista mieszkań, która wyświetla wszystkie dostępne mieszkania w bazie danych.

[2] Nad listą mieszkań znajduje się pasek z trzema przyciskami o etykietach "Zapisz", "Otwórz" i "Dodaj tyle losowych rekordów". Przycisk "Zapisz" służy do zapisywania aktualnej listy mieszkań do pliku, przycisk "Otwórz" służy do odczytania listy z pliku, a przycisk "Dodaj tyle losowych rekordów" służy do dodawania do listy określonej liczby losowych rekordów.

[3] Użytkownik może zaznaczyć dowolne mieszkanie w liście [1], aby wyświetlić jego szczegółowe informacje w formularzu po prawej stronie. Formularz ten pozwala również na edycję informacji o mieszkaniu lub usunięcie go z bazy danych.

[4] Pośrodku znajduje się filtr do wyszukiwania rekordów w liście mieszkań. Składa się on z rozwijanej listy pozwalającej wybrać pole, według którego ma być przeprowadzane wyszukiwanie oraz pola tekstowego lub checkboxu (w zależności od wyboru pola) do wprowadzenia kryteriów wyszukiwania. Obok tych elementów znajduje się przycisk, który uruchamia wyszukiwanie po naciśnięciu. W ten sposób użytkownik może szybko i łatwo wyszukać rekordy spełniające określone kryteria w liście mieszkań.

[5] Na dole znajduje się panel z kartami, w którym użytkownik może wybrać rodzaj wyszukiwania. Na każdej karcie znajduje się lista wyników wyszukiwania dla danego rodzaju. W przypadku wyszukiwania łańcuchowego i inwersyjnego na karcie znajduje się także dodatkowa informacja o strukturach indeksowych, które są używane do przeprowadzenia wyszukiwania. Ta funkcjonalność pozwala użytkownikowi szybko i łatwo przejrzeć wyniki wyszukiwania dla różnych rodzajów oraz zapoznać się z dodatkowymi szczegółami dotyczącymi implementacji poszczególnych rodzajów wyszukiwania.

[6] Na prawej stronie GUI znajduje się panel informujący użytkownika o czasach wykonywania różnych metod. Ten panel pokazuje, ile czasu zajmuje wykonanie poszczególnych metod wyszukiwania dla różnych danych wejściowych.

4. Wyszukiwanie liniowe

Wyszukiwanie liniowe polega na iteracyjnym przeszukiwaniu elementów kolekcji, jednego po drugim, w poszukiwaniu elementu spełniającego określone kryteria. Algorytm wyszukiwania liniowego jest bardzo prosty, ale jednocześnie stosunkowo wolny, ponieważ wymaga przeszukania każdego elementu kolekcji.

```
IndeksyLiniowe.Clear();
for (int i = 0; i < Lista.Count; i++) {
  if (comparer.Compare(Lista[i], dummy) == 0) {
    IndeksyLiniowe.Add(i);
  }
}
```

Fragment kodu, który został przedstawiony, implementuje wyszukiwanie liniowe w kolekcji `Lista`. Algorytm rozpoczyna się od ustawienia zmiennej `i` na wartość `0`, co oznacza, że rozpoczynamy przeszukiwanie od pierwszego elementu kolekcji. Następnie w pętli `for` sprawdzamy, czy aktualny element kolekcji `Lista[i]` spełnia kryteria wyszukiwania. Jeśli tak, dodajemy ten element do kolekcji Liniowe. Po sprawdzeniu każdego elementu, zwiększamy wartość `i` o `1` i ponawiamy sprawdzanie dla kolejnych elementów, aż do momentu przeszukania całej kolekcji.

Warto zauważyć, że do porównywania elementów kolekcji używana jest metoda `comparer.Compare(...)`, która jest implementacją interfejsu `IComparer<T>`, który został wcześniej wybrany przez użytkownika w interfejsie graficznym. Dzięki temu użytkownik ma możliwość wyboru, według jakiego kryterium chce przeprowadzić wyszukiwanie liniowe.

5. Wyszukiwanie binarne

Wyszukiwanie binarne to algorytm, który polega na posortowaniu elementów bazy danych według wybranego kryterium, a następnie na przeszukiwaniu ich przy użyciu pętli `while`, podzieleniu przedziału poszukiwań na pół i porównaniu szukanego elementu z elementem pośrodku. Jeśli szukany element jest mniejszy od elementu pośrodku, przeszukiwany jest lewy podprzedział, a jeśli większy - prawy. Algorytm powtarza się, aż do momentu znalezienia szukanego elementu lub wyczerpania przedziału poszukiwań.

```
int low = 0;
int high = ListaPosortowana.Count - 1;
IndeksyBinarne.Clear();
while (low <= high){
  int mid = low + (high - low) / 2;
  int comparison = comparer.Compare(ListaPosortowana[mid], dummy);
  if (comparison == 0)        {
    IndeksyBinarne.Add(mid);
    int tempIndex = mid + 1;
    while (tempIndex < ListaPosortowana.Count &&
           comparer.Compare(dummy, ListaPosortowana[tempIndex]) == 0){
      IndeksyBinarne.Add(tempIndex);
      tempIndex += 1;
    }
    tempIndex = mid - 1;
    while (tempIndex >= 0 &&
           comparer.Compare(dummy, ListaPosortowana[tempIndex]) == 0){
      IndeksyBinarne.Add(tempIndex);
      tempIndex -= 1;
    }
    break;
  }
  else if (comparison < 0) low = mid + 1;
  else high = mid - 1;
}
```

Ten fragment kodu jest implementacją wyszukiwania binarnego, który służy do znajdowania elementu `dummy` w posortowanej liście `ListaPosortowana`. Zmienna `low` przechowuje indeks początku aktualnie przeszukiwanego zakresu w liście, a zmienna `high` przechowuje indeks końca aktualnie przeszukiwanego zakresu. Pętla `while` jest wykonywana tak długo, jak `low` jest mniejsze lub równe `high`. Wewnątrz pętli obliczany jest indeks środka aktualnie przeszukiwanego zakresu i porównywany jest z elementem `dummy` za pomocą funkcji `comparer.Compare()`.

Jeśli wynik tego porównania jest równy zero, oznacza to, że znaleziono szukany element. W takim przypadku jego indeks jest dodawany do listy `IndeksyBinarne` i rozpoczynają się dwie pętle, które szukają dodatkowych wystąpień elementu `dummy` w liście. Pętle te działają tak długo, jak indeks `tempIndex` jest odpowiednio większy lub równe zero lub mniejszy niż rozmiar listy, a także warunek `comparer.Compare(dummy, ListaPosortowana[tempIndex]) == 0` jest spełniony. Po zakończeniu pętli, algorytm kończy działanie.

W przeciwnym razie, jeśli wynik porównania jest mniejszy niż zero, oznacza to, że szukany element jest po prawej stronie aktualnie przeszukiwanego zakresu, więc zmienna `low` jest ustawiana na `mid + 1`. Jeśli wynik porównania jest większy niż zero, oznacza to, że szukany element jest po lewej stronie aktualnie przeszukiwanego zakresu, więc zmienna `high` jest ustawiana na `mid - 1`. Pętla `while` jest następnie kontynuowana, aż zostanie znaleziony szukany element lub stwierdzone, że elementu nie ma w liście.

Uwaga: wyszukiwanie binarne jest szybsze niż wyszukiwanie liniowe, ale wymaga posortowania elementów bazy danych.

6. Wyszukiwanie łańcuchowe

Wyszukiwanie łańcuchowe to metoda wyszukiwania, w której elementy są przechowywane w łańcuchach, a ich klucze są używane jako indeksy do tablicy z łańcuchami. Elementy o tym samym kluczu są dodawane do tego samego łańcucha.

```
IndeksyLancuchowe.Clear();
if (poczatki.ContainsKey(key)){
  for (j = poczatki[key]; lancuchy[j] != -1; j = lancuchy[j])
    IndeksyLancuchowe.Add(j);
  IndeksyLancuchowe.Add(j);
}
```

Ten fragment kodu służy do wyszukiwania elementów o danym kluczu `key` w liście danych za pomocą indeksu zbudowanego poprzez wyszukiwanie łańcuchowe. Najpierw lista `IndeksyLancuchowe` jest czyszczona. Następnie sprawdzane jest, czy słownik `poczatki` zawiera klucz o nazwie `key`. Jeśli tak, rozpoczynana jest pętla `for`, która działa tak długo, jak wartość elementu o indeksie `j` w słowniku `lancuchy` jest różna od `-1`. W każdym przebiegu pętli indeks elementu o indeksie `j` jest dodawany do listy `IndeksyLancuchowe` i zmienna `j` jest ustawiana na wartość elementu o indeksie `j` w słowniku `lancuchy`. Po zakończeniu pętli, indeks elementu o indeksie `j` jest również dodawany do listy `IndeksyLancuchowe`. W ten sposób wszystkie indeksy elementów o danym kluczu są dodawane do tej listy.

```
Dictionary<string, int> poczatki = new();
List<int> lancuchy = new();
Dictionary<string, int> konce = new();
string klucz;
for (int i = 0; i < Lista.Count; i++){
  if (index == 0) klucz = Lista[i].Id.ToString();
  else if (index == 1) klucz = Lista[i].Nazwa;
  else if (index == 2) klucz = Lista[i].Opis;
  else if (index == 3) klucz = Lista[i].Powierzchnia.ToString();
  else if (index == 4) klucz = Lista[i].MaxOsob.ToString();
  else if (index == 5) klucz = Lista[i].Cena.ToString();
  else if (index == 6) klucz = Lista[i].MaPralke.ToString();
  else klucz = Lista[i].MaJacuzzi.ToString();
```

```
    if (!poczatki.ContainsKey(klucz)){
      poczatki.Add(klucz, i);
      konce.Add(klucz, i);
      lancuchy.Add(-1);
    }
    else{
      lancuchy[konce[klucz]] = i;
      lancuchy.Add(-1);
      konce[klucz] = i;
    }
  }
```

Indeks jest tworzony za pomocą trzech słowników: `poczatki`, `konce` i `lancuchy`. Słownik `poczatki` przechowuje indeksy początków łańcuchów elementów o danym kluczu, słownik `konce` przechowuje indeksy końców łańcuchów elementów o danym kluczu, a słownik `lancuchy` przechowuje informacje o tym, gdzie znajduje się następny element w łańcuchu dla każdego elementu. Pętla `for` jest wykonywana dla każdego elementu z listy `Lista`, a zmienna `klucz` jest ustawiana na odpowiednią wartość dla danego elementu na podstawie indeksu `index` (0 → Id, 1 → Nazwa, itp.).

Jeśli dana wartość klucza nie jest jeszcze obecna w indeksie, jest ona dodawana do słowników `poczatki` i `konce` oraz do listy `lancuchy` jest dodawana wartość -1, co oznacza koniec łańcucha. W przeciwnym razie, jeśli wartość klucza już jest obecna w indeksie, do listy `lancuchy` jest dodawana wartość indeksu aktualnego elementu, a wartość indeksu aktualnego elementu jest również dodawana do słownika `konce` jako indeks końca aktualnego łańcucha dla danej wartości klucza.

Uwaga: słownik `konce` jest używany tylko tymczasowo w procesie tworzenia indeksu do wyszukiwania łańcuchowego. Jego głównym celem jest umożliwienie szybszego dodawania nowych elementów do istniejących łańcuchów w indeksie. Dzięki temu, że indeks końca aktualnego łańcucha dla danej wartości klucza jest przechowywany w słowniku `konce`, możliwe jest szybkie odnalezienie tego indeksu i ustawienie odpowiedniej wartości w liście `lancuchy`, co pozwala na łączenie elementów w łańcuchy bez konieczności przeszukiwania całej listy `lancuchy` od początku. Dzięki temu proces tworzenia indeksu do wyszukiwania łańcuchowego jest bardziej efektywny pod względem czasowym. Po zakończeniu tworzenia indeksu, słownik `konce` nie jest już potrzebny i może zostać usunięty.

7. Wyszukiwanie metodą list inwersyjnych.

Wyszukiwanie metodą list inwersyjnych polega na tworzeniu specjalnej struktury danych, tzw. kartoteki, w której dla każdego możliwego klucza przechowywana jest lista indeksów elementów o tej wartości klucza. Podczas wyszukiwania elementu o określonym kluczu, sprawdzana jest obecność tego klucza w kartotece. Jeśli klucz jest obecny, oznacza to, że istnieją elementy o tej wartości klucza i ich indeksy są zawarte w odpowiedniej liście w kartotece. W ten sposób możliwe jest szybkie znalezienie wszystkich elementów o danym kluczu bez konieczności przeszukiwania całej kolekcji danych.

```
  IndeksyInwersyjne.Clear();
  if (kartoteka.ContainsKey(key)){
    IndeksyInwersyjne = kartoteka[key].ToList();
  }
```

Jest to metoda która wyszukuje klucz w strukturze danych typu słownik o nazwie `kartoteka` i zapisuje wartość powiązaną z tym kluczem w liście o nazwie `IndeksyInwersyjne`. Tworzenie struktury indeksowej wygląda następująco:

```
Dictionary<string, List<int>> kartoteka = new();
string klucz;
for (int i = 0; i < Lista.Count; i++){
  if (index == 0) klucz = Lista[i].Id.ToString();
  else if (index == 1) klucz = Lista[i].Nazwa;
  else if (index == 2) klucz = Lista[i].Opis;
  else if (index == 3) klucz = Lista[i].Powierzchnia.ToString();
  else if (index == 4) klucz = Lista[i].MaxOsob.ToString();
  else if (index == 5) klucz = Lista[i].Cena.ToString();
  else if (index == 6) klucz = Lista[i].MaPralke.ToString();
  else klucz = Lista[i].MaJacuzzi.ToString();
  if (kartoteka.ContainsKey(klucz)){
    kartoteka[klucz].Add(i);
  }
  else{
    kartoteka[klucz] = new List<int>();
    kartoteka[klucz].Add(i);
  }
}
```

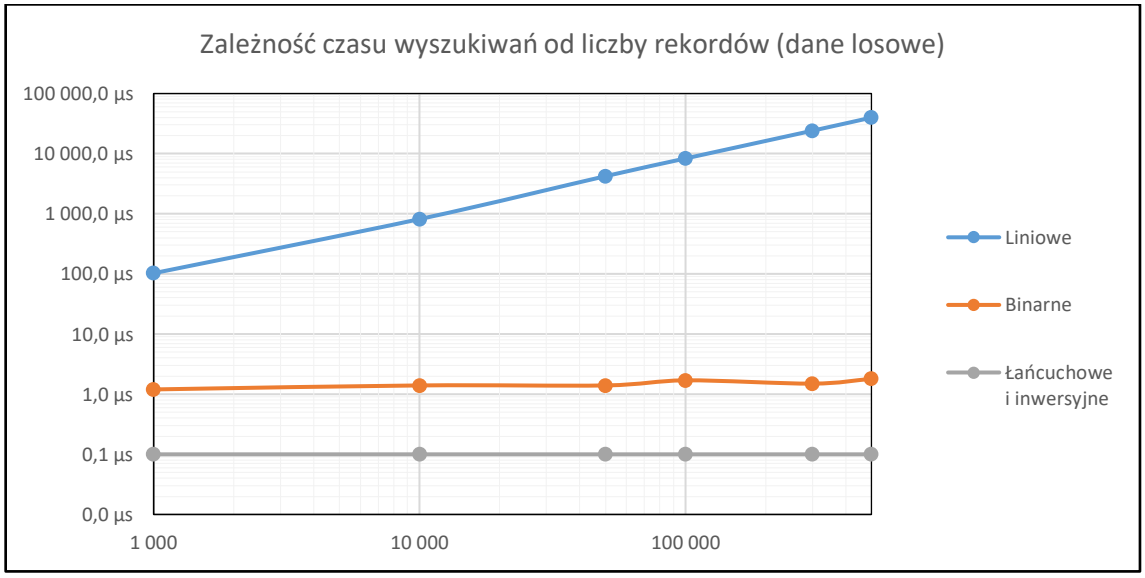
Ten kod tworzy nowy słownik o nazwie `kartoteka`, który przechowuje listy indeksów elementów z listy `Lista` jako wartości dla różnych kluczy. Zmienna `klucz` jest inicjowana na początku pętli `for` i ustawiana na odpowiednią wartość dla danego elementu na podstawie indeksu `index` (0 → Id, 1 → Nazwa, itp.). Następnie sprawdzane jest, czy kartoteka już zawiera klucz o tej nazwie. Jeśli tak, indeks elementu jest dodawany do już istniejącej listy dla tego klucza. W przeciwnym razie tworzona jest nowa lista dla tego klucza i indeks elementu jest dodawany do niej. Pętla `for` jest kontynuowana dla każdego elementu z listy `Lista`, aż zostaną przetworzone wszystkie elementy.

8. Porównanie poszukiwań

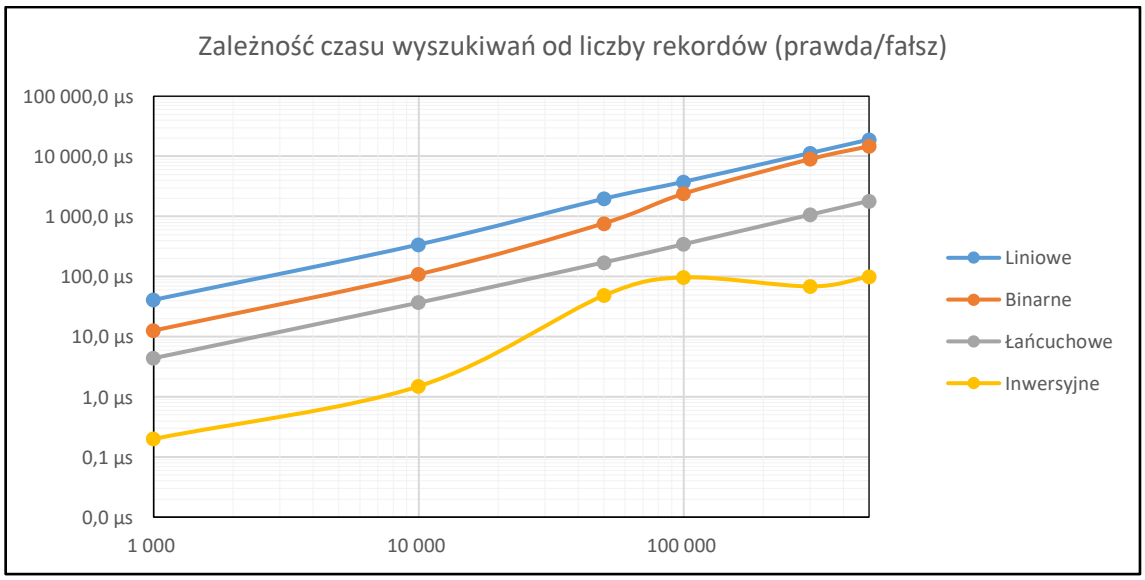
Porównywanie algorytmów poszukiwań polega na sprawdzeniu, który z dostępnych algorytmów danego typu jest najszybszy w danych warunkach. Można to zrobić poprzez zmierzenie czasu, jaki każdy z algorytmów potrzebuje do wyszukania określonego elementu lub zestawu elementów w danych. Należy pamiętać, aby porównywać algorytmy w podobnych warunkach, tj. na podobnych danych lub w tych samych warunkach. Pozwoli to uzyskać bardziej obiektywny wynik i uniknąć błędów w interpretacji.

Na wykresie przedstawiono porównanie skuteczności 4 różnych algorytmów wyszukiwania dla różnych liczb rekordów. Dla celów porównania przyjęto dwa warunki: dane z całkowicie losowymi wartościami oraz dane z wartościami z określonego przedziału (tj. "prawda/fałsz" w naszym przypadku). Warto zwrócić uwagę na to, jak algorytmy radzą sobie w obu tych przypadkach i czy ich wydajność różni się w zależności od warunków.

Na poniższych wykresach przedstawiono czasy działania różnych algorytmów wyszukiwania dla różnych ilości danych. Na osi x znajduje się liczba rekordów, a na osi y czas wyszukiwania w mikrosekundach. Osi są logarytmiczne, co umożliwia lepsze zobrazowanie dużych różnic między wartościami.



1. Rysunek: Wykres porównawczy zaimplementowanych metod wyszukiwania dla różnych liczb rekordów dla kolumny z wartościami całkowicie losowymi



2. Rysunek: Wykres porównawczy zaimplementowanych metod wyszukiwania dla różnych liczb rekordów dla kolumny z wartościami z danego przedziału (prawda/fałsz)

Opis przedstawionych wykresów

Porównując czasy wyszukiwania liniowego i binarnego, możemy zauważyć, że są one uzależnione od liczby rekordów. Im więcej rekordów, tym dłuższy jest czas wyszukiwania. Algorytm wyszukiwania liniowego ma złożoność $O(n)$, natomiast wyszukiwanie binarne $O(log(n))$. Z kolei czasy wyszukiwania łańcuchowego i inwersyjnego są niezależne od liczby rekordów i wynoszą $O(1)$. Możemy to zaobserwować na wykresie przedstawiającym wartości całkowicie losowe (1. Rysunek). Należy jednak pamiętać, że wszystkie wyszukiwania są uzależnione od liczby trafień. Możemy to zaobserwować na wykresie przedstawiającym wartości z danego przedziału (2. Rysunek).

Wnioski

Wyszukiwanie liniowe jest najprostszą metodą, ale również jedną z najwolniejszych. Jego wydajność maleje wraz ze wzrostem liczby rekordów, co skutkuje nieefektywnym działaniem w przypadku dużych baz danych. Z tego powodu nie jest ono zalecane do użytku w takich przypadkach.

W porównaniu z wyszukiwaniem liniowym, wyszukiwanie binarne jest znacznie szybsze i cechuje się mniejszą zmiennością wydajności wraz ze wzrostem liczby rekordów. Dlatego też jest ono dobrą opcją do szybkiego wyszukiwania w posortowanych kolekcjach danych. Jedyną wadą tej metody jest konieczność posortowania zbioru przed jej użyciem.

Wyszukiwanie łańcuchowe oraz wyszukiwanie metodą list inwersyjnych są uważane za najszybsze metody wyszukiwania dostępne. Ich wydajność jest niezależna od liczby rekordów, co sprawia, że są one bardzo efektywne nawet w przypadku dużych baz danych. W praktyce są one często stosowane w przypadku, gdy mamy do czynienia z dużymi zbiorami danych, a szybkość działania jest kluczowa. Te metody charakteryzują się podobnymi wadami. Obie metody wymagają utworzenia dodatkowych struktur danych do przechowywania elementów zbioru, co może prowadzić do zwiększenia kosztów pamięciowych oraz spowolnienia działania algorytmu. Obie metody nie są dobrze dostosowane do dynamicznych zbiorów danych, w których elementy często są dodawane lub usuwane, ponieważ po każdej zmianie danych w zbiorze, niezbędne jest także przeprowadzenie aktualizacji indeksów. W przeciwnym razie, dane w indeksach mogą być nieaktualne i wyszukiwanie może dawać nieprawidłowe rezultaty. Dlatego też utrzymanie indeksów w aktualnym stanie jest ważnym elementem pracy z bazami danych.

Appendix: Dane do wykresów

1. Tabela: czasy wyszukiwania dla różnych liczb rekordów dla kolumny z wartościami całkowicie losowymi

Liczba rekordów [szt]	1 000	10 000	50 000	100 000	300 000	500 000
Liniowe [µs]	102,8	816,9	4 223,9	8 332,2	24 094,5	39 823,1
Binarne [µs]	1,2	1,4	1,4	1,7	1,5	1,8
Łańcuchowe i inwersyjne [µs]	<0,1	<0,1	<0,1	<0,1	<0,1	<0,1
Liczba trafień [szt]	1	1	1	1	1	1

2. Tabela: czasy wyszukiwania dla różnych liczb rekordów dla kolumny z wartościami z danego przedziału (prawda/fałsz)

Liczba rekordów [szt]	1 000	10 000	50 000	100 000	300 000	500 000
Liniowe [µs]	41,1	341,8	1 976,9	3 783,2	11 368,7	19 010,2
Binarne [µs]	12,7	109,3	770,9	2 398,6	9 007,8	14 709,6
Łańcuchowe [µs]	4,4	36,9	172,6	346,5	1 072,6	1 818,3
Inwersyjne [µs]	0,2	1,5	48,7	97,1	68,7	100,0
Liczba trafień [szt]	497	5 011	24 795	49 274	148 486	247 335