1 Lista 4, Zadanie 1

Mamy pokazać, że w modelu CM (Comparison Model) przeszukiwanie posortowanej tablicy wymaga $\Omega(\log n)$ porównań. Co oznacza, że zachowując pełną ogólność, mamy pokazać, że algorytm zadający pytanie $A[i] \geq z$ w najgorszym przypadku wykona $\Omega(\log n)$ porównań.

1.1 Dowód

Oznaczmy:

- A - dowolny algorytm wyszukiwania elementu.
- T - posortowana tablica.
- n - długość tablicy.
- x - wyszukiwany element.

Algorytm wykorzystując jedynie zapytanie postaci $A[i] \geq z$, gdzie z to element tablicy, zwraca $T[i] == x$ lub informacje o braku elementu x w tablicy.
Algorytm w każdym kroku zadaje jedynie pytanie wyżej wymienionej postaci i na podstawie wyniku zapytania decyduje, który elementy rozważyć następnie. Zatem sekwencje zapytania można przedstawić jako drzewo binarne gdzie węzły odpowiadają sekwecnji wykonanych porównań, natomiast liście zawierają rezultat programu. Należy zwrócić uwagę, że algorytm A może wyprodukować drzewo niepełne.
Liście zawierają rezultat algorytmu, wiemy, że odpowiedzi jest n, ponieważ szukany element może znajdować się na każdej z n pozycji tablicy, zatem drzewo będzie miało co najmniej n liści. Drzewo binarne o n liściach ma wysokość rzędu $\Omega(\log n)$, toteż ilość porównań algorytmu jest rzędu $\Omega(\log n)$, co należało pokazać.

Binary-Search ma dokładnie n liści, co pokazuje, że jest on optymalnym algorytmem do przeszukiwania posortowanej tablicy.

Nie możemy zadać pytania $A[i] == x$, natomiast zadanie pytania $A[i] \geq z$ & $z \geq A[i]$ jest tym co nam to zastąpi. Ponadto nie musimy martwić się o złożoność zadawania dwóch pytań, ponieważ są to pytania podstawowe, o złożonosci jednostkowej.

2 Dodatek o Binary-Search

Pomędrkuję o złożoności algorytmu Binary-Search i pokażę, że w CM wyszukiwanie jest $\Omega(\log n)$. Innymi słowy chcę pokazać, że algorytm Binary-Search ma ograniczenie dolne $k \cdot \log n$, dla pewnej stałej k.

2.1 Dowód

Przedstawmy kolejne (i zarazem wszytskie możliwe) wartości w przeszukiwanej tablicy za pomocą drzewa binarnego, gdzie węzeł to indeks sprawdzanego elementu. Pokażę na przykładzie dla $n = 15$, (co jest bardzo życzeniowe, ale pokazuje dokładnie co się dzieje), jak wartości tablicy są przeglądane. Używanie do zapisu n robi się szybko bardzo nieczytelne.

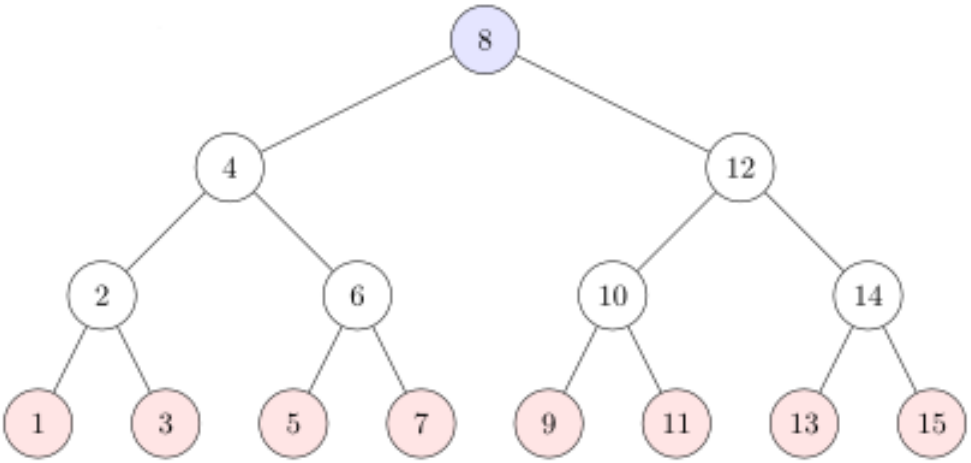

Figure 1: Drzewo indeksów dla n = 15

Możemy zauważyć, że:

1. Każdy indeks tablicy występuje w drzewie. Dla $n = 2^k - 1$, $k > 0$ drzewo będzie pełne, dla pozostałych niepełne.

2. Wysokość drzewa to $\lfloor \log 15 \rfloor + 1 = 4$. W ogólności wysokość drzewa to $\lfloor \log n \rfloor + 1$ wynika to z tego, że każdy $x \in [n]$ wystąpi w drzewie tylko raz.

Algorytm Binary-Search przechodzi drzewo w głąb aż natrafi na szukaną wartość lub dojdzie do węzła bez dziecka i zwróci informacje o braku występienia elementu w tablicy.
Na przeszukiwanie w głąb możemy sobie pozwolić, dzięki posortowanej tablicy i zadanemu porządkowi na wartościach tablicy, ponieważ gwarantuje on nam, że na lewo od elementu $A[i]$ są tylko wartości mnijesze niż $A[i]$.

Ilość, wywołań Binary-Search w implementacji rekurencyjnej zależy, oczywiście, od wartości elementu którego suzkamy. W najgorszym przypadku, powiedzmy $index(x) = 1$, BS zostanie wywołany $\lfloor \log n \rfloor + 1$ razy. Naturalnie algorytm może trafić wcześniej na szukany element, natomiast w rzeczywistych przypadkach, radosne spotkanie pouszukiwanej wartości z algorytmem nastąpi w najniższych

poziomach drzewa, proszę zwrócić uwagę, że dla kazdego poziomu l drzewa ilość elementów zawartych na niższych poziomach $before$, to $before = 2^{l-1} - 1$ (w drzewach pełnych, ale w drzewach niepełnych znacząca różnica jest zachowana), więc w naszym drzewie na poziomie 4 mamy — 53,(3)% wszytskich elementów na 3 — 26,(6)%, na 2 — 13,(3)%, a na 1 zaledwie 6,(6)%. W takim razie dla BS złożoność $T(n) = \Omega(\log n)$, ponieważ poszukiwaną wartość spodziewamy się znaleźć na najniższych poziomach rekurencji, a dla pozostałych przypadków istnieje stała k dla, której $T(n) \geq k \cdot \log n$.