ASD, Wyk 09, 2022.04.21 - Struktury słownikowe

Wyszukiwanie binarne i drzewa BST (Binary Search Tree)

Mamy daną tablicę $t[0..n)$, uporządkowaną względem porządku $\leq$.

Chcemy w tej tablicy wyszukać element key.

Mamy gwarancję, że szukany element, jeśli jest w tablicy, to jest w części $t[left..right]$.

Na początku $left = 0$, $right = n - 1$.

Upraszczanie takiej pracy nam redukuje o połowę rozmiar problemu

Obliczam element środkowy przedziału

$$mid = left + \left\lfloor \frac{right - left + 1}{2} \right\rfloor$$

① $t[mid] = key \rightarrow$ return mid

② $t[mid] > key \rightarrow$ czyli key jest na lewo od mid $\rightarrow$ right = mid − 1.

③ $t[mid] < key \rightarrow$ Analogicznie, left = mid + 1.

Kod binary search

```
-- binary search in array t[0..size-1]

int binary_search(int* t, int size, int key){
int left=0;
int right=size-1;
while(left<=right){
        int mid = left +  floor((right - left+1)/2);
        if(t[mid]=key)  return mid;
        if(t[mid]>key)  right=mid-1;
         if(t[mid]<key) left=mid+1;
 }
 return -1;
}
```

Niezmiennik pętli: jeśli w tablicy jest element key, to jest on w części t[left..right].

Czas działania jest logarytmiczny bo z każdą iteracją pętli zmniejszamy o połowę długość przedziału, w którym szukamy klucza key.

Struktura słownika

Operacje ① wyszukiwanie elementów słownika po ich indeksach (kluczach)

② wstawianie nowych elementów

③ usuwanie elementów.

Co jeszcze:

sprawdź czy słownik jest pusty

połącz dwa słowniki

skopiuj słownik.

Żeby słownik działał efektywnie
musimy założyć coś o kluczach.
Załóżmy, że mamy dany
liniowy porządek na kluczach.

Operacje słownikowe

Elem find(Key key)
void insert(Elem e)
void delete(Key key)
Bool empty()

Czy uporządkowana tablica

to dobra struktura słownikowa?

find = $O(\log_2 n)$, gdzie n to rozmiar tablicy.

insert, delete

mają koszt (pesymistyczny)

insert(k)

① znajdujemy miejsce w tablicy dla k
wyszukiwanie binarne → $O(\log_2 n)$

② musimy przesunąć o jeden wyżej elementy
żeby zrobić miejsce → $O(n)$.

delet(k) – podobnie koszt $O(n)$, jeśli usuwamy element

lub

koszt $O(\log_2 n)$ jeśli element tylko zaznaczamy jako usunięty,

ale wtedy mamy z czasem coraz więcej elementów zaznaczonych jako usunięte co spowalnia działanie słownika.

Inne rozwiązanie

Słownik w tablicy nieuporządkowanej

ale tutaj już koszt wyszukania jak

liniowy.

Czy można wykonać te trzy operacje
słownikowe szybciej?

Tezka

Na poziomie implementacji słownika
przez drzewa wyszukiwań binarnych (BST)

Co to jest drzewo BST?

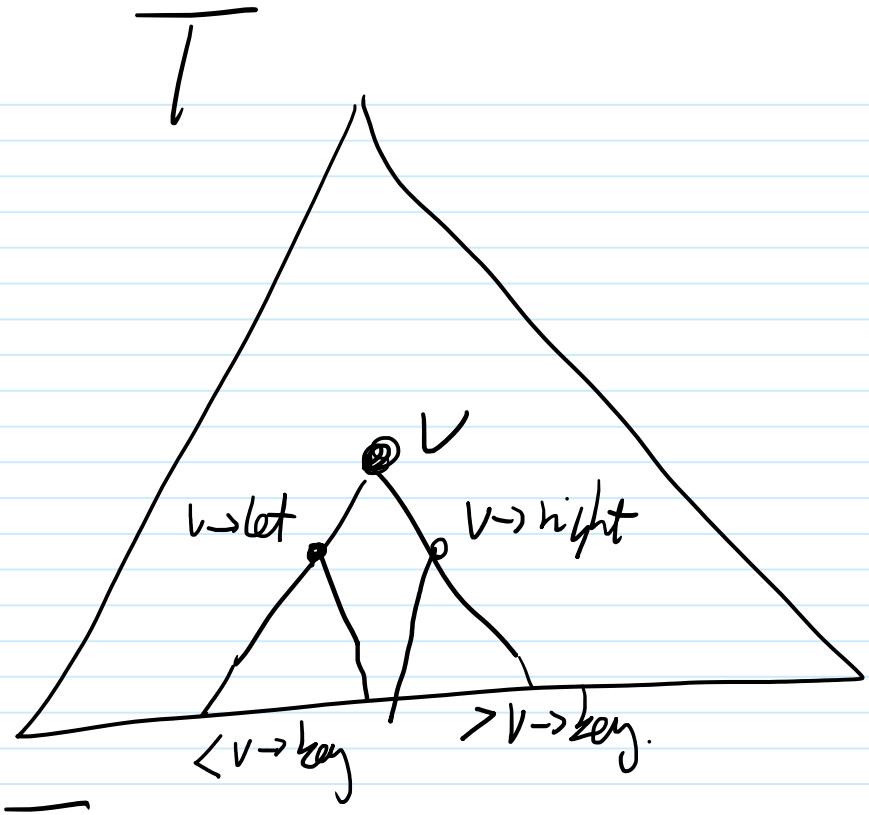
Elem = key = int, dla uproszczenia

Def

Drzewo BST to drzewo binarne takie, że każdy wierzchołek v w tym drzewie ma następującą własność:

① wszystkie klucze w lewym podrzewie v są mniejsze (ostro) od $v \to key$

② wszystkie klucze w prawym podrzewie v są ostro większe od $v \to key$.



Niech $v \in T$

$v \to key$ – klucz v.

$Tree(v)$ – poddrzewo T o korzeniu v

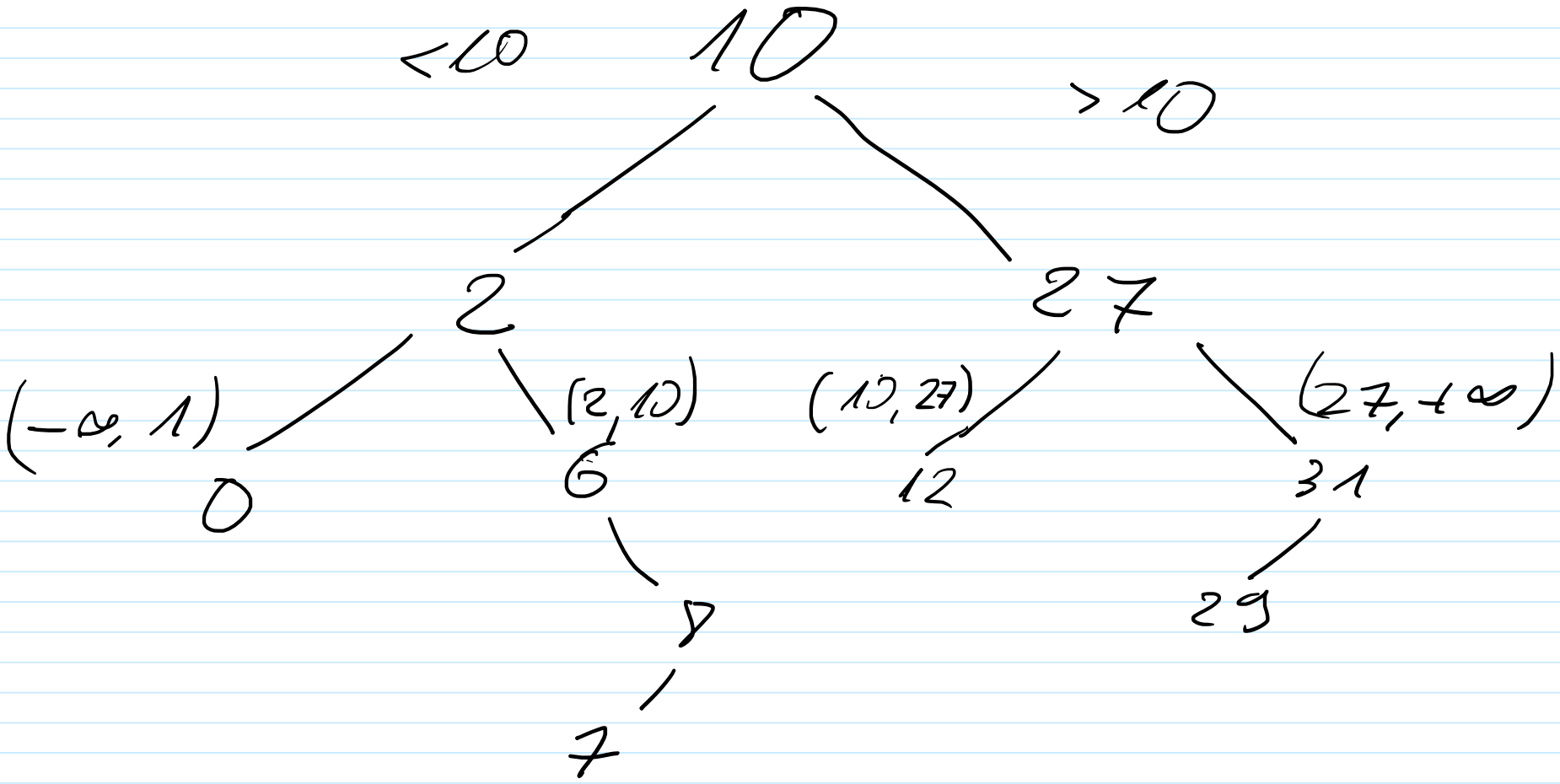
$v \to left$ – lewe dziecko v

$v \to right$ – prawe dziecko v

Def T jest BST gdy

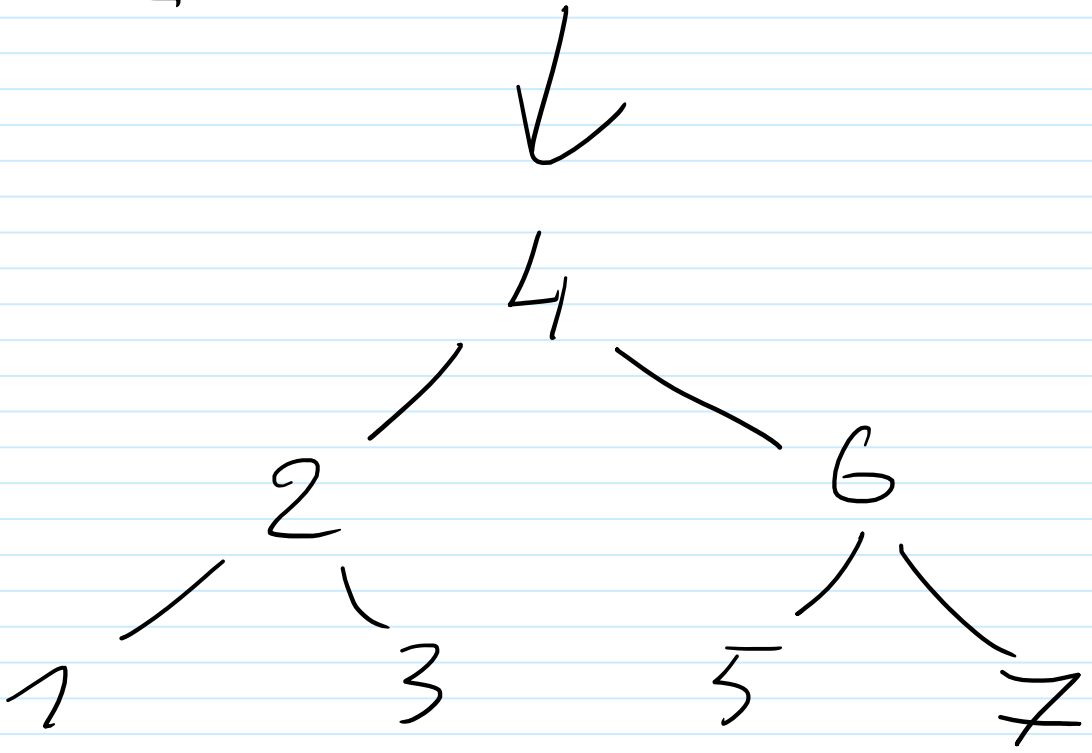
$$\forall v \in T \left((\forall w \in Tree(v \to left) \; w \to key < v \to key) \wedge (\forall w \in Tree(v \to right) \; w \to key > v \to key) \right)$$

Jeśli $Tree(v \to left)$ lub $Tree(v \to right)$ są puste, to odpowiedni warunek jest spełniony z a darmo.

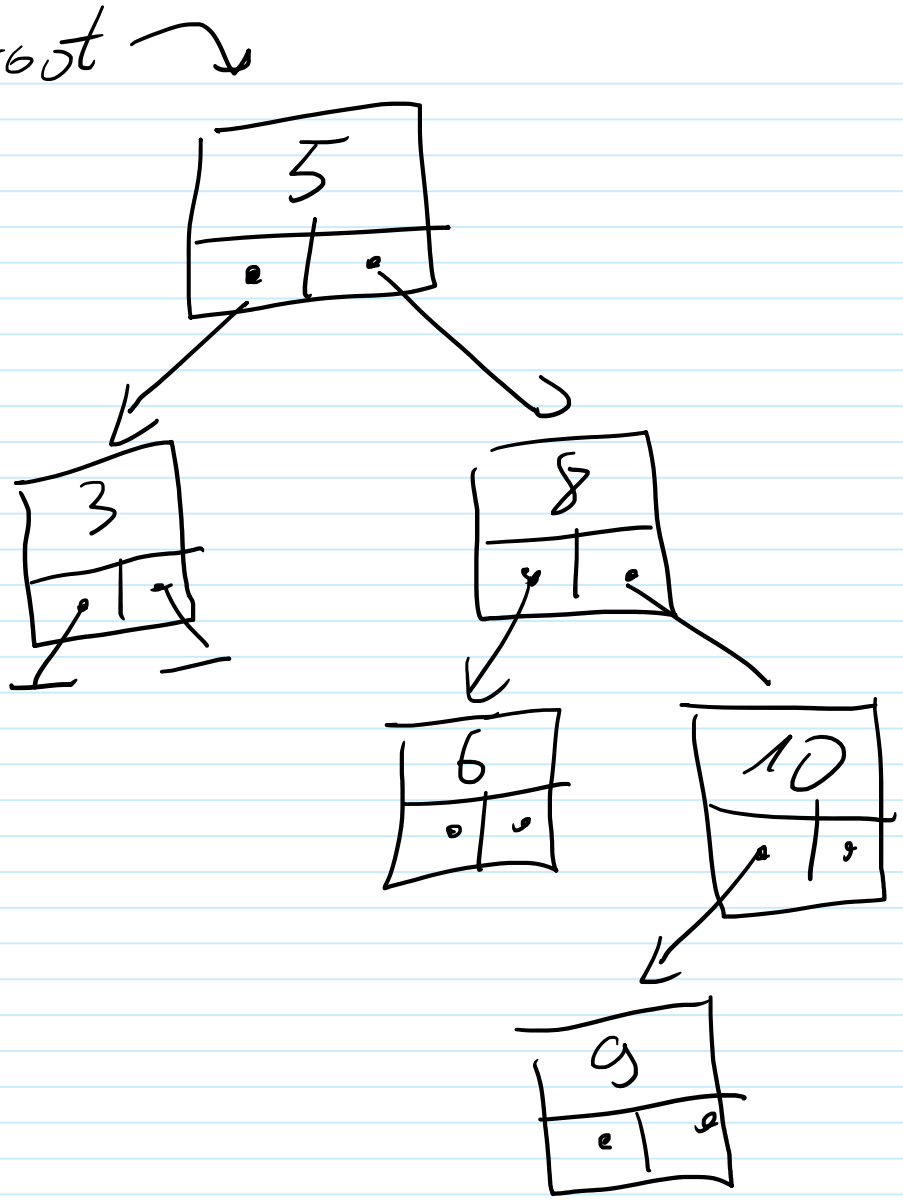

< 10
10
> 10
2
27
(−∞, 1)
0
[2, 10)
6
(10, 27)
12
(27, +∞)
31
7
29

Analogia z wyszukiwaniem potrójnym.

[1, 2, 3, 4, 5, 6, 7]



Implementacja BST




```
struct Node{
      int key;
      Data info o elemencie o kluczu key;
      Node *left;
       Node *right;
}Node;

Node *root=NULL;
```

Wyszukiwanie

```
Node* find(int k){
     Node *ptr=root;
     while(ptr!=NULL){
          if(ptr->key==k) return ptr;
          if(ptr->key>k) ptr=ptr->left;
          if(ptr->key<k) ptr=ptr->right;
     }
    return NULL;
}
```