Drzewa wyszukiwań binarnych

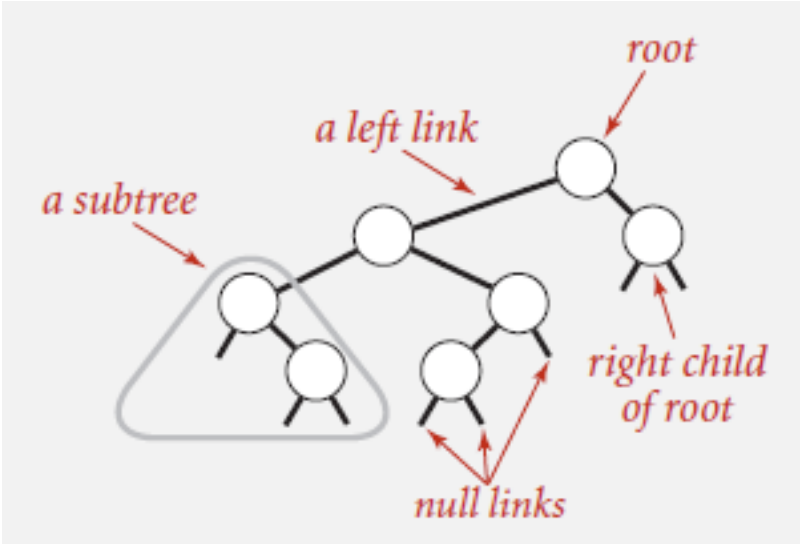
Drzewo wyszukiwań binarnych (*ang. binary search tree* - BST) jest to **drzewo binarne** z **symetrycznym porządkiem**.

Drzewo binarne:

- Zawiera dwa rozłączne drzewa binarne (lewe i prawe)

  lub

- Jest puste

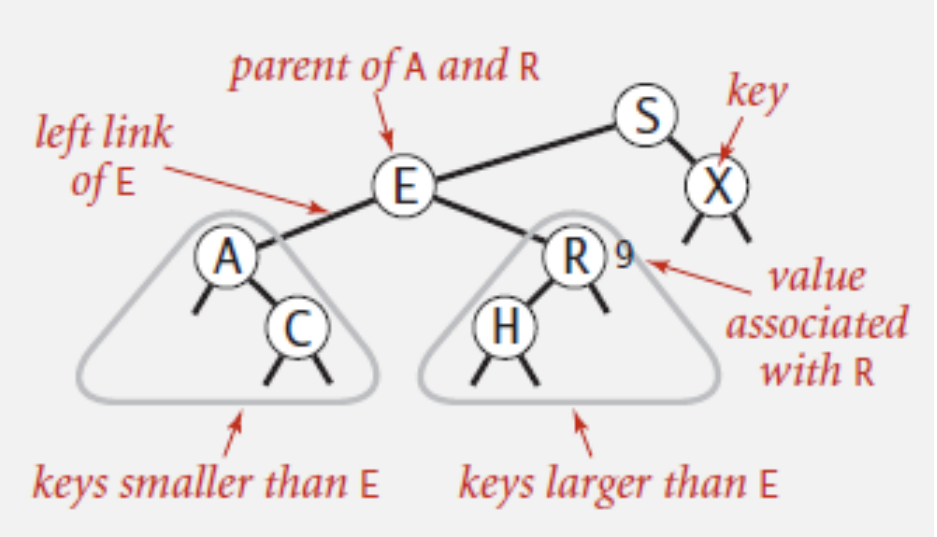


[SW-2018]

Drzewa wyszukiwań binarnych

Porządek symetryczny:

- Każdy wierzchołek zawiera klucz (+ implementacja `Comparable`)
- Klucz w dowolnym wierzchołku jest:
  - Większy niż wszystkie klucze w jego lewym poddrzewie
  - Mniejszy niż wszystkie klucze w jego prawym poddrzewie



[SW-2018]

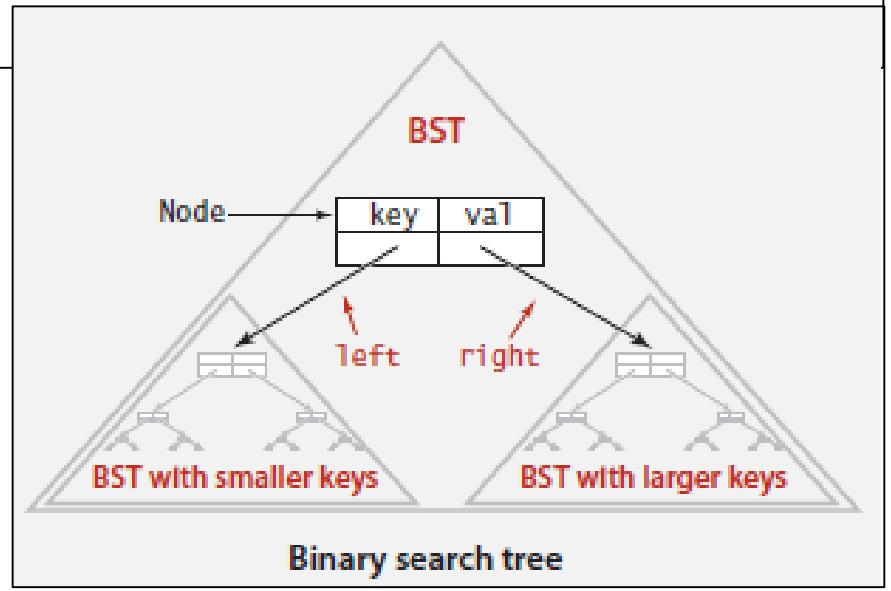
Drzewa wyszukiwań binarnych

Implementacja BST za pomocą listy powiązanej

`Node` składa się z czterech pól:

- `Key` oraz `Value`
- Referencję do lewego i prawego poddrzewa

```
public class BST {
   private Node root;

   private class Node {
      private Key key;
      private Value val;
      private Node left, right;
   } ...
}
```



[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja BST za pomocą listy powiązanej

```
public class BST<Key extends Comparable<Key>,
Value>
{
   private Node root;
   private class Node {
       …
   }

   public Value get(Key key) {
   }
   public void put(Key key, Value val) {
   }
   public void delete(Key key) {
   }
}
```

Drzewa wyszukiwań binarnych

Operacja `get()`

Zwraca wartość odpowiadającą danemu kluczowi, ew. zwraca null, gdy nie znajdzie klucza.

Liczba porównań wynosi:

1+głębokość drzewa (dokładniej wierzchołka)

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if (cmp < 0) x = x.left;
      else if (cmp > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

Drzewa wyszukiwań binarnych

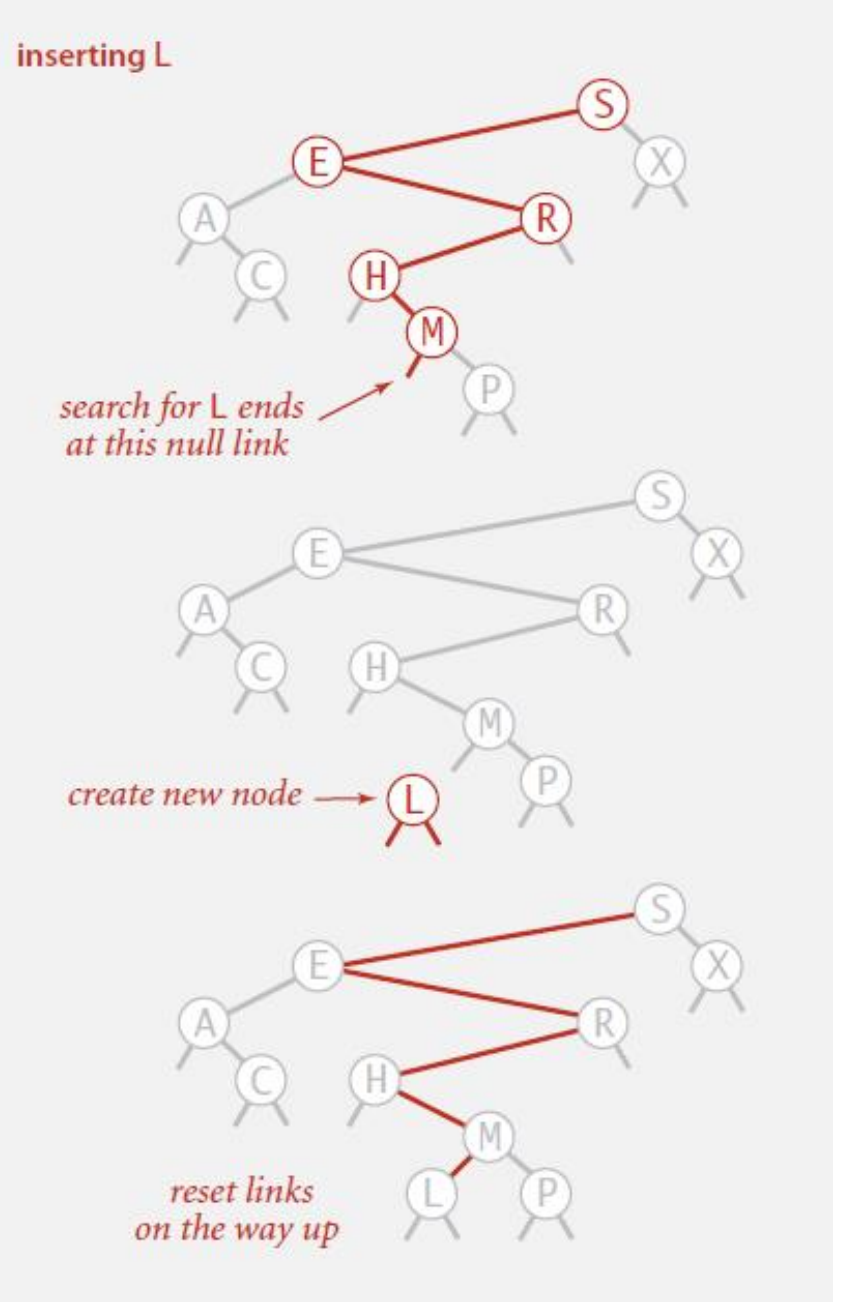
Operacja `put()`

Wstawia parę (klucz, wartość):

- Gdy klucz istnieje w drzewie – nadpisz wartość
- Gdy klucz nie istnieje w drzewie – dodaj nowy wierzchołek

Liczba porównań wynosi:

1+głębokość drzewa



[SW-2018]

Drzewa wyszukiwań binarnych

Operacja `put()`

Wstawia parę (klucz, wartość):

- Gdy klucz istnieje w drzewie – nadpisz wartość
- Gdy klucz nie istnieje w drzewie – dodaj nowy wierzchołek

Liczba porównań wynosi:

1+głębokość drzewa

```
public void put(Key key, Value val)
{
   root = put(root, key, val);
}

private Node put(Node x, Key key, Value val)
{
   if (x == null)
      return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
     x.left = put(x.left, key, val);
   else if (cmp > 0)
     x.right = put(x.right, key, val);
   else
     x.val = val;
   return x;
}
```

Drzewa wyszukiwań binarnych

Operacja `deleteMin()`

- Schodzimy rekurencyjnie do lewych poddrzew aż napotkamy wierzchołek z referencją do lewego poddrzewa równą null
- Zastąp tę referencję referencją do jego prawego poddrzewa

```
public void deleteMin()
{
   root = deleteMin(root);
}

private Node deleteMin(Node x)
{
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   return x;
}
```

Drzewa wyszukiwań binarnych

Operacja `delete()`

Usuwa wierzchołek *t* o wskazanym kluczu *key*

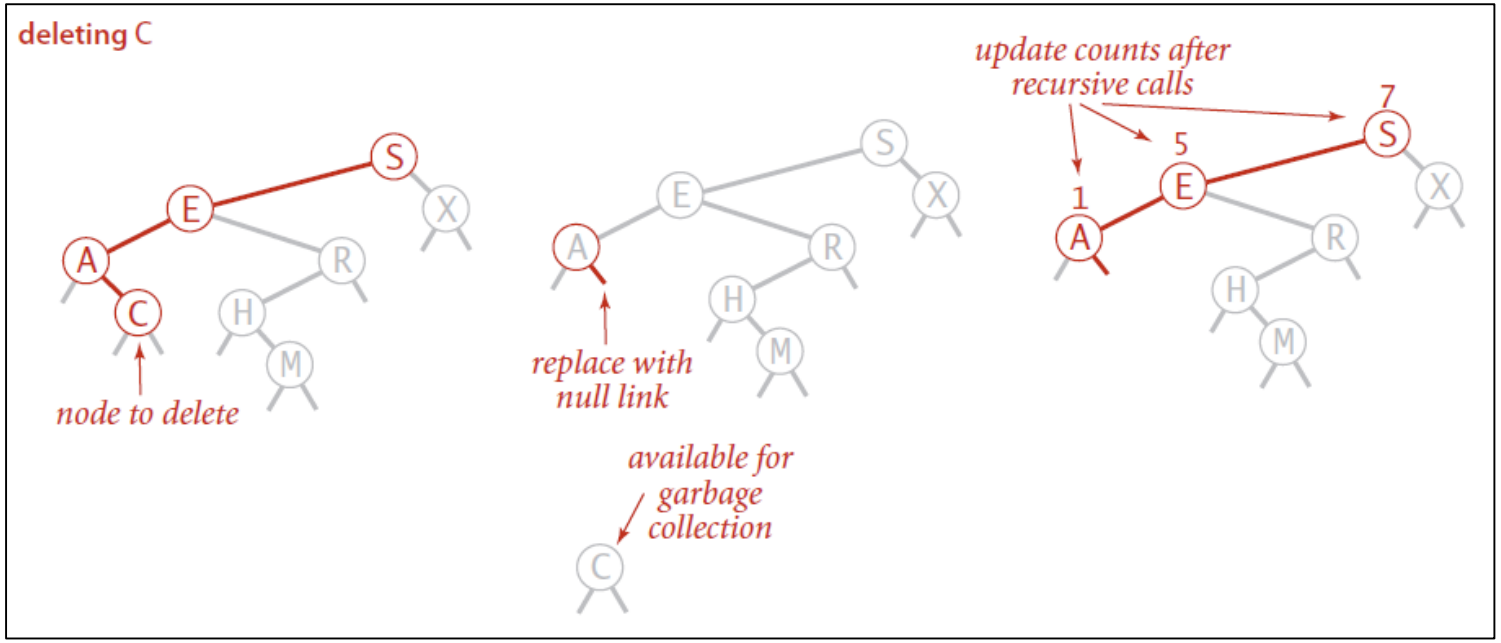
- Znajdź ten wierzchołek
- Usuń go

Drzewa wyszukiwań binarnych

Operacja `delete()`

Przypadek 1: Znaleziony wierzchołek *t* ma 0 potomków

- Usuń *t* przez ustawienie w wierzchołku przodka linku do tego drzewa (wierzchołka) potomka na null



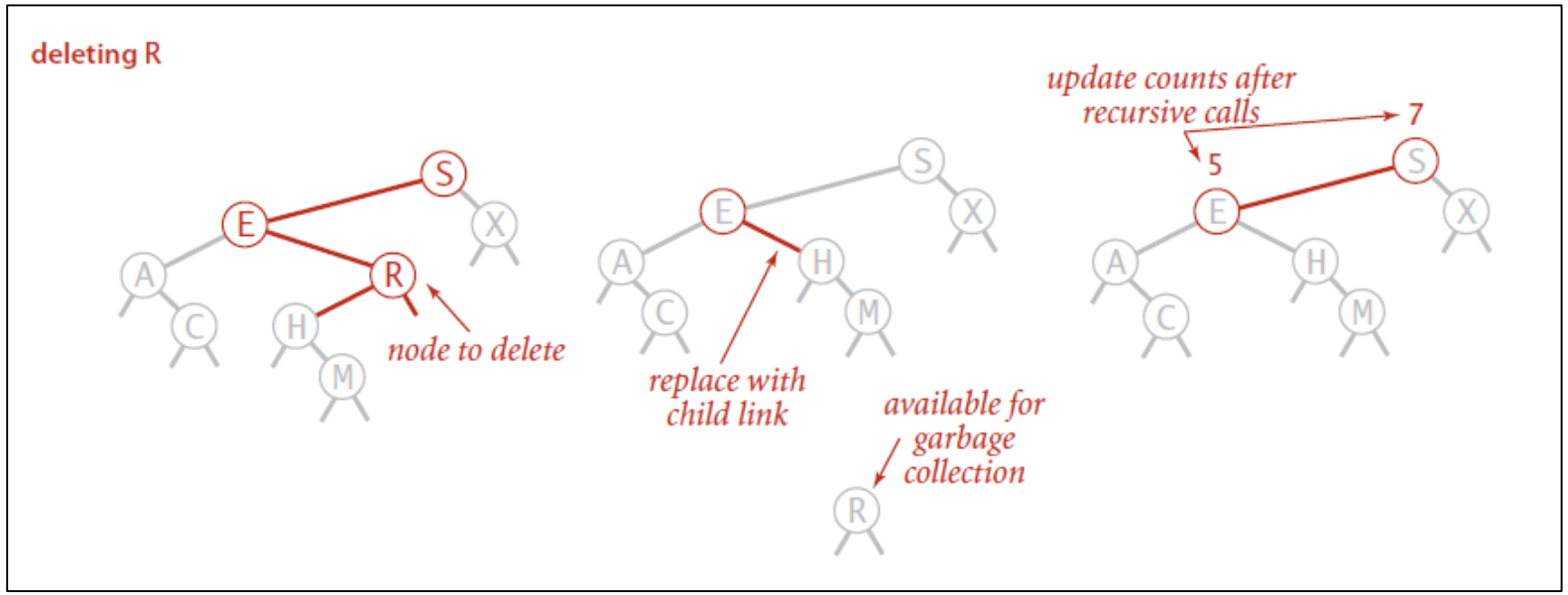
[SW-2018]

Drzewa wyszukiwań binarnych

Operacja `delete()`

Przypadek 2: Znaleziony wierzchołek *t* ma 1 potomka

- Usuń *t* przez przekierowanie w wierzchołku przodka linku do tego potomka drzewa (wierzchołka) potomka na potomka usuwanego wierzchołka *t*



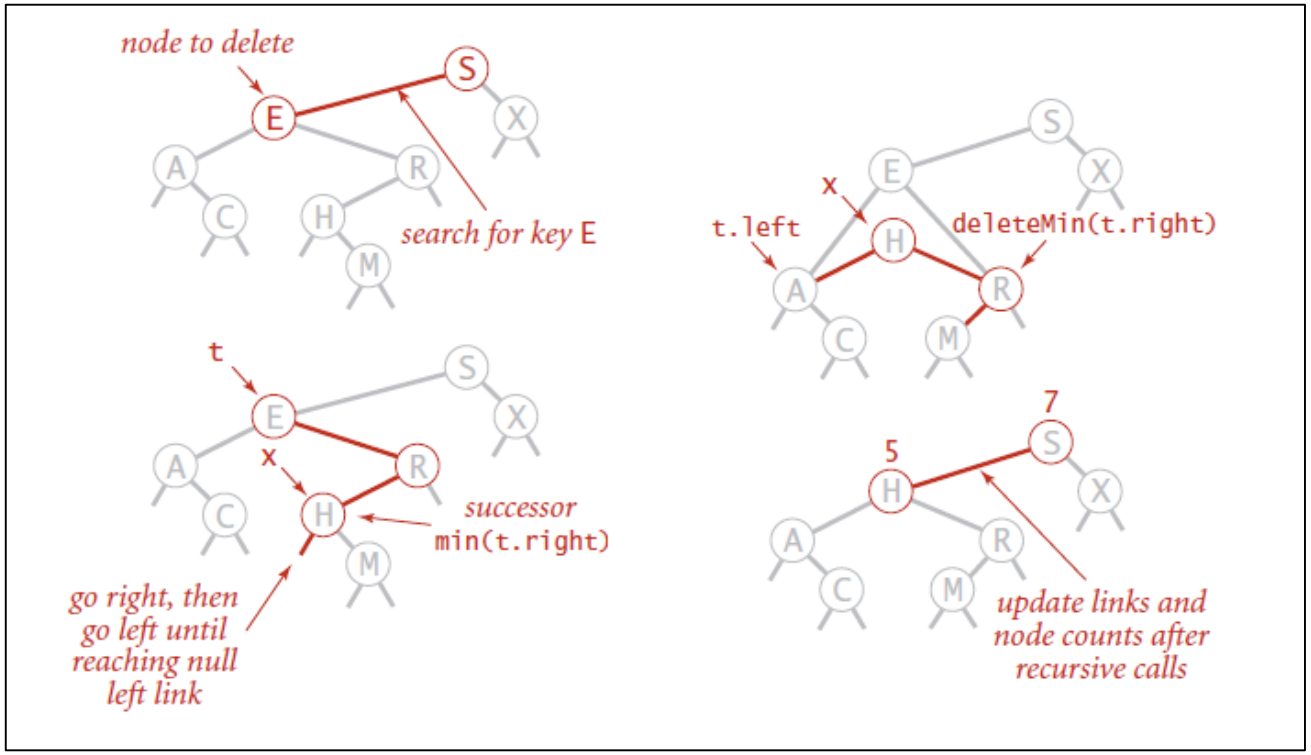
[SW-2018]

Drzewa wyszukiwań binarnych

Operacja `delete()`

Przypadek 3: Znaleziony wierzchołek *t* ma 2 potomków

- Znajdź następnik *x* wierzchołka *t* (następnik – wierzchołek o min kluczu w prawym poddrzewie)
- Usuń min w prawym poddrzewie wierzchołka t
- Wstaw *x* w miejsce wierzchołka *t*



[SW-2018]

Drzewa wyszukiwań binarnych

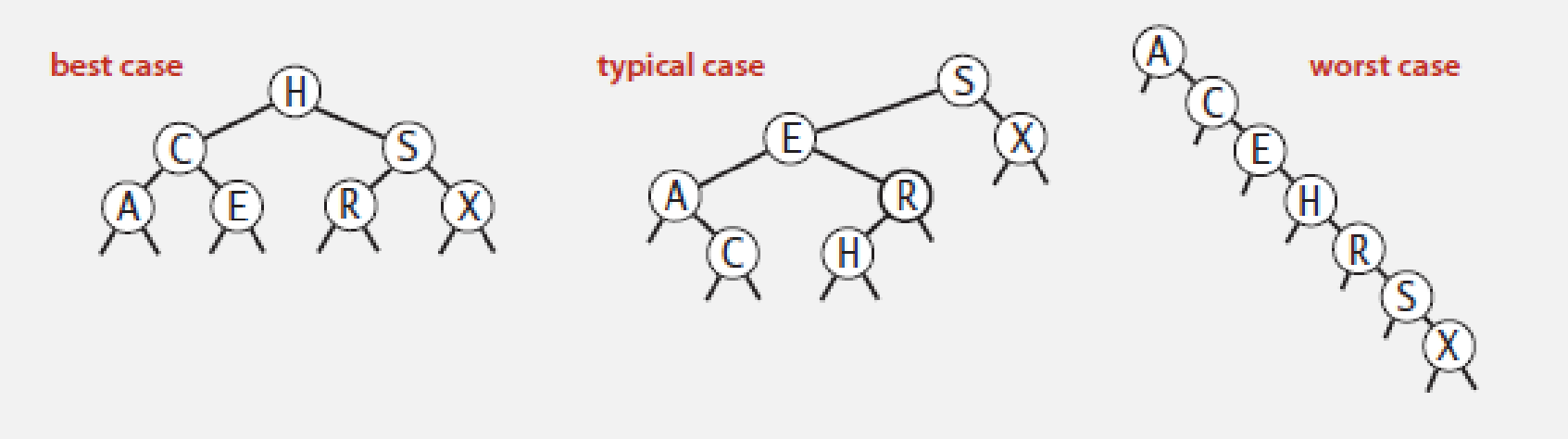
Operacja `delete()`

```
public void delete(Key key){
   root = delete(root, key);
}

private Node delete(Node x, Key key) {
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
      x.left = delete(x.left, key);
   else if (cmp > 0)
      x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   return x;
}
```

Drzewa wyszukiwań binarnych

Kształt drzewa

- Temu samemu zbiorowi kluczy odpowiada wiele drzew BST
- Kształt drzewa zależy od kolejności wstawiania elementów



[SW-2018]

- Liczba porównań dla operacji `get()/put()` wynosi 1+głębokość drzewa

Drzewa wyszukiwań binarnych

Kształt drzewa

Pomysł:

Budując drzewo wstawiamy elementy w porządku losowym ich kluczy.

Konsekwencje:

Wyszukiwanie/wstawienie w drzewie zbudowanym z *N* losowych kluczy wymaga ok. 1.39 log *N* porównań średnio.

Drzewa wyszukiwań binarnych

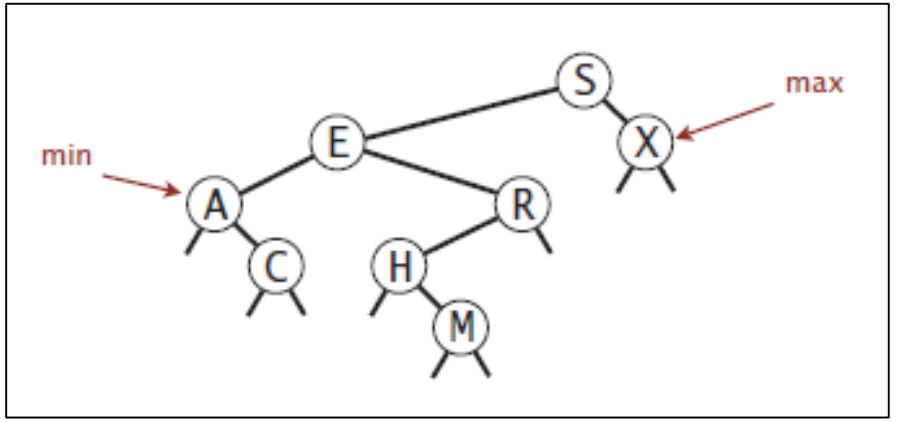
Implementacja pozostałych operacji

- `min(), max()`
- `floor(), ceiling()`
- `rank(), select(), size()`
- Iteracyjne przechodzenie po strukturze

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

- `min()` – najmniejszy klucz w tablicy
- `max()` – największy klucz w tablicy



[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

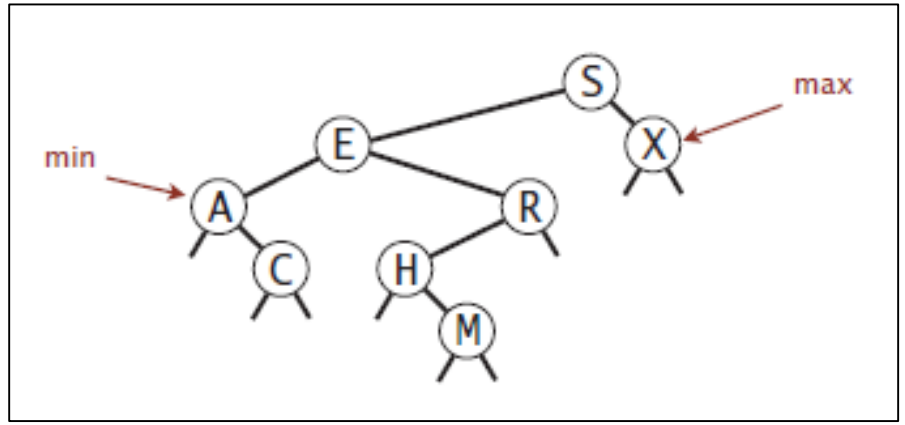
`min()` – najmniejszy klucz w tablicy

Przypadek 1: Referencja w korzeniu do lewego poddrzewa jest null

- Funkcja zwraca wartość klucza z korzenia

Przypadek 2: Referencja w korzeniu do lewego poddrzewa nie jest null

- Szukaj odpowiedzi w lewym poddrzewie



[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

Przykład

`min()`

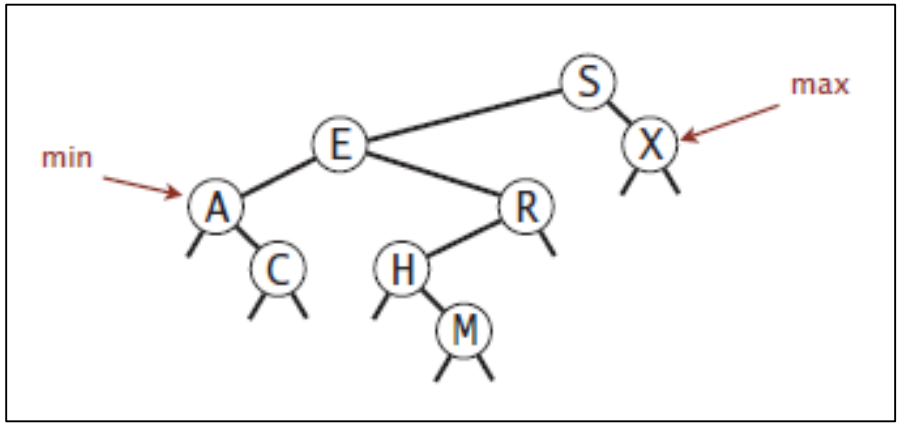
```
public Key min() {
   return min(root).key;


private Node min(Node x) {
   if (x.left == null)
      return x;
   else
      return min(x.left);
}
```

Przypadek 1: Referencja w korzeniu do lewego poddrzewa jest null

- Funkcja zwraca wartość klucza z korzenia

Przypadek 2: Referencja w korzeniu do lewego poddrzewa nie jest null

- Szukaj odpowiedzi w lewym poddrzewie



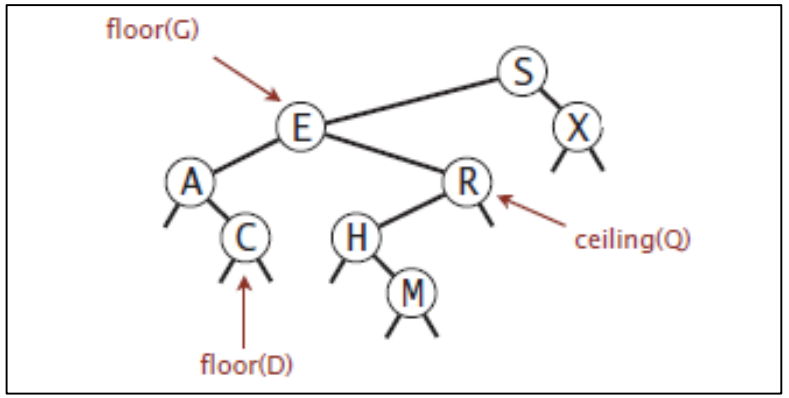
[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

- `floor()` – największy klucz mniejszy lub równy danemu kluczowi
- `ceiling()` – najmniejszy klucz większy lub równy danemu kluczowi

```
public Key floor(Key key)
{
   ???
}

public Key ceiling(Key key)
{
   ???
}
```



[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

- `rank()` – zwraca liczbę kluczy mniejszych lub równych *k*
- `select()` – zwraca klucz z pozycji (rank) *k*
- `size()` – zwraca liczbę wierzchołków

Jak efektywnie je zaimplementować?

Wskazówka: przypomnij sobie sposób implementacji operacji `size()` w strukturze kolejki czy stosu.

Drzewa wyszukiwań binarnych

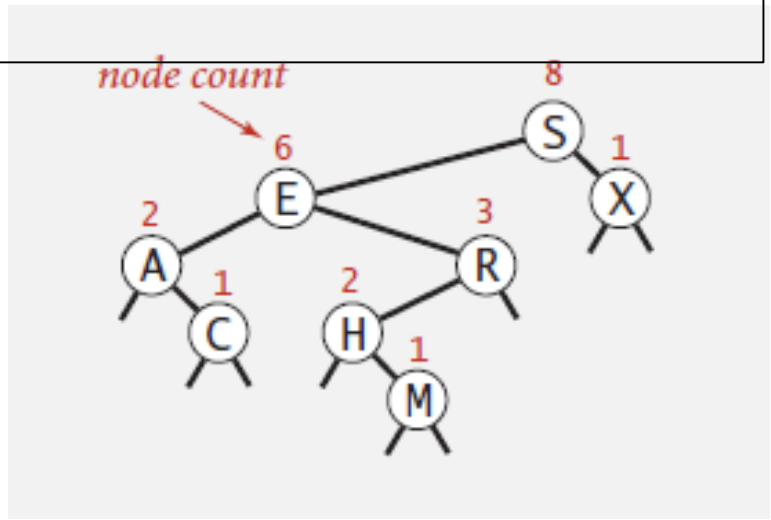
Implementacja pozostałych operacji

`rank(), select(), size()`

Jak efektywnie je zaimplementować?

W każdym wierzchołku pamiętamy liczbę wierzchołków w poddrzewie, którego korzeniem jest ten wierzchołek.

```
public class BST {
   private Node root;

   private class Node {
      private Key key;
      private Value val;
      private Node left, right;
      private int count;
   } ...
}
```



[SW-2018]

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

`rank()`, `select()`, `size()`

```
private Node put(Node x, Key key, Value
val)
{
   if (x == null)
      return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
     x.left = put(x.left, key, val);
   else if (cmp > 0)
     x.right = put(x.right, key, val);
   else
     x.val = val;
   x.count =
      1 + size(x.left) + size(x.right);
   return x;
}
```

Uwaga na operacje `put()` i `delete()`

w kontekście funkcji `size()`

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

Przykład

`rank()` – zwraca liczbę kluczy mniejszych lub równych *k*

```
public int rank(Key key) {
   return rank(key, root);
}

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if (cmp < 0)
      return rank(key, x.left);
   else if (cmp > 0)
      return 1 + size(x.left) +
         rank(key, x.right);
   else
      return size(x.left);
}
```

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

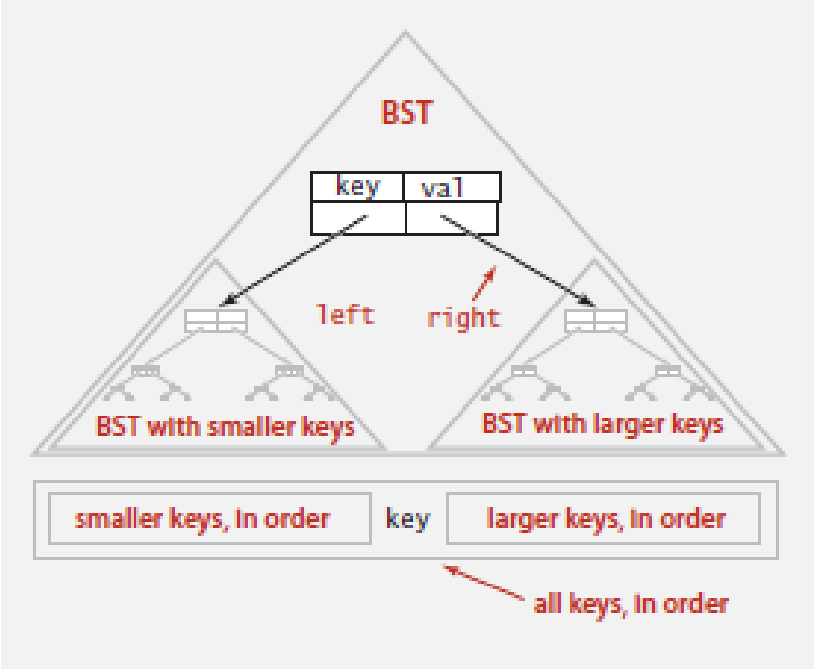
Iteracyjne przechodzenie po strukturze (porządek INORDER):

- Przejdź po lewym poddrzewie
- Zapisz klucz do kolejki
- Przejdź po prawym poddrzewie

Drzewa wyszukiwań binarnych

Implementacja pozostałych operacji

Iteracyjne przechodzenie po strukturze (porządek INORDER):



[SW-2018]

```
public Iterable<Key> keys()
{
   Queue<Key> q = new Queue<Key>();
   inorder(root, q);
   return q;
}
private void inorder(Node x,
Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

Przechodzenie w porządku INORDER ustawia klucze w porządku rosnącym.