Tölvunarfræði 2

Vika 8

Eiríkur Ernir Þorsteinsson

Háskóli Íslands

Vor 2017

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ

Yfirlit

Inngangur

Hrúgur

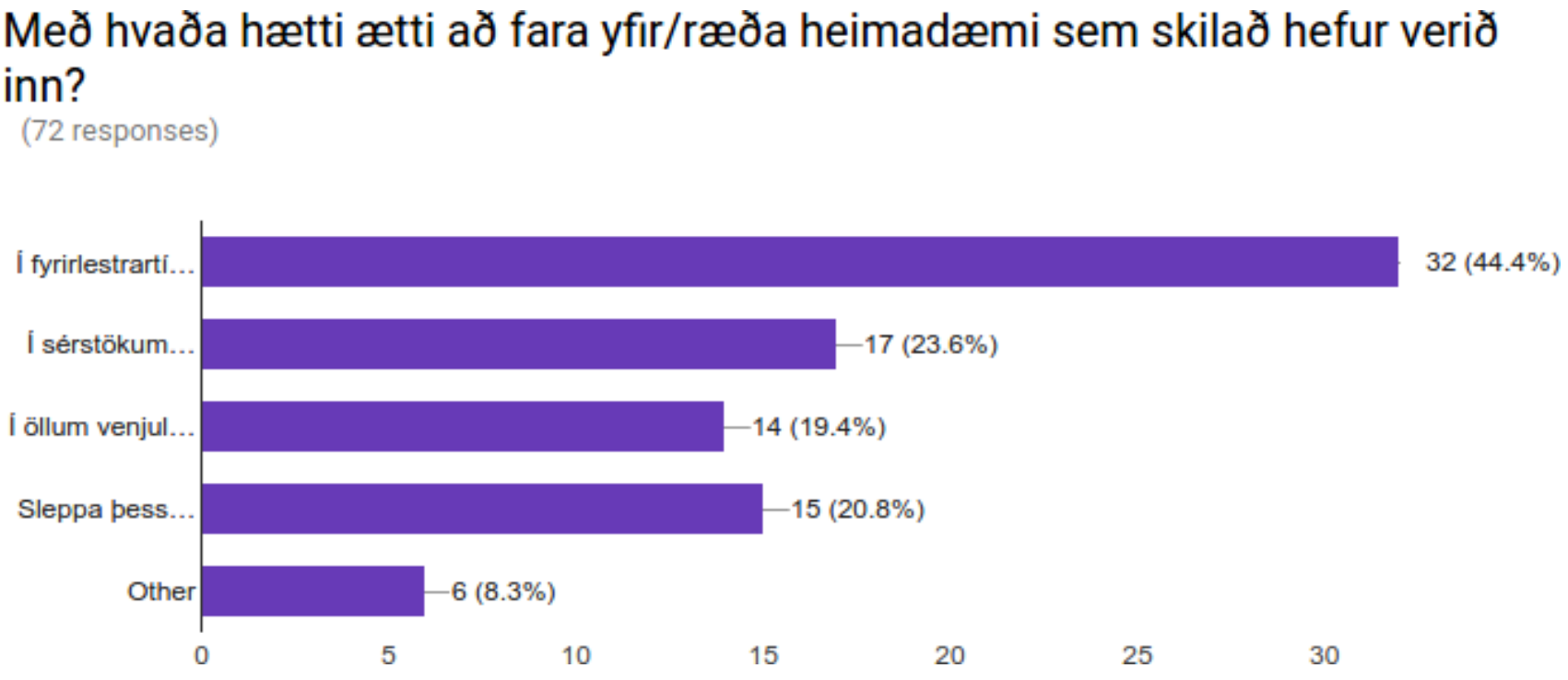
Minnisuppbygging tölva

Nafnatöflur

Helmingunarleit

Tvíleitartré

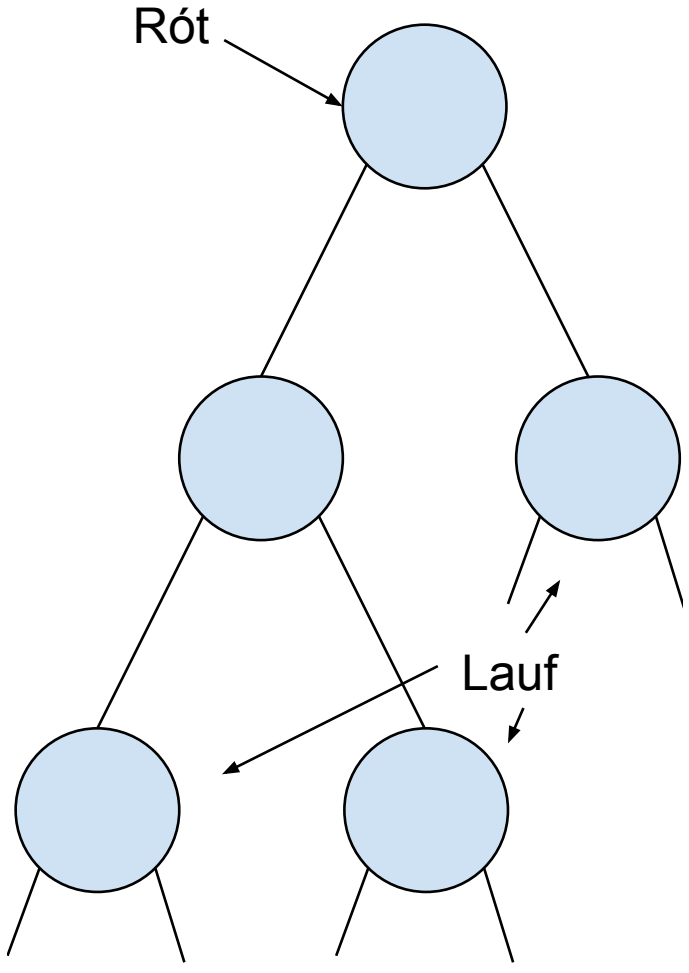
Skoðanakönnun úr síðasta tíma



Í samræmi við niðurstöður - ræðum skil 7 undir lok tímans

Upprifjun: Tré

- Algengt er í tölvunarfræði að raða gögnum upp í tré
- Hugmyndin eins og við munum nota hana:
  - Höfum hnúta, hver hnútur inniheldur vísun í tvö eða fleiri *börn*
    - Vísunin getur verið tóm
  - Getum kallað hnút sem er ekki barn neins annars *rót*, hnút sem á engin ekki-tóm börn *lauf*
- Sértilvik: tvíundartré (e. *binary tree*), þar sem barnafjöldinn er 2



Yfirlit

Inngangur

Hrúgur

Minnisuppbygging tölva

Nafnatöflur

Helmingunarleit

Tvíleitartré

Hrúgur

Hrúga (e. *heap*) er tvíundartré sem uppfyllir hrúguskilyrði (e. *heap property*). Við notum fylki til að geyma hrúgur.
Skoðum glærur 16-17 í PriorityQueues.

Binary heap representations

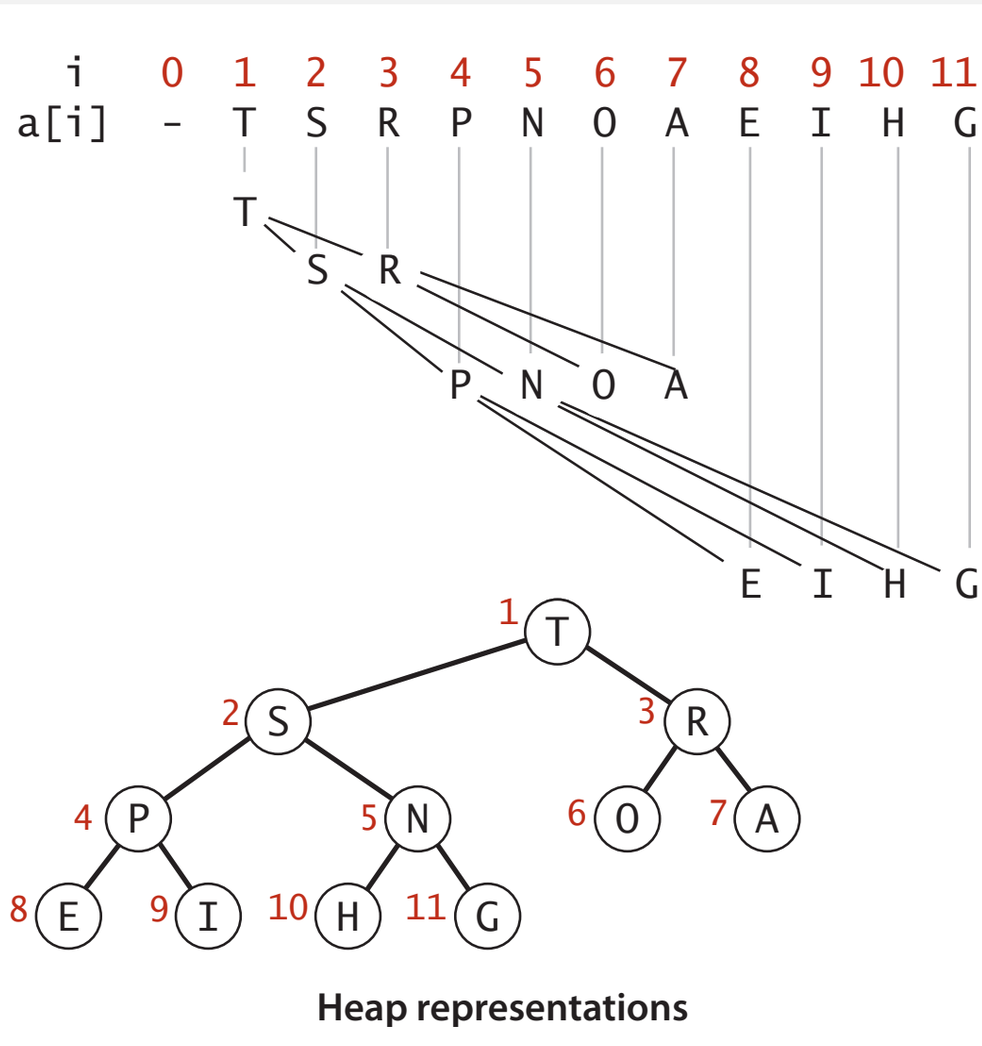
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no smaller than children's keys.

Array representation.

- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!



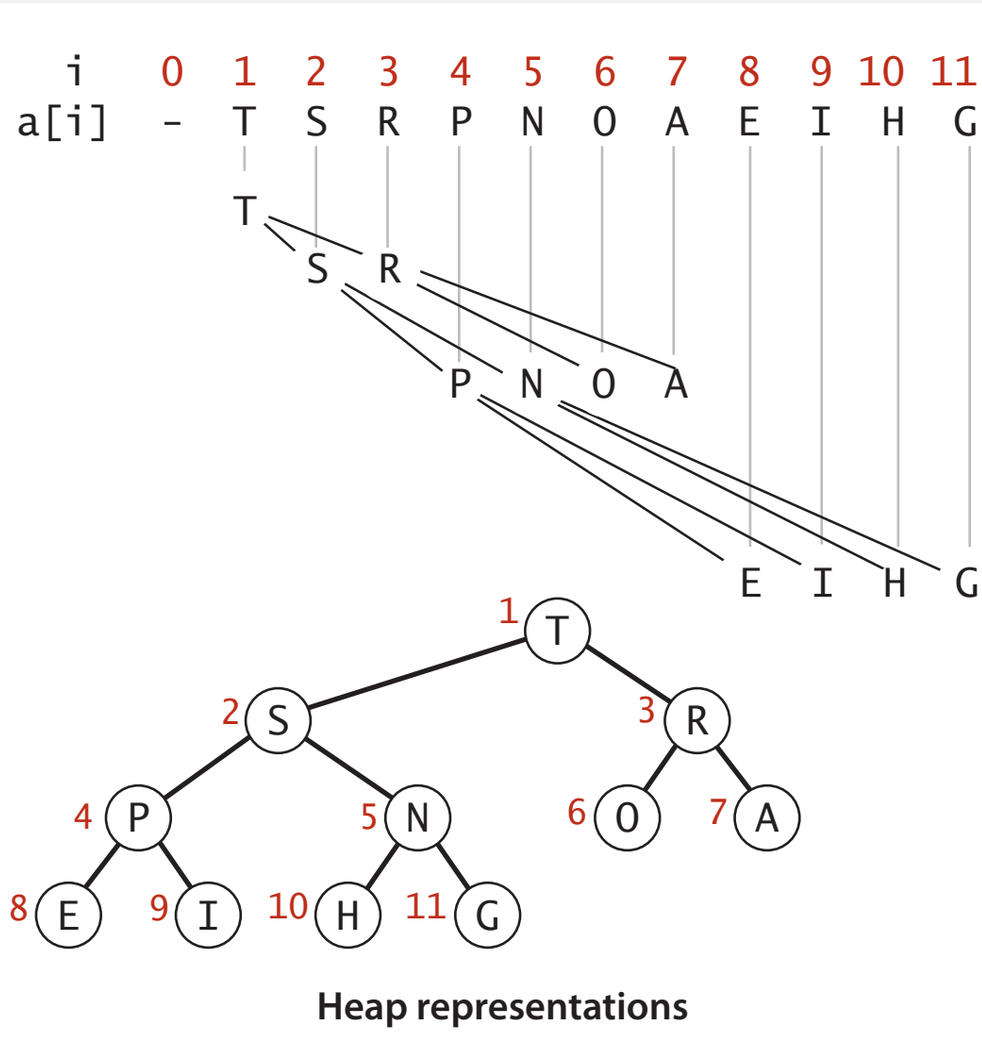
Heap representations

Binary heap properties

Proposition. Largest key is `a[1]`, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at `k` is at `k/2`.
- Children of node at `k` are at `2k` and `2k+1`.



Heap representations

Aðgerðir

- Notkun hrúgu til að útfæra forgangsbiðröð krefst tveggja aðgerða
- Innsetning
  - Setjum stakið aftast og látum það synda upp
- Eyðing stærsta staks
  - Skiptum á rót og síðasta stakinu í fylkinu, sökkvum því sem við færðum upp
- Skoðum glærur 20-23 í PriorityQueues.

Promotion in a heap

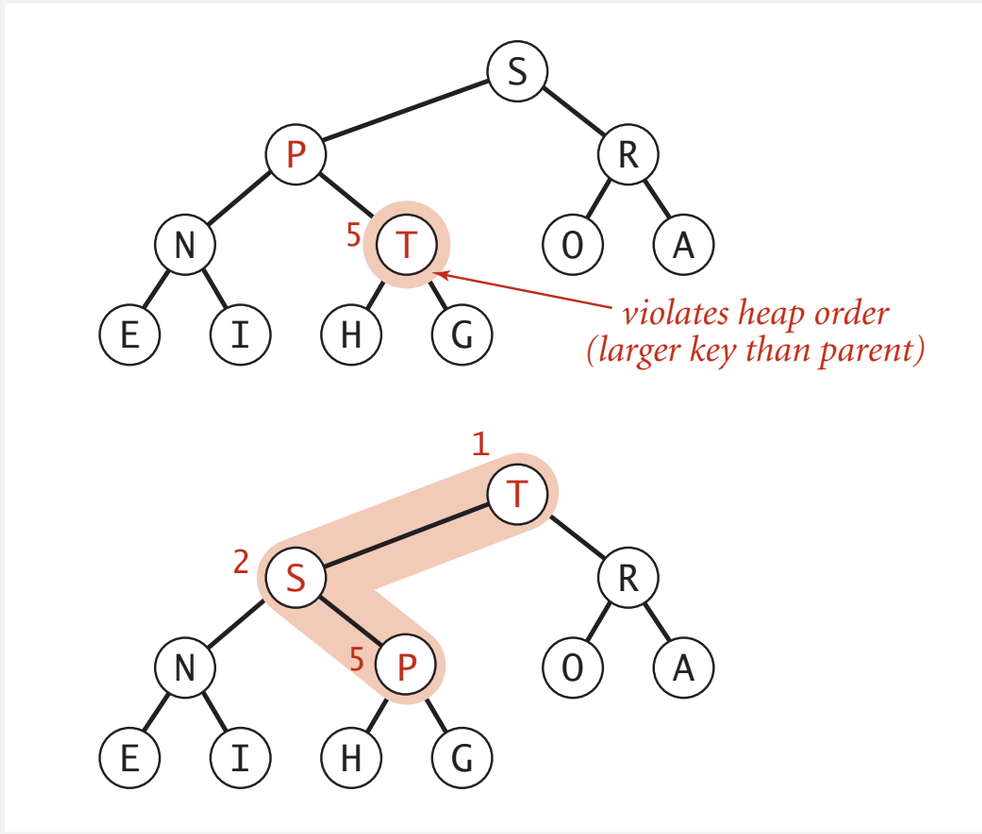
Scenario. Child's key becomes larger key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2



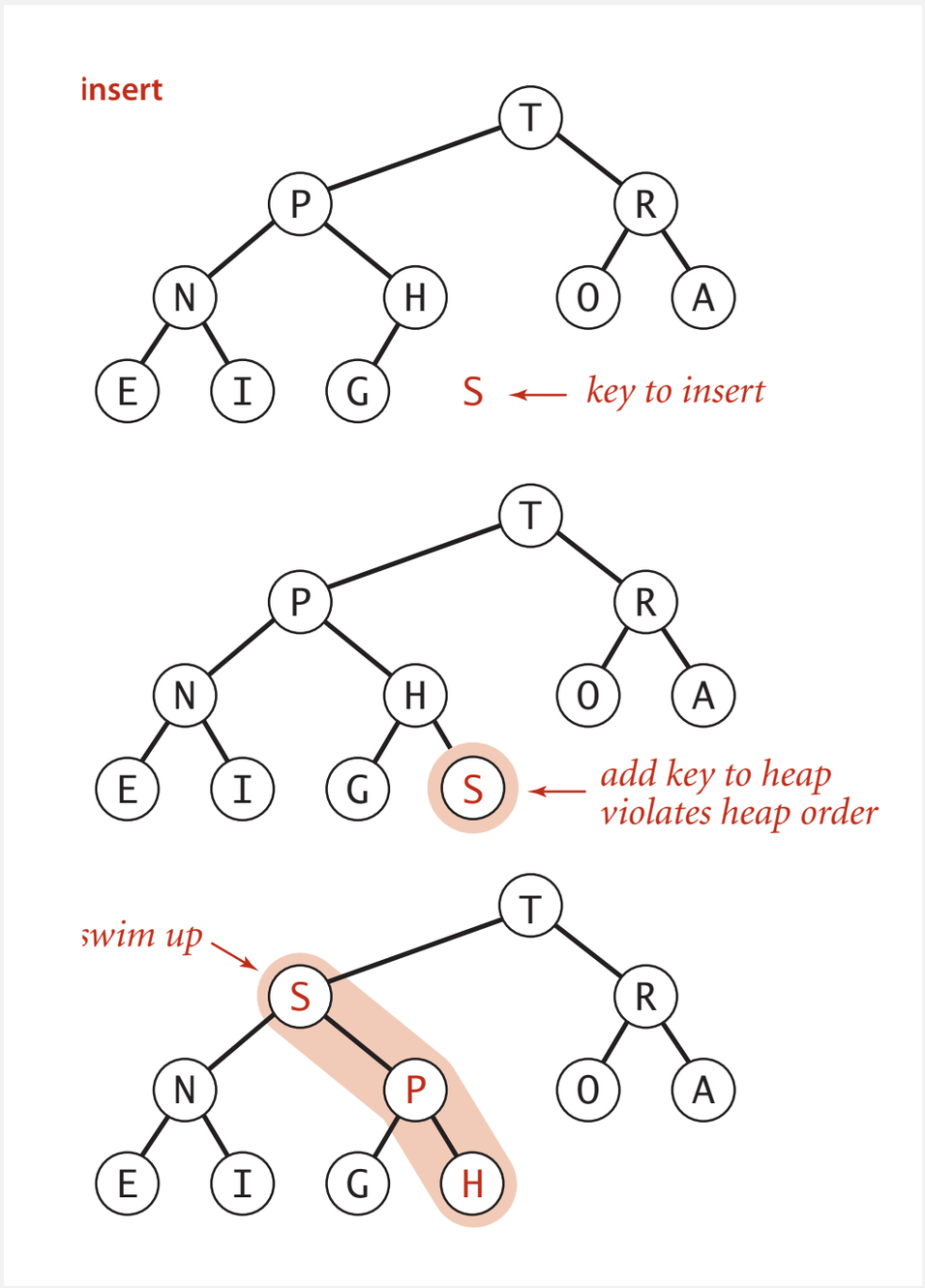
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.

Cost. At most $1 + \lg N$ compares.

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```



Demotion in a heap

Scenario. Parent's key becomes smaller than one (or both) of its children's.

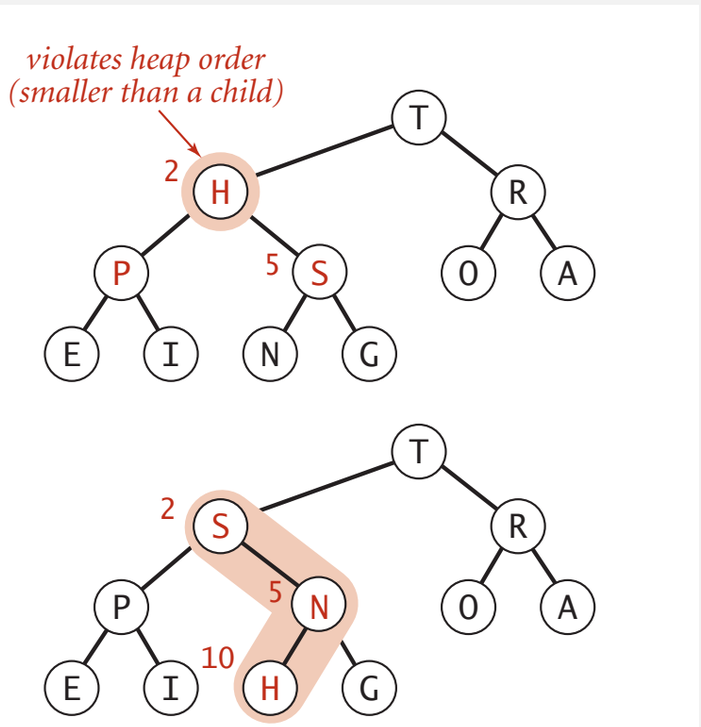
To eliminate the violation:

why not smaller child?

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node at k are 2k and 2k+1



Top-down reheapify (sink)

Power struggle. Better subordinate promoted.

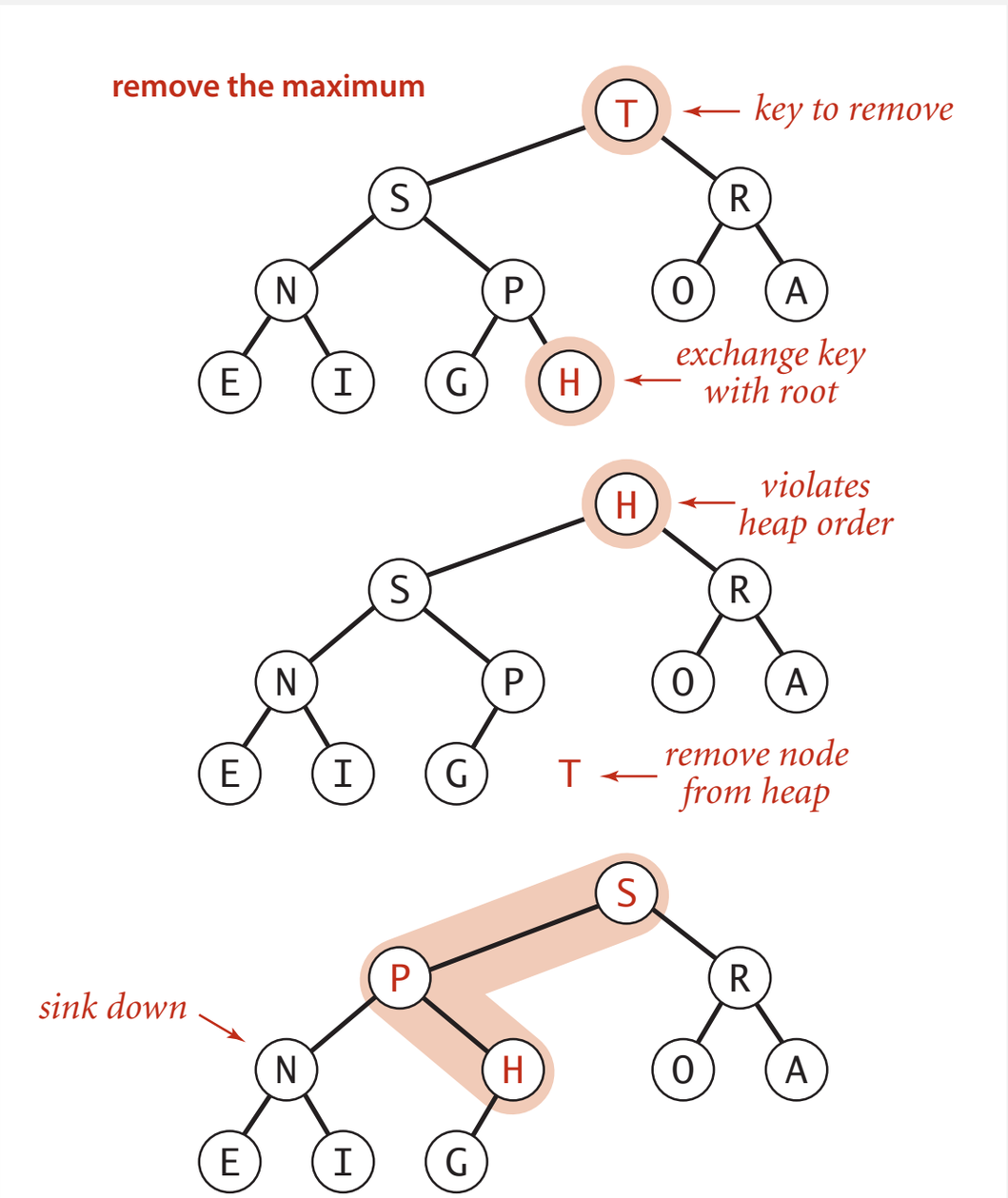
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

prevent loitering



Tími

- Athugum - um fullskipuð tré er að ræða
- Innsetning og eyðing felur í sér færslu á milli hæða í trénu
- Fjöldi aðgerða takmarkast af hæð trésins
  - Hæð fullskipaðs trés með N hnútum er $\lfloor \log_2 n \rfloor$
- Praktísk vandræði við útfærsluna eins og henni hefur verið lýst er að langt er á milli staka, hentar illa fyrir minnisuppbyggingu raun

Yfirlit

Inngangur

Hrúgur

Minnisuppbygging tölva

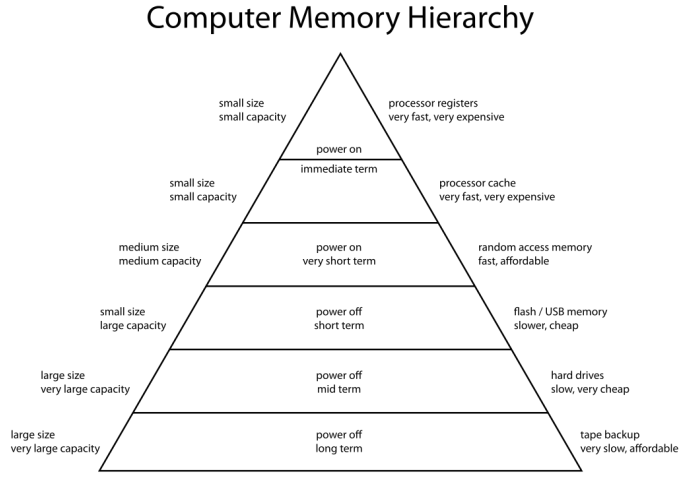
Nafnatöflur

Helmingunarleit

Tvíleitartré

Minnisuppbygging

- Sumir geymslustaðir fyrir gögn eru hraðvirkari en aðrir
- Gróf uppbygging í nútímatölvu:
  1. Minni örgjörvans (e. *internal memory*) - gisti (e. *registers*) og skyndiminni (e. *cache*)
  2. Aðalminni (e. *main memory*) - RAM
  3. Aukageymsla (e. *secondary storage*) - SSD diskar/harðir diskar
  4. Útvær geymsla (e. *off-line storage*) - geymsla utan stjórnar örgjörvans



Mynd af Wikipedia

Yfirlit

Inngangur

Hrúgur

Minnisuppbygging tölva

Nafnatöflur

Helmingunarleit

Tvíleitartré

Nafnatöflur

- Nafnatafla (e. *symbol table*) er mjög almenn hugræn gagnagrind
- Snýst um að tengja saman lykla (e. *keys*) og gildi (e. *values*)
  - "Hvert er gildið fyrir þennan lykil?"
- Höfum þegar séð útfærslu á nafnatöflu - `std::map` í C++

DNS tafla

Lykill	Gildi
mbl.is	92.43.192.110
visir.is	82.221.81.10
hi.is	130.208.165.207
ru.is	52.48.55.82

API

Möguleg skil fyrir nafnatöflu:

public class ST<Key, Value>		
-	`ST()`	Smiður, býr til tóma nafnatöflu
`void`	`void put(Key k, Value v)`	Setja lykil-gildis par í töfluna
`Value`	`get(Key key)`	Skilar gildinu sem svarar til `key`

Auk þess mætti skilgreina `delete(Key key)`, `contains(Key key)`, `size()` og `isEmpty()` aðferðir

Útfærsla á nafnatöflu

- Við gætum sett gögnin okkar í óraðað fylki eða eintengdan lista
  - Sjá: SequentialSearchST.java
  - Verður tímafrekt fyrir stórar töflur
- Getum notað öflugri aðferðir ef við getum gert ráð fyrir meiru um lyklana
  - Viljum hafa lykla sem eru samanburðarhæfir (sjá `Comparable`)
  - Seinna: Lykla sem hægt er að taka af `hashCode`
- Almennt ráðlagt: Nota óbreytanlega (e. *immutable*) lykla
  - Gott: `Integer`, `String`, `Double`,...
  - Slæmt: T.d. fylki

Yfirlit

Inngangur

Hrúgur

Minnisuppbygging tölva

Nafnatöflur

Helmingunarleit

Tvíleitartré

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ

Helmingunarleit

- Hingað til höfum við notað línulega leit til að finna staðsetningar í fylkjum og listum
- Getum gert betur í röðuðum fylkjum, hugmyndin er:
  - Berum "miðju" leitarbilsins saman við leitarstakið
  - Sé miðjan stærri en leitarstakið vitum við að stakið er ekki að finna í stærri helming leitarbilsins (og öfugt)
  - Notum helmingunarleit endurkvæmt á þann helming leitarbilsins sem eftir er
- Leitaraðferðin er kölluð helmingunarleit (e. *binary search*) því hún helmingar stærð leitarbilsins í hverri ítrun
- Grundvallaraðferðafræði sem kemur víða við!
- Skoðum BinarySearchST.java vandlega

Endurkvæm helmingunarleit

Möguleg endurkvæm útfærsla á helmingunarleit:

```
public int rank(Key key, int lo, int hi)
{
   if (hi < lo) return lo;
   int mid = lo + (hi - lo) / 2;
   int cmp = key.compareTo(keys[mid]);
   if      (cmp < 0)
        return rank(key, lo, mid-1);
   else if (cmp > 0)
        return rank(key, mid+1, hi);
   else return mid;
}
```

Sjá bls. 380 í Algorithms.

Tími

- Getum lýst fjölda samanburða í helmingunarleit í N staka fylki með rakningarvenslum:
  - $C(0) = 0,\ C(1) = 1\ ,\ C(N) = C(\lfloor N \rfloor) + 1$
  - Fáum $C(N) \sim \log N$
- Útfærslan á nafnatöflunni í `BinarySearchST.java` leyfir skilvirkar uppflettingar (logratími)
- Innsetningar og eyðingar krefjast hins vegar enn línulegs tíma

Yfirlit

Inngangur

Hrúgur

Minnisuppbygging tölva

Nafnatöflur

Helmingunarleit

Tvíleitartré

Tvíleitartré

- Við getum auðveldað helmingunarleit með því að geyma gögnin í tvíundartré
  - Í þetta skiptið - táknað með hnútum og bendum, líkt og við höfum gert þegar við höfum útfært eintengda lista
- Til að útfæra nafnatöflu látum við hnúta innihalda samanburðarhæfan lykil, notum uppbyggingu trésins til að auðvelda leitina
  - Látum vinstra barn hvers hnúts hafa minni lykil en hnúturinn, hægra barnið stærri lykil
- Köllum þessi tvíundartré tvíleitartré (e. *binary search tree*)
- Skoðum nafnatöflu útfærða með tvíleitartré, BST.java

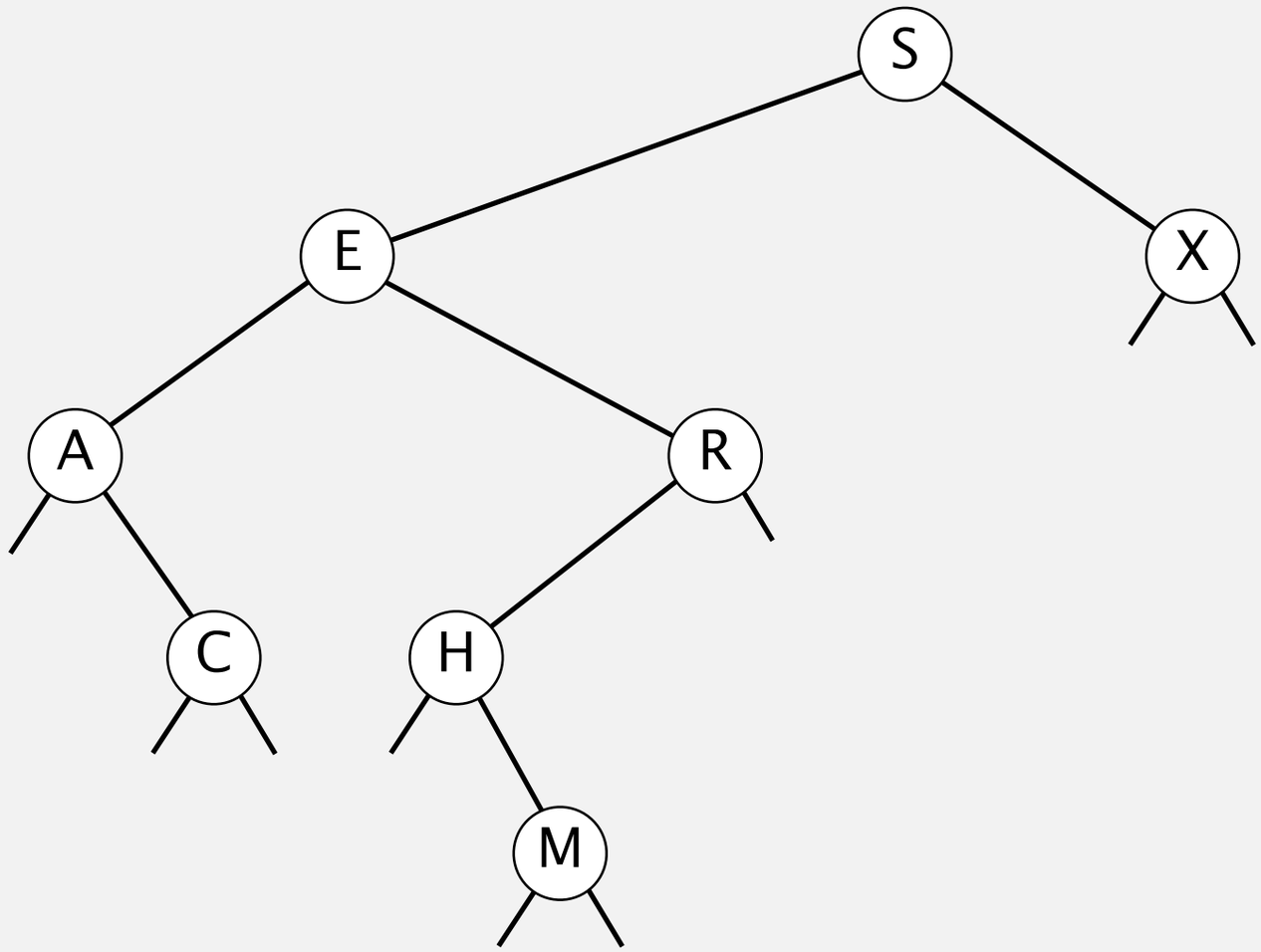
Aðgerðir á tvíleitartré

- Til að uppfylla skil fyrir nafnatöflu þurfum við að geta:
  - Leitað eftir lykli (get)
    - Berum leitarlykilinn saman við lykil hvers hnúts
    - Förum í vinstra eða hægra undirtré eftir hvern samanburð
    - Hættum þegar lykillinn finnst eða þegar við finnum `null`
  - Innsetning (put)
    - Svipað og leit, en þegar við finnum `null` setjum við stakið inn
- Sjá glærur 4-10 í BinarySearchTrees
- Eyðing er kafli út af fyrir sig (sjá glærur 31-34 í BinarySearchTrees)

Binary search tree demo

Search. If less, go left; if greater, go right; if equal, search hit.

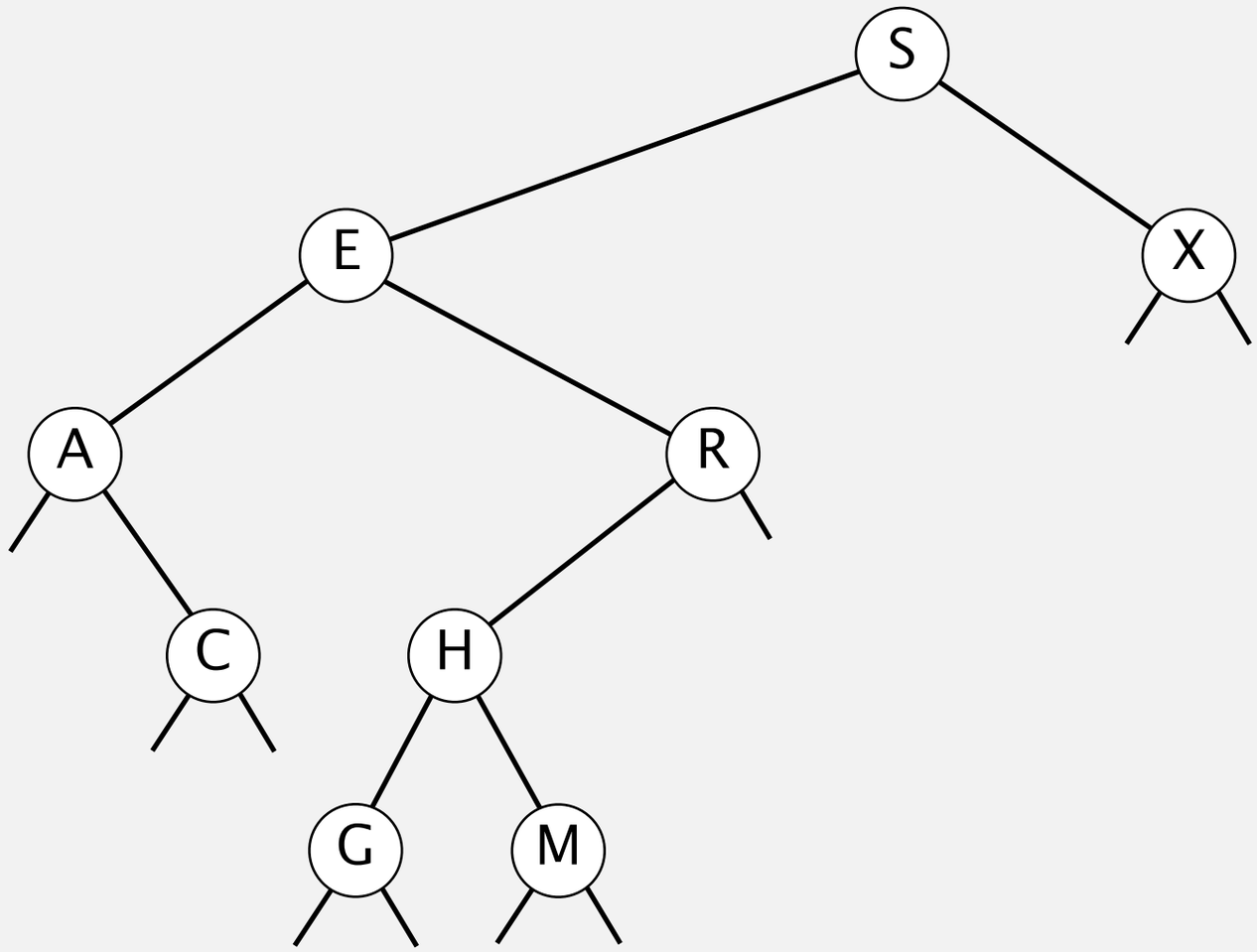
successful search for H



Binary search tree demo

Insert. If less, go left; if greater, go right; if null, insert.

insert G



BST representation in Java

Java definition. A BST is a reference to a root `Node`.

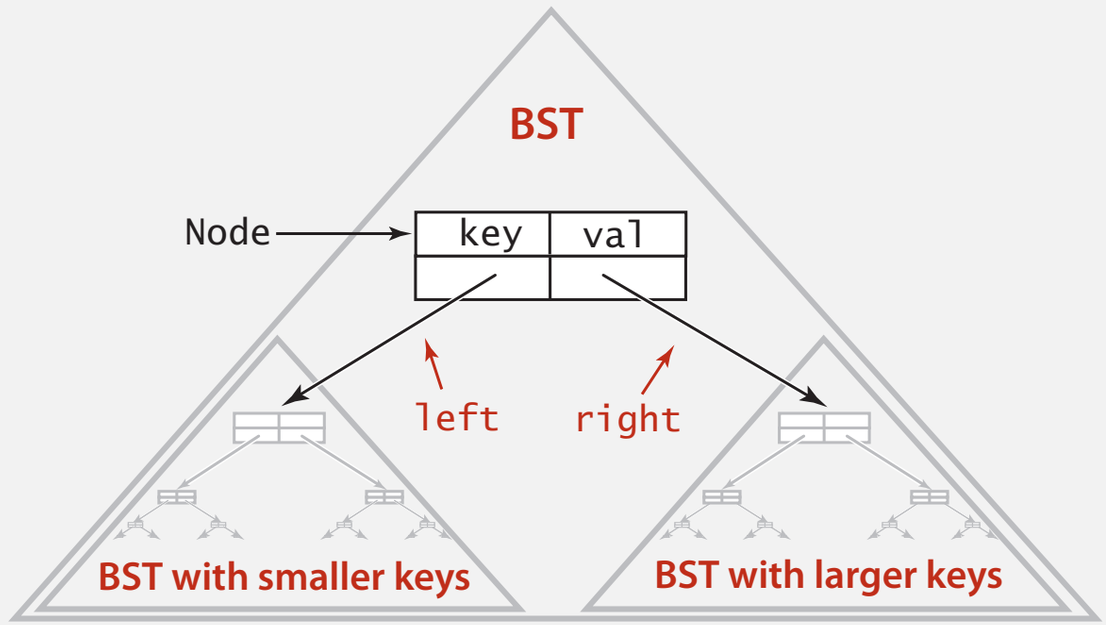
A `Node` is composed of four fields:

- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys larger keys

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;
   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

Key and Value are generic types; Key is Comparable



Binary search tree

BST implementation (skeleton)

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

   private class Node
   {  /* see previous slide */  }

   public void put(Key key, Value val)
   {  /* see next slides */  }

   public Value get(Key key)
   {  /* see next slides */  }

   public void delete(Key key)
   {  /* see next slides */  }

   public Iterable<Key> iterator()
   {  /* see next slides */  }

}
```

root of BST

BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
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;
}
```

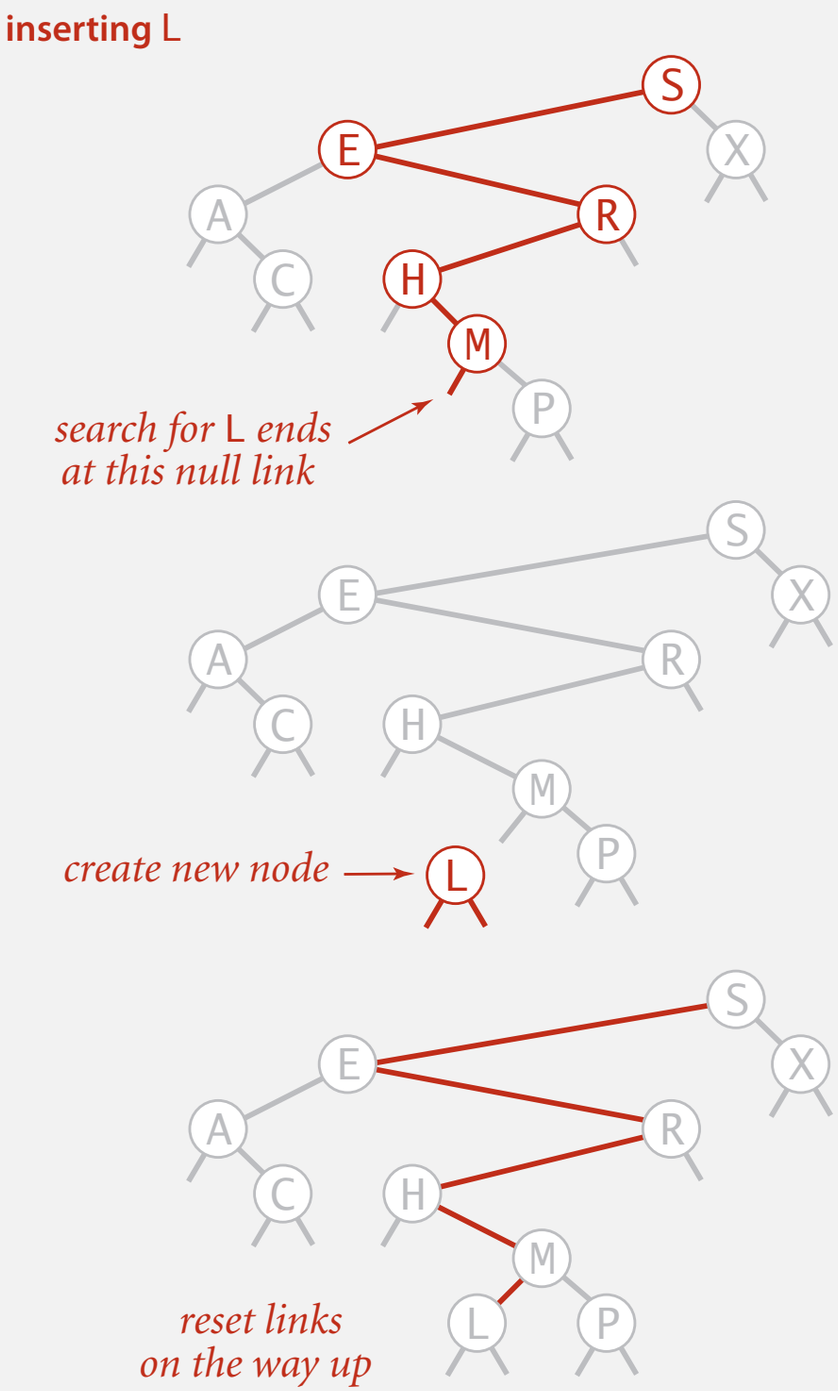
Cost. Number of compares is equal to 1 + depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.



Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

```
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 if (cmp == 0)
      x.val = val;
   return x;
}
```

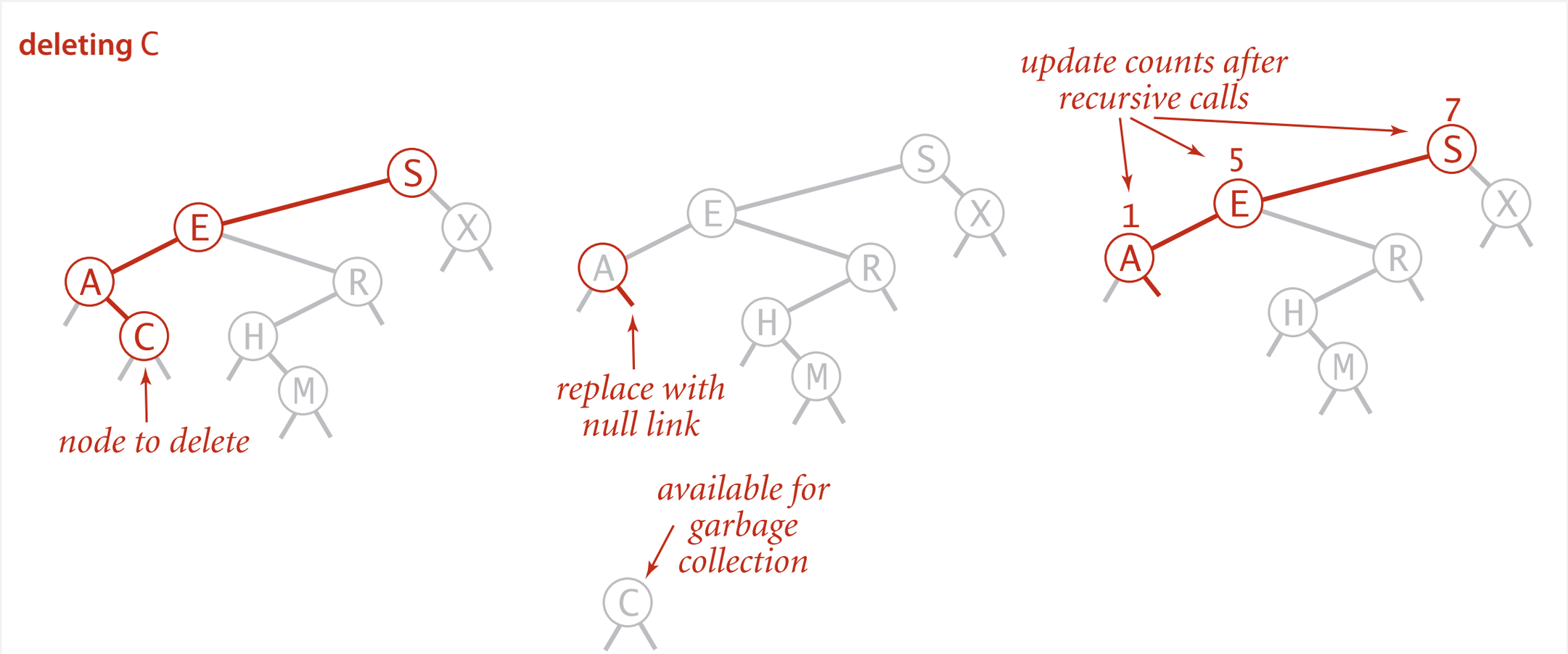
concise, but tricky, recursive code; read carefully!

Cost. Number of compares is equal to 1 + depth of node.

Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

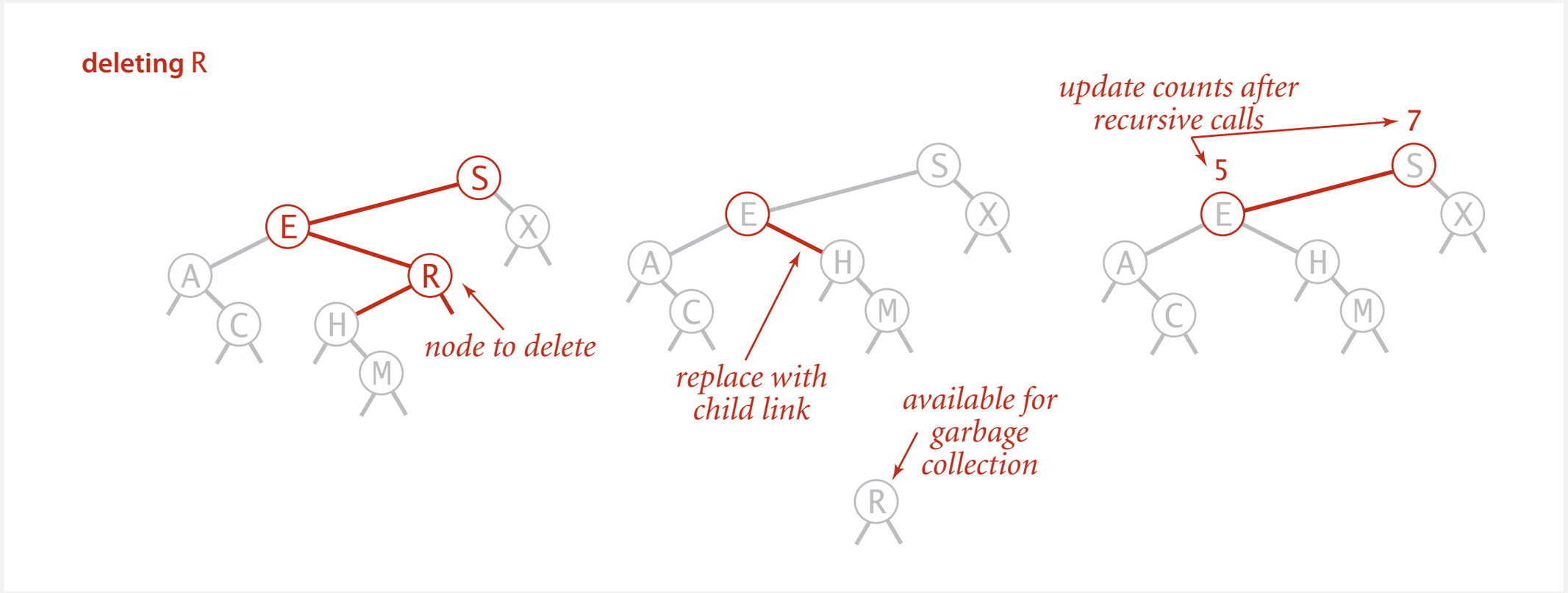
Case 0. [0 children] Delete `t` by setting parent link to null.



Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 1. [1 child] Delete `t` by replacing parent link.

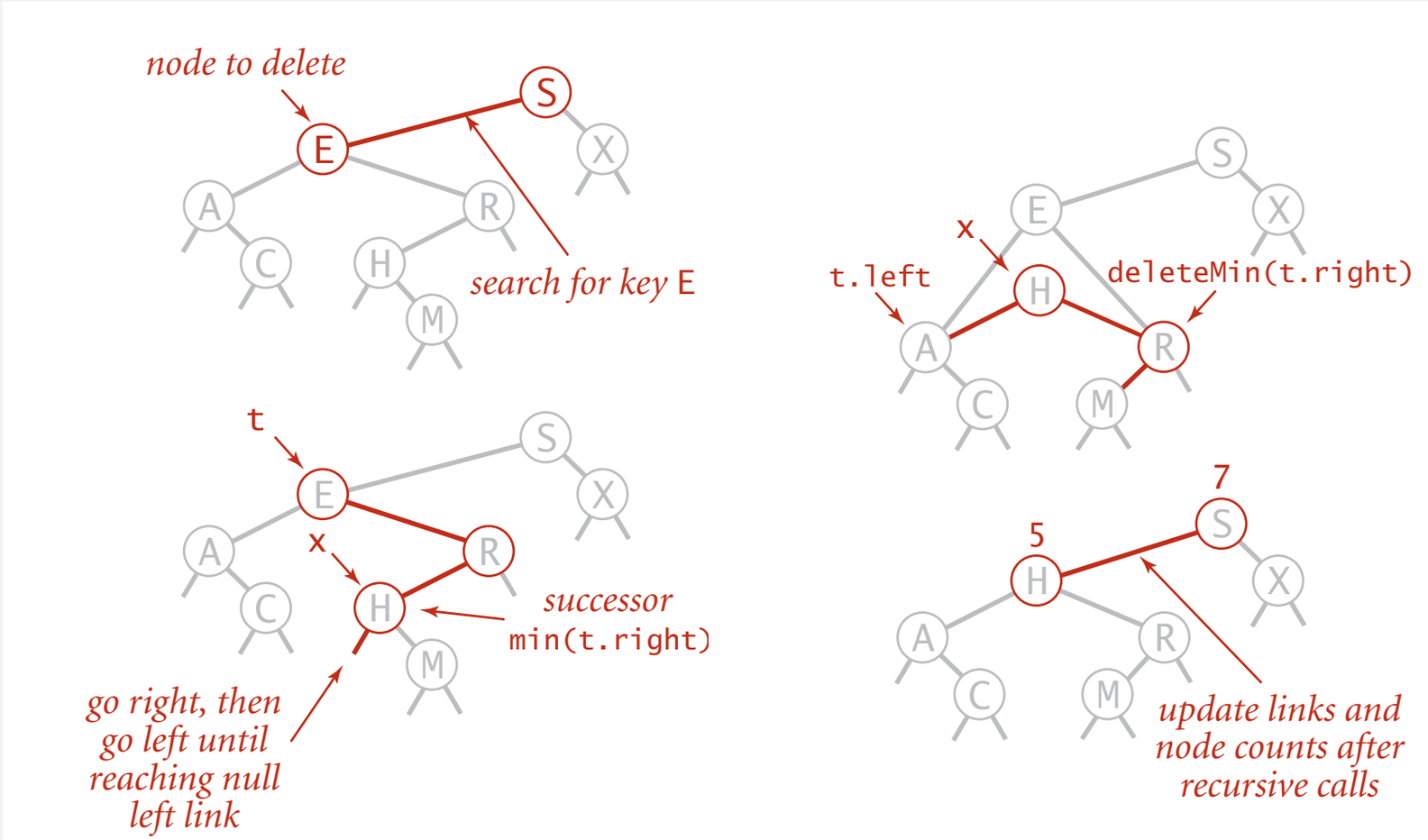


Hibbard deletion

To delete a node with key `k`: search for node `t` containing key `k`.

Case 2. [2 children]

- Find successor `x` of `t`. ← x has no left child
- Delete the minimum in `t`'s right subtree. ← but don't garbage collect x
- Put `x` in `t`'s spot. ← still a BST



Hibbard deletion: Java implementation

```
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;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

no left child

replace with successor

update subtree counts

Tengill á fyrirlestraræfingu:
https://goo.gl/forms/xgWGlDlmEvGlbcu93

Kóða fyrir algs4 reiknirit má finna á
http://algs4.cs.princeton.edu/code/

Næst

Meira um tvíleitartré, hakkatöflur.