Heapar och prioritetsköer

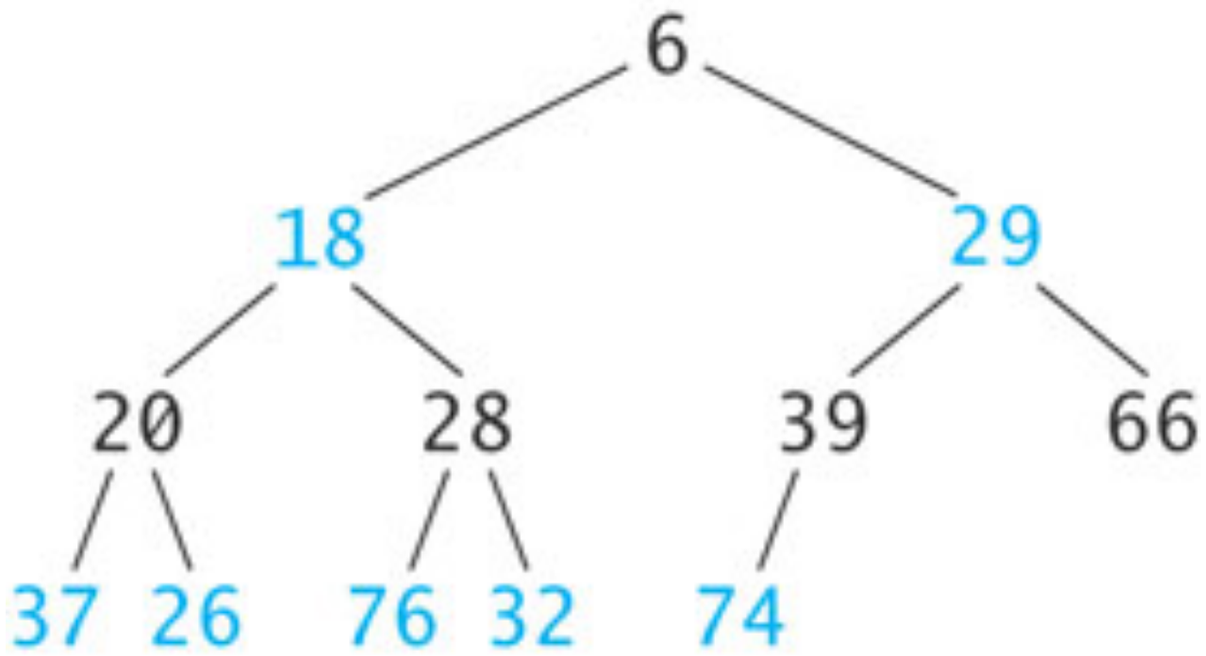
Koffman & Wolfgang

- kapitel 6, avsnitt 5–6

Heapar

En heap är ett *nivåbalanserat* (complete) binärt träd med följande egenskaper (eller invariant):

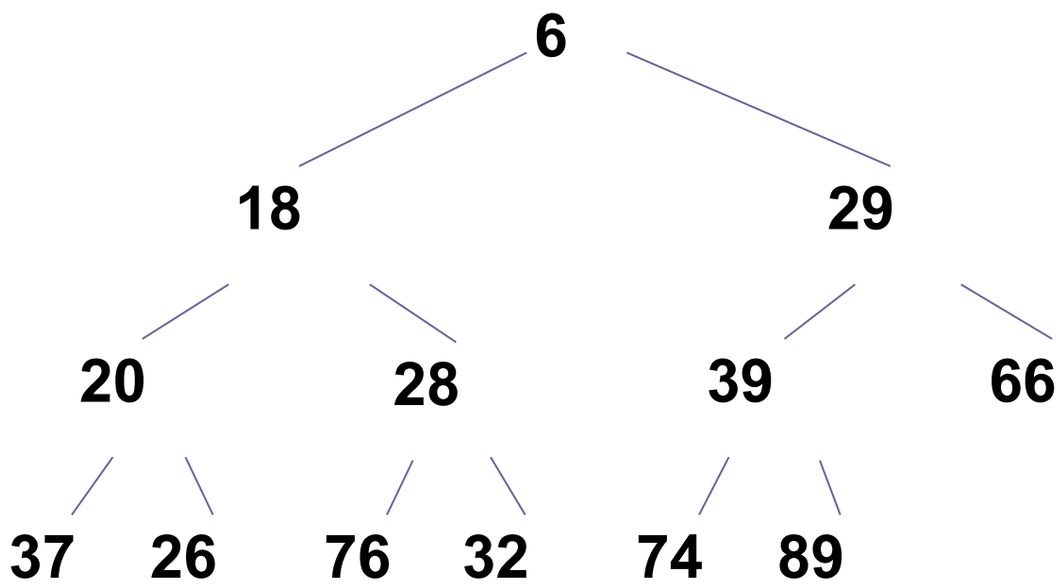
- roten innehåller det minsta elementet i trädet
- varje icketomt underträd är en heap



Att lägga till element i en heap

Algoritm för att lägga till ett nytt element:

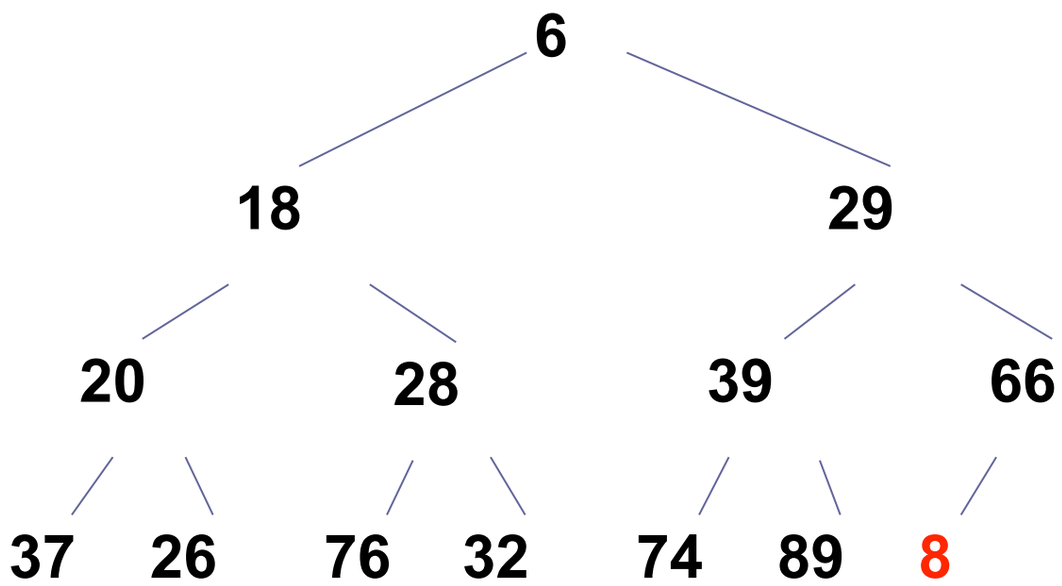
- lägg till det nya elementet i nästa tomma position längst ner i heapen
- så länge som det nya elementet är mindre än sin förälder:
  - flytta upp elementet genom att byta ut det mot sin förälder



Att lägga till element i en heap

Algoritm för att lägga till ett nytt element:

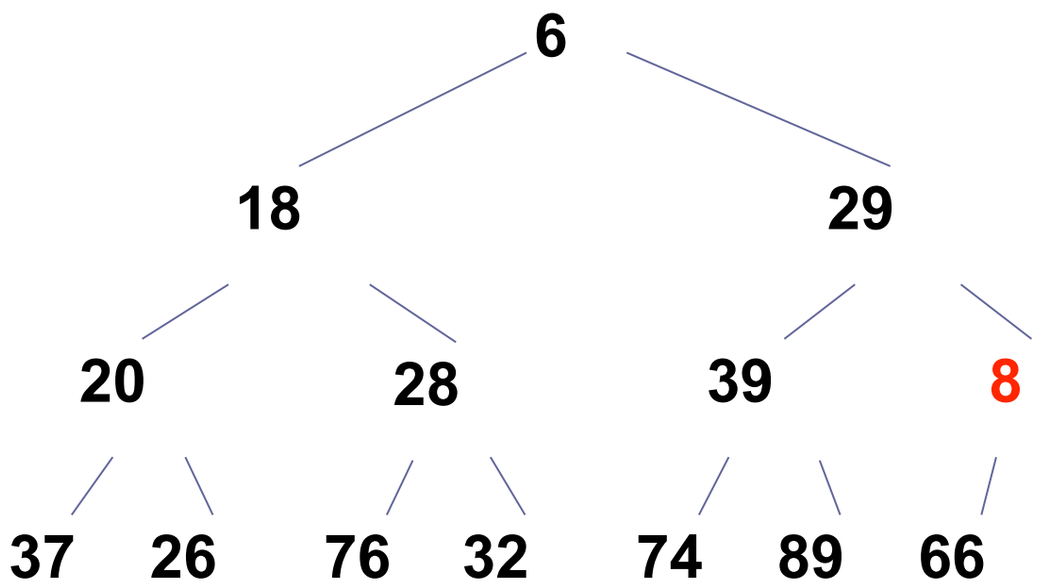
- lägg till det nya elementet i nästa tomma position längst ner i heapen
- så länge som det nya elementet är mindre än sin förälder:
  - flytta upp elementet genom att byta ut det mot sin förälder



Att lägga till element i en heap

Algoritm för att lägga till ett nytt element:

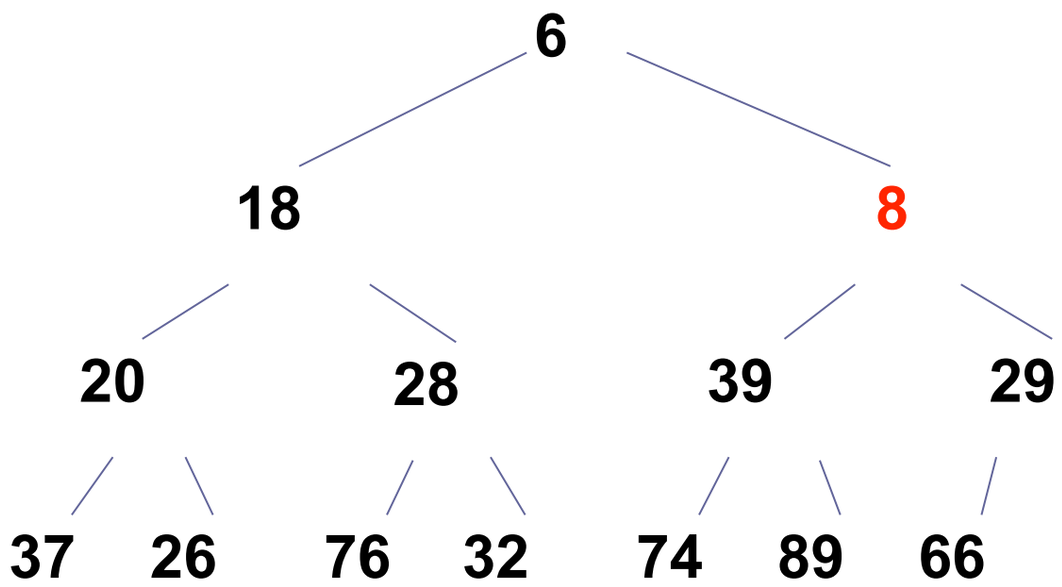
- lägg till det nya elementet i nästa tomma position längst ner i heapen
- så länge som det nya elementet är mindre än sin förälder:
  - flytta upp elementet genom att byta ut det mot sin förälder



Att lägga till element i en heap

Algoritm för att lägga till ett nytt element:

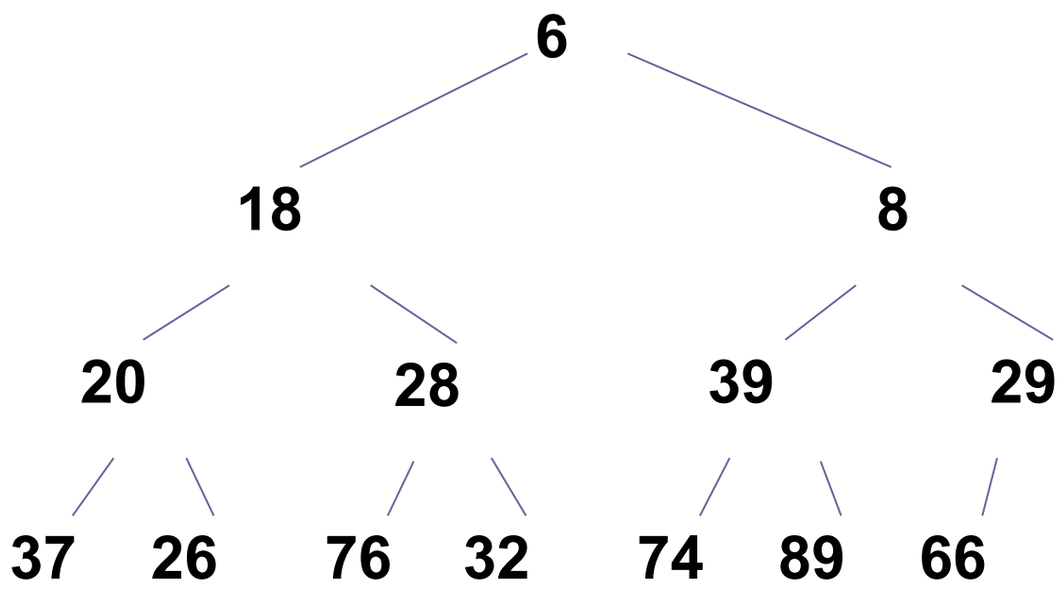
- lägg till det nya elementet i nästa tomma position längst ner i heapen
- så länge som det nya elementet är mindre än sin förälder:
  - flytta upp elementet genom att byta ut det mot sin förälder



Att lägga till element i en heap

Algoritm för att lägga till ett nytt element:

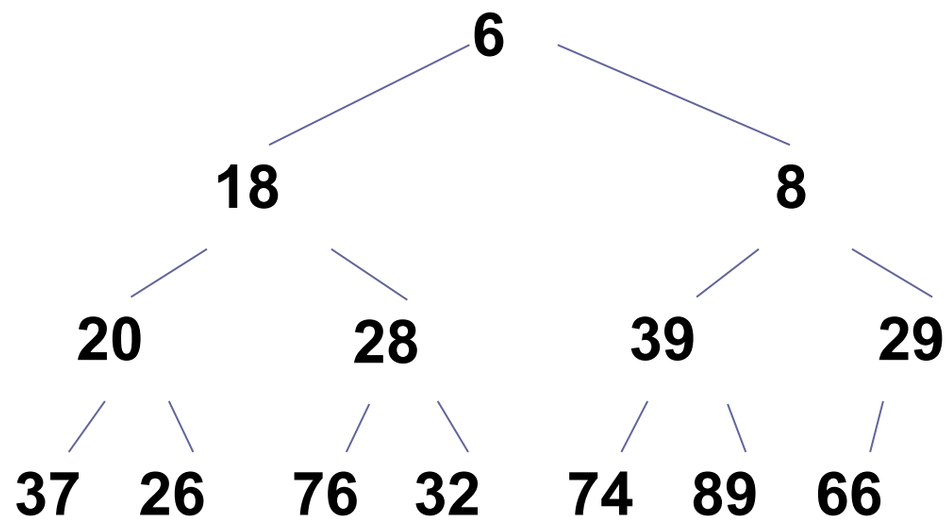
- lägg till det nya elementet i nästa tomma position längst ner i heapen
- så länge som det nya elementet är mindre än sin förälder:
  - flytta upp elementet genom att byta ut det mot sin förälder



Ta bort det minsta elementet

Algoritm för att ta bort rotnoden från en heap:

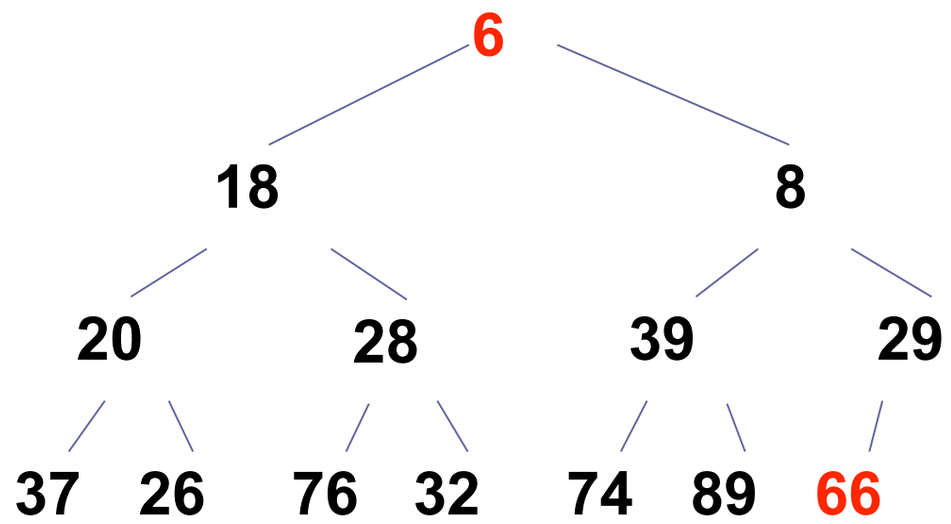
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Ta bort det minsta elementet

Algoritm för att ta bort rotnoden från en heap:

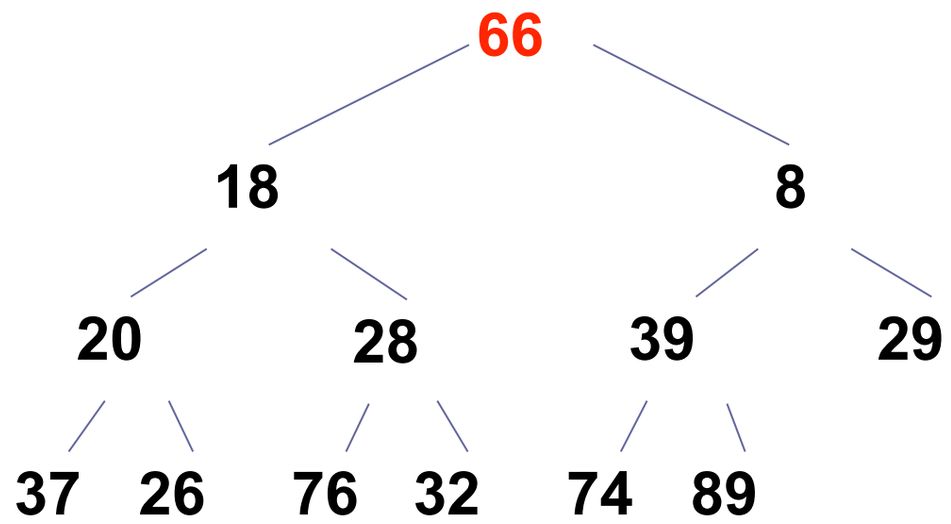
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Ta bort det minsta elementet

Algoritm för att ta bort rotnoden från en heap:

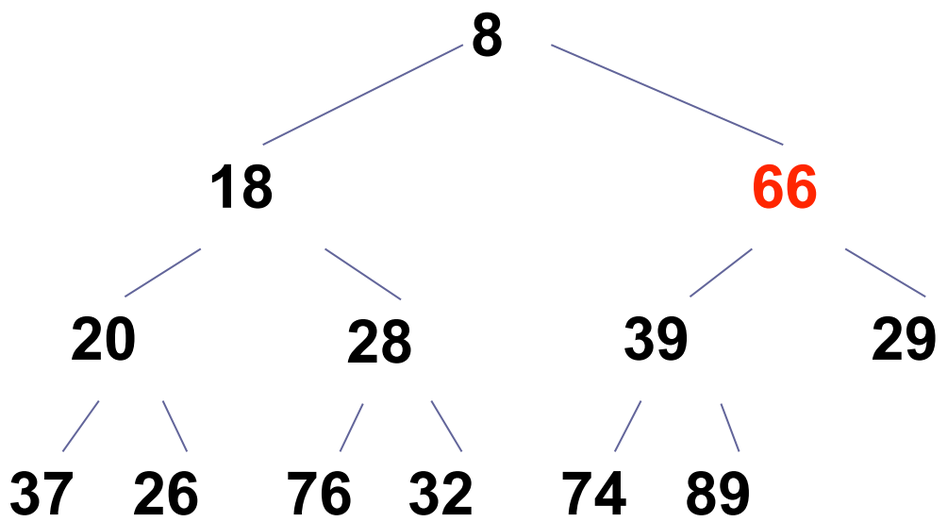
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Ta bort det minsta elementet

Algoritm för att ta bort rotnoden från en heap:

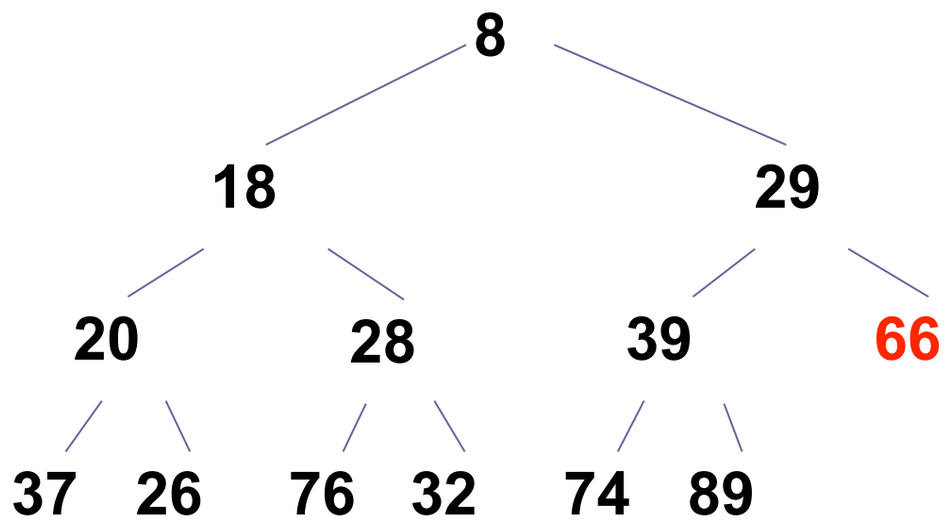
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Ta bort det minsta elementet

Algoritm för att ta bort rotnoden från en heap:

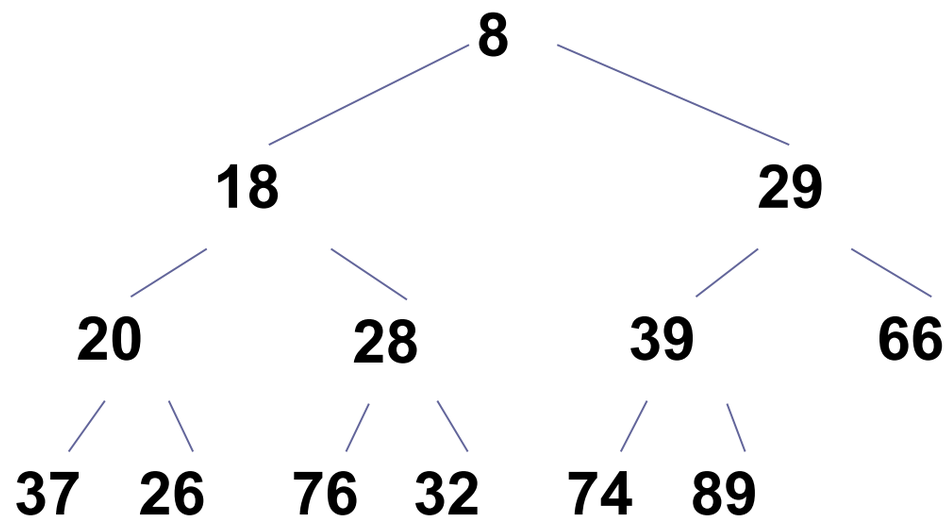
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Ta bort det minsta elementet

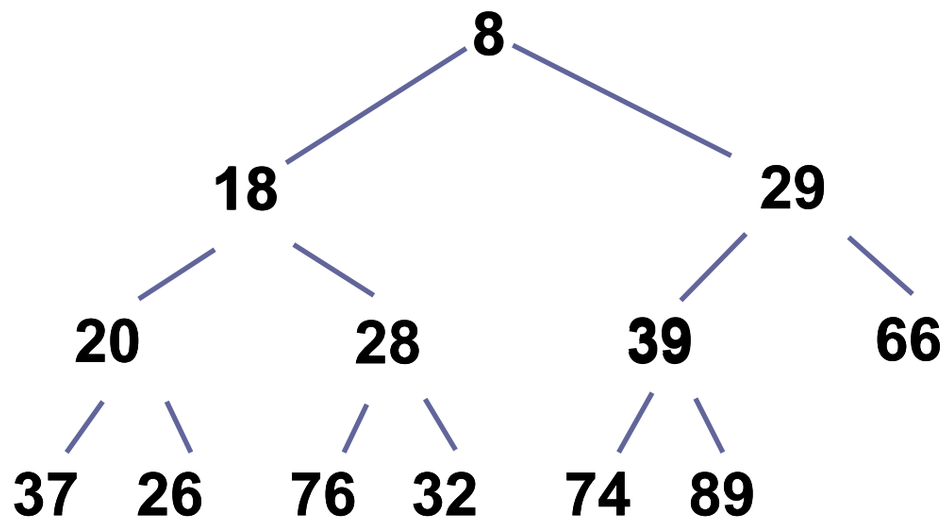
Algoritm för att ta bort rotnoden från en heap:

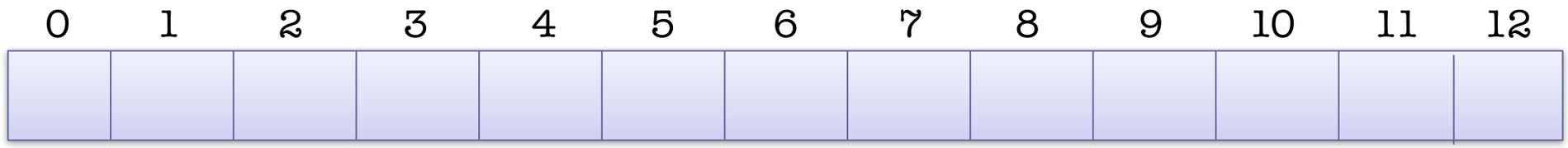
- ta bort rotnoden genom att ersätta den med det sista elementet i heapen (LEH)
- så länge som LEH har barn och LEH är större än något av barnen:
  - flytta ner LEH i heapen genom att byta ut LEH mot det minsta barnet



Att implementera en heap

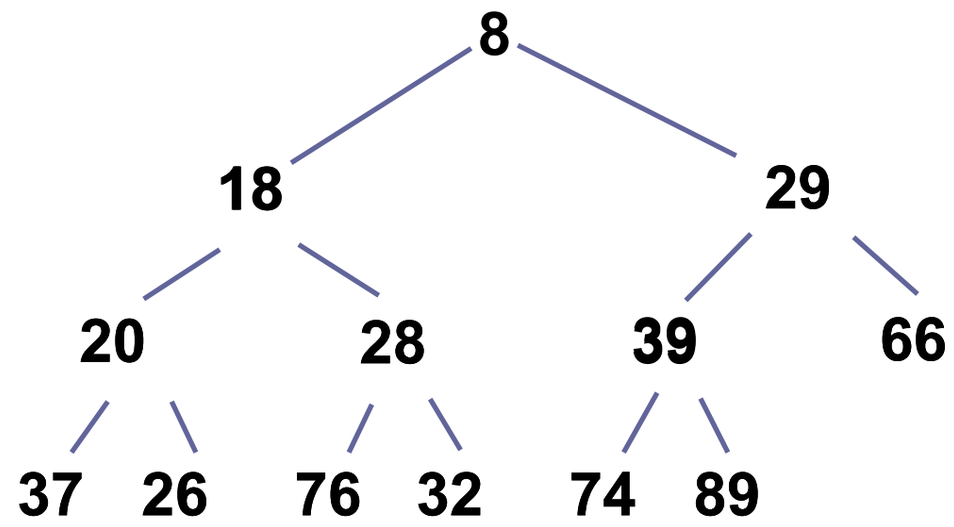
Eftersom en heap är ett *nivåbalanserat* binärt träd, kan den implementeras effektivt med en array istället för en länkad struktur

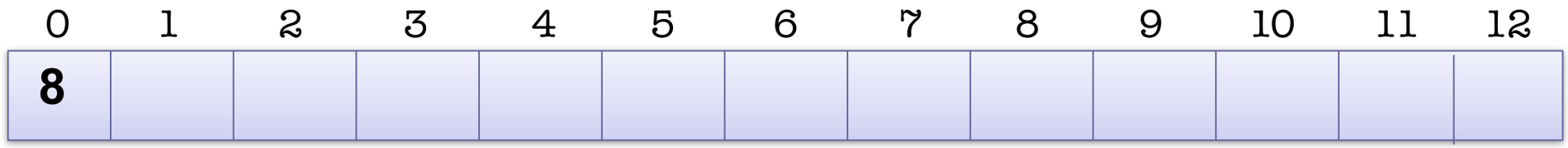




Att implementera en heap

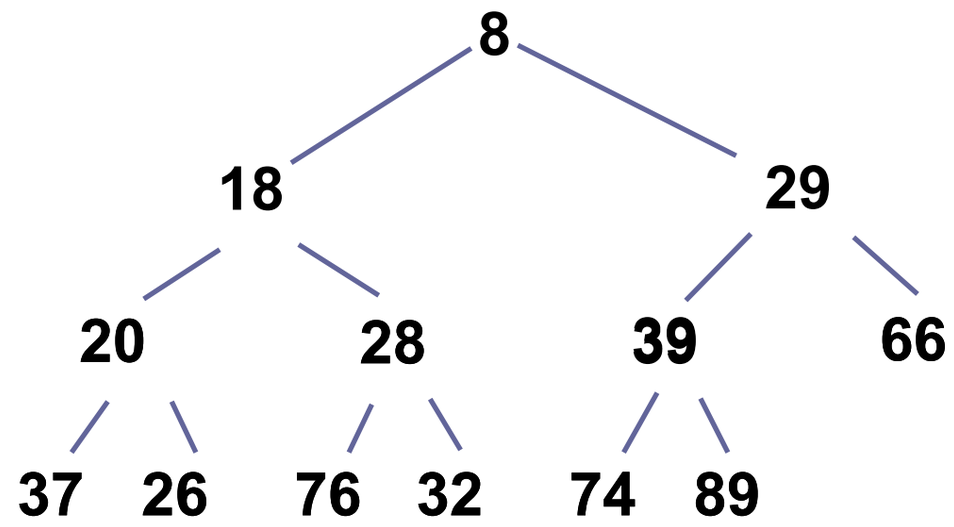
Eftersom en heap är ett *nivåbalanserat* binärt träd, kan den implementeras effektivt med en array istället för en länkad struktur

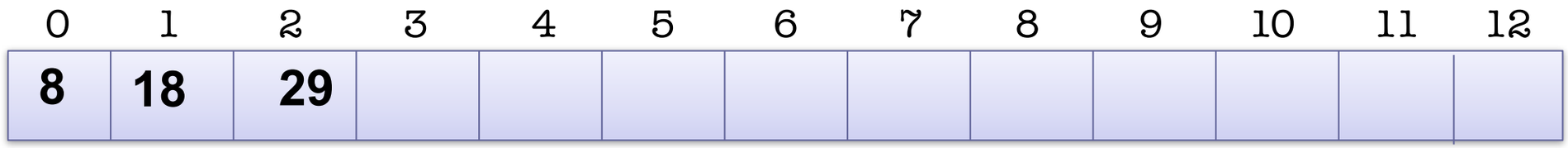




Att implementera en heap

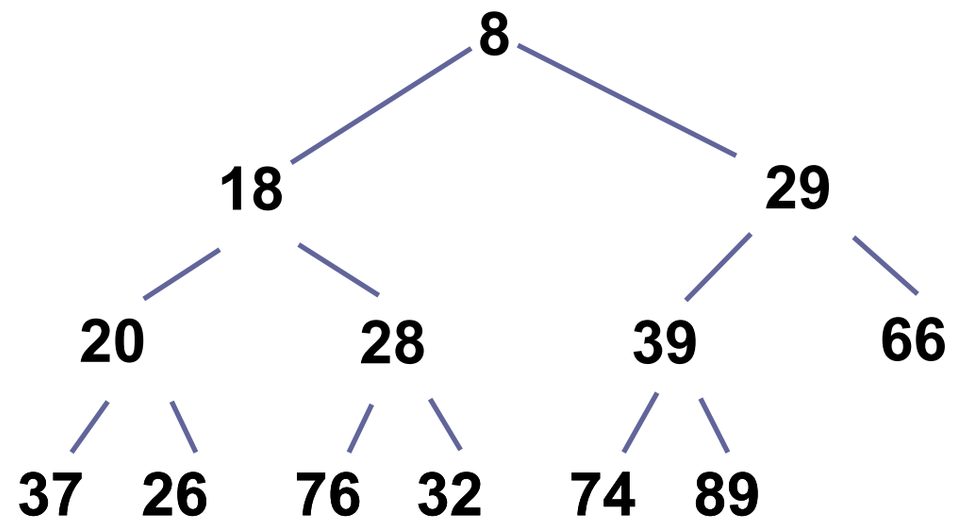
Eftersom en heap är ett *nivåbalanserat* binärt träd, kan den implementeras effektivt med en array istället för en länkad struktur

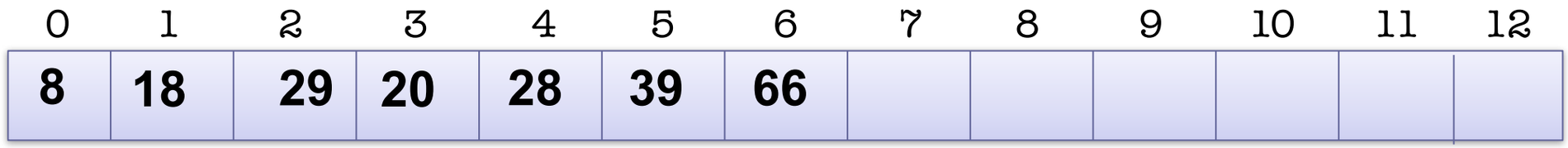




Att implementera en heap

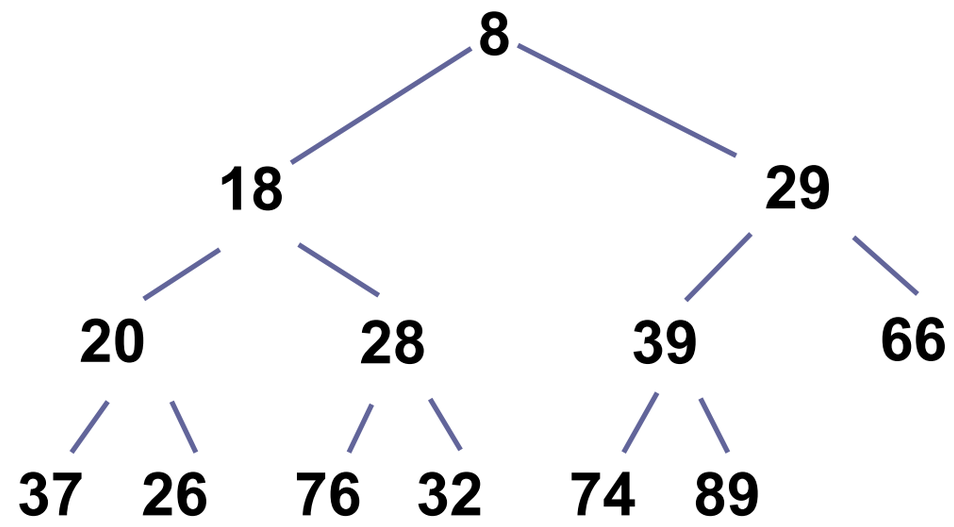
Eftersom en heap är ett *nivåbalanserat* binärt träd, kan den implementeras effektivt med en array istället för en länkad struktur





Att implementera en heap

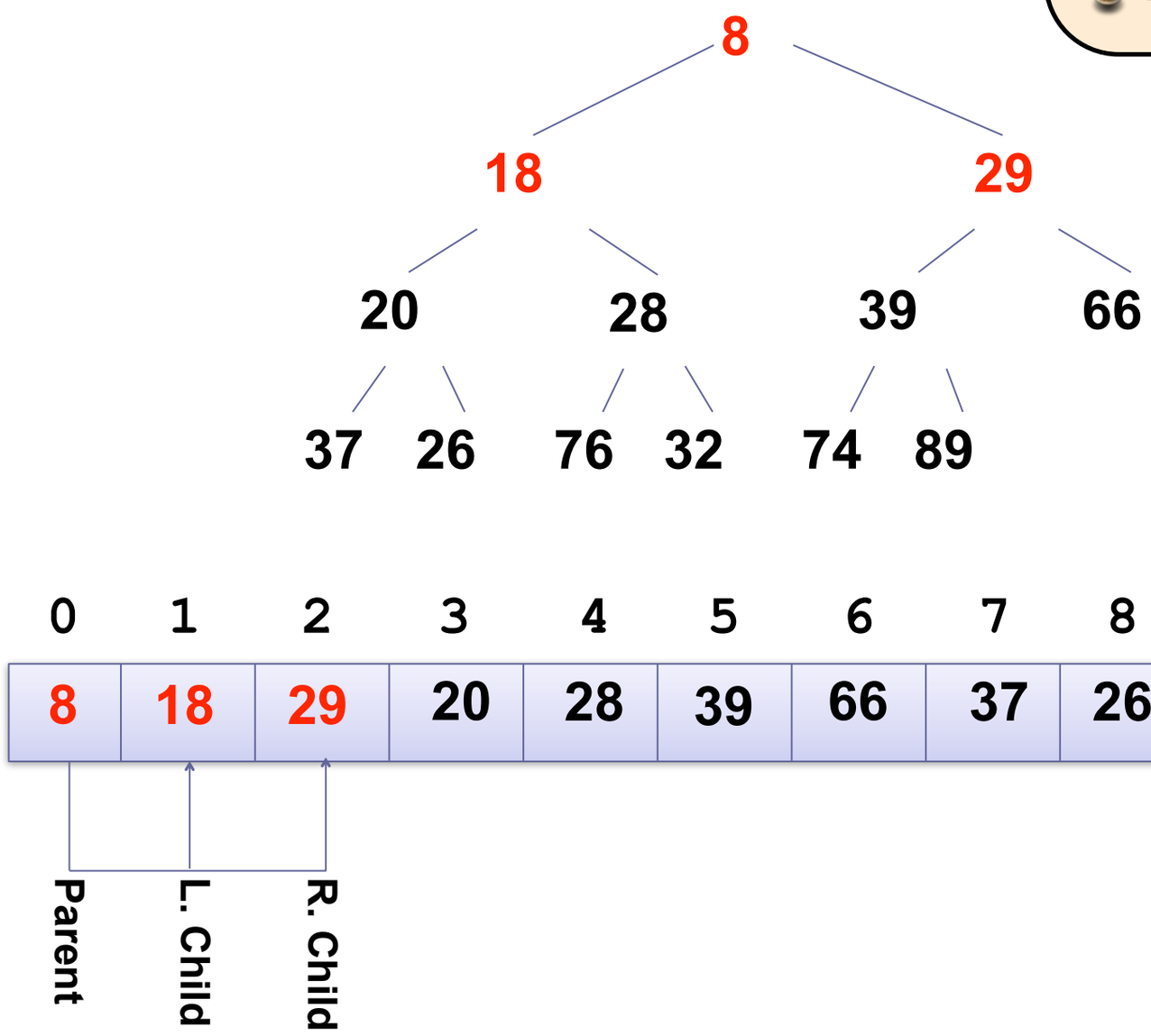
Eftersom en heap är ett *nivåbalanserat* binärt träd, kan den implementeras effektivt med en array istället för en länkad struktur



0	1	2	3	4	5	6	7	8	9	10	11	12
8	18	29	20	28	39	66	37	26	76	32	74	89

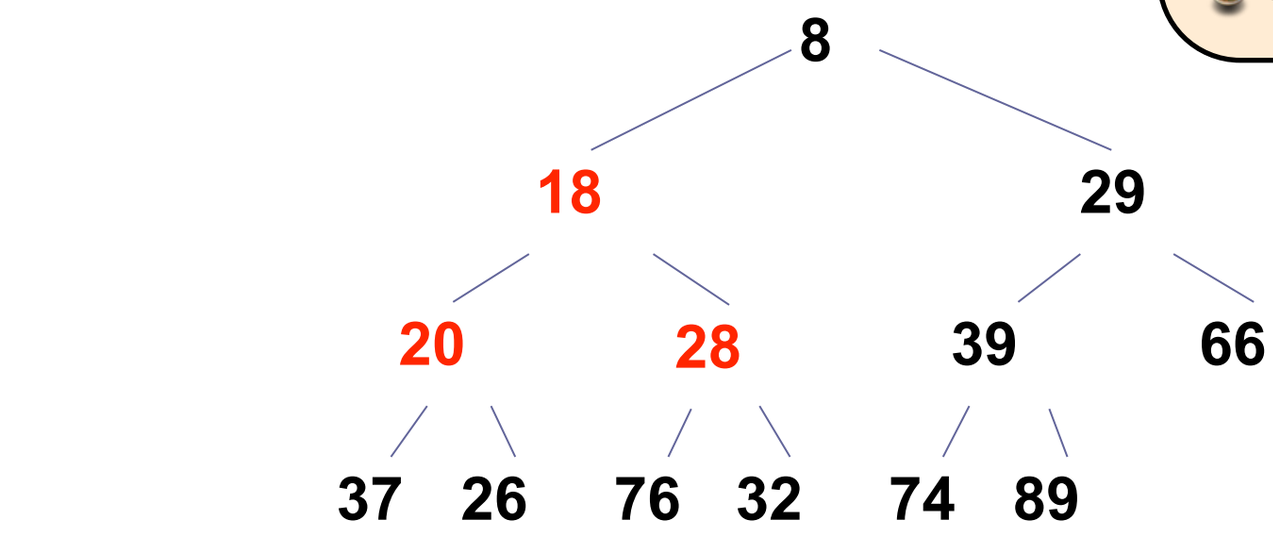
Barnens position

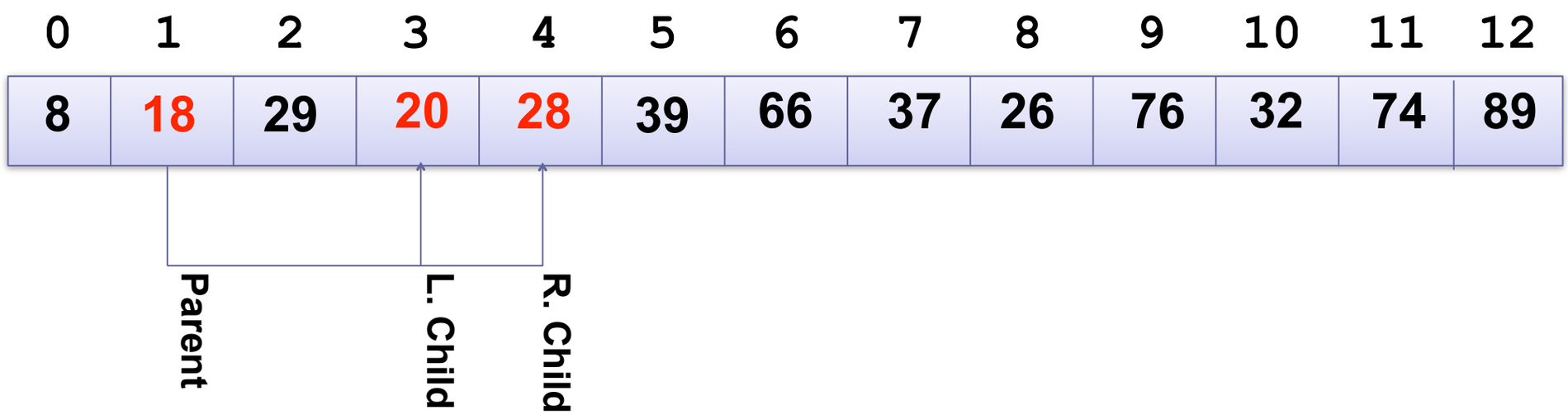
För en nod på position p gäller:
- vänster barns position: $2p + 1$
- höger barns position: $2p + 2$



Barnens position

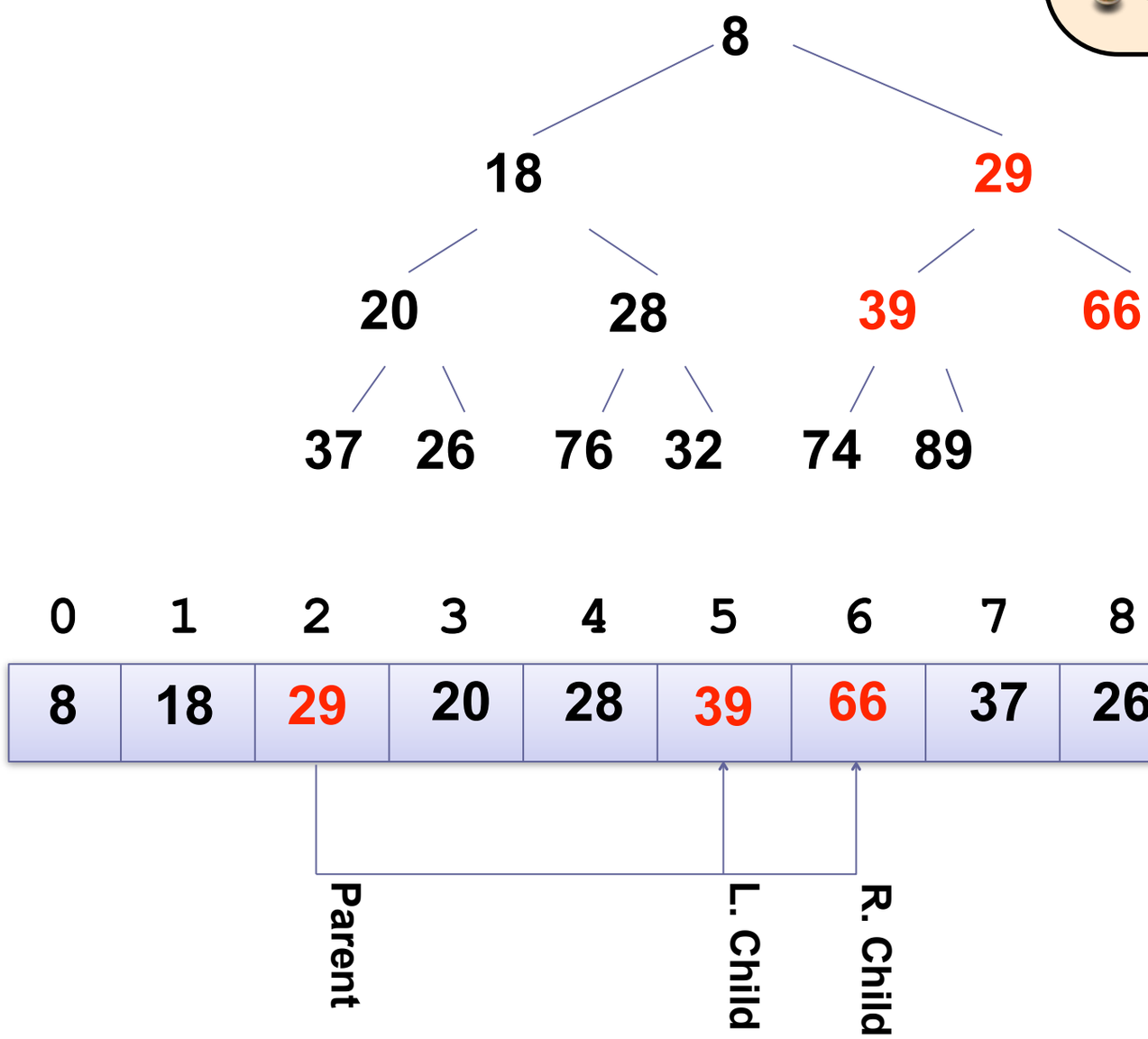
För en nod på position p gäller:
- vänster barns position: $2p + 1$
- höger barns position: $2p + 2$





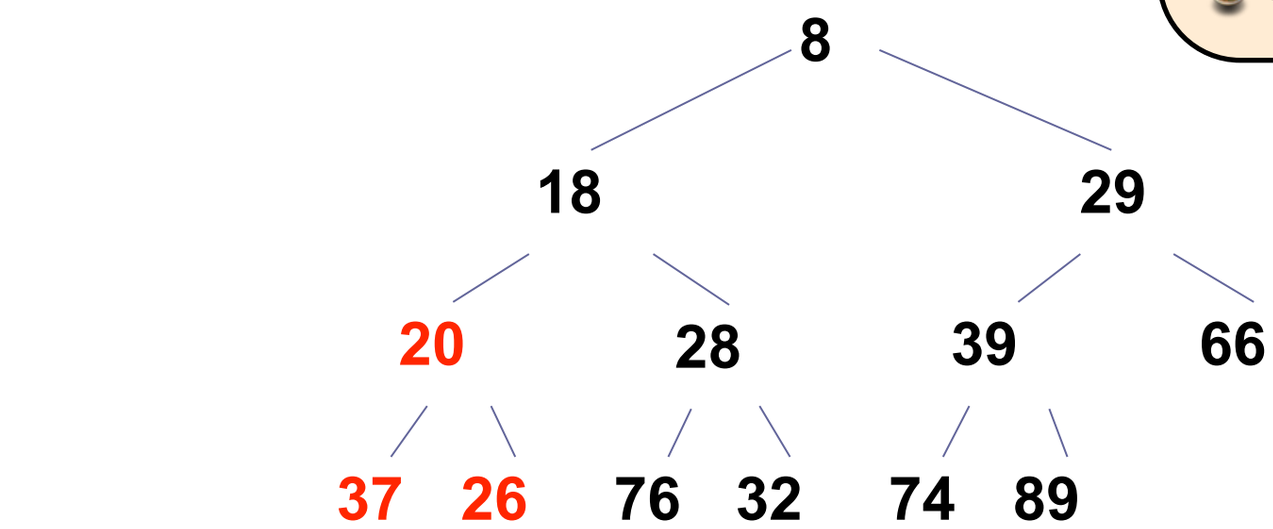
Barnens position

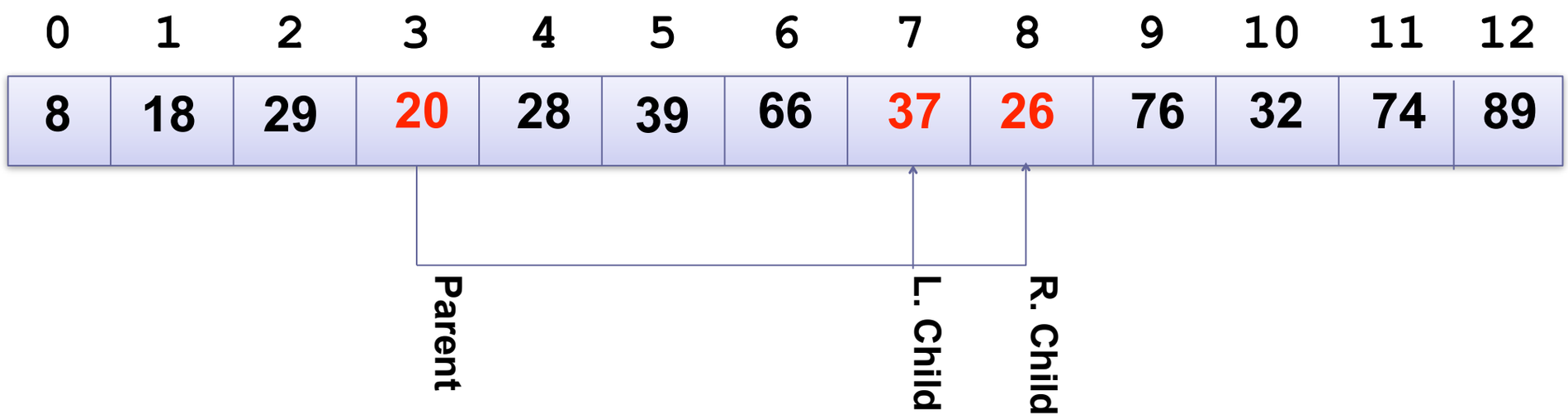
För en nod på position p gäller:
- vänster barns position: $2p + 1$
- höger barns position: $2p + 2$



Barnens position

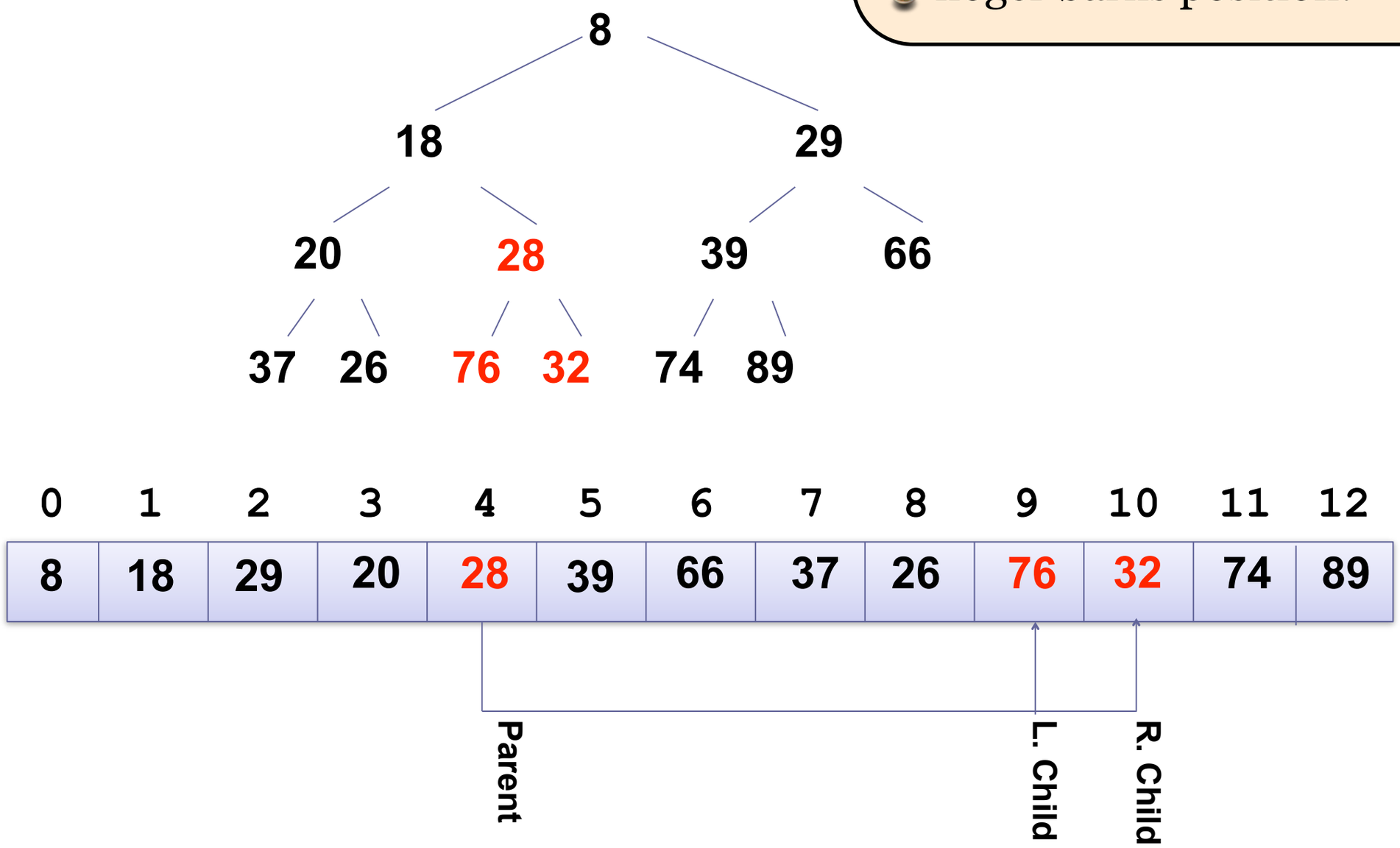
För en nod på position *p* gäller:
- vänster barns position: $2p + 1$
- höger barns position: $2p + 2$



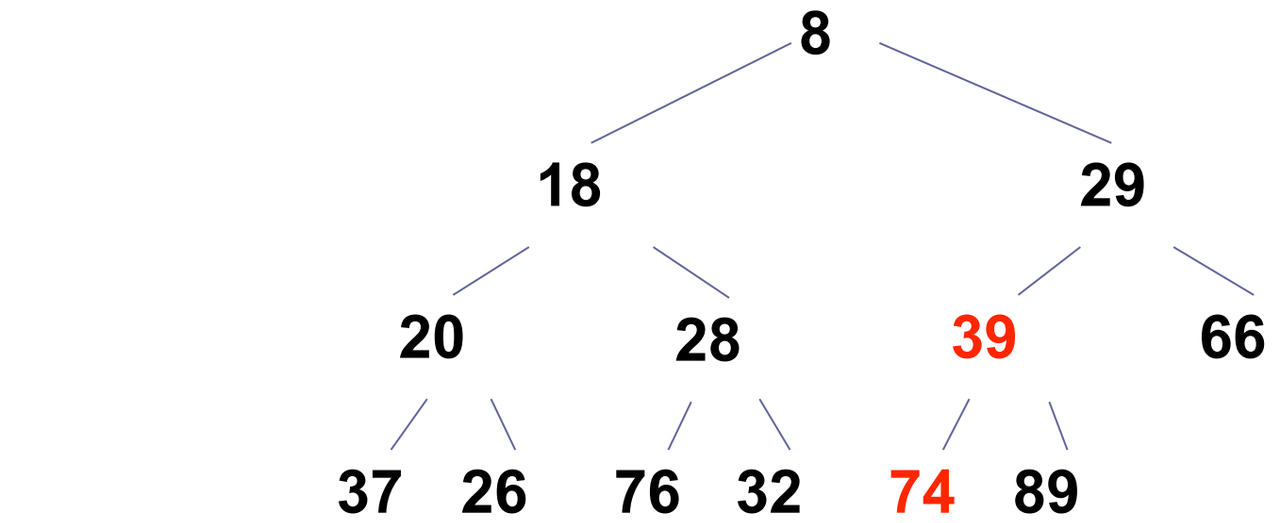


Barnens position

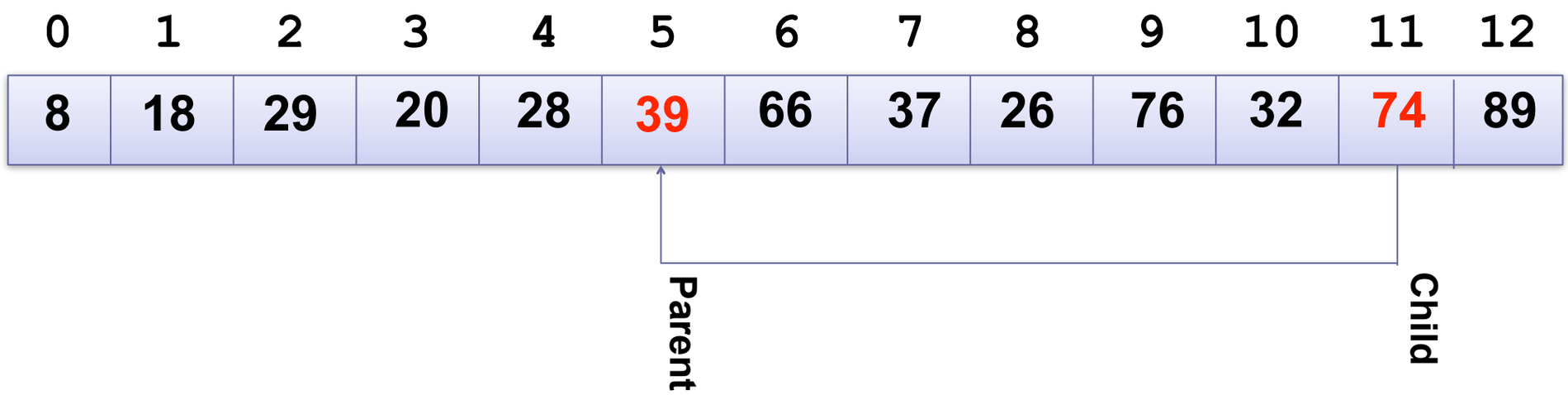
För en nod på position *p* gäller:
- vänster barns position: $2p + 1$
- höger barns position: $2p + 2$



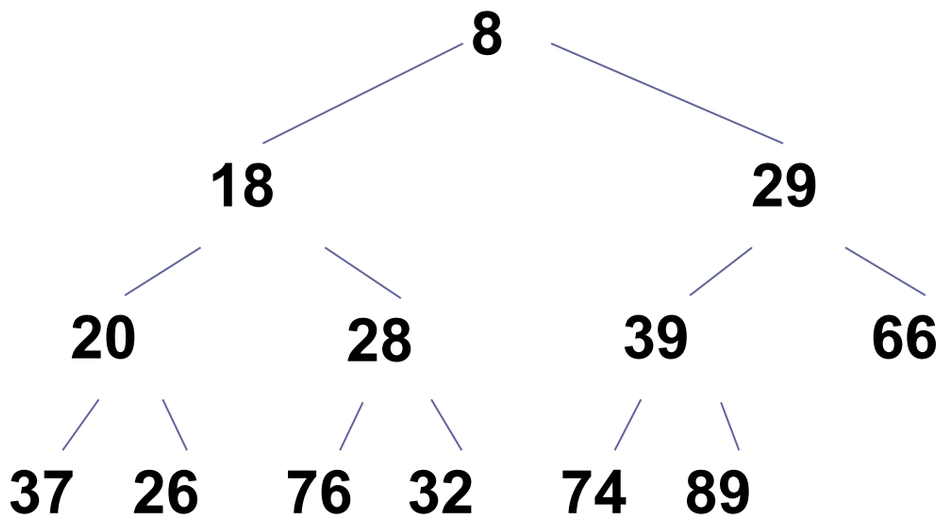
Förälderns position



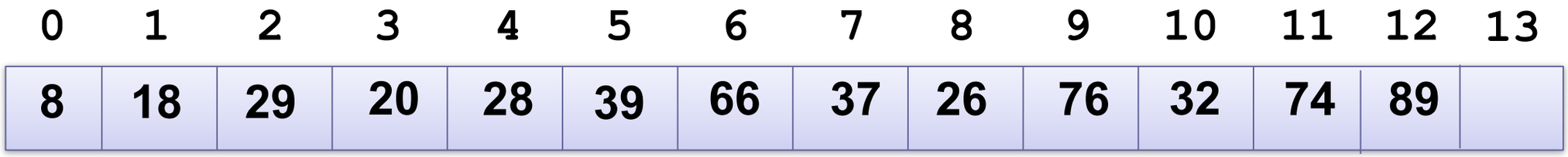
En nod på position *p* kan hitta sin förälder på positionen $(p - 1) / 2$



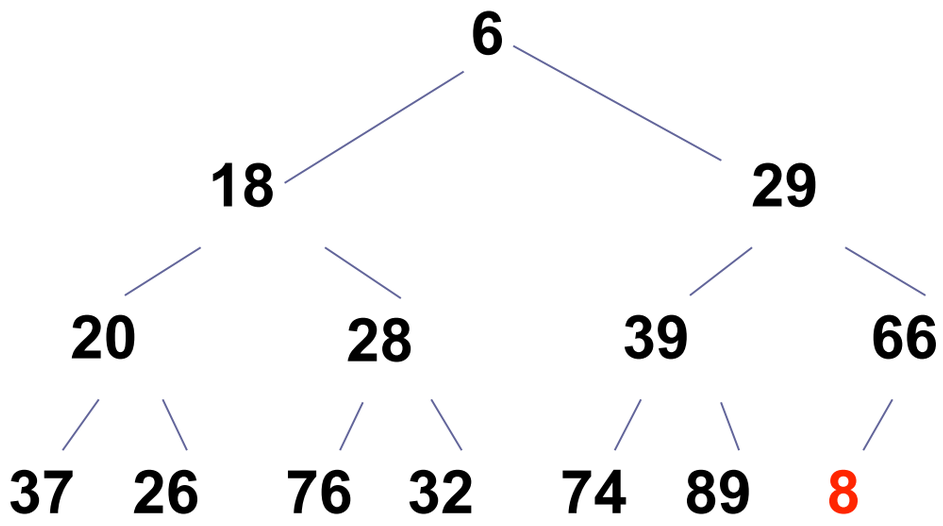
Att stoppa in element i en heap implementerad som en ArrayList



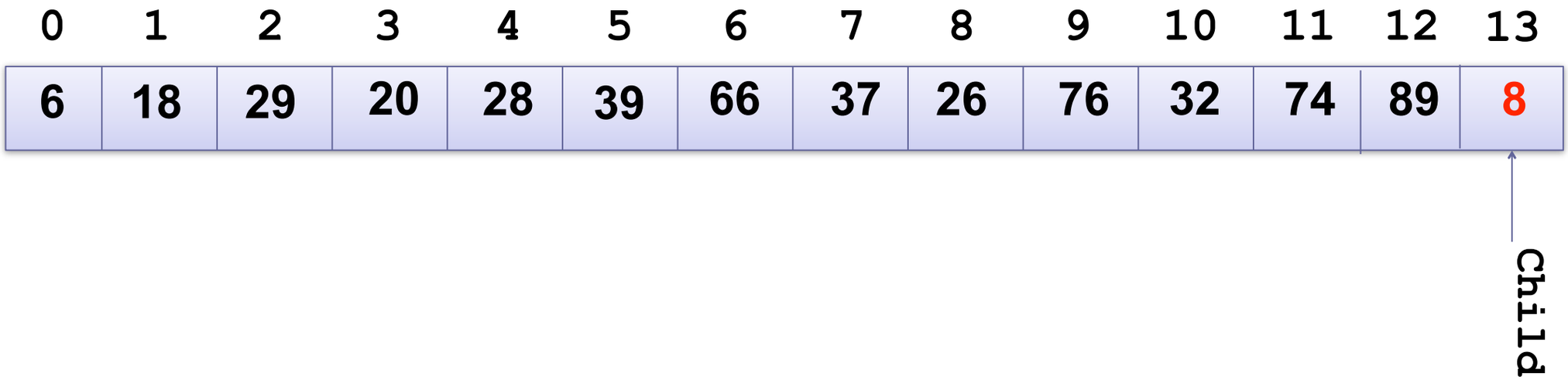
1. **Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`**



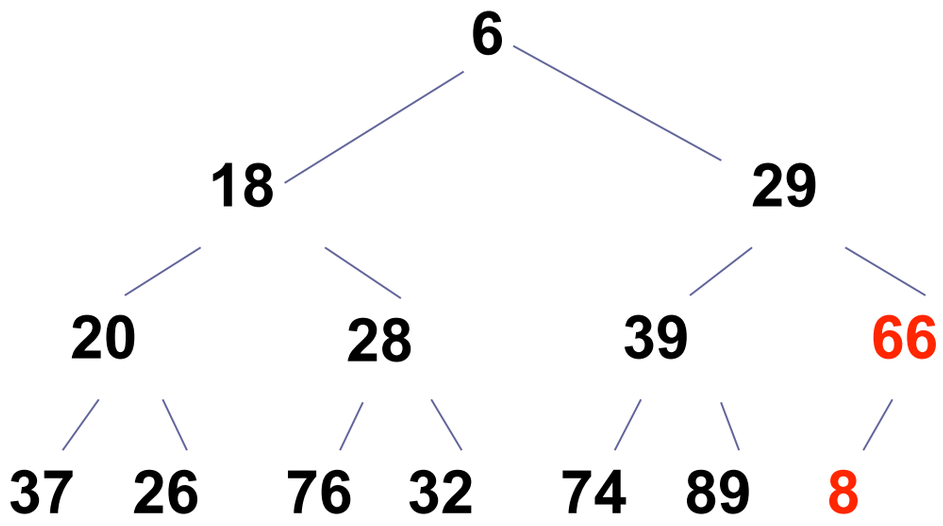
Stoppa in element (forts.)



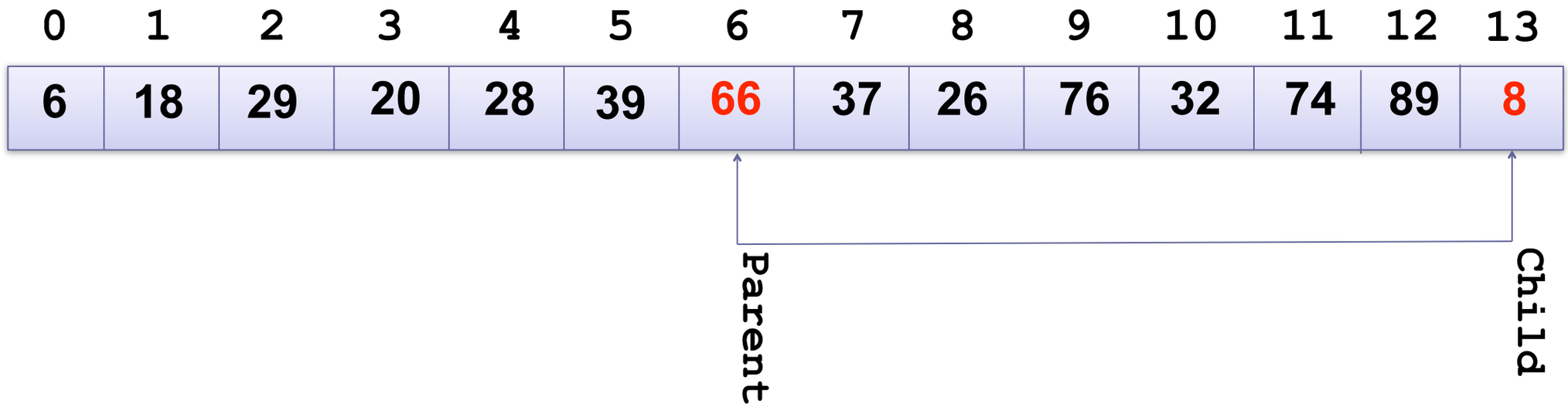
1. **Insert the new element at the end of the `ArrayList` and set `child` to `table.size() - 1`**



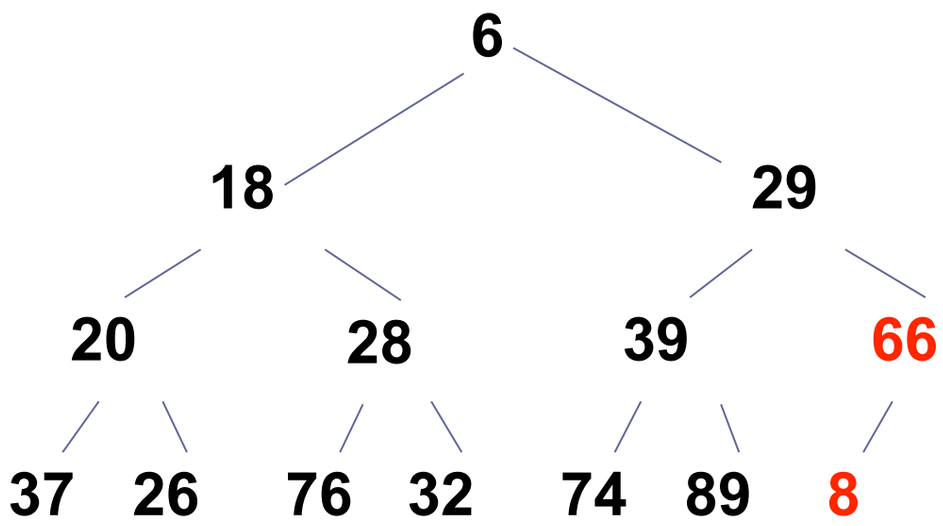
Stoppa in element (forts.)



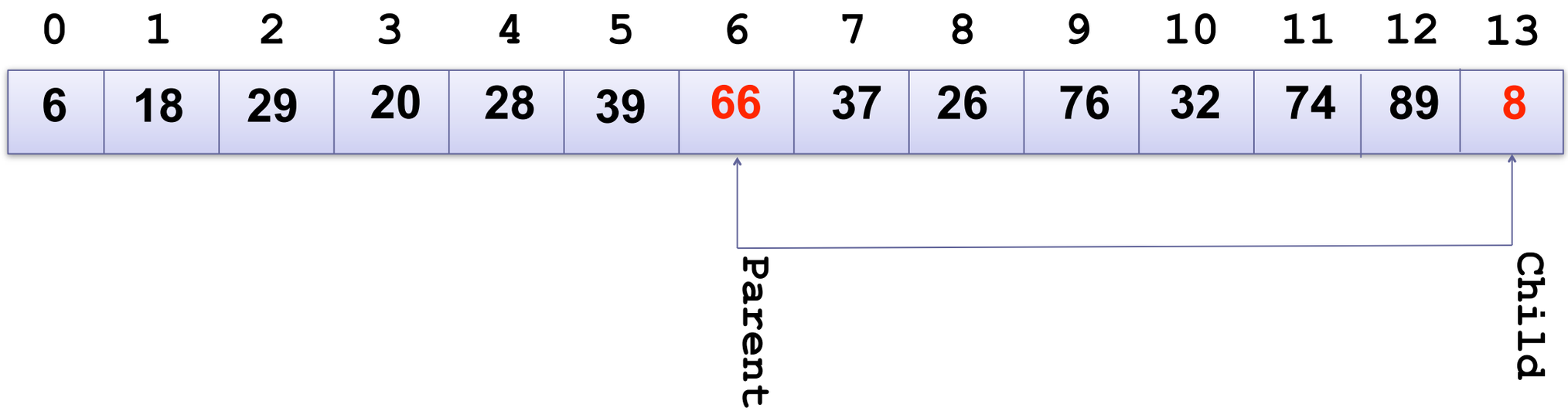
2. Set `parent` to `(child - 1)/ 2`



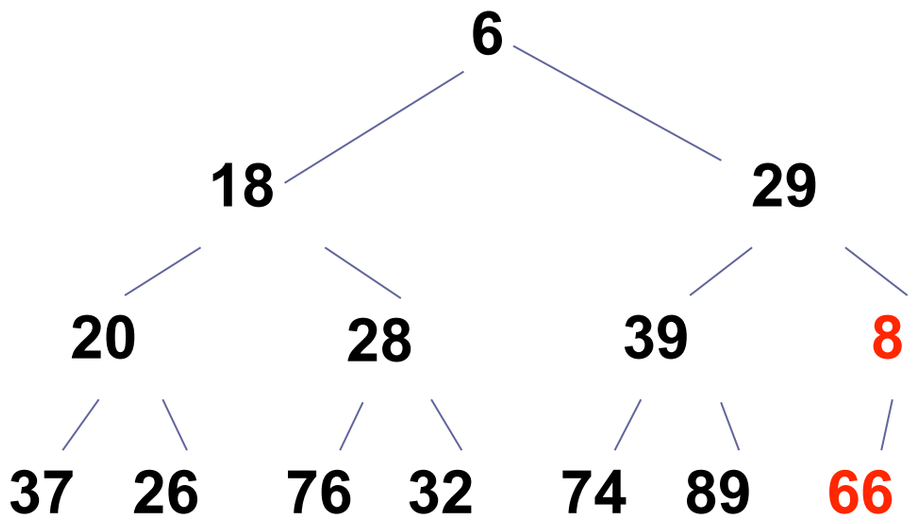
Stoppa in element (forts.)



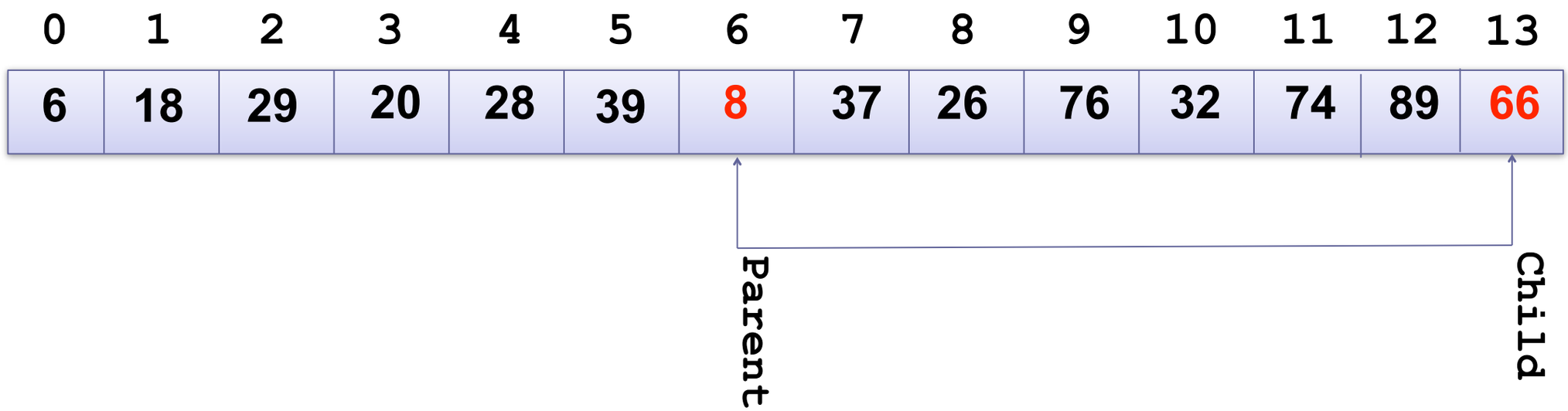
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



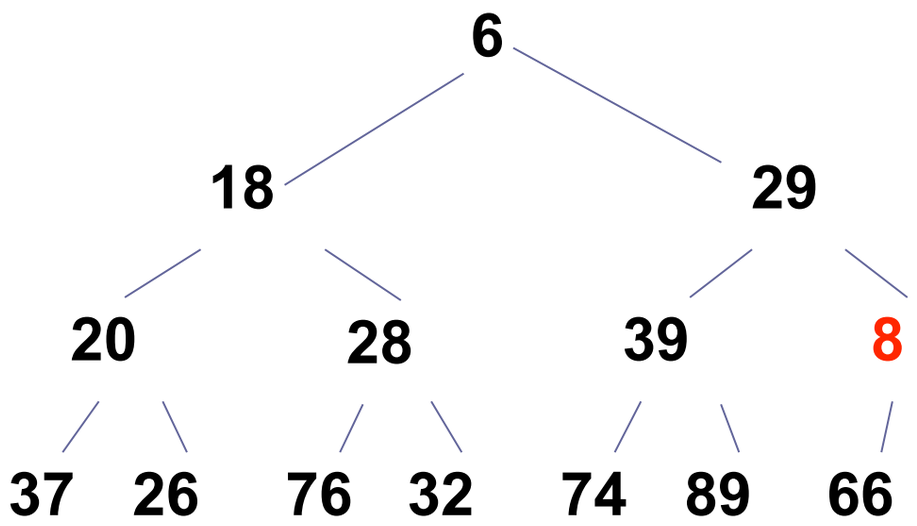
Stoppa in element (forts.)



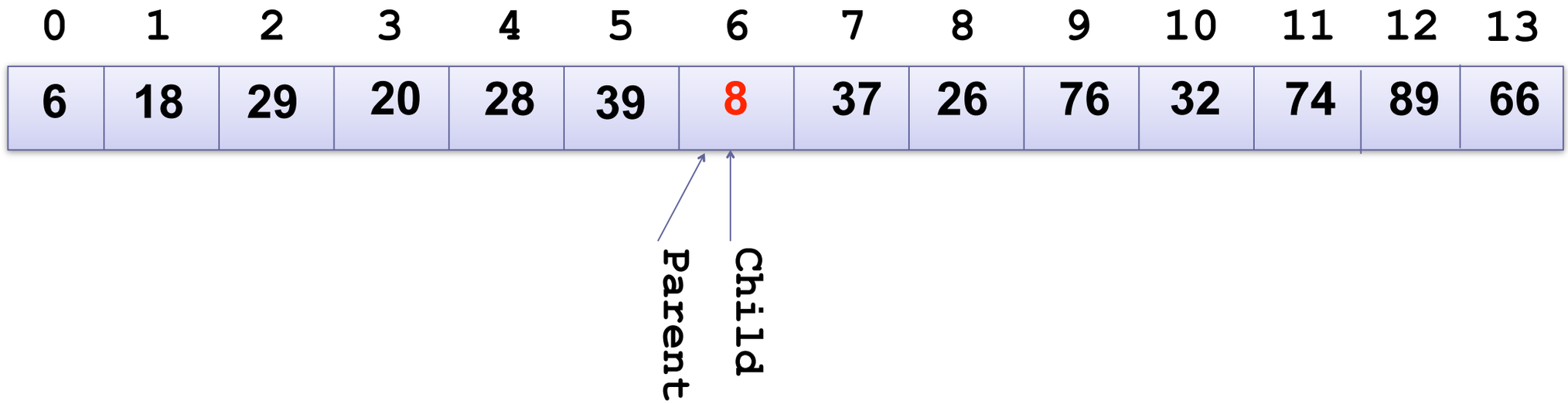
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



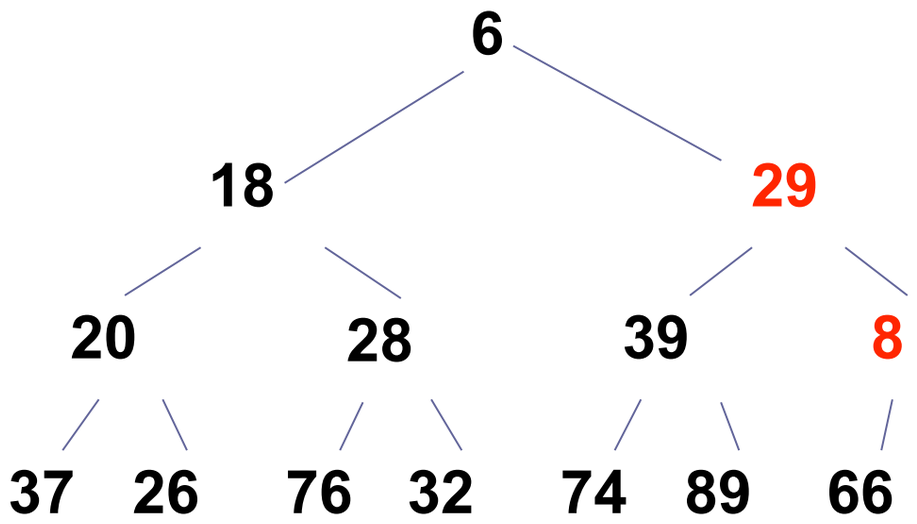
Stoppa in element (forts.)



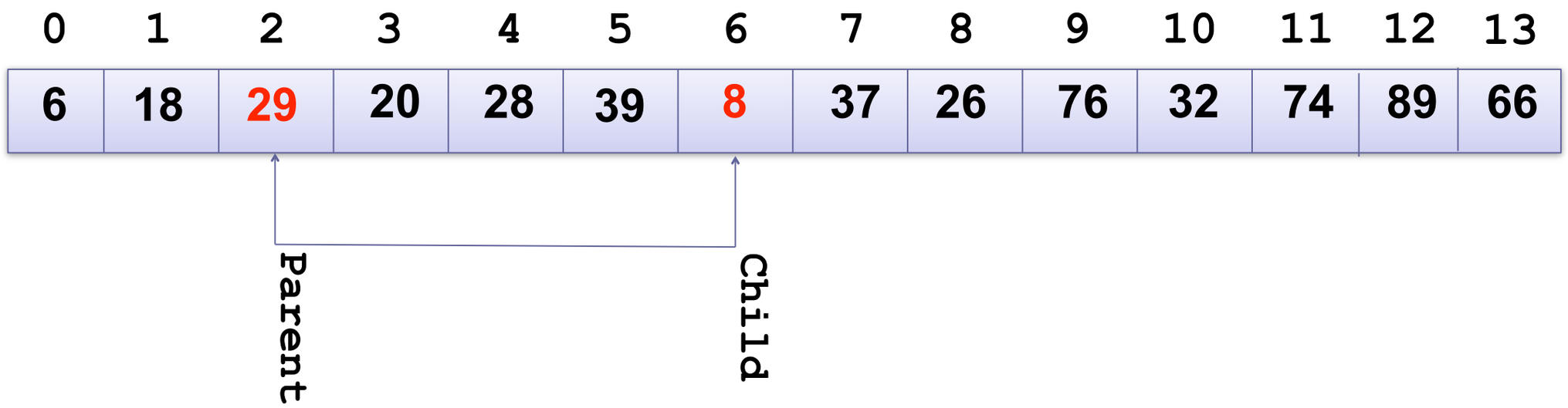
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



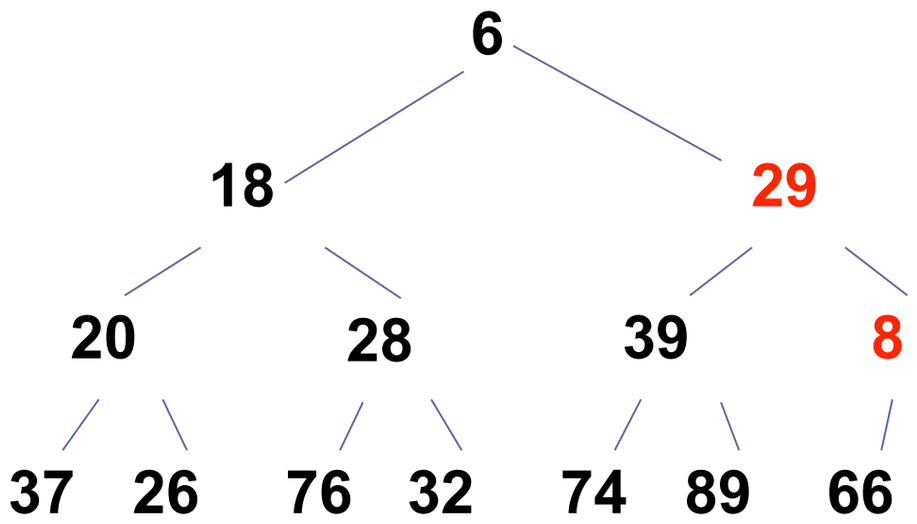
Stoppa in element (forts.)



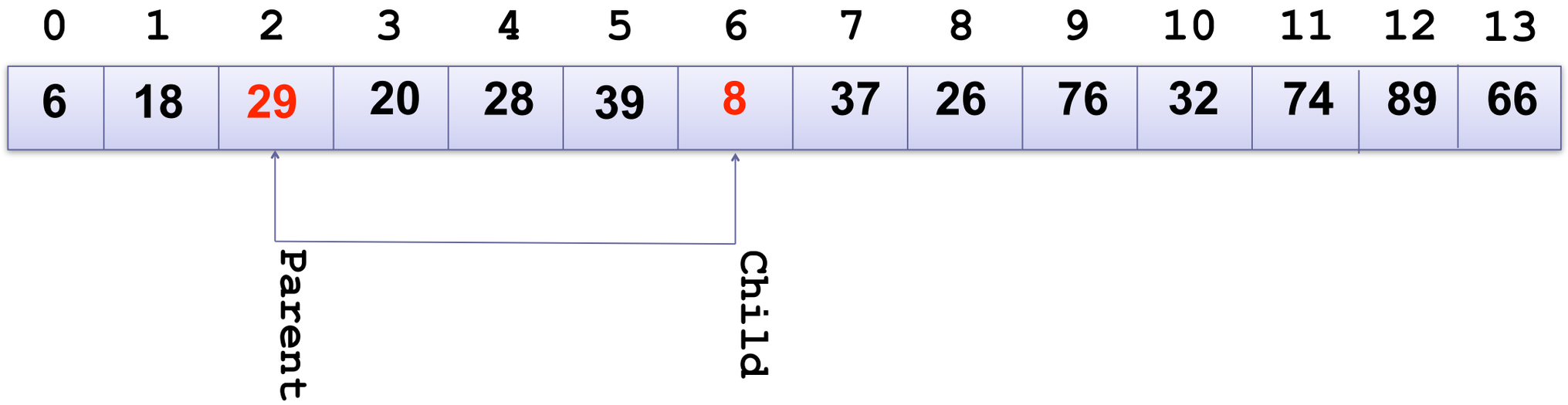
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



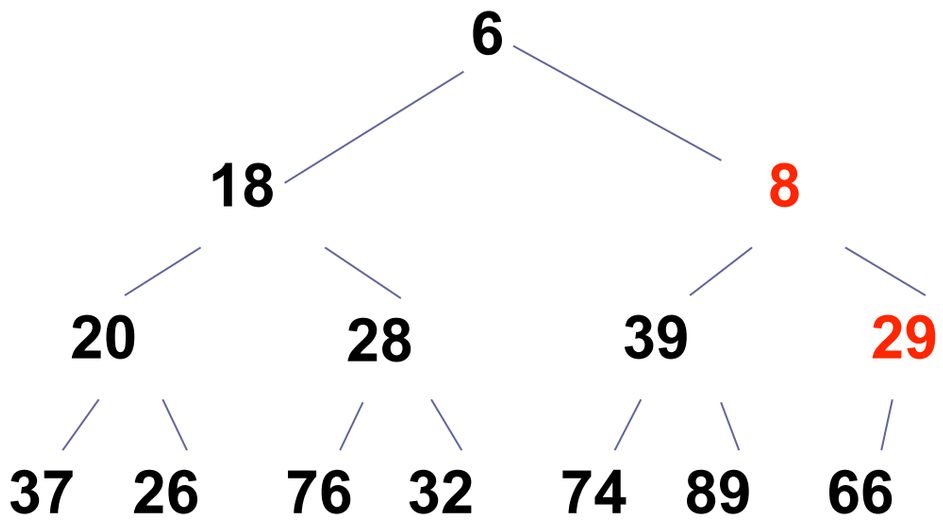
Stoppa in element (forts.)



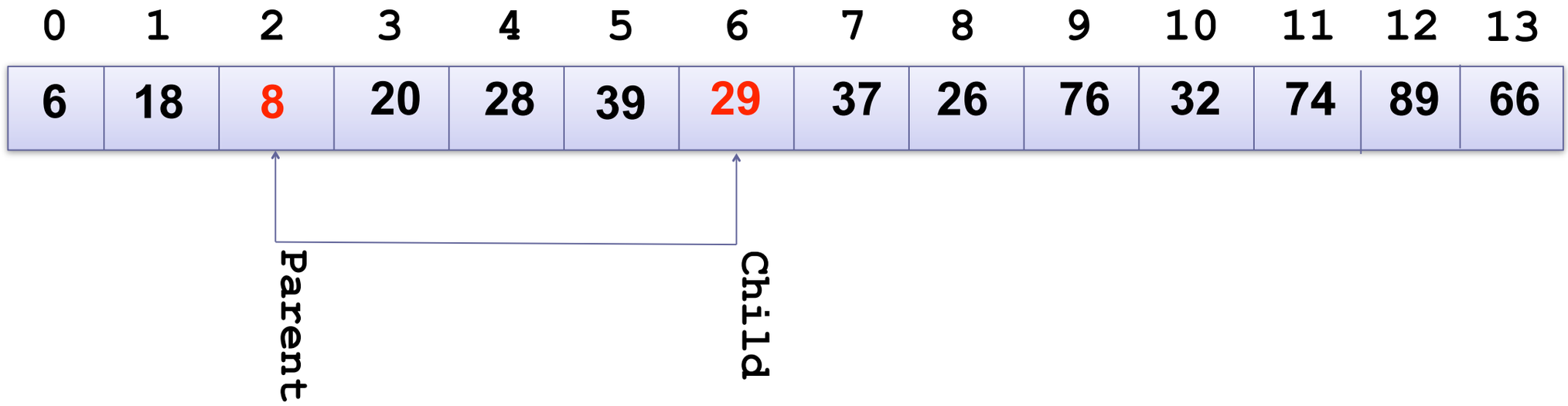
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



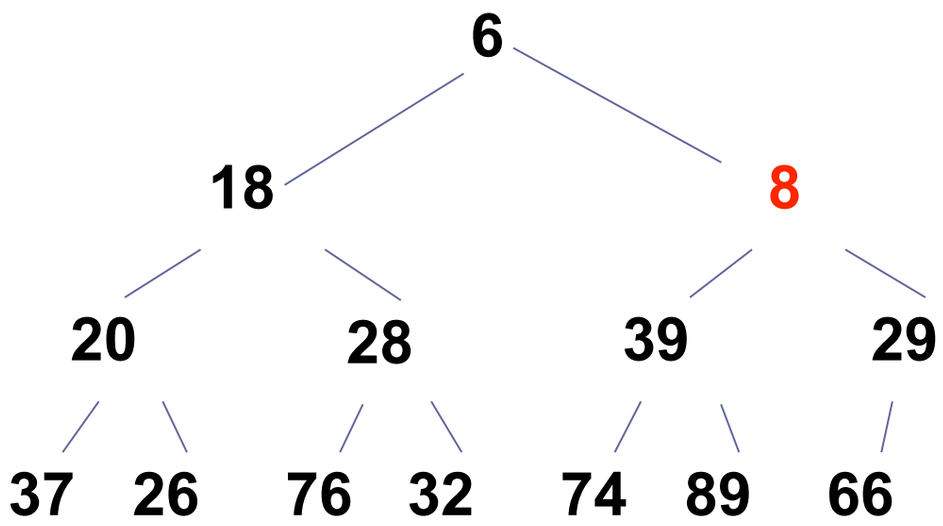
Stoppa in element (forts.)



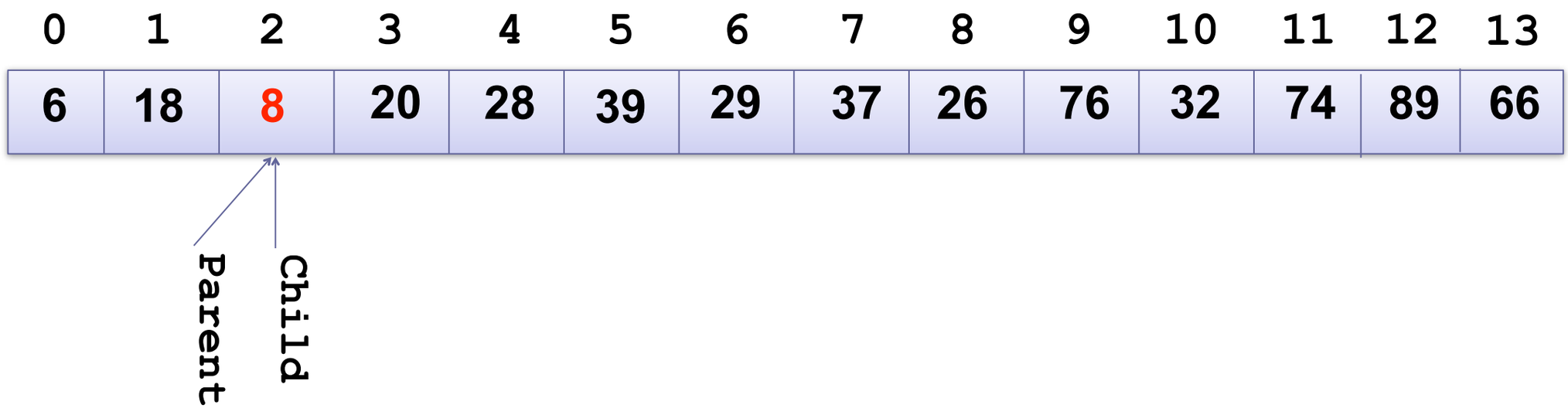
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



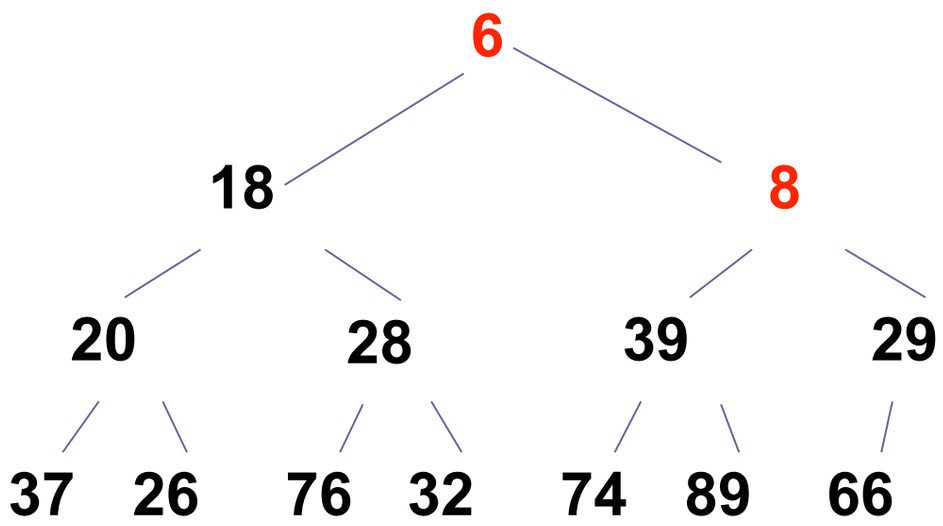
Stoppa in element (forts.)



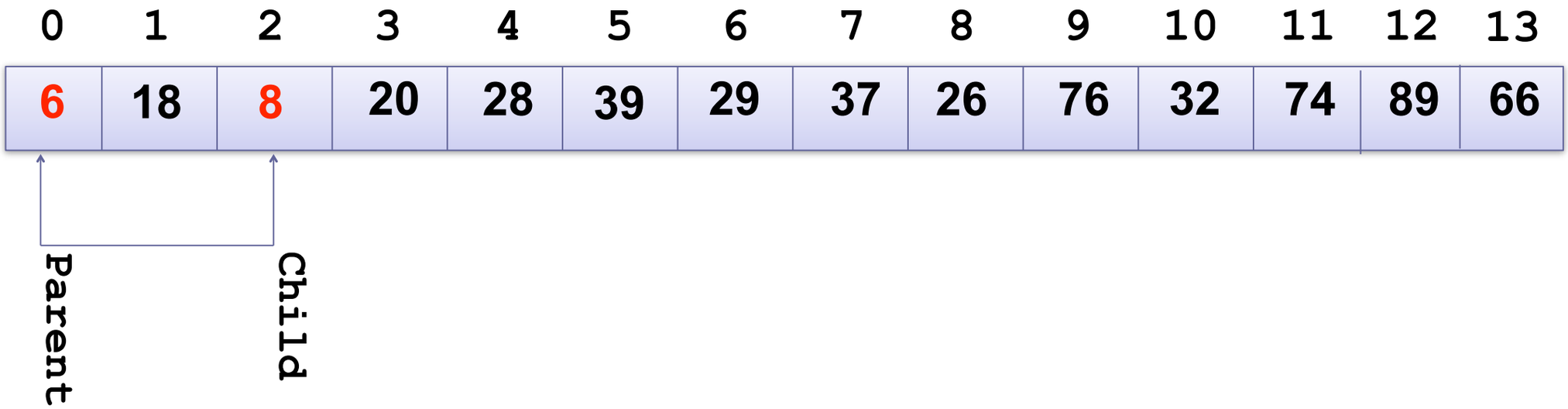
3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



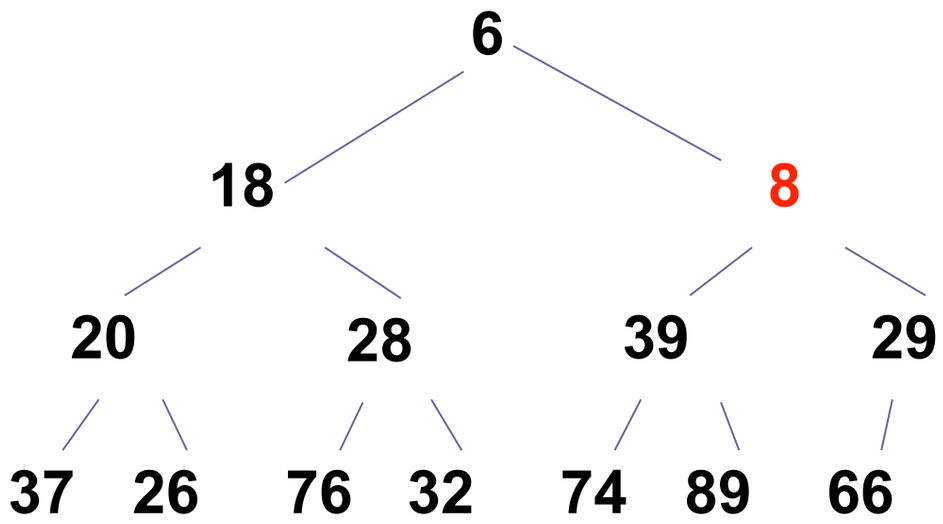
Stoppa in element (forts.)



3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`



Stoppa in element (forts.)



3. **while** (`parent >= 0` **and** `table[parent] > table[child]`)
4. **Swap** `table[parent]` **and** `table[child]`
5. **Set** `child` **equal to** `parent`
6. **Set** `parent` **equal to** `(child-1)/2`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	8	20	28	39	29	37	26	76	32	74	89	66

Effektivitet

Hur effektivt är insättning resp. borttagning?

- insert går en stig från ett löv till roten
- remove går en stig från roten till ett löv
- stigen är högst h steg, där h är trädets höjd
- ett nivåbalanserat träd med höjd h har minst 2^{h-1} noder
- dvs, $2^{h-1} \leq n$
- dvs, $h - 1 \leq \log n$

Alltså: insättning resp. borttagning är logaritmiskt, O(log n)

- där n är antalet noder i trädet

Prioritetsköer

I en skrivarkö kan det ibland vara bättre att låta små dokument tränga sig före stora dokument, även om de har dykt upp efteråt

- precis som i en kassakö, där snälla människor med fulla kundvagnar kan låta en stackare med bara tre varor gå före
- de snälla människorna har då en *prioritetskö* i sina hjärnceller, istället för en vanlig enkel kö
- en prioritetskö brukar implementeras som en heap

Klassen PriorityQueue<E>

java.util.PriorityQueue<E>
implementerar gränssnittet Queue<E>

Method	Behavior
`boolean offer(E item)`	Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the smallest entry and returns it. If the queue is empty, returns **`null`**.
`E peek()`	Returns the smallest entry without removing it. If the queue is empty, returns **`null`**.
`E element()`	Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`.

Klassen KWPriorityQueue<E>

Boken ger en egen implementation med följande variabler, konstruerare och privata metoder:

Data Field	Attribute
`ArrayList<E> theData`	An `ArrayList` to hold the data.
`Comparator<E> comparator`	An optional object that implements the `Comparator<E>` interface by providing a `compare` method.

Method	Behavior
`KWPriorityQueue()`	Constructs a heap-based priority queue that uses the elements' natural ordering.
`KWPriorityQueue (Comparator<E> comp)`	Constructs a heap-based priority queue that uses the `compare` method of `Comparator comp` to determine the ordering of the elements.
`private int compare(E left, E right)`	Compares two objects and returns a negative number if object `left` is less than object `right`, zero if they are equal, and a positive number if object `left` is greater than object `right`.
`private void swap(int i, int j)`	Exchanges the object references in `theData` at indexes `i` and `j`.

Metoden offer(E)

```
/** Insert an item into the priority queue.
    pre: The ArrayList theData is in heap order.
    post: The item is in the priority queue and
         theData is in heap order.
    @param item The item to be inserted
    @throws NullPointerException if the item to be inserted is null.
*/

@Override
public boolean offer(E item) {
    // Add the item to the heap.
    theData.add(item);
    // child is newly inserted item.
    int child = theData.size() - 1;
    int parent = (child - 1) / 2; // Find child's parent.
    // Reheap
    while (parent >= 0 && compare(theData.get(parent),
                                  theData.get(child)) > 0) {
        swap(parent, child);
        child = parent;
        parent = (child - 1) / 2;
    }
    return true;
}
```

Metoden poll()

```
/** Remove an item from the priority queue
    pre: The ArrayList theData is in heap order.
    post: Removed smallest item, theData is in heap order.
    @return The item with the smallest priority value or null if empty.
*/

@Override
public E poll() {
    if (isEmpty()) {
        return null;
    }
    // Save the top of the heap.
    E result = theData.get(0);
    // If only one item then remove it.
    if (theData.size() == 1) {
        theData.remove(0);
        return result;
    }

    ...
```

Metoden poll() [forts.]

```
// Remove the last item from the ArrayList and place it into the first position.
theData.set(0, theData.remove(theData.size() - 1));
// The parent starts at the top.
int parent = 0;
while (true) {
    int leftChild = 2 * parent + 1;
    if (leftChild >= theData.size()) {
        break; // Out of heap.
    }
    int rightChild = leftChild + 1;
    int minChild = leftChild; // Assume leftChild is smaller.
    // See whether rightChild is smaller.
    if (rightChild < theData.size()
        && compare(theData.get(leftChild), theData.get(rightChild)) > 0) {
        minChild = rightChild;
    }
    // assert: minChild is the index of the smaller child.
    // Move smaller child up heap if necessary.
    if (compare(theData.get(parent), theData.get(minChild)) > 0) {
        swap(parent, minChild);
        parent = minChild;
    } else { // Heap property is restored.
        break;
    }
  }
  return result;
}
```

Metoden compare(E, E)

```
/** Compare two items using either a Comparator object's compare method
    or their natural ordering using method compareTo.
    pre: If comparator is null, left and right implement Comparable<E>.
    @param left One item
    @param right The other item
    @return Negative int if left less than right,
            0 if left equals right,
            positive int if left > right
    @throws ClassCastException if items are not Comparable
*/

private int compare(E left, E right) {
    if (comparator != null) {
        // A Comparator is defined.
        return comparator.compare(left, right);
    } else {
        // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

Exempel: Huffmankodning

Huffmankodning baseras på bokstavsfrekvenser:

- vanliga bokstäver får korta bitsekvenser (t.ex. *e* får 010)
- ovanliga bokstäver får långa (t.ex. *z* får 1100001010)

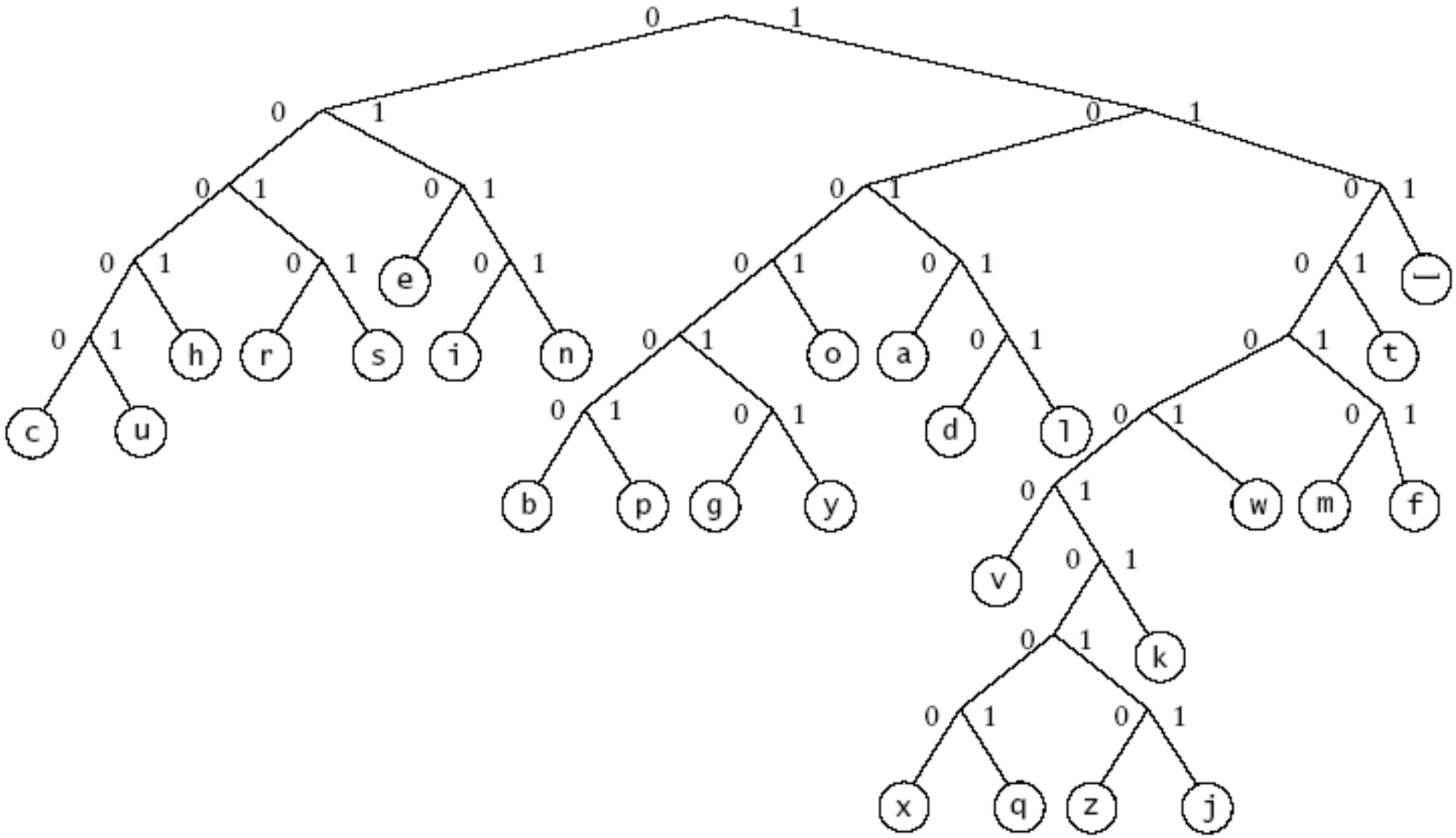
Här är en tabell med frekvenser i engelsk text:

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
␣	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

Huffmanträd

Huffmankoden lagras som ett binärt träd, ett Huffmanträd

Här är ett Huffmanträd baserat på bokstavsfrekvenser i engelsk text:



Att bygga ett Huffmanträd

Vi kan bygga ett Huffmanträd med hjälp av en prioritetskö:

- elementen i prioritetskön är delträd av det slutliga Huffmanträdet
- kön är ordnad efter delträdens frekvenser
- vi börjar med att stoppa in alla löven i kön
- så länge som kön har mer än ett element:
  - ta bort de två första elementen (som är binära träd)
  - skapa ett binärt träd av dem
  - trädets frekvens blir summan av delträdens frekvenser
  - stoppa in det nya trädet i kön

Pseudokod

Algorithm for Building a Huffman Tree

1. Construct a set of trees with root nodes that contain each of the individual symbols and their weights.
2. Place the set of trees into a priority queue.
3. `while` the priority queue has more than one item:
4. Remove the two trees with the smallest weights.
5. Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.
6. Insert the newly created tree back into the priority queue.

Huffmanträd: Exempel

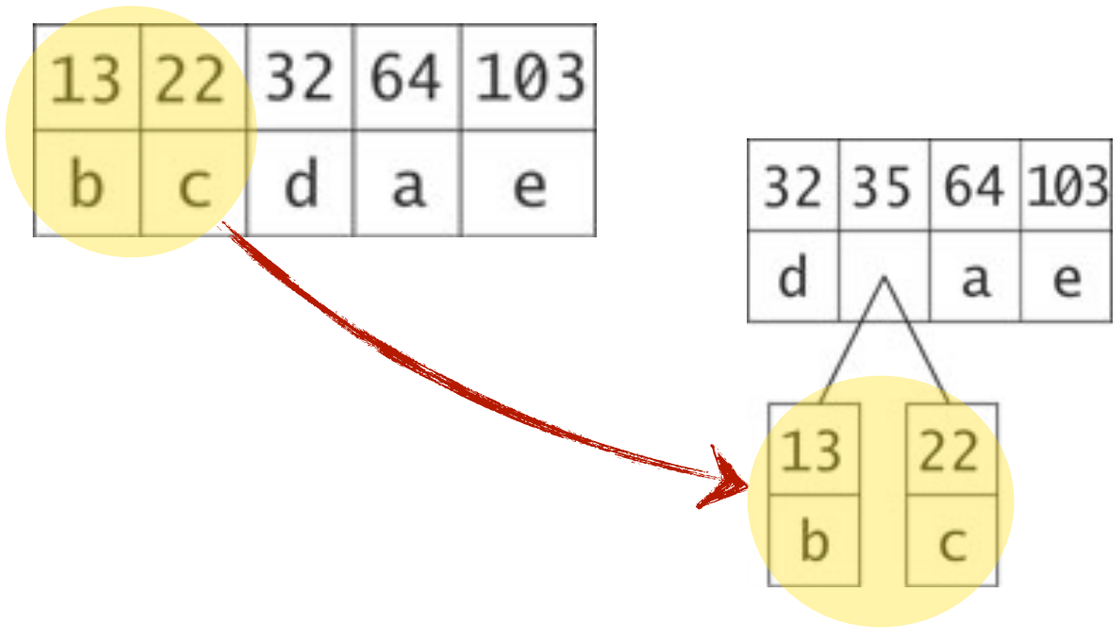
Huffmanträd: Exempel

13	22	32	64	103
b	c	d	a	e

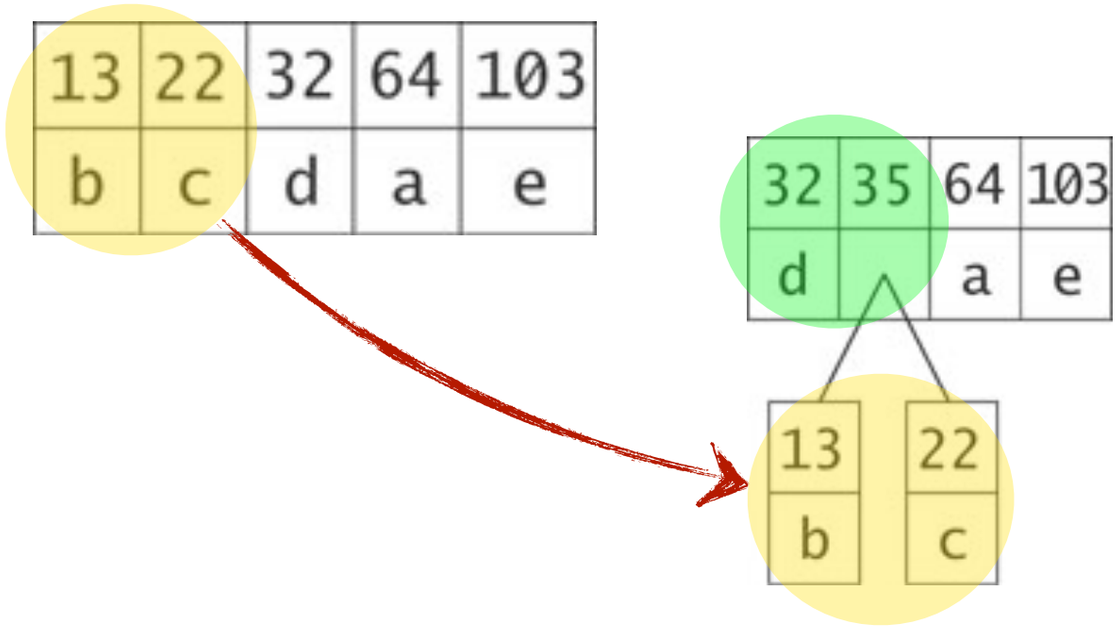
Huffmanträd: Exempel

13	22	32	64	103
b	c	d	a	e

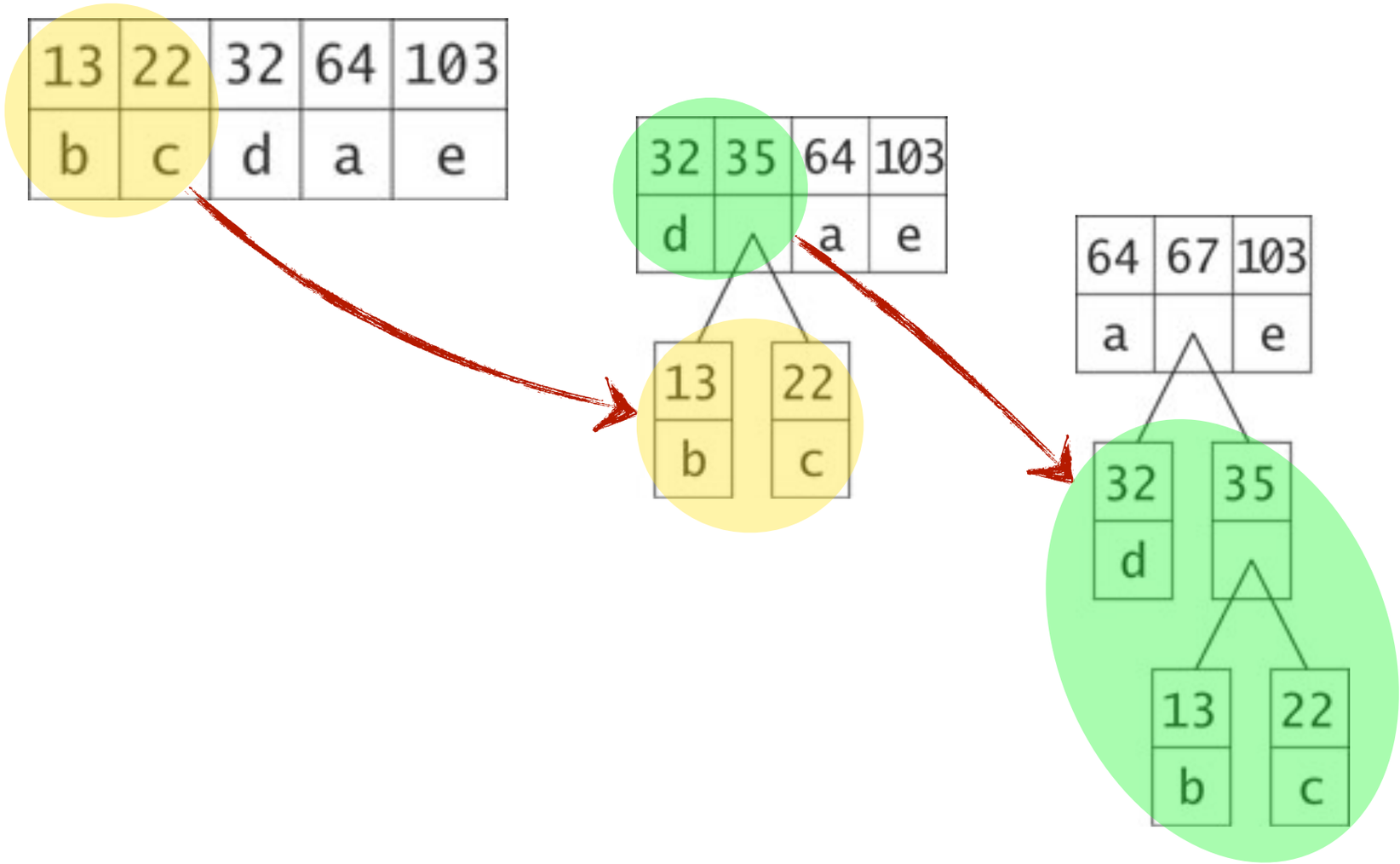
Huffmanträd: Exempel



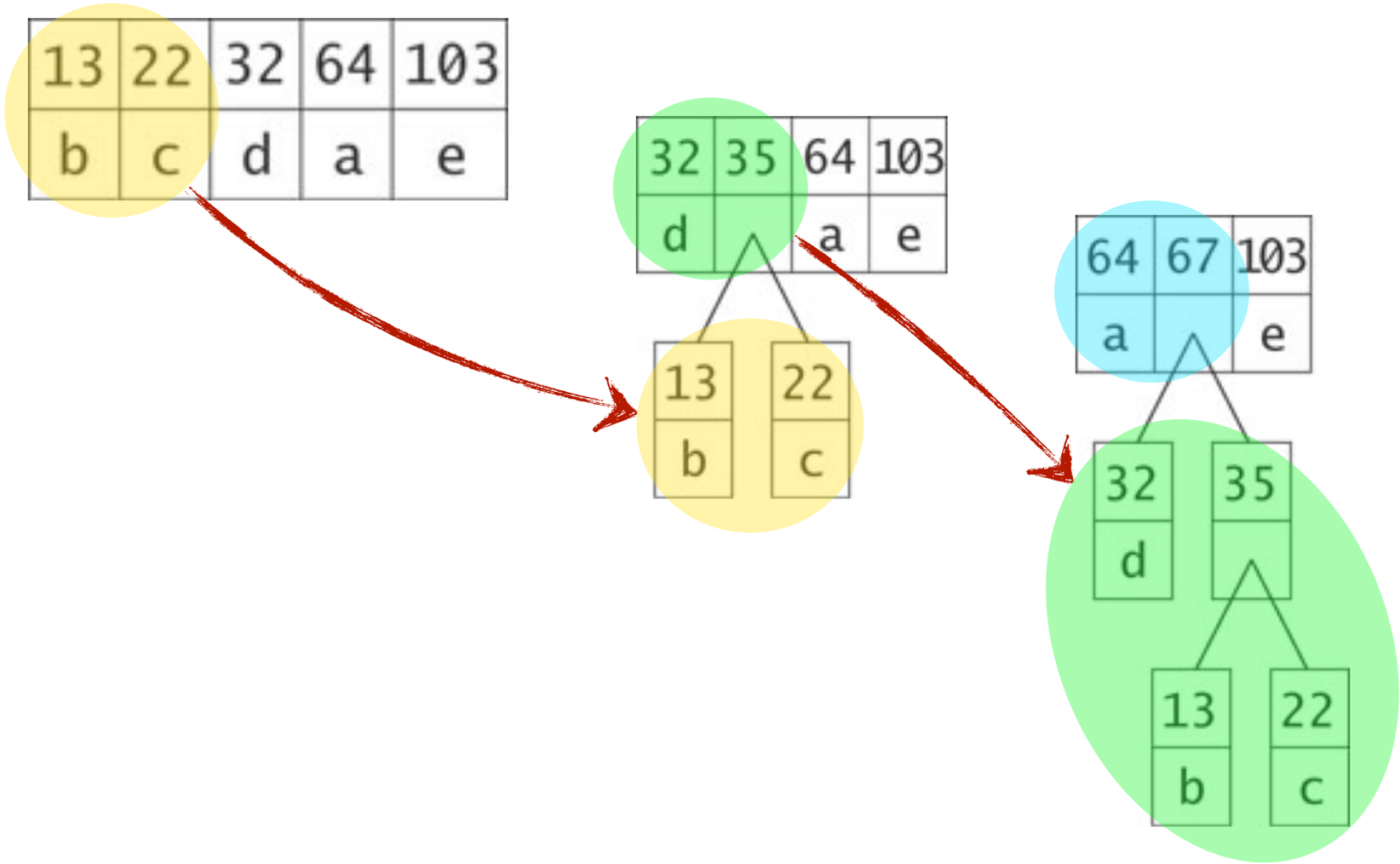
Huffmanträd: Exempel



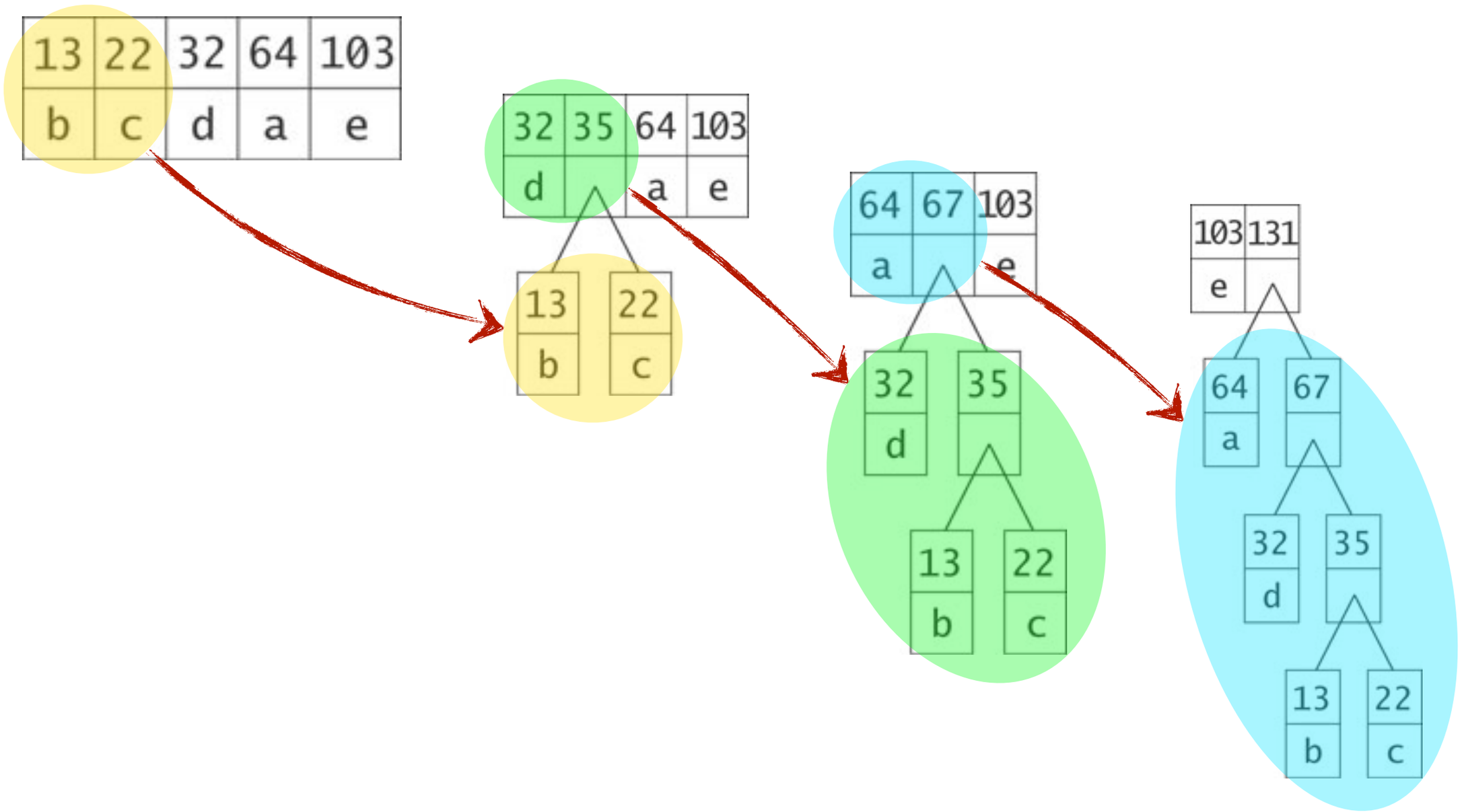
Huffmanträd: Exempel



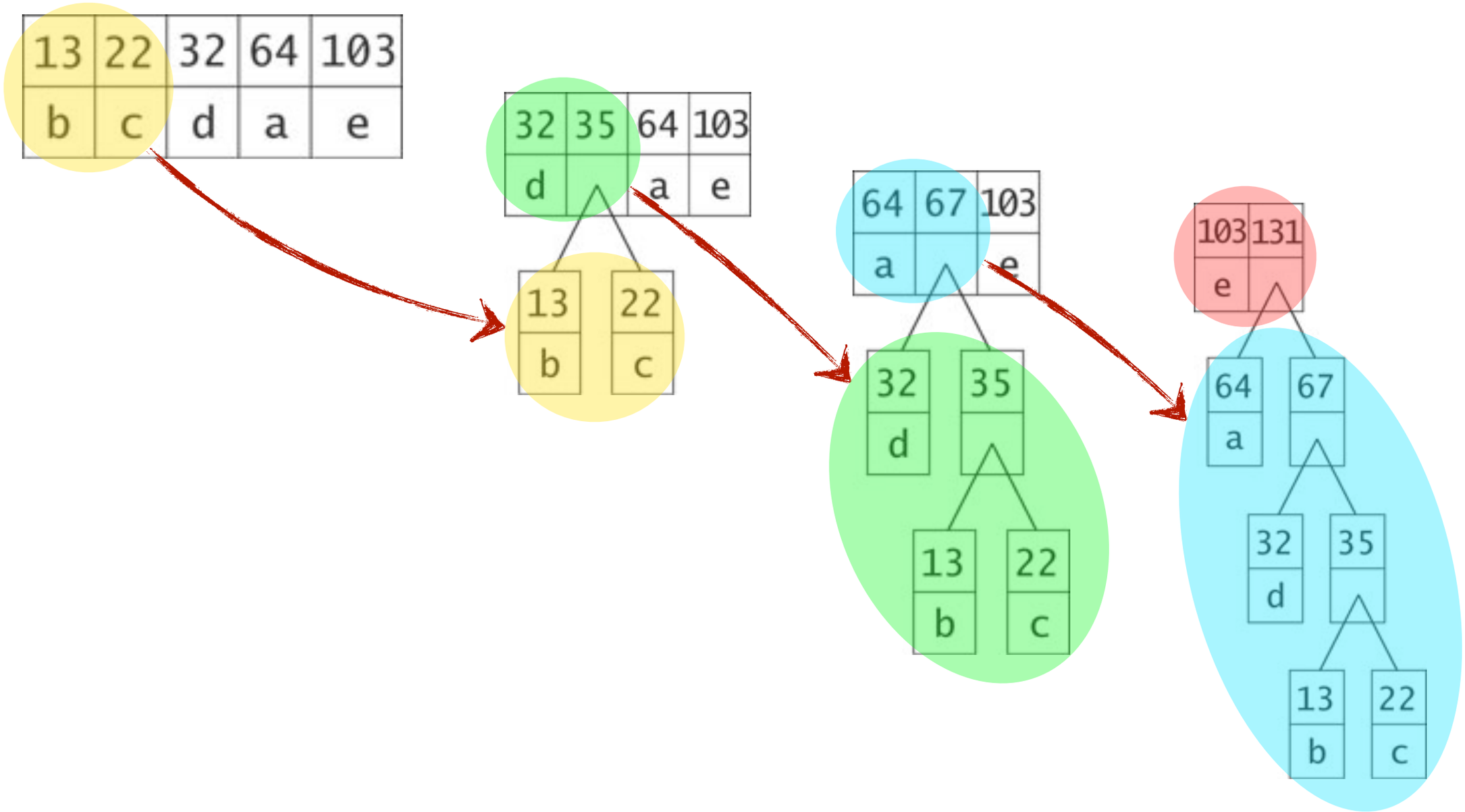
Huffmanträd: Exempel



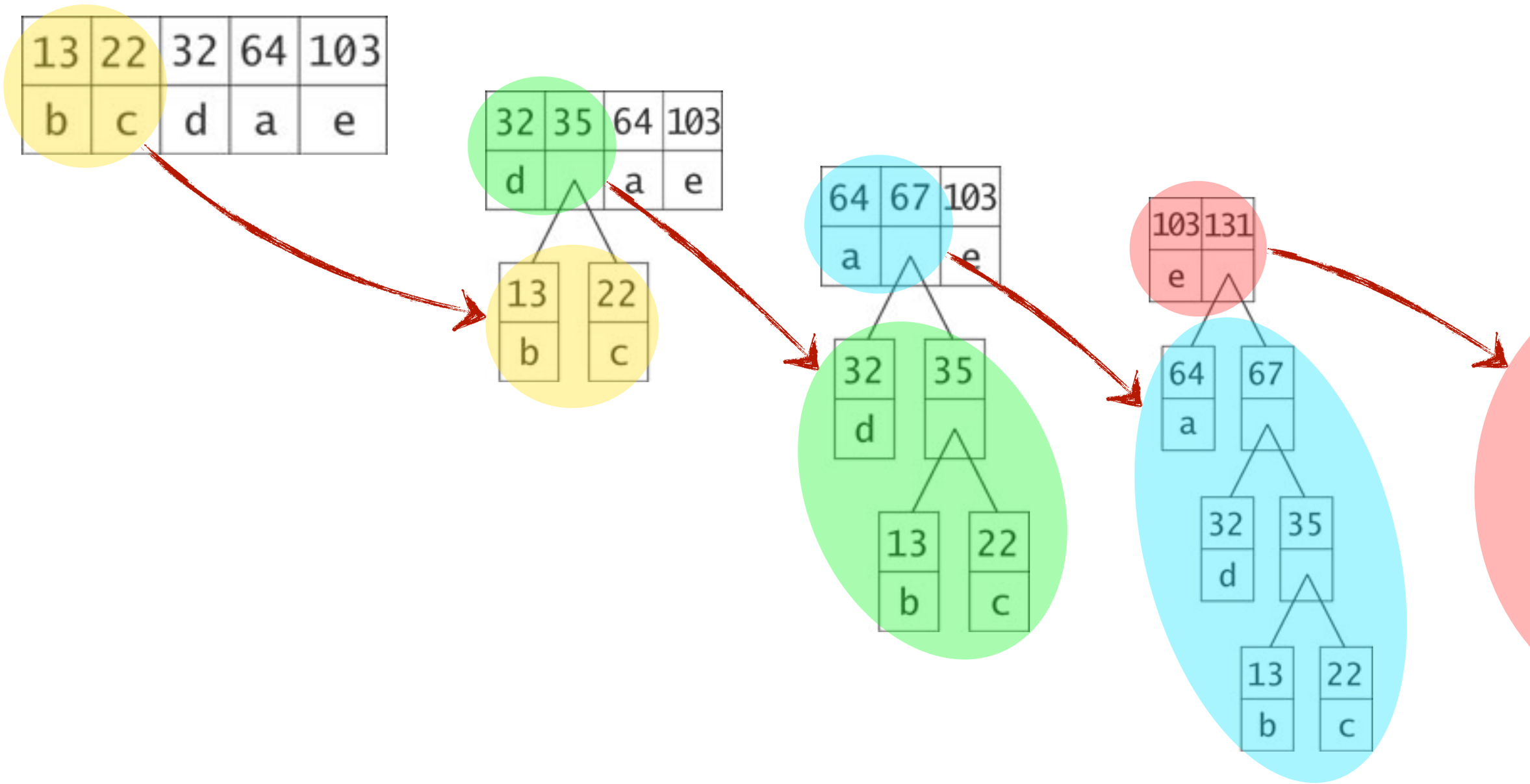
Huffmanträd: Exempel



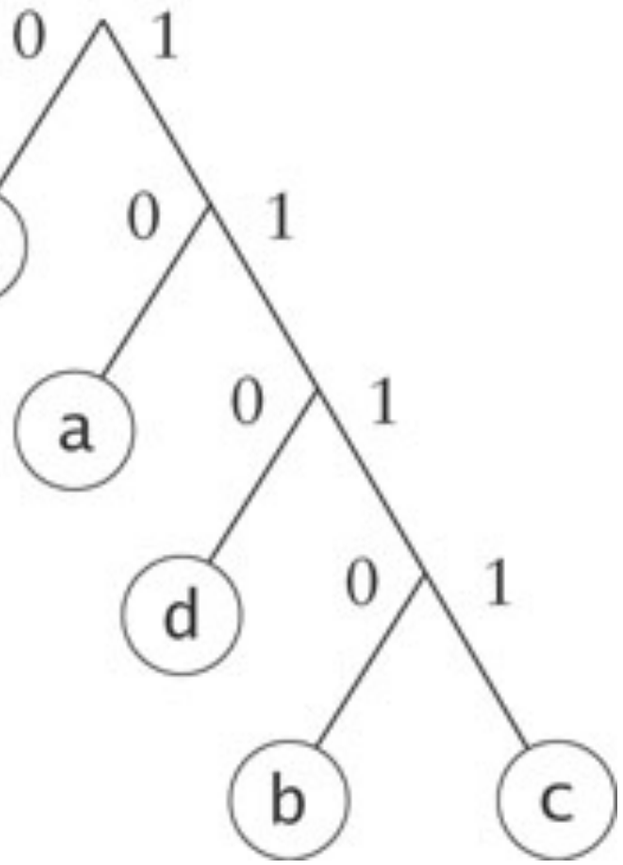
Huffmanträd: Exempel



Huffmanträd: Exempel



Huffmanträd: Slutresultatet



Symbol	Code
a	10
b	1110
c	1111
d	110
e	0