

Introduction to Algorithms and Data Structures

Tutorial 3

Oliver Soeser and John Longley

University of Edinburgh

22 October 2026

Scan to check-in



Expires at 22/10/2026 18:10

Q1: List implementation via *Extensible Array*

Initial array capacity of 1000. Increase by 100 each time an expansion is needed.

Append operator: adds one single element at the right hand of the array.

Copying operator: copies a single element from one array to another.

Q1: List implementation via *Extensible Array*

Initial array capacity of 1000. Increase by 100 each time an expansion is needed.

Append operator: adds one single element at the right hand of the array.

Copying operator: copies a single element from one array to another.

(a) Number of **copyings** for **5000** appends.

- ▶ When do we apply **append**?
- ▶ What about **copying**?

Array size: $1000 \rightarrow 1100 \rightarrow 1200 \rightarrow \dots \rightarrow 4900 \rightarrow 5000$

Q1: List implementation via *Extensible Array*

Initial array capacity of 1000. Increase by 100 each time an expansion is needed.

Append operator: adds one single element at the right hand of the array.

Copying operator: copies a single element from one array to another.

(a) Number of **copyings** for **5000** appends.

- ▶ When do we apply **append**?
- ▶ What about **copying**?

Array size: 1000 $\rightarrow$ 1100 $\rightarrow$ 1200 $\rightarrow$... $\rightarrow$ 4900 $\rightarrow$ 5000

$$100\left(\sum_{i=1}^{49} i - \sum_{i=1}^9 i\right) = 100((49 \times 50)/2 - (9 \times 10)/2)$$

$$100(1225 - 45) = 11800 \text{ copyings}$$

Q1: List implementation via *Extensible Array*

(b) Number of copying for n appends.

We perform an expansion on the $(1000 + 100j + 1)$ th append.

For each $j = 0, 1, \dots$, we copy the $1000 + 100j$ elements already present.

Q1: List implementation via *Extensible Array*

(b) *Number of copying for n appends.*

We perform an expansion on the $(1000 + 100j + 1)$ th append.

For each $j = 0, 1, \dots$, we copy the $1000 + 100j$ elements already present.

Summing them all up:

$$\sum_{i=10}^{\lfloor (n-1)/100 \rfloor} 100i = 50 \lfloor (n-1)/100 \rfloor \lfloor (n-1)/100 + 1 \rfloor - 4500$$

Q1: List implementation via *Extensible Array*

(b) *Number of copying for n appends.*

We perform an expansion on the $(1000 + 100j + 1)$ th append.

For each $j = 0, 1, \dots$, we copy the $1000 + 100j$ elements already present.

Summing them all up:

$$\sum_{i=10}^{\lfloor (n-1)/100 \rfloor} 100i = 50 \lfloor (n-1)/100 \rfloor \lfloor (n-1)/100 + 1 \rfloor - 4500$$

Asymptotically: $\Theta(n^2)$

Amortized cost: $\Theta(n)$

Q1: List Implementation via *Extensible Array*

*(c) Contraction policy: if the current array capacity is **2000** or more, and the proportion of the array in use dips below **0.5**, move the contents to an array of half the size.*

What bad behaviour might result from this policy?

Suggest a better contraction policy.

- ▶ What happens when we append **1001** elements?
- ▶ What if we follow up with two deletes and then two appends?

Q1: List Implementation via *Extensible Array*

*(c) Contraction policy: if the current array capacity is **2000** or more, and the proportion of the array in use dips below **0.5**, move the contents to an array of half the size.*

What bad behaviour might result from this policy?

Suggest a better contraction policy.

- ▶ What happens when we append **1001** elements?
- ▶ What if we follow up with two deletes and then two appends?

After 4 list operators we would have done **1999** copyings! Bad idea.

Q1: List Implementation via *Extensible Array*

(c) *Contraction policy*: if the current array capacity is **2000** or more, and the proportion of the array in use dips below **0.5**, move the contents to an array of half the size.

What bad behaviour might result from this policy?

Suggest a better contraction policy.

- ▶ What happens when we append **1001** elements?
- ▶ What if we follow up with two deletes and then two appends?

After 4 list operators we would have done **1999** copyings! Bad idea.

Improvements: use dips < 0.25 before contracting by 0.5.

At least $n/2 - 1$ list operations before the next expansion/contraction.

Amortized $\Theta(1)$ cost per operation.

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[47]
Bucket 3:	[]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[47]
Bucket 3:	[93]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[17, 47]
Bucket 3:	[93]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[17, 47]
Bucket 3:	[143, 93]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[777,17, 47]
Bucket 3:	[143,93]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

Q2: Bucket-style Hash Tables

(a) Perform the following sequence of set operations in a hash table of size 10 using the hash function: $h(n) = n \bmod 10$:

insert(47), insert(93), insert(17), insert(143), insert(777), contains(93), contains(7)

- ▶ How many non-empty buckets do we have by the end ?
- ▶ How does the bucket list look like after 5 operators?

Bucket 1:	[]	Bucket 6:	[]
Bucket 2:	[]	Bucket 7:	[777,17, 47]
Bucket 3:	[143,93]	Bucket 8:	[]
Bucket 4:	[]	Bucket 9:	[]
Bucket 5:	[]	Bucket 0:	[]

contains(93): look in bucket 3 $\rightarrow$ *True*

contains(7): look in bucket 7 $\rightarrow$ *False*

Q2: Probing in Hash Tables

We can represent such sets by storing the elements within the hash table itself, using **probing** to resolve collisions. We may use -1 to indicate a blank entry in the table. We use the **hash-probe** function:

$$g(n, i) = (n + (i \times F(n)) \bmod 10$$

where F is some magical function such that:

$$F(47) = 1, F(93) = 3, F(17) = 3, F(143) = 7, F(777) = 3, F(7) = 1$$

(b) What happens with the given hash/probe scheme?

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe
- **insert(93):** slot 3 is free

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	93	-1	-1	-1	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe
- **insert(93):** slot 3 is free
- **insert(17):** 7 is taken. $F(17) = 3$ so $g(17, 1) = 0$

0	1	2	3	4	5	6	7	8	9
17	-1	-1	93	-1	-1	-1	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe
- **insert(93):** slot 3 is free
- **insert(17):** 7 is taken. $F(17) = 3$ so $g(17, 1) = 0$
- **insert(143):** 3 is taken. $F(143) = 7$, the successive probe values are 3, 0, 7, 4, 1, ... $g(143, 3) = 4$ is free.

0	1	2	3	4	5	6	7	8	9
17	-1	-1	93	143	-1	-1	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe
- **insert(93):** slot 3 is free
- **insert(17):** 7 is taken. $F(17) = 3$ so $g(17, 1) = 0$
- **insert(143):** 3 is taken. $F(143) = 7$, the successive probe values are 3, 0, 7, 4, 1, ... $g(143, 3) = 4$ is free.
- **insert(777):** 7 is taken. Have $F(777) = C(1) = 3$, so probes are 7, 0, 3, 6, 9, ...
. Of these, 6 is the first free one.

0	1	2	3	4	5	6	7	8	9
17	-1	-1	93	143	-1	777	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47)**: table empty, successful probe
- **insert(93)**: slot 3 is free
- **insert(17)**: 7 is taken. $F(17) = 3$ so $g(17, 1) = 0$
- **insert(143)**: 3 is taken. $F(143) = 7$, the successive probe values are 3, 0, 7, 4, 1, ... $g(143, 3) = 4$ is free.
- **insert(777)**: 7 is taken. Have $F(777) = C(1) = 3$, so probes are 7, 0, 3, 6, 9, ...
. Of these, 6 is the first free one.
- **contains(93)**: first probe 3, True

0	1	2	3	4	5	6	7	8	9
17	-1	-1	93	143	-1	777	47	-1	-1

Q2: Probing in Hash Tables

- **insert(47):** table empty, successful probe
- **insert(93):** slot 3 is free
- **insert(17):** 7 is taken. $F(17) = 3$ so $g(17, 1) = 0$
- **insert(143):** 3 is taken. $F(143) = 7$, the successive probe values are 3,0,7,4,1,... $g(143, 3) = 4$ is free.
- **insert(777):** 7 is taken. Have $F(777) = C(1) = 3$, so probes are 7,0,3,6,9,...
. Of these, 6 is the first free one.
- **contains(93):** first probe 3, True
- **contains(7):** $C_7 = C(0) = 1$ False

0	1	2	3	4	5	6	7	8	9
17	-1	-1	93	143	-1	777	47	-1	-1

Q2: Probing in Hash Tables

(c)* *Why not $C(-) = 5$?*

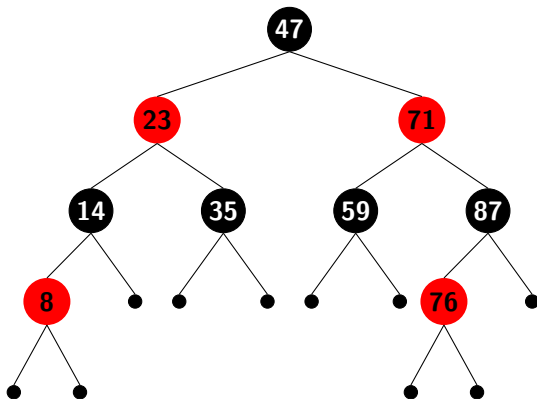
Q3: Red-Black Tree Operations

Red-Black Tree Rules:

- ▶ Every single node in the tree must be either **red** or **black**.
- ▶ The root node of the tree must always be **black**.
- ▶ Two **red** nodes can never appear consecutively, one after another.
- ▶ Every branch path — the path from a root node to a leaf node — must pass through the exact same number of **black** nodes.

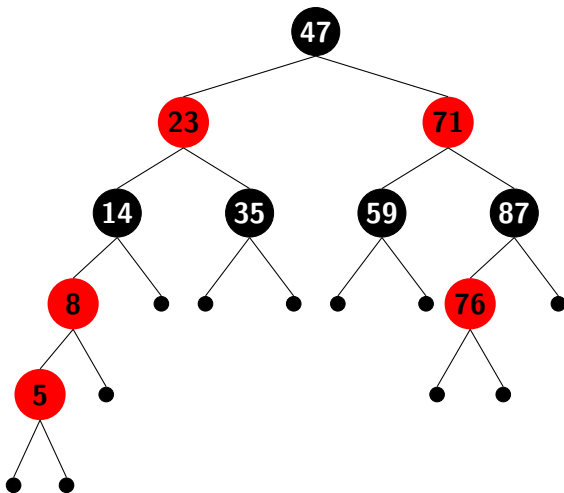
Q3: Red-Black Tree Operations

(a) What happens when the following are performed in sequence: *insert(5)*, *insert(17)*?



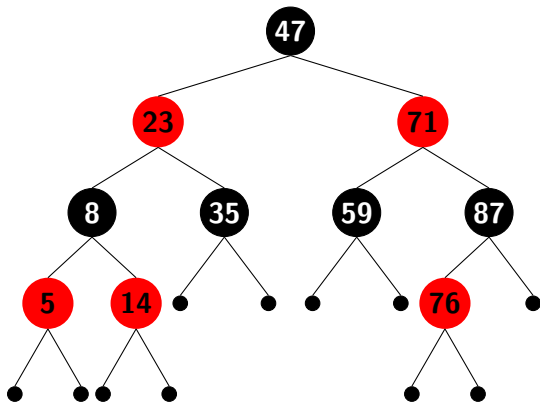
Q3: Red-Black Tree Operations

insert(5)



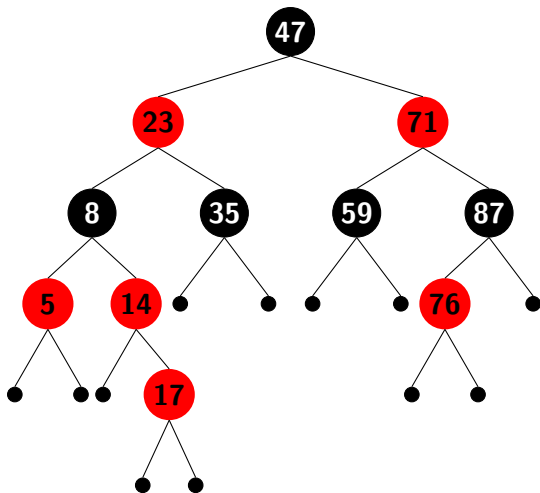
Q3: Red-Black Tree Operations

Rebalanced tree!



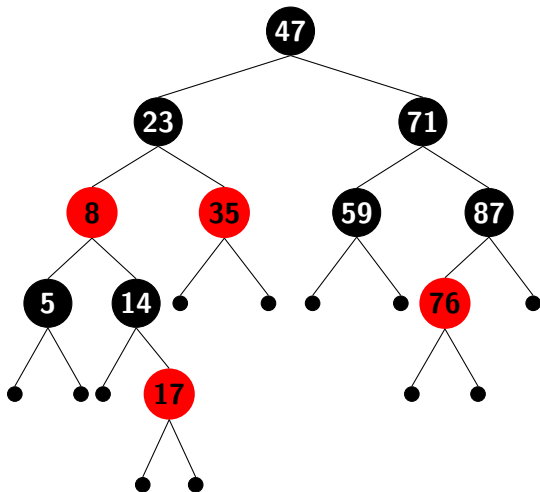
Q3: Red-Black Tree Operations

insert(17)



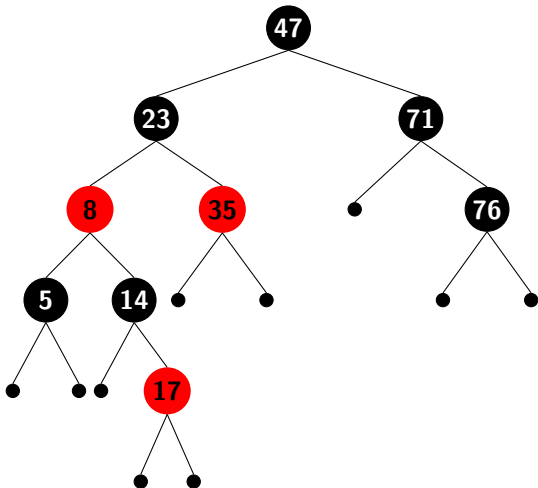
Q3: Red-Black Tree Operations

Rebalanced tree!



Q3: Red-Black Tree Operations

(a) What happens when the following are performed in sequence: *insert*(5), *insert*(17), *delete*(87), ***delete***(59)?



Q3: Red-Black Tree Operations

(b) How would we use red-black trees to implement lists?

Q3: Red-Black Tree Operations

(b) How would we use red-black trees to implement lists?

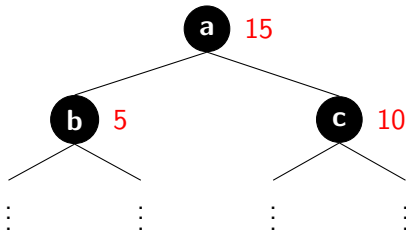
Idea: each internal node should store, in addition to its key value, the number of list elements in the sub-tree rooted at that node. It does require a tree update for each operation.

Q3: Red-Black Tree Operations

(b) *How would we use red-black trees to implement lists?*

Idea: each internal node should store, in addition to its key value, the number of list elements in the sub-tree rooted at that node. It does require a tree update for each operation.

- ▶ How would **get(i)** operation work?
- ▶ Using the graph below explain how **get(8)** would work.



Scan to check-in



Expires at 22/10/2026 18:10