Searching

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

McMaster University

Winter 2024

Recap

- Fundamental analysis of algorithms and data structures.
  Correctness, complexity (average, amortized, expected), recurrences, recurrence trees.
- Basic algorithms.
  LinearSearch, BinarySearch, InsertionSort, SelectionSort.
- Collection types.
  Bag, stack, queue, double-ended queue, priority queue.
- Data structures.
  Ring buffer, linked lists, dynamic arrays, trees and heaps.
- Fast data analysis algorithms.
  MergeSort, Merge, QuickSort, Partition, Select, HeapSort.

Next: Sets and dictionaries

Fundamental tools in the arsenal of programmers.

Most-commonly implemented using either *search trees* or *hash tables*:
vastly different classes of data structures with vastly different properties.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

$\textsc{Add}(S, v)$	add value v to the set S;
$\textsc{Delete}(S, v)$	remove value v from the set S;
$\textsc{Contains}(S, v)$	return `true` if set S holds value v (if $v \in S$);
$\textsc{Size}(S)$	returns the number of values in S.

In addition, one can iterate over the values in S.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

$\textsc{Add}(S, v)$	add value v to the set S;
$\textsc{Delete}(S, v)$	remove value v from the set S;
$\textsc{Contains}(S, v)$	return `true` if set S holds value v (if $v \in S$);
$\textsc{Size}(S)$	returns the number of values in S.

In addition, one can iterate over the values in S.

Ordered Set: a set that provides *ordered iteration*:
one can iterate over the values in S in sorted order.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

$\textsc{Add}(S, v)$	add value v to the set S;
$\textsc{Delete}(S, v)$	remove value v from the set S;
$\textsc{Contains}(S, v)$	return `true` if set S holds value v (if $v \in S$);
$\textsc{Size}(S)$	returns the number of values in S.

In addition, one can iterate over the values in S.

Ordered Set: a set that provides *ordered iteration*:
one can iterate over the values in S in sorted order.

Sets and Contains

- We typically write $v \in S$ instead of $\textsc{Contains}(S, v)$.
- We typically write $v \notin S$ instead of $\neg\textsc{Contains}(S, v)$.
- We typically write $|S|$ instead of $\textsc{Size}(S)$.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

ADD(S, v)	add value v to the set S;
DELETE(S, v)	remove value v from the set S;
CONTAINS(S, v)	return `true` if set S holds value v (if $v \in S$);
SIZE(S)	returns the number of values in S.

In addition, one can iterate over the values in S.

Ordered Set: a set that provides *ordered iteration*:
one can iterate over the values in S in sorted order.

Sets often also support *set operations* such as

UNION(S_1, S_2)	compute the set $S_1 \cup S_2$.
INTERSECT(S_1, S_2)	compute the set $S_1 \cap S_2$.
DIFFERENCE(S_1, S_2)	compute the set $S_1 \setminus S_2$.

Collection types: Sets

Set: collection to which values can be added and removed,
and in which one can *check value membership*.

ADD(S, v)	add value v to the set S;
DELETE(S, v)	remove value v from the set S;
CONTAINS(S, v)	return `true` if set S holds value v (if $v \in S$);
SIZE(S)	returns the number of values in S.

In addition, one can iterate over the values in S.

Ordered Set: a set that provides *ordered iteration*:
one can iterate over the values in S in sorted order.

Sets often also support *set operations* such as

UNION(S_1, S_2)	compute the set $S_1 \cup S_2$.
INTERSECT(S_1, S_2)	compute the set $S_1 \cap S_2$.
DIFFERENCE(S_1, S_2)	compute the set $S_1 \setminus S_2$.

We will *not* focus on set operations.

A use-case for sets

Algorithm Dedup(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

words := an empty set.
for all words w from *stream* **do**
    if $w \notin$ *words* **then**
        ADD(*words*, w).
        Output w.

Result: output each unique word in *stream* once.

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

words

Algorithm Dedup(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

Example: Dedup("a word is just a word or is it not just a word")

A use-case for sets

words
a

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

words := an empty set.
for all words *w* from *stream* **do**
  if $w \notin words$ **then**
    ADD(*words*, *w*).
    Output *w*.

Result: output each unique word in *stream* once.

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

1: *words* := an empty set.
2: **for all** words *w* from *stream* **do**
3:     **if** $w \notin words$ **then**
4:         ADD(*words*, *w*).
5:         Output *w*.

Result: output each unique word in *stream* once.

words
a
word
is

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

1: *words* := an empty set.
2: **for all** words w from *stream* **do**
3:     **if** $w \notin words$ **then**
4:         ADD(*words*, w).
5:         Output w.

Result: output each unique word in *stream* once.

words
a
word
is
just

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):
Input: *stream* is a sequence of words.
1: *words* := an empty set.
2: **for all** words w from *stream* **do**
3: **if** $w \notin words$ **then**
4: ADD($words$, w).
5: Output w.
Result: output each unique word in *stream* once.

words
a
word
is
just

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word
is
just

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):
Input: *stream* is a sequence of words.
1: *words* := an empty set.
2: **for all** words w from *stream* **do**
3: **if** $w \notin words$ **then**
4: ADD(*words*, w).
5: Output w.
Result: output each unique word in *stream* once.

words
a
word
is
just
or

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word
is
just
or

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):
Input: *stream* is a sequence of words.
1: *words* := an empty set.
2: **for all** words *w* from *stream* **do**
3: **if** $w \notin words$ **then**
4: ADD(*words*, *w*).
5: Output *w*.
Result: output each unique word in *stream* once.

words
a
word
is
just
or
it

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        ADD(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word
is
just
or
it
not

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        ADD(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word
is
just
or
it
not

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        ADD(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

words
a
word
is
just
or
it
not

Example: DEDUP(“a word is just a word or is it not just a word”)

A use-case for sets

Algorithm DEDUP(*stream*):
Input: *stream* is a sequence of words.
1: *words* := an empty set.
2: **for all** words w from *stream* **do**
3: **if** $w \notin words$ **then**
4: ADD(*words*, w).
5: Output w.
Result: output each unique word in *stream* once.

words
a
word
is
just
or
it
not

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):
Input: *stream* is a sequence of words.
1: *words* := an empty set.
2: **for all** words w from *stream* **do**
3: **if** $w \notin words$ **then**
4: ADD($words, w$).
5: Output w.
Result: output each unique word in *stream* once.

words
a
word
is
just
or
it
not

Unordered.

Example: DEDUP("a word is just a word or is it not just a word")

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

Runtime complexity

A use-case for sets

Algorithm DEDUP(*stream*):

Input: *stream* is a sequence of words.

```
words := an empty set.
for all words w from stream do
    if w ∉ words then
        Add(words, w).
        Output w.
```

Result: output each unique word in *stream* once.

Runtime complexity

Let $N = |stream|$ be the *number of words* in *stream*.

Let $U = |output|$ be the *number of unique words* in *stream* written to the output.

A use-case for sets

Algorithm DEDUP(*stream*)**:**

Input: *stream* is a sequence of words.

words := an empty set.
for all words w from *stream* **do**
    if $w \notin words$ **then**
        ADD($words, w$).
        Output w.

Result: output each unique word in *stream* once.

N CONTAINS operations.
U ADD operations.

Runtime complexity

Let $N = |stream|$ be the *number of words* in *stream*.

Let $U = |output|$ be the *number of unique words* in *stream* written to the output.

Collection types: Dictionary (or map, symbol table, …)

Dictionary: collection to which (key, value)-pairs (*kv-pairs*) can be added and removed, and in which one can *look-up and modify values* by key.

Collection types: Dictionary (or map, symbol table, ...)

Dictionary: collection to which (key, value)-pairs (*kv-pairs*) can be added and removed, and in which one can *look-up and modify values* by key.

PUT(D, ($k \mapsto v$))	add kv-pair $k \mapsto v$ to the dictionary D;
GET(D, k)	return the value v for kv-pair $k \mapsto v$ in dictionary D;
DELETE(D, k)	remove the kv-pair for key k from dictionary D;
CONTAINS(D, k)	return `true` if dictionary D holds a kv-pair for key k;
SIZE(D)	returns the number of kv-pairs in D.

In addition, one can iterate over the kv-pairs in D.

Collection types: Dictionary (or map, symbol table, ...)

Dictionary: collection to which (key, value)-pairs (*kv-pairs*) can be added and removed, and in which one can *look-up and modify values* by key.

PUT(D, $(k \mapsto v)$)	add kv-pair $k \mapsto v$ to the dictionary D;
GET(D, k)	return the value v for kv-pair $k \mapsto v$ in dictionary D;
DELETE(D, k)	remove the kv-pair for key k from dictionary D;
CONTAINS(D, k)	return `true` if dictionary D holds a kv-pair for key k;
SIZE(D)	returns the number of kv-pairs in D.

In addition, one can iterate over the kv-pairs in D.

Ordered Dictionary: a dictionary that provides *ordered iteration*: one can iterate over the kv-pairs in D in sorted order (on key).

Collection types: Dictionary (or map, symbol table, ...)

Dictionary: collection to which (key, value)-pairs (*kv-pairs*) can be added and removed, and in which one can *look-up and modify values* by key.

PUT(D, $(k \mapsto v)$) add kv-pair $k \mapsto v$ to the dictionary D;

GET(D, k) return the value v for kv-pair $k \mapsto v$ in dictionary D;

DELETE(D, k) remove the kv-pair for key k from dictionary D;

CONTAINS(D, k) return `true` if dictionary D holds a kv-pair for key k;

SIZE(D) returns the number of kv-pairs in D.

In addition, one can iterate over the kv-pairs in D.

Ordered Dictionary: a dictionary that provides *ordered iteration*: one can iterate over the kv-pairs in D in sorted order (on key).

GETing and modifying values

- We typically write $D[k]$ instead of GET(D, k).
- We typically write $D[k] := v$ to change the value of a kv-pair in D.
- We typicaly write $|D|$ instead of SIZE(D).

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

counts := an empty dictionary.
for all words w from *stream* **do**
    if ¬Contains(*counts*, w) **then**
        Put(*counts*, $(w \mapsto 1)$).
    else
        $counts[w] := counts[w] + 1$.
output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

A use-case for dictionaries

Algorithm WordCount(*stream*):
Input: *stream* is a sequence of words.

counts := an empty dictionary.
for all words w from *stream* **do**
    if $\neg$Contains(*counts*, w) **then**
        Put(*counts*, $(w \mapsto 1)$).
    else
        $counts[w] := counts[w] + 1$.
output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

counts

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3:     **if** $\neg$Contains(*counts*, w) **then**
4:         Put(*counts*, $(w \mapsto 1)$).
5:     **else**
6:         $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts
(a $\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3:     **if** $\neg$Contains(*counts*, w) **then**
4:         Put(*counts*, $(w \mapsto 1)$).
5:     **else**
6:         $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts		
(	a	$\mapsto$ 1)
(	word	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts		
(a	$\mapsto$	1)
(word	$\mapsto$	1)
(is	$\mapsto$	1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):
Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words *w* from *stream* **do**
3: **if** $\neg$Contains(*counts*, *w*) **then**
4: Put(*counts*, ($w \mapsto 1$)).
5: **else**
6: *counts*[*w*] := *counts*[*w*] + 1.
7: output each pair ($w \mapsto$ *counts*[*w*]) in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts		
(	a	$\mapsto$ 1)
(	word	$\mapsto$ 1)
(	is	$\mapsto$ 1)
(	just	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):
Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts
(a $\mapsto$ 2)
(word $\mapsto$ 1)
(is $\mapsto$ 1)
(just $\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3: **if** $\neg$Contains(*counts*, w) **then**
4: Put(*counts*, $(w \mapsto 1)$).
5: **else**
6: $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts	
(a	$\mapsto$ 2)
(word	$\mapsto$ 2)
(is	$\mapsto$ 1)
(just	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts
(a $\mapsto$ 2)
(word $\mapsto$ 2)
(is $\mapsto$ 1)
(just $\mapsto$ 1)
(or $\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):
Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3: **if** $\neg$Contains(*counts*, w) **then**
4: Put(*counts*, $(w \mapsto 1)$).
5: **else**
6: $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts	
(a	$\mapsto$ 2)
(word	$\mapsto$ 2)
(is	$\mapsto$ 2)
(just	$\mapsto$ 1)
(or	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts		
(	a	$\mapsto$ 2)
(	word	$\mapsto$ 2)
(	is	$\mapsto$ 2)
(	just	$\mapsto$ 1)
(	or	$\mapsto$ 1)
(	it	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WORDCOUNT(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬CONTAINS(counts, w) then
        PUT(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts
(a $\mapsto$ 2)
(word $\mapsto$ 2)
(is $\mapsto$ 2)
(just $\mapsto$ 1)
(or $\mapsto$ 1)
(it $\mapsto$ 1)
(not $\mapsto$ 1)

Example: WORDCOUNT(“a word is just a word or is it not just a word”)

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

counts := an empty dictionary.

for all words w from *stream* **do**

  if $\neg$Contains(*counts*, w) **then**

    Put(*counts*, $(w \mapsto 1)$).

  else

    $counts[w] := counts[w] + 1$.

output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts		
(	a	$\mapsto$ 2)
(	word	$\mapsto$ 2)
(	is	$\mapsto$ 2)
(	just	$\mapsto$ 2)
(	or	$\mapsto$ 1)
(	it	$\mapsto$ 1)
(	not	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3: **if** $\neg$Contains(*counts*, w) **then**
4: Put(*counts*, $(w \mapsto 1)$).
5: **else**
6: $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts
(a $\mapsto$ 3)
(word $\mapsto$ 2)
(is $\mapsto$ 2)
(just $\mapsto$ 2)
(or $\mapsto$ 1)
(it $\mapsto$ 1)
(not $\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

Result: output the number of occurances of each unique word in *stream*.

counts	
(a	$\mapsto$ 3)
(word	$\mapsto$ 3)
(is	$\mapsto$ 2)
(just	$\mapsto$ 2)
(or	$\mapsto$ 1)
(it	$\mapsto$ 1)
(not	$\mapsto$ 1)

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3:     **if** $\neg$Contains(*counts*, w) **then**
4:         Put(*counts*, $(w \mapsto 1)$).
5:     **else**
6:         *counts*$[w]$:= *counts*$[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

counts
(a ↦ 3)
(word ↦ 3)
(is ↦ 2)
(just ↦ 2)
(or ↦ 1)
(it ↦ 1)
(not ↦ 1)

Output: Unordered.

Example: WordCount("a word is just a word or is it not just a word")

A use-case for dictionaries

Algorithm WORDCOUNT(*stream*):

Input: *stream* is a sequence of words.

1: *counts* := an empty dictionary.
2: **for all** words w from *stream* **do**
3:     **if** $\neg$CONTAINS(*counts*, w) **then**
4:         PUT(*counts*, $(w \mapsto 1)$).
5:     **else**
6:         $counts[w] := counts[w] + 1$.
7: output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

Runtime complexity

Let $N = |stream|$ be the *number of words* in *stream*.

Let $U = |output|$ be the *number of pairs* written to the output.

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

```
counts := an empty dictionary.
for all words w from stream do
    if ¬Contains(counts, w) then
        Put(counts, (w ↦ 1)).
    else
        counts[w] := counts[w] + 1.
output each pair (w ↦ counts[w]) in counts.
```

(for the loop:)
N Contains operations.
U Put operations.
$N - U$ *updates to values*.

Result: output the number of occurances of each unique word in *stream*.

Runtime complexity

Let $N = |stream|$ be the *number of words* in *stream*.

Let $U = |output|$ be the *number of pairs* written to the output.

A use-case for dictionaries

Algorithm WordCount(*stream*):

Input: *stream* is a sequence of words.

counts := an empty dictionary.
for all words w from *stream* **do**
  if $\neg$Contains(*counts*, w) **then**
    Put(*counts*, $(w \mapsto 1)$).
  else
    counts$[w]$:= *counts*$[w] + 1$.
output each pair $(w \mapsto counts[w])$ in *counts*.

Result: output the number of occurances of each unique word in *stream*.

(for lines 2–6:)
N Contains operations.
U Put operations.
$N - U$ *updates to values* ($\Theta(1)$, for “free”).

Runtime complexity

Let $N = |stream|$ be the *number of words* in *stream*.

Let $U = |output|$ be the *number of pairs* written to the output.

After finding a key (e.g., Contains, Get), updating the value is typically $\Theta(1)$.

Dictionaries and sets

Dictionaries and sets are closely related.

Dictionaries and sets

Dictionaries and sets are closely related.

Consider a data structure that can implement a set

In the implementation, we can store extra data v alongside each element k in the set
$\rightarrow$ the pair (k, v) is a kv-pair and we end up with a dictionary!

Dictionaries and sets

Dictionaries and sets are closely related.

Consider a data structure that can implement a set

In the implementation, we can store extra data v alongside each element k in the set
$\longrightarrow$ the pair (k, v) is a kv-pair and we end up with a dictionary!

Consider a data structure that can implement a dictionary

We can use a piece of dummy data for all values $\longrightarrow$ a set (of keys)!

Dictionaries and sets

Dictionaries and sets are closely related.

Consider a data structure that can implement a set

In the implementation, we can store extra data v alongside each element k in the set
$\rightarrow$ the pair (k, v) is a kv-pair and we end up with a dictionary!

Consider a data structure that can implement a dictionary

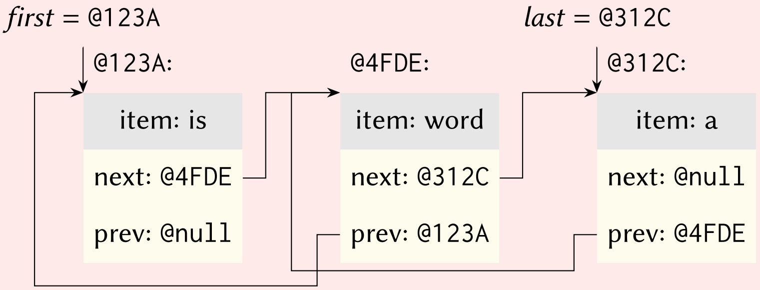
We can use a piece of dummy data for all values $\rightarrow$ a set (of keys)!

To simplify presentation, we focus on the details of data structures that implement *sets*.

Implementing sets with lists

Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

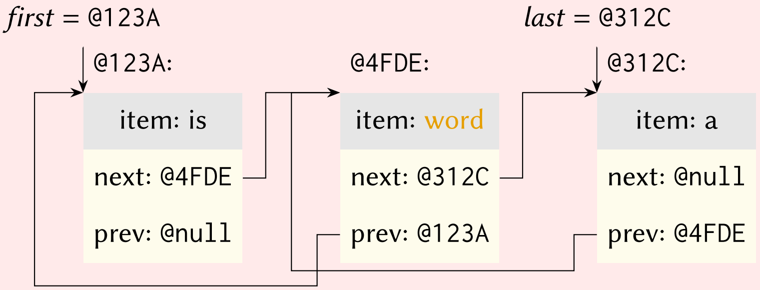


Implementing sets with lists

Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

Contains(S, v) return `true` if one can find a list node n in S with $n.item = v$.



Contains(S, word).

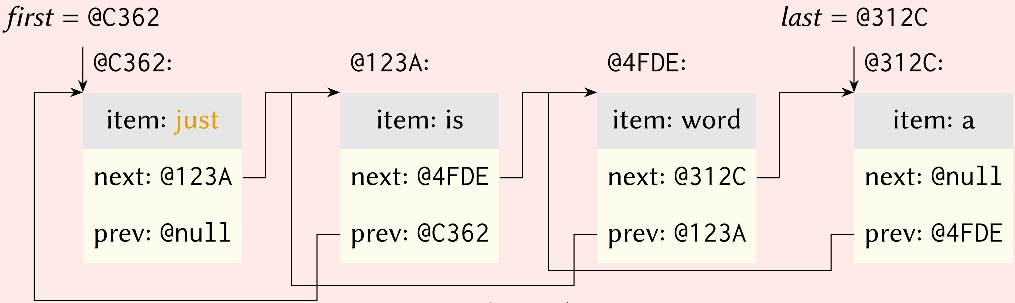
Implementing sets with lists

Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

CONTAINS(S, v) return `true` if one can find a list node n in S with $n.item = v$.

ADD(S, v) (after testing $v \notin S$) PUSHFRONT(S, v).



ADD(S, just).

Implementing sets with lists

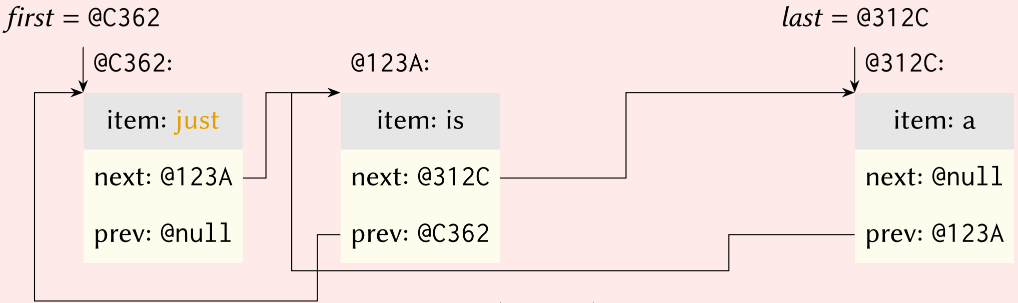
Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

CONTAINS(S, v) return `true` if one can find a list node n in S with $n.item = v$.

ADD(S, v) (after testing $v \notin S$) PUSHFRONT(S, v).

DELETE(S, v) (assuming $v \in S$) search the list node n with $n.item = v$ and remove n.



DELETE(S, word).

Implementing sets with lists

Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

CONTAINS(S, v) return `true` if one can find a list node n in S with $n.item = v$.

ADD(S, v) (after testing $v \notin S$) PUSHFRONT(S, v).

DELETE(S, v) (assuming $v \in S$) search the list node n with $n.item = v$ and remove n.

ADD in $\Theta(1)$!

Worst-case CONTAINS and DELETE traverse the entire list: $\Theta(|S|)$.

Implementing sets with lists

Idea: We can easily *add or remove values* from doubly linked lists.

Let S be a double linked list representing a set

CONTAINS(S, v) return `true` if one can find a list node n in S with $n.item = v$.

ADD(S, v) (after testing $v \notin S$) PUSHFRONT(S, v).

DELETE(S, v) (assuming $v \in S$) search the list node n with $n.item = v$ and remove n.

ADD in $\Theta(1)$!

Worst-case CONTAINS and DELETE traverse the entire list: $\Theta(|S|)$.

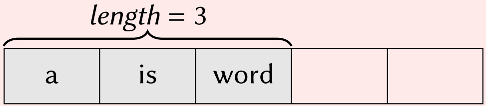
Implementation on top of a dynamic array: similarly bad.

Implementing sets with BinarySearch

Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.



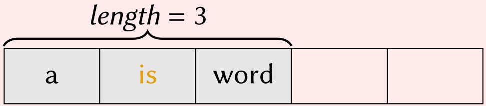
Implementing sets with BinarySearch

Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.



Contains(S, is).

Implementing sets with BinarySearch

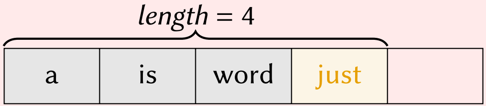
Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.

Add(S, v) (after testing $v \notin S$) PushBack(S, v) and move v to its in-order position.



Add(S, just).

Implementing sets with BinarySearch

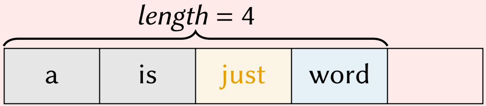
Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.

Add(S, v) (after testing $v \notin S$) PushBack(S, v) and move v to its in-order position (we can use the inner loop of InsertionSort for this).



Add(S, just).

Implementing sets with BinarySearch

Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

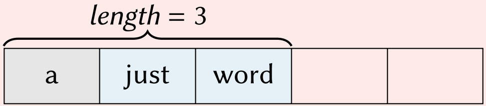
Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.

Add(S, v) (after testing $v \notin S$) PushBack(S, v) and move v to its in-order position (we can use the inner loop of InsertionSort for this).

Delete(S, v) search the position p with $S[p] = v$, move all succeeding values one position to the left, PopBack(S).



Delete(S, is).

Implementing sets with BinarySearch

Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.

Add(S, v) (after testing $v \notin S$) PushBack(S, v) and move v to its in-order position (we can use the inner loop of InsertionSort for this).

Delete(S, v) search the position p with $S[p] = v$, move all succeeding values one position to the left, PopBack(S).

Contains in $\log_2(|S|)$!

Worst-case Add and Delete have to move all values: $\Theta(|S|)$.

Implementing sets with BinarySearch

Idea: We can easily *check value membership* using BinarySearch *if* we maintain the set as an ordered list.

Let S be a dynamic array representing a set

We maintain that S is ordered.

Contains(S, v) return `true` if BinarySearch returns a position.

Add(S, v) (after testing $v \notin S$) PushBack(S, v) and move v to its in-order position (we can use the inner loop of InsertionSort for this).

Delete(S, v) search the position p with $S[p] = v$, move all succeeding values one position to the left, PopBack(S).

Contains in $\log_2(|S|)$!

Worst-case Add and Delete have to move all values: $\Theta(|S|)$.

If we maintain that sets are ordered:

we can use variants of Merge for union, intersection, and difference of sets.

Comparing set implementations

Complexity of DEDUP(*stream*)

N CONTAINS operations with $N = |stream|$.

U ADD operations with U the number of unique words in *stream*.

Comparing set implementations

Complexity of DEDUP(*stream*) and WORDCOUNT(*stream*)

N CONTAINS operations with $N = |stream|$.

U ADD operations with U the number of unique words in *stream*.

Comparing set implementations

Complexity of DEDUP(*stream*) and WORDCOUNT(*stream*)

N CONTAINS operations with $N = |stream|$.

U ADD operations with U the number of unique words in *stream*.

	Doubly-linked list	Sorted dynamic array
N CONTAINS	$\Theta(N \cdot U)$	$\Theta(N \cdot \log U)$
U ADD	$\Theta(U)$	$\Theta(U^2)$
DEDUP	$\Theta(N \cdot U)$	$\Theta(N \cdot \log U + U^2)$.

Comparing set implementations

Complexity of DEDUP(*stream*) and WORDCOUNT(*stream*)

N CONTAINS operations with $N = |stream|$.

U ADD operations with U the number of unique words in *stream*.

	Doubly-linked list	Sorted dynamic array
N CONTAINS	$\Theta(N \cdot U)$	$\Theta(N \cdot \log U)$
U ADD	$\Theta(U)$	$\Theta(U^2)$
DEDUP	$\Theta(N \cdot U)$	$\Theta(N \cdot \log U + U^2)$.

Conclusion

- List implementation (doubly linked, dynamic array): practical only for tiny datasets.
- Sorted dynamic array implementation: only practical if usage of CONTAINS dominates.

Toward a better set data structure

- Linked lists can easily be modified due to usage of pointers.
- BinarySearch can quickly find values even in huge datasets.

Toward a better set data structure

- Linked lists can easily be modified due to usage of pointers.
- BinarySearch can quickly find values even in huge datasets.

Combining the principles of linked lists and BinarySearch

Pointer-based for ease-of-modification.

Branching at each value: we can go *left* (smaller values) or *right* (larger values).

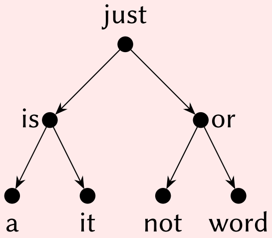
Toward a better set data structure

Combining the principles of linked lists and BinarySearch

Pointer-based for ease-of-modification.

Branching at each value: we can go *left* (smaller values) or *right* (larger values).

A natural fit: a *tree*.



Toward a better set data structure

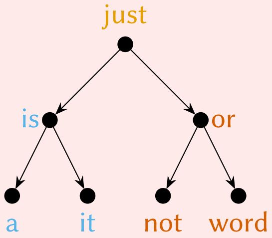
Combining the principles of linked lists and BinarySearch

Pointer-based for ease-of-modification.

Branching at each value: we can go *left* (smaller values) or *right* (larger values).

A natural fit: a *tree* with the *binary search tree* property:

For each node *n*, the nodes in the left subtree all have smaller values; and
the nodes in the right subtree all have larger values.



Toward a better set data structure

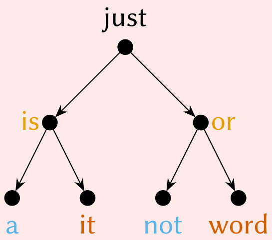
Combining the principles of linked lists and BinarySearch

Pointer-based for ease-of-modification.

Branching at each value: we can go *left* (smaller values) or *right* (larger values).

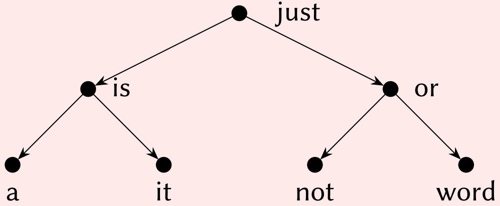
A natural fit: a *tree* with the *binary search tree* property:

For each node *n*, the nodes in the left subtree all have smaller values; and
the nodes in the right subtree all have larger values.



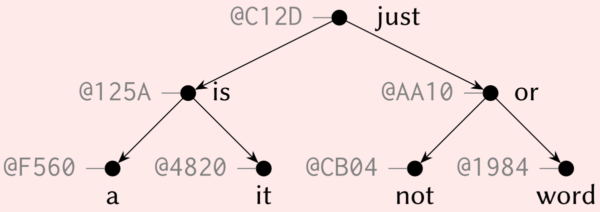
Intermezzo: Binary trees

Binary tree: a data structure that can hold values, each stored in a *binary tree node*.



Intermezzo: Binary trees

Binary tree: a data structure that can hold values, each stored in a *binary tree node*.
Here, we consider a *pointer*-based variant.

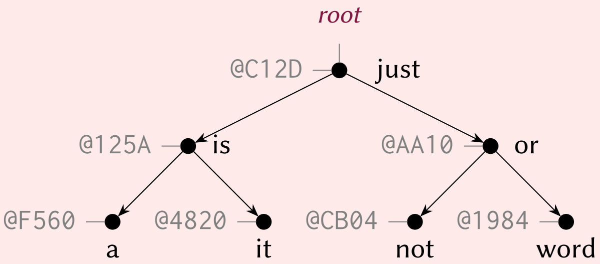


Intermezzo: Binary trees

Binary tree: a data structure that can hold values, each stored in a *binary tree node*.

Here, we consider a *pointer*-based variant.

All values represented by a binary tree are reachable from the *root node*.



Intermezzo: Binary trees

Binary tree: a data structure that can hold values, each stored in a *binary tree node*.

Here, we consider a *pointer*-based variant.

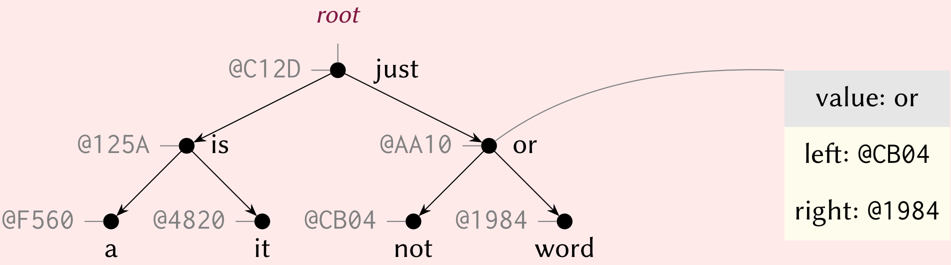
All values represented by a binary tree are reachable from the *root node*.

Each value in a binary tree is stored in a *binary tree node*:

value The value held by the binary tree node.

left A pointer to the *left child* of the node, if any.

right A pointer to the *right child* of the node, if any.



Intermezzo: Binary trees

Binary tree: a data structure that can hold values, each stored in a *binary tree node*.

Here, we consider a *pointer*-based variant.

All values represented by a binary tree are reachable from the *root node*.

Each value in a binary tree is stored in a *binary tree node*:

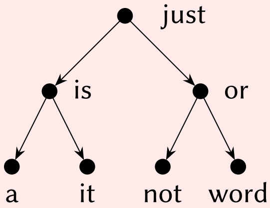
value The value held by the binary tree node.

left A pointer to the *left child* of the node, if any.

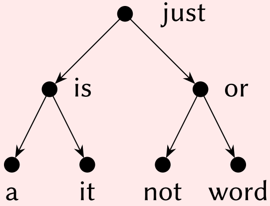
right A pointer to the *right child* of the node, if any.

A binary search tree is represented by a pointer to the *root node*.
If the tree is empty, this pointer is `@null`.

Intermezzo: Traversing binary trees



Intermezzo: Traversing binary trees



Output

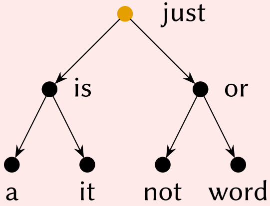
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output

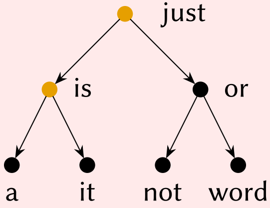
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
if n.left ≠ @null then
    INORDERTRAVERSE(n.left, A).
A(n).
if n.right ≠ @null then
    INORDERTRAVERSE(n.right, A).
```

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output

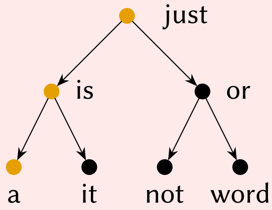
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
if n.left ≠ @null then
    INORDERTRAVERSE(n.left, A).
A(n).
if n.right ≠ @null then
    INORDERTRAVERSE(n.right, A).
```

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output

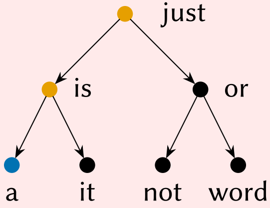
Algorithm InorderTraverse(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    InorderTraverse($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    InorderTraverse($n.right$, A).

InorderTraverse(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a

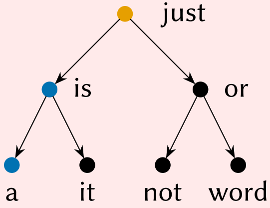
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
if n.left ≠ @null then
    INORDERTRAVERSE(n.left, A).
A(n).
if n.right ≠ @null then
    INORDERTRAVERSE(n.right, A).
```

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is

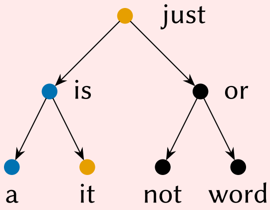
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
  INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
  INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is

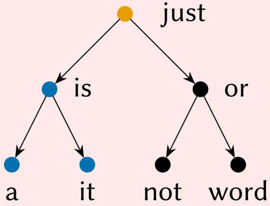
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it

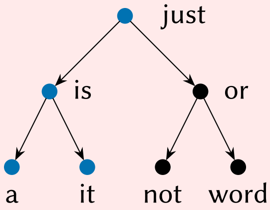
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
  INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
  INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just

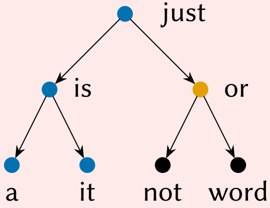
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

1: **if** $n.left \neq$ `@null` **then**
2: INORDERTRAVERSE($n.left$, A).
3: $A(n)$.
4: **if** $n.right \neq$ `@null` **then**
5: INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just

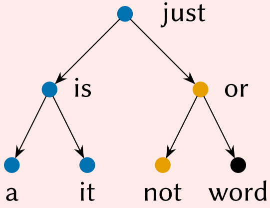
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
if n.left ≠ @null then
    INORDERTRAVERSE(n.left, A).
A(n).
if n.right ≠ @null then
    INORDERTRAVERSE(n.right, A).
```

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just

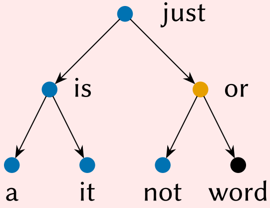
Algorithm InorderTraverse(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    InorderTraverse($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    InorderTraverse($n.right$, A).

InorderTraverse(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just
not

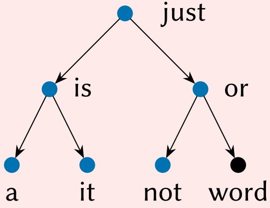
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just
not
or

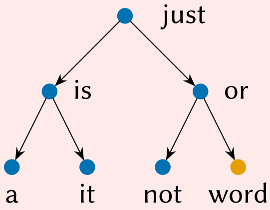
Algorithm InorderTraverse(*n*, action *A*):

Input: *n* is a pointer to a node.

if *n.left* ≠ `@null` **then**
    InorderTraverse(*n.left*, *A*).
A(*n*).
if *n.right* ≠ `@null` **then**
    InorderTraverse(*n.right*, *A*).

InorderTraverse(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just
not
or

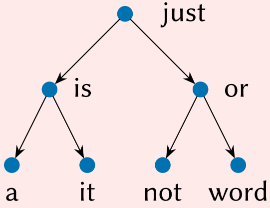
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
if n.left ≠ @null then
    INORDERTRAVERSE(n.left, A).
A(n).
if n.right ≠ @null then
    INORDERTRAVERSE(n.right, A).
```

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
is
it
just
not
or
word

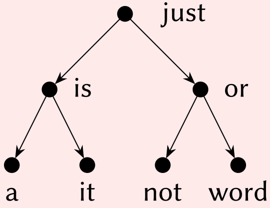
Algorithm INORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    INORDERTRAVERSE($n.left$, A).
$A(n)$.
if $n.right \neq$ `@null` **then**
    INORDERTRAVERSE($n.right$, A).

INORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees

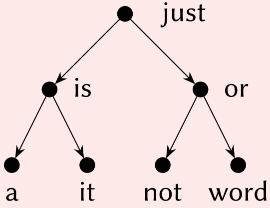


Algorithm PREORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

```
A(n).
if n.left ≠ @null then
    PreorderTraverse(n.left, A).
if n.right ≠ @null then
    PreorderTraverse(n.right, A).
```

Intermezzo: Traversing binary trees



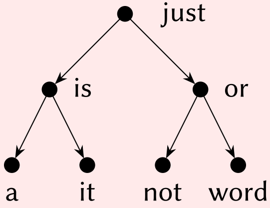
Algorithm PREORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

1: $A(n)$.
2: **if** $n.left \neq$ `@null` **then**
3: PREORDERTRAVERSE($n.left$, A).
4: **if** $n.right \neq$ `@null` **then**
5: PREORDERTRAVERSE($n.right$, A).

PREORDERTRAVERSE(root, "output n.value").

Intermezzo: Traversing binary trees



Output
just
is
a
it
or
not
word

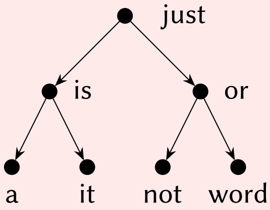
Algorithm PreorderTraverse(n, action A):

Input: n is a pointer to a node.

```
A(n).
if n.left ≠ @null then
    PreorderTraverse(n.left, A).
if n.right ≠ @null then
    PreorderTraverse(n.right, A).
```

PreorderTraverse(root, "output n.value").

Intermezzo: Traversing binary trees

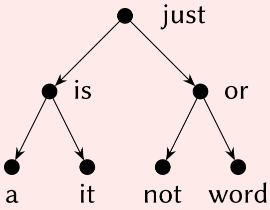


Algorithm PostorderTraverse(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
    PostorderTraverse($n.left$, A).
if $n.right \neq$ `@null` **then**
    PostorderTraverse($n.right$, A).
$A(n)$.

Intermezzo: Traversing binary trees



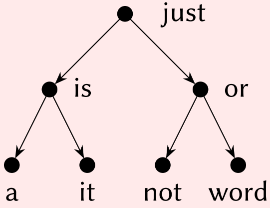
Algorithm PostorderTraverse(n, action A):

Input: n is a pointer to a node.

1: **if** $n.left \neq$ `@null` **then**
2: PostorderTraverse($n.left$, A).
3: **if** $n.right \neq$ `@null` **then**
4: PostorderTraverse($n.right$, A).
5: $A(n)$.

PostorderTraverse(root, "output n.value").

Intermezzo: Traversing binary trees



Output
a
it
is
not
word
or
just

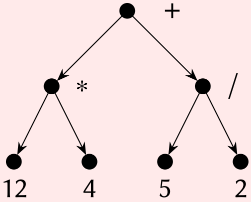
Algorithm POSTORDERTRAVERSE(n, action A):

Input: n is a pointer to a node.

if $n.left \neq$ `@null` **then**
  POSTORDERTRAVERSE($n.left$, A).
if $n.right \neq$ `@null` **then**
  POSTORDERTRAVERSE($n.right$, A).
$A(n)$.

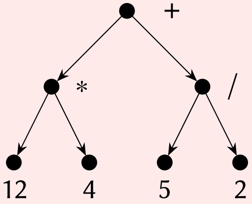
POSTORDERTRAVERSE(root, “output n.value”).

Intermezzo: Traversing binary trees



Let A := "output *n.value*".

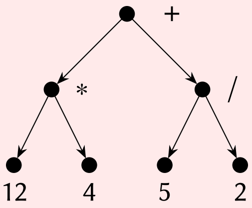
Intermezzo: Traversing binary trees



Let A := "output *n.value*".

- InorderTraverse(root, A)
- PreorderTraverse(root, A)
- PostorderTraverse(root, A)

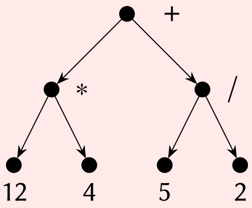
Intermezzo: Traversing binary trees



Let A := "output *n.value*".
For readabilty, we added parentheses and commas.

- InorderTraverse(root, A) $\longrightarrow$ `(12 * 4) + (5 / 2).`
- PreorderTraverse(root, A) $\longrightarrow$ `+(*(12, 4), /(5, 2)).`
- PostorderTraverse(root, A) $\longrightarrow$ `12 4 * 5 2 / +.`

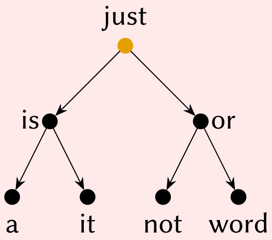
Intermezzo: Traversing binary trees



Let A := "output *n.value*".
For readabilty, we added parentheses and commas.

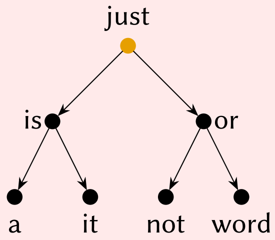
- InorderTraverse(root, A) $\longrightarrow$ `(12 * 4) + (5 / 2)`.
  ("daily" notation)
- PreorderTraverse(root, A) $\longrightarrow$ `+(*(12, 4), /(5, 2))`.
  (prefix notation: function calls)
- PostorderTraverse(root, A) $\longrightarrow$ `12 4 * 5 2 / +`.
  (postfix notation)

Intermezzo: Properties of binary search trees



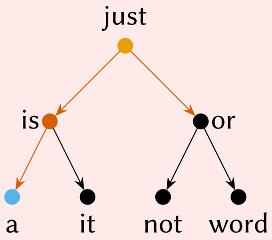
Consider the binary search tree rooted at node n.

Intermezzo: Properties of binary search trees



Consider the binary search tree rooted at node n.

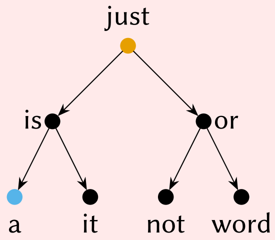
Intermezzo: Properties of binary search trees



Consider the binary search tree rooted at node *n*.

Minimum value Walk down via the left children.

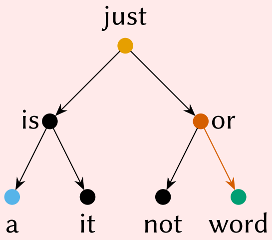
Intermezzo: Properties of binary search trees



Consider the binary search tree rooted at node n.

Minimum value Walk down via the left children.

Intermezzo: Properties of binary search trees

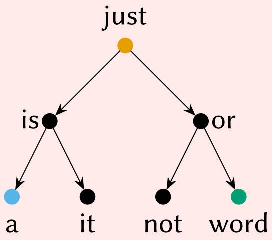


Consider the binary search tree rooted at node n.

Minimum value Walk down via the left children.

Maximum value Walk down via the right children.

Intermezzo: Properties of binary search trees



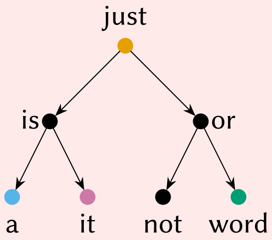
Consider the binary search tree rooted at node n.

Minimum value Walk down via the left children.

Maximum value Walk down via the right children.

Preceding value The maximum value that is smaller than *n.value* (if any).

Intermezzo: Properties of binary search trees



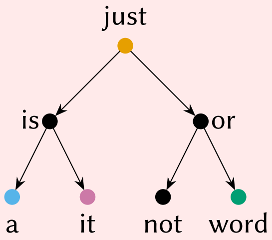
Consider the binary search tree rooted at node *n*.

Minimum value Walk down via the left children.

Maximum value Walk down via the right children.

Preceding value The maximum value that is smaller than *n.value* (if any).
Find the *maximum value* in the binary search tree rooted at *n.left*.

Intermezzo: Properties of binary search trees



Consider the binary search tree rooted at node *n*.

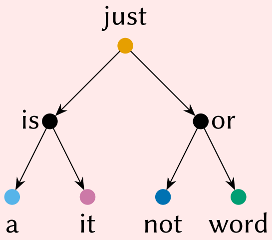
Minimum value Walk down via the left children.

Maximum value Walk down via the right children.

Preceding value The maximum value that is smaller than *n.value* (if any).
Find the *maximum value* in the binary search tree rooted at *n.left*.

Succeding value The minimum value that is larger than *n.value* (if any).

Intermezzo: Properties of binary search trees



Consider the binary search tree rooted at node *n*.

Minimum value Walk down via the left children.

Maximum value Walk down via the right children.

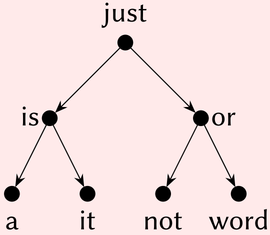
Preceding value The maximum value that is smaller than *n.value* (if any).
Find the *maximum value* in the binary search tree rooted at *n.left*.

Succeding value The minimum value that is larger than *n.value* (if any).
Find the *minimum value* in the binary search tree rooted at *n.right*.

Finding values in binary search trees

How to find a value v?

Adjust binary search to work on trees.



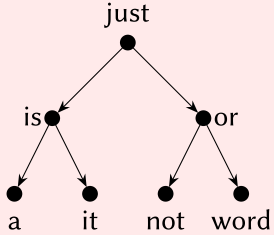
Finding values in binary search trees

Algorithm BSTSearchR(n, v):

Input: n points to a binary search tree node.

if n = `@null` **or** $n.value = v$ **then**
    return n.
else if $n.value < v$ **then**
    return BSTSearchR($n.right$, v).
else
    return BSTSearchR($n.left$, v).

Result: return the node that holds v
(or `@null` if no such node exists).



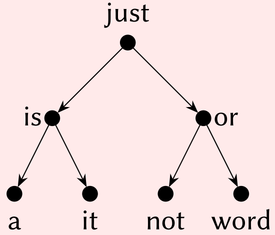
Finding values in binary search trees

Algorithm BSTSearchR(n, v):

Input: n points to a binary search tree node.

```
if n = @null or n.value = v then
    return n.
else if n.value < v then
    return BSTSearchR(n.right, v).
else
    return BSTSearchR(n.left, v).
```

Result: return the node that holds v (or @null if no such node exists).



BinarySearchR(root, "it").

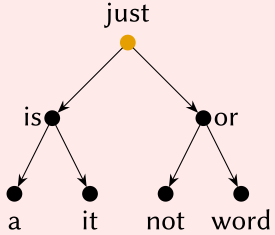
Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):

Input: n points to a binary search tree node.

if $n = $ `@null` **or** $n.value = v$ **then**
    return n.
else if $n.value < v$ **then**
    return BSTSEARCHR($n.right$, v).
else
    return BSTSEARCHR($n.left$, v).

Result: return the node that holds v (or `@null` if no such node exists).



BINARYSEARCHR(root, “it”).

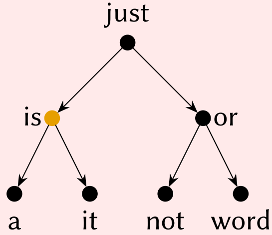
Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):

Input: n points to a binary search tree node.

if n = `@null` **or** $n.value = v$ **then**
    return n.
else if $n.value < v$ **then**
    return BSTSEARCHR($n.right$, v).
else
    return BSTSEARCHR($n.left$, v).

Result: return the node that holds v (or `@null` if no such node exists).



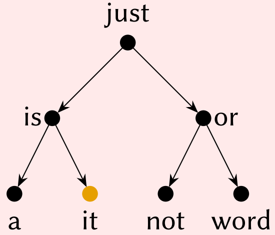
BINARYSEARCHR(root, "it").

Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):
Input: n points to a binary search tree node.

1: **if** n = @null **or** $n.value = v$ **then**
2: **return** n.
3: **else if** $n.value < v$ **then**
4: **return** BSTSEARCHR($n.right$, v).
5: **else**
6: **return** BSTSEARCHR($n.left$, v).

Result: return the node that holds v
(or @null if no such node exists).



BINARYSEARCHR(root, “it”).

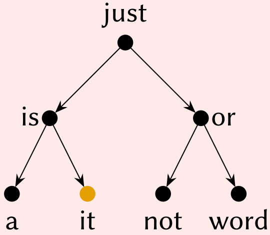
Finding values in binary search trees

Algorithm BSTSearchR(n, v):

Input: n points to a binary search tree node.

if $n =$ `@null` **or** $n.value = v$ **then**
  return n.
else if $n.value < v$ **then**
  return BSTSearchR($n.right$, v).
else
  return BSTSearchR($n.left$, v).

Result: return the node that holds v (or `@null` if no such node exists).



BinarySearchR(root, "it").

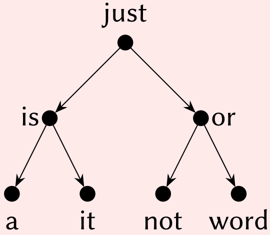
Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):

Input: n points to a binary search tree node.

```
if n = @null or n.value = v then
    return n.
else if n.value < v then
    return BSTSearchR(n.right, v).
else
    return BSTSearchR(n.left, v).
```

Result: return the node that holds v (or `@null` if no such node exists).



Runtime complexity

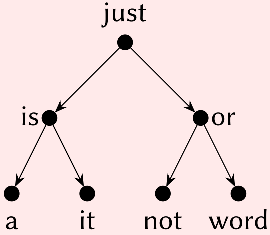
Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):

Input: n points to a binary search tree node.

1: **if** n = `@null` **or** $n.value = v$ **then**
2:     **return** n.
3: **else if** $n.value < v$ **then**
4:     **return** BSTSEARCHR($n.right$, v).
5: **else**
6:     **return** BSTSEARCHR($n.left$, v).

Result: return the node that holds v
(or `@null` if no such node exists).



Runtime complexity

Length of path from *root* to a leaf.

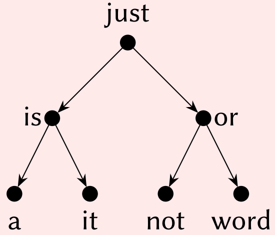
Finding values in binary search trees

Algorithm BSTSearchR(n, v):

Input: n points to a binary search tree node.

```
if n = @null or n.value = v then
    return n.
else if n.value < v then
    return BSTSearchR(n.right, v).
else
    return BSTSearchR(n.left, v).
```

Result: return the node that holds v (or `@null` if no such node exists).



Runtime complexity

Length of path from *root* to a leaf $\rightarrow \lceil \log_2(N) \rceil$ *if* a tree with N nodes is "balanced".

Balanced tree: any path from the root to a leaf has length at-most $\lceil \log_2(N) \rceil$.

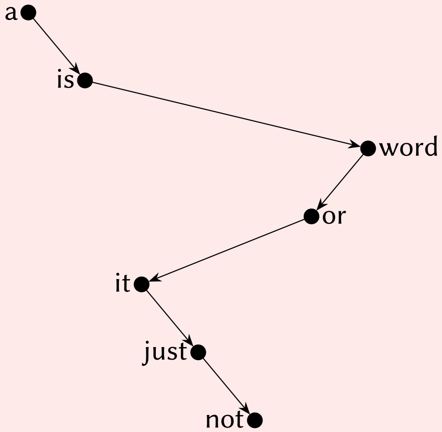
Finding values in binary search trees

Algorithm BSTSEARCHR(n, v):

Input: n points to a binary search tree node.

```
if n = @null or n.value = v then
    return n.
else if n.value < v then
    return BSTSearchR(n.right, v).
else
    return BSTSearchR(n.left, v).
```

Result: return the node that holds v (or `@null` if no such node exists).



Runtime complexity

Length of path from *root* to a leaf $\longrightarrow$ worst-case N.

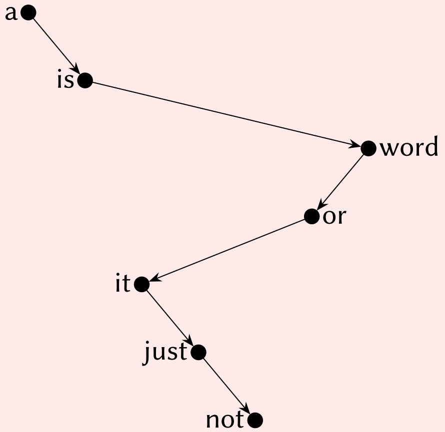
Finding values in binary search trees

Algorithm BSTSearchR(n, v):

Input: n points to a binary search tree node.

```
if n = @null or n.value = v then
    return n.
else if n.value < v then
    return BSTSearchR(n.right, v).
else
    return BSTSearchR(n.left, v).
```

Result: return the node that holds v (or `@null` if no such node exists).



Runtime complexity

Length of path from *root* to a leaf $\longrightarrow$ worst-case N.

Challenge: Our algorithms to modify trees must assure (close to) *balance*.

Finding values in binary search trees

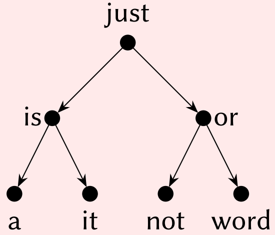
A recursion-free BSTSEARCHR:

Algorithm BSTSEARCH(n, v):

Input: n points to a binary search tree node.

```
while n ≠ @null and n.value ≠ v do
    if n.value < v then
        n := n.right.
    else
        n := n.left.
return n.
```

Result: return the node that holds v
(or `@null` if no such node exists).



Adding values to binary search trees

High-level sketch: adding value v

1. Make a node m for value v.
2. Find the node that will become parent p of node m.
3. Based on $p.value$, add m as either the left or right child of p.

Adding values to binary search trees

Find a candidate parent p to hold a node with value v

Adding values to binary search trees

Find a candidate parent p to hold a node with value v

We cannot use BSTSEARCH to find v: always returns `@null`!

Algorithm BSTSEARCH(n, v):

Input: n points to a binary search tree node.

```
while n ≠ @null and n.value ≠ v do
    if n.value < v then
        n := n.right.
    else
        n := n.left.
return n.
```

Adding values to binary search trees

Find a candidate parent p to hold a node with value v

We cannot use BSTSearch to find v: always returns `@null`!

Idea: keep track of parents p of nodes n visited by BSTSearch.

Algorithm BSTSearch(n, v):

Input: n points to a binary search tree node.

```
while n ≠ @null and n.value ≠ v do
    if n.value < v then
        n := n.right.
    else
        n := n.left.
return n.
```

Adding values to binary search trees

Find a candidate parent p to hold a node with value v

We cannot use BSTSEARCH to find v: always returns `@null`!

Idea: keep track of parents p of nodes n visited by BSTSEARCH.

Algorithm BSTFINDPARENT(n, v):

Input: n points to a binary search tree node.

```
p := @null.
while n ≠ @null do
    p := n.
    if n.value < v then
        n := n.right.
    else
        n := n.left.
return p.
```

Adding values to binary search trees

Add m as either the left or right child of p

Let p := BSTFindParent(root, v).

Let m point to a fresh binary search tree node with $m.value := v$.

We have three cases:

Adding values to binary search trees

Add m as either the left or right child of p

Let p := BSTFindParent(root, v).

Let m point to a fresh binary search tree node with $m.value := v$.

We have three cases:

1. If p = `@null`: empty tree, make m the root of the tree.

Adding values to binary search trees

Add m as either the left or right child of p

Let $p := \textsc{BSTFindParent}(\text{root}, v)$.

Let m point to a fresh binary search tree node with $m.value := v$.

We have three cases:

1. If $p =$ `@null`: empty tree, make m the root of the tree.
2. If $v < p.value$: set $p.left := m$.
3. If $v > p.value$: set $p.right := m$.

Adding values to binary search trees

Adding values: Good case

We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.

Adding values to binary search trees

Adding values: Good case

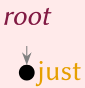
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.

p = `@null` *root*

Adding values to binary search trees

Adding values: Good case

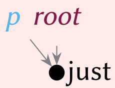
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

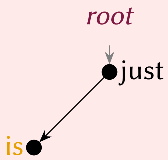
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

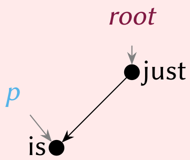
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

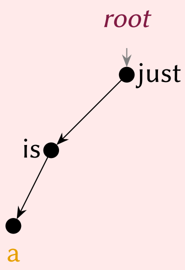
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

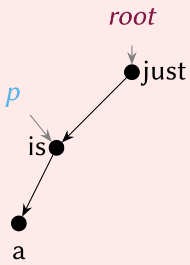
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

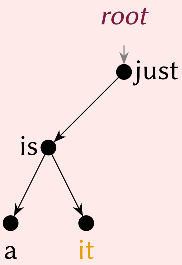
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

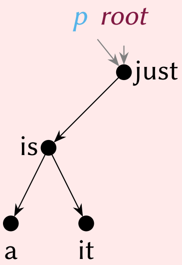
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

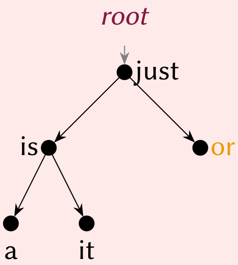
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

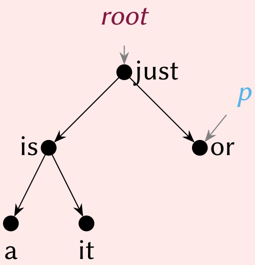
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

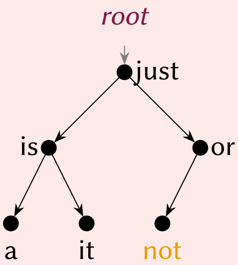
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

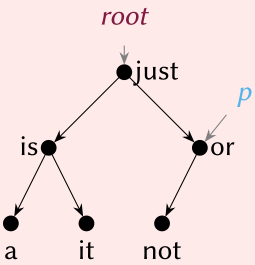
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

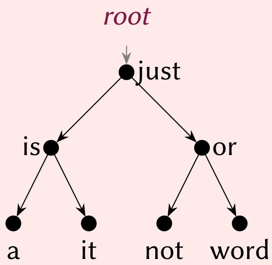
We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Good case

We add “just”, “is”, “a”, “it”, “or”, “not”, “word”.



Adding values to binary search trees

Adding values: Bad case

We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.

Adding values to binary search trees

Adding values: Bad case

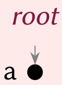
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.

p = `@null` *root*

Adding values to binary search trees

Adding values: Bad case

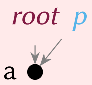
We add "a", "is", "it", "just", "not", "or", "word".



Adding values to binary search trees

Adding values: Bad case

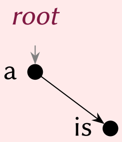
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

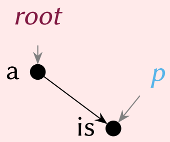
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

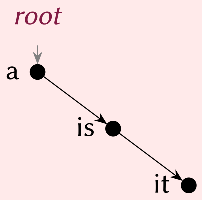
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

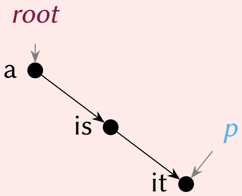
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

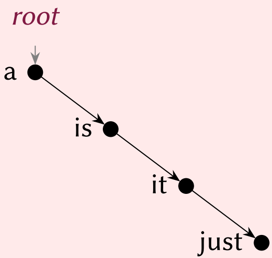
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

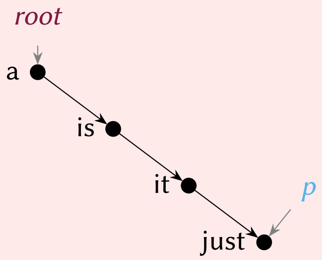
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

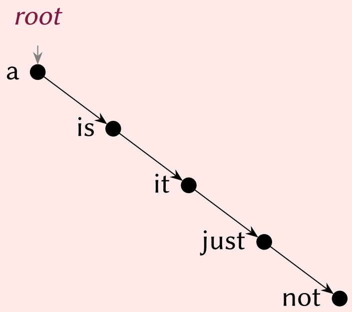
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

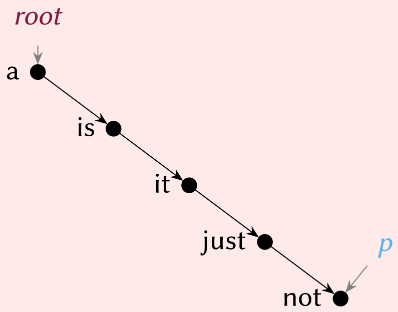
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

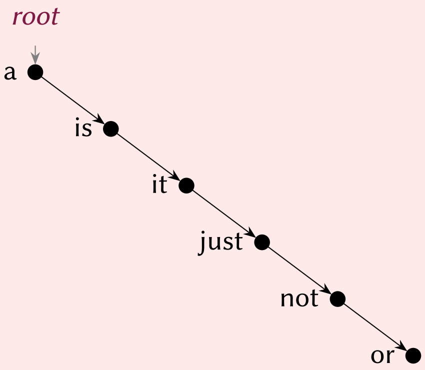
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

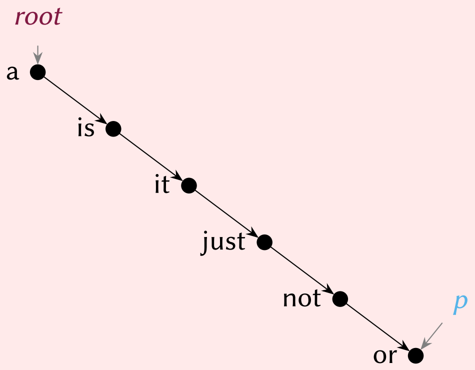
We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Adding values to binary search trees

Adding values: Bad case

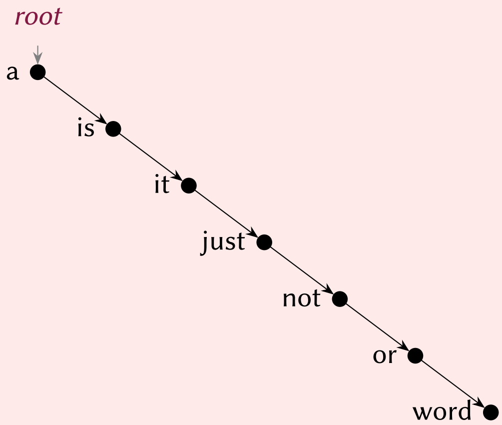
We add "a", "is", "it", "just", "not", "or", "word".



Adding values to binary search trees

Adding values: Bad case

We add “a”, “is”, “it”, “just”, “not”, “or”, “word”.



Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Average cost of adding values

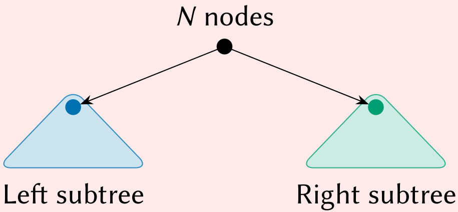
Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p.

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

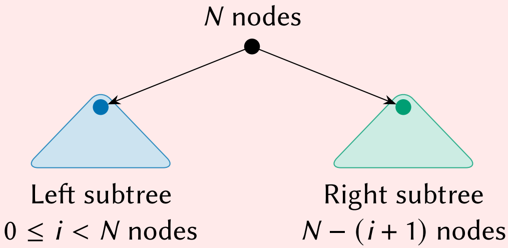
Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.



Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

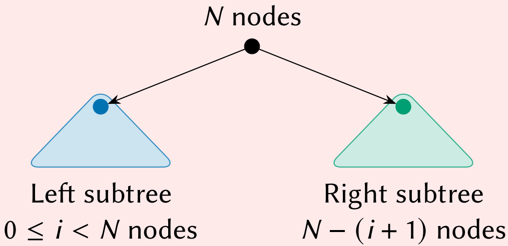
Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.



Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.

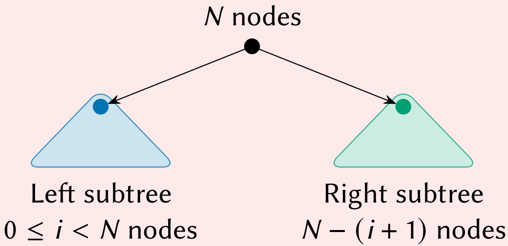


We write a *recurrence* $T(N)$ for the average value of L.

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.



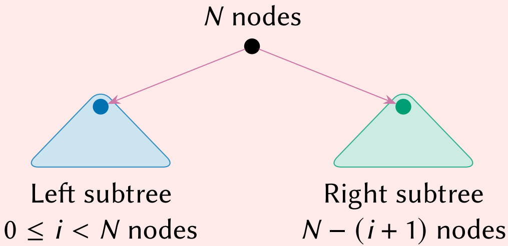
We write a *recurrence* $T(N)$ for the average value of L:

- Either v ends up in the left subtree.
- Or v ends up in the right subtree.

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
 $\longrightarrow$ Number of nodes L on the path from root to p.



We write a *recurrence* $T(N)$ for the average value of L:

- Either v ends up in the left subtree $\longrightarrow$ average length $T(i) + 1$.
- Or v ends up in the right subtree $\longrightarrow$ average length $T(N - (i + 1)) + 1$.

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.

We write a *recurrence* $T(N)$ for the average value of L:

- Either v ends up in the left subtree $\longrightarrow$ average length $T(i) + 1$.
- Or v ends up in the right subtree $\longrightarrow$ average length $T(N - (i + 1)) + 1$.

$$T(N) = \begin{cases} 0 & \text{if } N = 0; \\ 1 & \text{if } N = 1; \\ \frac{1}{2N}\left(\sum_{i=0}^{N-1} T(i) + 1 + T(N - (i + 1)) + 1\right) & \text{if } N > 1. \end{cases}$$

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.

We write a *recurrence* $T(N)$ for the average value of L:

- Either v ends up in the left subtree $\longrightarrow$ average length $T(i) + 1$.
- Or v ends up in the right subtree $\longrightarrow$ average length $T(N - (i + 1)) + 1$.

$$T(N) = \begin{cases} 0 & \text{if } N = 0; \\ 1 & \text{if } N = 1; \\ \dfrac{1}{2N}\left(\displaystyle\sum_{i=0}^{N-1} T(i) + 1 + T(N - (i + 1)) + 1\right) & \text{if } N > 1. \end{cases}$$

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right)$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} \underbrace{c\,\mathrm{LOG}_2(i)}_{} + d\right) + 1$$

$\mathrm{LOG}_2(i) = \log_2(i)$, except $\mathrm{LOG}_2(0) = 0$.

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\log_2(k) + d\right) + 1$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\log_2(k) + d\right) + 1 = c\log_2(k) + d + 1.$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\log_2(k) + d\right) + 1 = \underbrace{c\log_2(k) + d + 1.}_{\text{Need to get rid of “+1”!}}$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\sum_{i=0}^{k-1} \underbrace{c\log_2(k)}_{\text{Extreme upper bound.}} + d\right) + 1 = \underbrace{c\log_2(k) + d + 1.}_{\text{Need to get rid of "+1"!}}$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c \log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c \log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\left(\sum_{i=0}^{k \operatorname{div} 2} c \log_2\left(\frac{k}{2}\right)\right) + \left(\sum_{i=k \operatorname{div} 2+1}^{k-1} c \log_2(k)\right)\right) + d + 1$$

Average cost of adding values

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \frac{1}{N}\left(\sum_{i=0}^{N-1} T(i)\right) + 1.$$

Show that $T(N) = \Theta(\log_2(N))$ using induction

Induction hypothesis For some c, d, $T(N) \leq c\log_2(N) + d$ for all $2 \leq N < k$.

Induction step Prove $T(k) \leq c\log_2(k) + d$.

$$T(k) = \frac{1}{k}\left(\sum_{i=0}^{k-1} T(i)\right) \leq \frac{1}{k}\left(\sum_{i=0}^{k-1} c\,\mathrm{LOG}_2(i) + d\right) + 1$$

$$\leq \frac{1}{k}\left(\left(\sum_{i=0}^{k \operatorname{div} 2} c\log_2\left(\frac{k}{2}\right)\right) + \left(\sum_{i=k \operatorname{div} 2+1}^{k-1} c\log_2(k)\right)\right) + d + 1 \leq c\log_2(k) + d + 1 - \frac{c}{2}.$$

Average cost of adding values

Assume we build a binary search tree of *random values* by adding values v one at a time.

Cost of adding N-th value v: finding the parent p
$\longrightarrow$ Number of nodes L on the path from root to p.

We write a *recurrence* $T(N)$ for the average value of L.

$$T(N) = \Theta\left(\log_2(N)\right).$$

Removing values from binary search trees

We have already seen how

- to traverse a binary search tree;
- to search for values in a binary search tree;
- to add values to a binary search tree;
- to find the minimum and maximum values in a binary search tree; and
- to find the preceding and succeding values of values in a binary search tree.

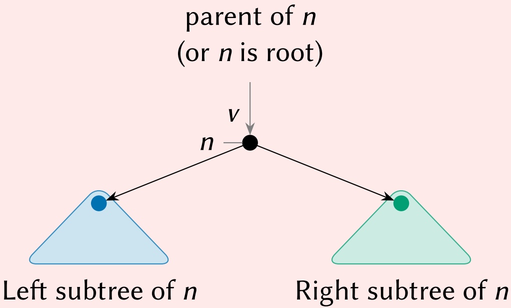
Removing values from binary search trees

We have already seen how

- to traverse a binary search tree;
- to search for values in a binary search tree;
- to add values to a binary search tree;
- to find the minimum and maximum values in a binary search tree; and
- to find the preceding and succeding values of values in a binary search tree.

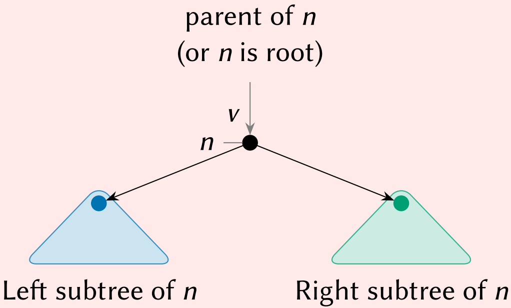
We have not yet seen how to *remove* values.

Removing values from binary search trees



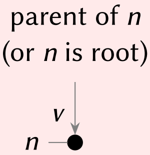
Say we want to remove the node n holding v from the tree.

Removing values from binary search trees



Say we want to remove the node n holding v from the tree.
Based on the number of children of n, we have three cases to consider:

Removing values from binary search trees



Say we want to remove the node *n* holding *v* from the tree.
Based on the number of children of *n*, we have three cases to consider:

1. *n* has zero children.

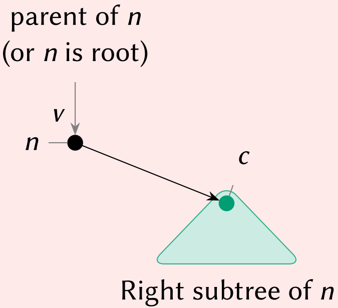
Removing values from binary search trees

parent of n
(or n is root)

Say we want to remove the node n holding v from the tree.
Based on the number of children of n, we have three cases to consider:

1. n has zero children.
   Easy: Just remove node n from the parent p, then remove n.

Removing values from binary search trees

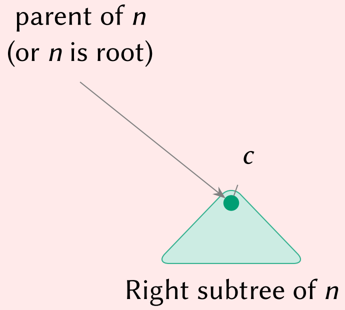


Say we want to remove the node n holding v from the tree.

Based on the number of children of n, we have three cases to consider:

2. n has one child c.

Removing values from binary search trees



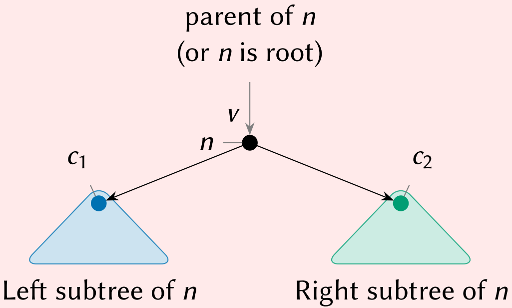
Say we want to remove the node n holding v from the tree.

Based on the number of children of n, we have three cases to consider:

2. n has one child c.

   Easy: Just replace node n in the parent p by c, then remove n.

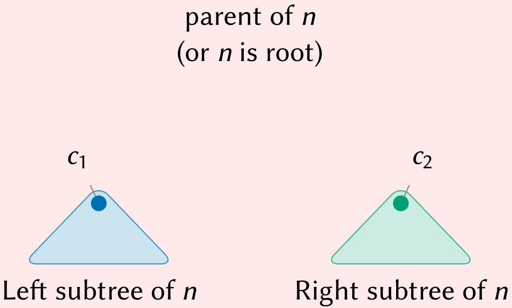
Removing values from binary search trees



Say we want to remove the node n holding v from the tree.
Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.

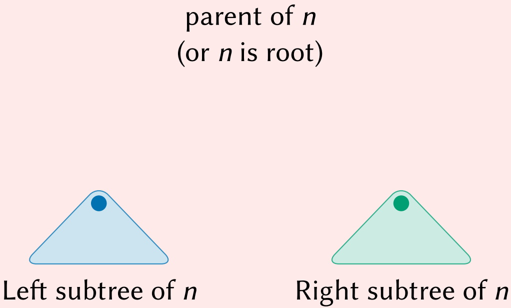
Removing values from binary search trees



Say we want to remove the node n holding v from the tree.
Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.
   Hard: What to do with the children?

Removing values from binary search trees

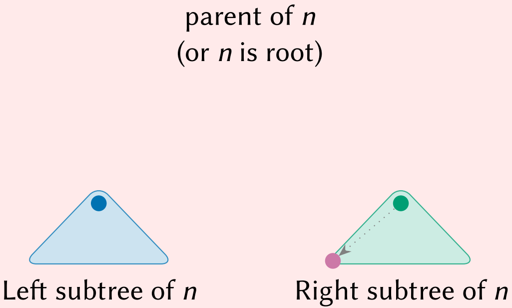


Say we want to remove the node n holding v from the tree.
Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.

   Hard: What to do with the children? *Make a new parent n' for c_1, c_2.*

Removing values from binary search trees



Say we want to remove the node n holding v from the tree.

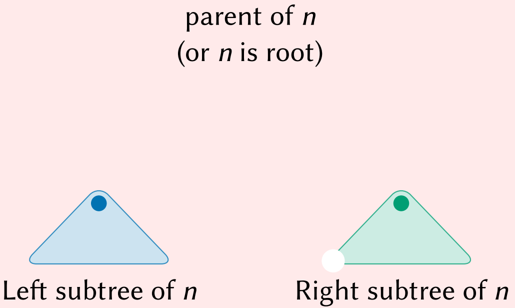
Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.

   Hard: What to do with the children? *Make a new parent n' for c_1, c_2.*

   - The subtree of c_2 has a node m that holds the succeding value w of v.

Removing values from binary search trees



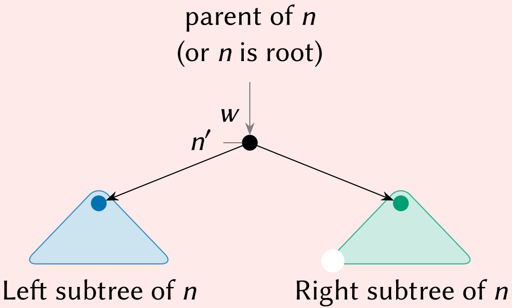
Say we want to remove the node n holding v from the tree.

Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.

   Hard: What to do with the children? *Make a new parent n' for c_1, c_2.*
   - The subtree of c_2 has a node m that holds the succeding value w of v.
   - Node m either has no children or has a right child: easy to remove.

Removing values from binary search trees



Say we want to remove the node n holding v from the tree.

Based on the number of children of n, we have three cases to consider:

3. n has two children c_1, c_2.

   Hard: What to do with the children? *Make a new parent n' for c_1, c_2 with value w.*
   - The subtree of c_2 has a node m that holds the succeding value w of v.
   - Node m either has no children or has a right child: easy to remove.

Binary search trees

Consider a binary search tree with N values.

Runtime complexity

Adding, searching, or removing values:
worst-case: number of nodes on the path from the root to a leaf.

Binary search trees

Consider a binary search tree with N values.

Runtime complexity

Adding, searching, or removing values:
worst-case: number of nodes on the path from the root to a leaf.

Worst-case: N. Expected-case: $\Theta(\log_2(N))$ *if* random values are added and removed.

Binary search trees

Consider a binary search tree with N values.

Runtime complexity

Adding, searching, or removing values:
worst-case: number of nodes on the path from the root to a leaf.

Worst-case: N. Expected-case: $\Theta(\log_2(N))$ *if* random values are added and removed.

Practical limitation

A lot of data sets are *not random*: e.g., partially-sorted inputs.