

Algoritmi di ricerca su array (*search algorithms*)

Liceo G.B. Brocchi
Classi seconde Scientifico - opzione scienze applicate
Bassano del Grappa, Ottobre 2022

Motivazioni

- Il nostro obiettivo è **cercare** la prima occorrenza di un elemento all'interno di una struttura dati:
 - l'unica struttura dati che conosciamo è l'**array**, per cui effettueremo la ricerca su array. Per semplicità lavoreremo su array di interi
 - nella realtà si lavora spesso su strutture simili ad array, contenenti però dati molto più complessi dei semplici interi
 - i principi di funzionamento sono comunque identici
- Questo problema ha innumerevoli applicazioni in informatica:
 - *ricerca di un dato in un database*
 - *ricerca di un file all'interno di un file system*
 - *ricerca di un gene in un genoma sequenziato*
 - *ricerca di una parola all'interno di un dizionario*
 - *ricerca di una parola all'interno della Divina Commedia*
 - *ricerca di un'impronta digitale all'interno di un archivio di impronte di pregiudicati*
 - *ricerca di un profilo Instagram con un determinato nome (sì, purtroppo anche queste cose inutili)*

Ricerca lineare (*linear search*)

6	5	9	23	5	7	6	7	78	4	56	98
---	---	---	----	---	---	---	---	----	---	----	----

- l'array non è ordinato
- dobbiamo cercare la prima occorrenza dell'elemento con valore 7 e memorizzare il suo indice

```
const int items_size = 12;
int items[items_size] = {6, 5, 9, 23, 5, 7, 6, 7, 78, 4, 56, 98};
bool found = false;
int index_of_searched_item = -1;
int searched_item = 7;

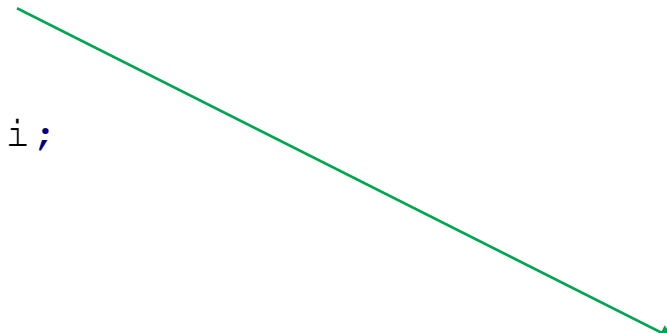
for (int i = 0; i < items_size && found == false; i++) {
    if (items[i] == searched_item) {
        found = true;
        index_of_searched_item = i;
    }
}
```

Ricerca lineare (*linear search*)

- l'array non è ordinato
- dobbiamo cercare la prima occorrenza dell'elemento con valore 7 e memorizzare il suo indice

```
const int items_size = 12;
int items[items_size] = {6, 5, 9, 23, 5, 7, 6, 7, 78, 4, 56, 98};
bool found = false;
int index_of_searched_item = -1;
int searched_item = 7;

for (int i = 0; i < items_size && !found; i++) {
    if (items[i] == searched_item) {
        found = true;
        index_of_searched_item = i;
    }
}
```



Equivalente è found == false

Ricerca lineare (*linear search*)

6	5	9	23	5	0	6	-2	78	4	-56	98
---	---	---	----	---	---	---	----	----	---	-----	----

- se l'elemento cercato non è presente, la ricerca lineare restituisce
 - `index_of_searched_item` (indice dell'elemento cercato): **-1**
 - variabile booleana `found`: **false**
- Avremmo potuto utilizzare anche soltanto `index_of_searched_item`. Utilizzare 2 variabili per indicare l'indice e la risposta *trovato/non trovato* rende il programma più leggibile

Ricerca lineare (*linear search*)

6	5	9	23	5	0	6	-2	78	4	-56	98
---	---	---	----	---	---	---	----	----	---	-----	----

- Stiamo lavorando sull'array rappresentato qui sopra
- Vogliamo sapere se è presente e dove è presente il primo elemento con valore 107
- La dimensione dell'array è $n = 12$

Prima di arrivare a dire che 107 non è presente nell'array, abbiamo dovuto effettuare un numero di confronti pari alla dimensione dell'array

È la situazione in cui l'algoritmo deve eseguire più calcoli prima di arrivare al risultato: il CASO PEGGIORE (WORST CASE)

Ricerca lineare (*linear search*)

6	5	9	23	5	0	6	-2	78	4	-56	107
---	---	---	----	---	---	---	----	----	---	-----	-----

- Stiamo lavorando sull'array rappresentato qui sopra
- Vogliamo sapere se è presente e dove è presente il primo elemento con valore 107
- La dimensione dell'array è $n = 12$

Prima di arrivare a dire che 107 è presente nell'array, abbiamo dovuto effettuare un numero di confronti pari alla dimensione dell'array

È la situazione in cui l'algoritmo deve eseguire più calcoli prima di arrivare al risultato: il CASO PEGGIORE (WORST CASE)

Ricerca lineare (*linear search*)

107	5	9	23	5	0	6	-2	78	4	-56	107
-----	---	---	----	---	---	---	----	----	---	-----	-----

- Stiamo lavorando sull'array rappresentato qui sopra
- Vogliamo sapere se è presente e dove è presente il primo elemento con valore 107
- La dimensione dell'array è $n = 12$

Prima di arrivare a dire che 107 è presente nell'array, abbiamo dovuto effettuare un numero di confronti pari a 1, perché 107 è presente nella prima posizione

È la situazione in cui l'algoritmo deve eseguire più calcoli prima di arrivare al risultato: il CASO MIGLIORE (BEST CASE)

Ricerca lineare (*linear search*)

107	5	9	23	5	0	6	-2	78	4	-56	107
-----	---	---	----	---	---	---	----	----	---	-----	-----

- In media, se non possiamo fare ipotesi sulla distribuzione degli interi all'interno dell'array, prima di trovare l'elemento cercato dovremmo fare **$n/2$** confronti

Per analizzare l'efficienza di un algoritmo, nella maggior parte dei casi, si tiene conto del numero di operazioni base che fa nel caso peggiore.

La ricerca lineare nel caso peggiore fa n confronti, dove n è la dimensione dell'array

E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
indice 0	indice 1	indice 2			indice 5						

L'elemento
centrale

- Ora l'array è **ordinato** in senso crescente, secondo l'ordinamento definito per i numeri interi
Assumiamo che sia ordinato e basta. Vedremo che ordinare gli array in realtà non è per niente banale (gli americani dicono *there ain't no such thing as a free lunch*, "nessuno dà niente per niente")
- Cerchiamo di visualizzare l'array in un modo che permetta di velocizzare la ricerca, senza dover più partire dal primo elemento e scorrerlo tutto
- Questo array ha 12 elementi. L'ultimo elemento ha indice 11
 - l'elemento **centrale** ha indice $11 / 2 = 5$
- In generale, dato un array di dimensione n :
 - l'elemento centrale ha indice $(n-1) / 2$, dove $/$ è la divisione intera
- Il numero degli elementi è pari, per cui l'elemento centrale non è veramente al «centro», nel senso che ha sinistra k elementi, e a destra $k + 1$. Lo chiamiamo comunque **centrale**

E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70
---	----	----	----	----	----	----	----	----	----	----

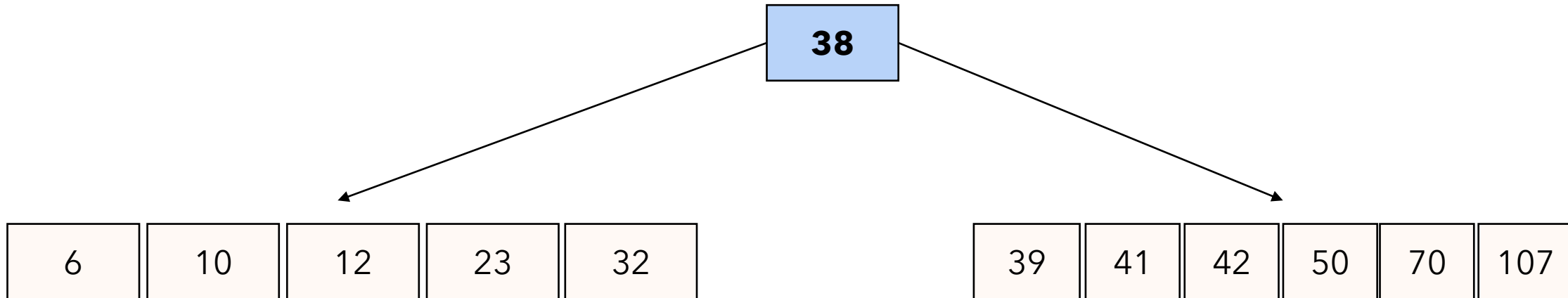
L'elemento
centrale

- Questo array ha 11 elementi. L'ultimo elemento ha indice 10
 - l'elemento *centrale* ha indice $10 / 2 = 5$
- In generale, dato un array di dimensione n :
 - l'elemento centrale ha indice $(n-1) / 2$, dove $/$ è la divisione intera
- Qui l'elemento centrale è veramente centrale, perché ha k elementi alla sua sinistra e k alla sua destra

E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

Proviamo a vedere l'array come un **albero**, dove la radice è l'elemento centrale, il figlio sinistro è il *sottoarray* a sinistra dell'elemento centrale, e il figlio destro è il *sottoarray* a destra dell'elemento centrale. Questa particolare costruzione è molto utilizzata in informatica e prende il nome di **albero binario**

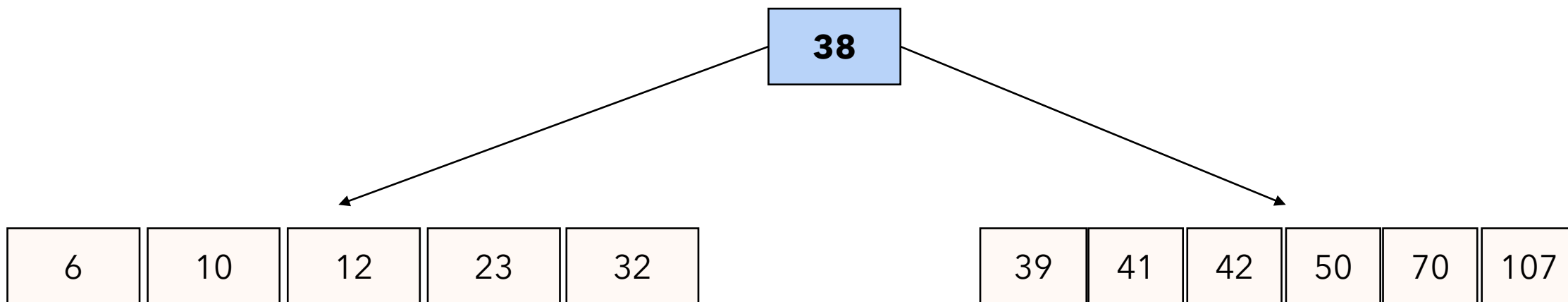


E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

- **Domande:**

- che caratteristica ha il sottoarray sinistro (il *figlio sinistro della radice*)
- che caratteristica ha il sottoarray destro (il *figlio destro della radice*)

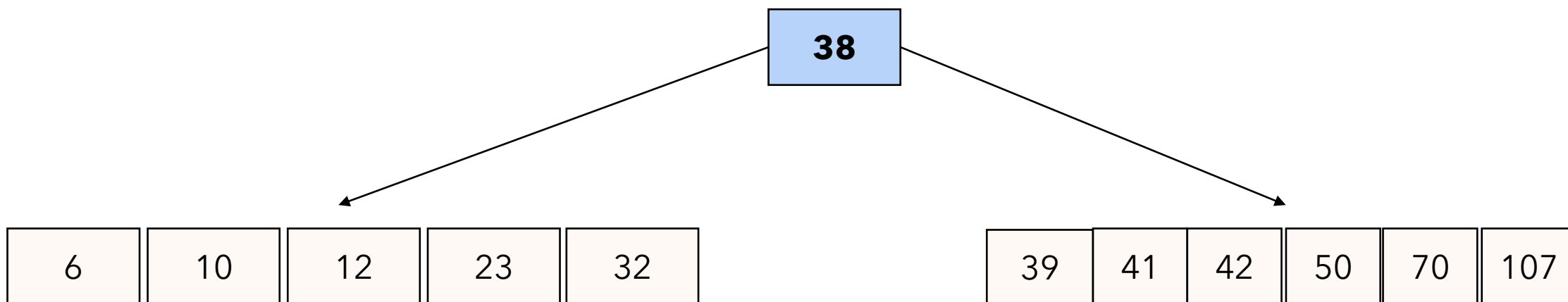


E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

- **Come è fatto l'albero:**

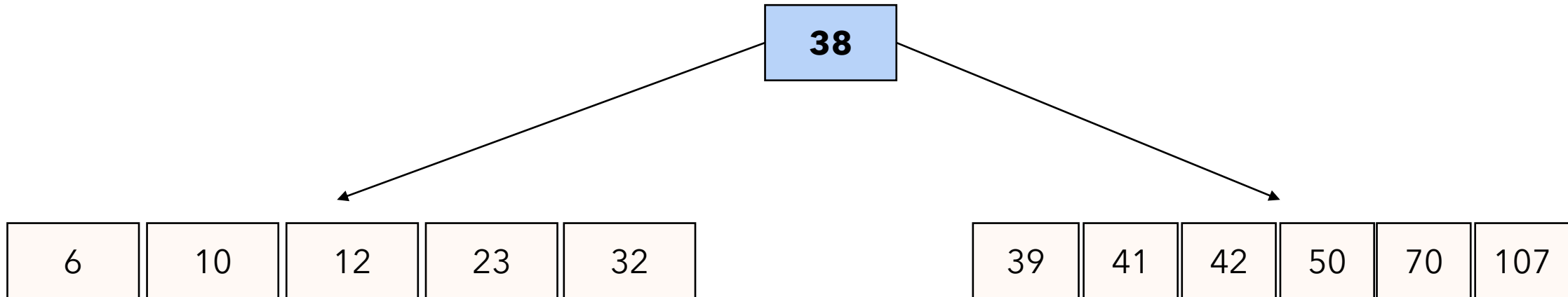
- Il sottoarray sinistro è costituito da elementi *minori o uguali all'elemento radice*
- Il sottoarray destro è costituito da elementi *maggiori o uguali all'elemento radice*



E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

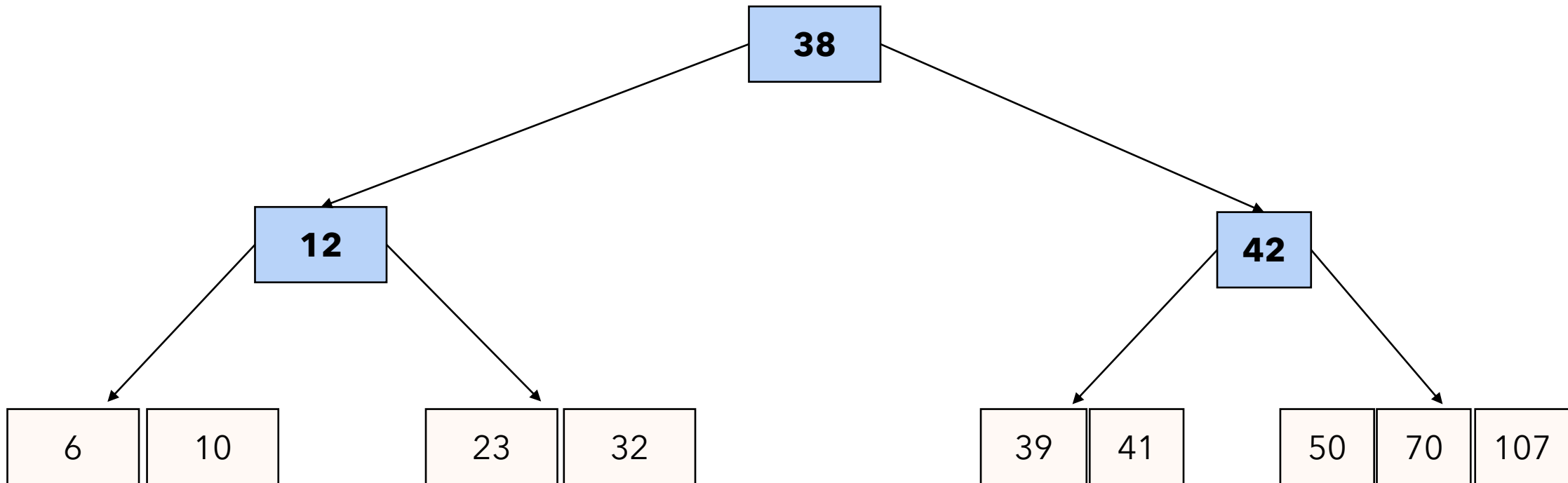
- Potremmo pensare di vedere come alberi anche i sottoarray figli...



E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

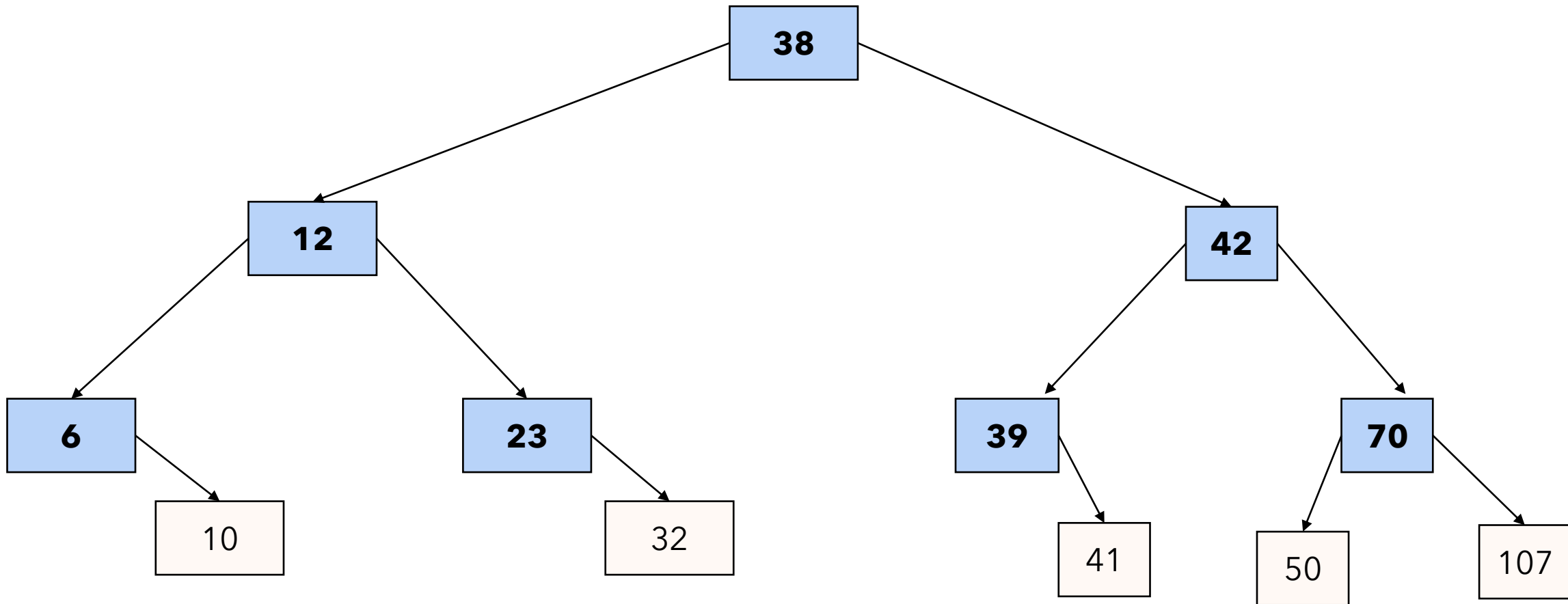
- Potremmo pensare di vedere come alberi anche i sottoarray figli...



E se l'array fosse ordinato?

6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

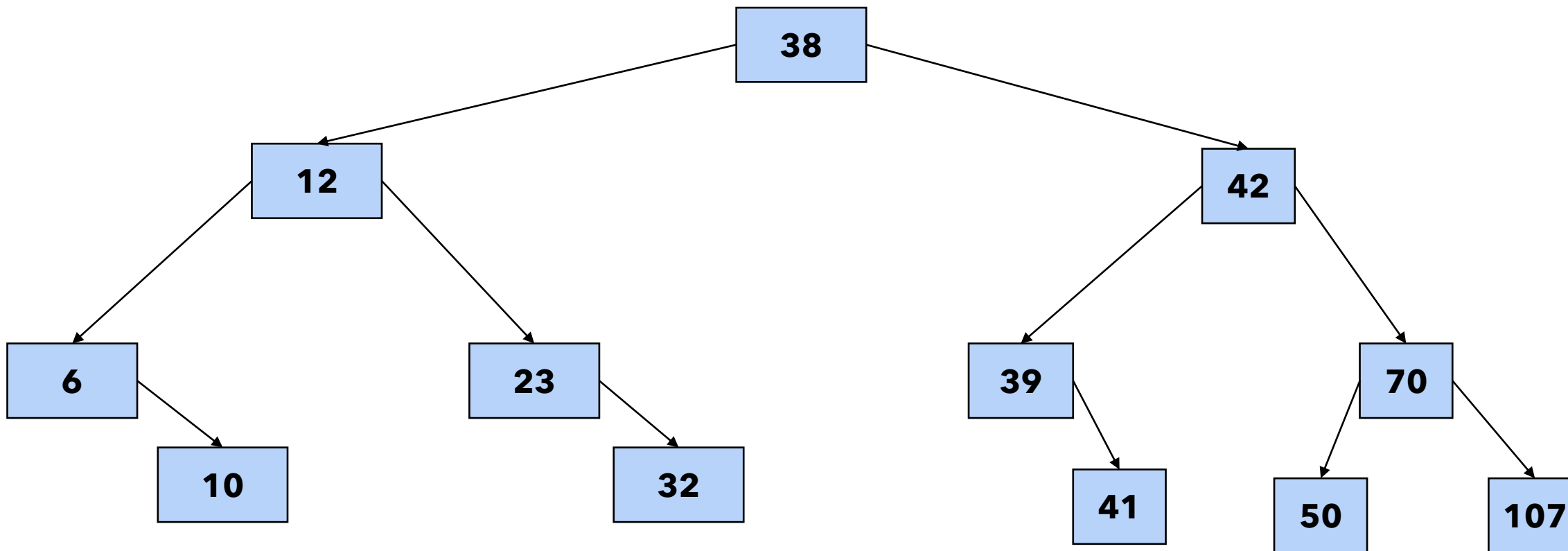
- Potremmo pensare di vedere come alberi anche i sottoarray figli...
- NB: se un array ha 2 elementi, l'elemento «centrale» è quello di indice 0 ($(2 - 1) / 2$)



E se l'array fosse ordinato?

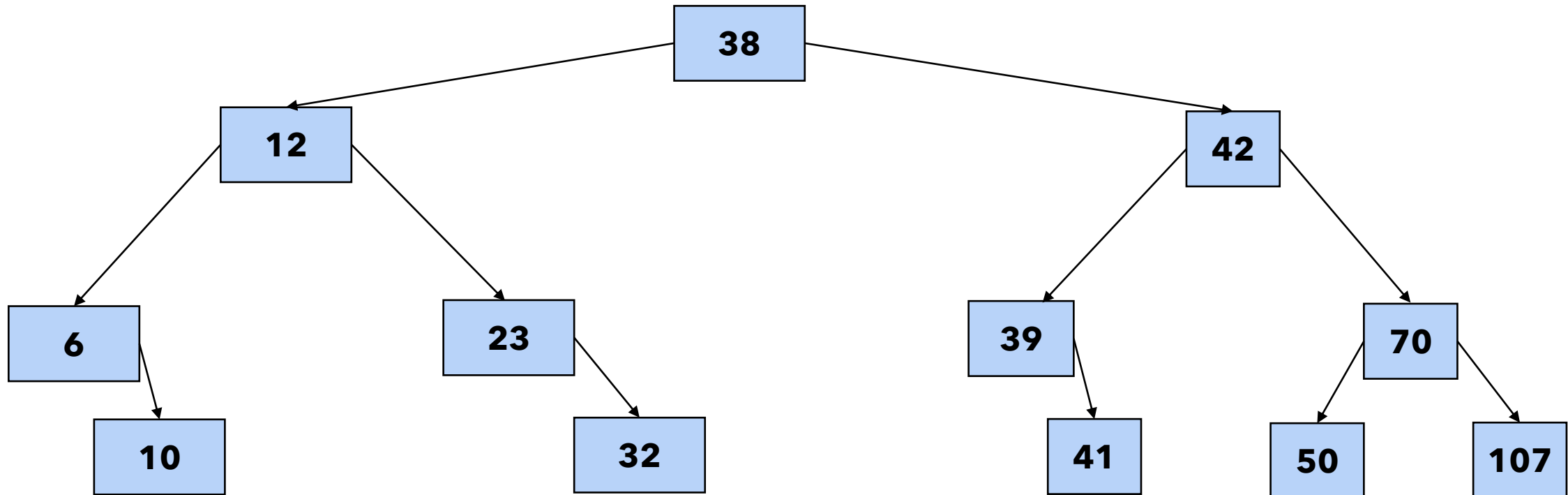
6	10	12	23	32	38	39	41	42	50	70	107
---	----	----	----	----	----	----	----	----	----	----	-----

- Abbiamo «esploso» tutto l'array sotto forma di albero, fermandoci ai sottoarray di dimensione 1
- Procedimenti di questo tipo vengono detti **ricorsivi** (*si applica a pezzi sempre più piccoli il procedimento che si era applicato al pezzo grande iniziale*)



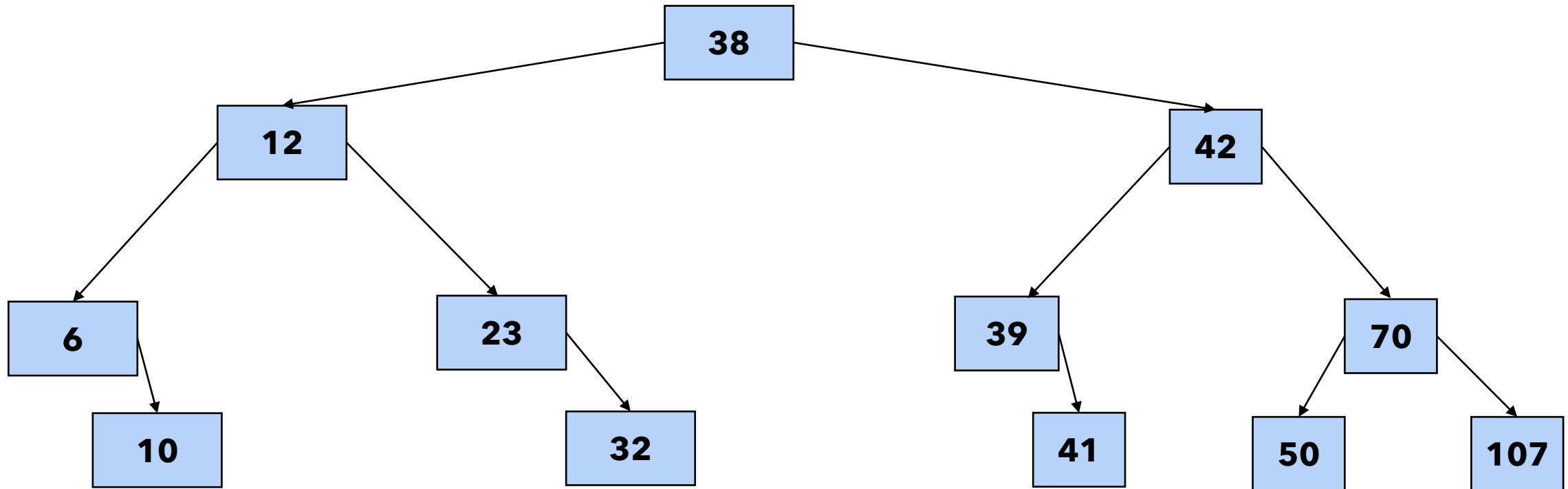
Come sfruttare questo albero

- **NB:** l'albero che abbiamo costruito esiste solo nella nostra mente, come modello di calcolo
- L'algoritmo di ricerca lo facciamo sul solito vecchio array
- Ipotizziamo di dover cercare l'elemento **39**



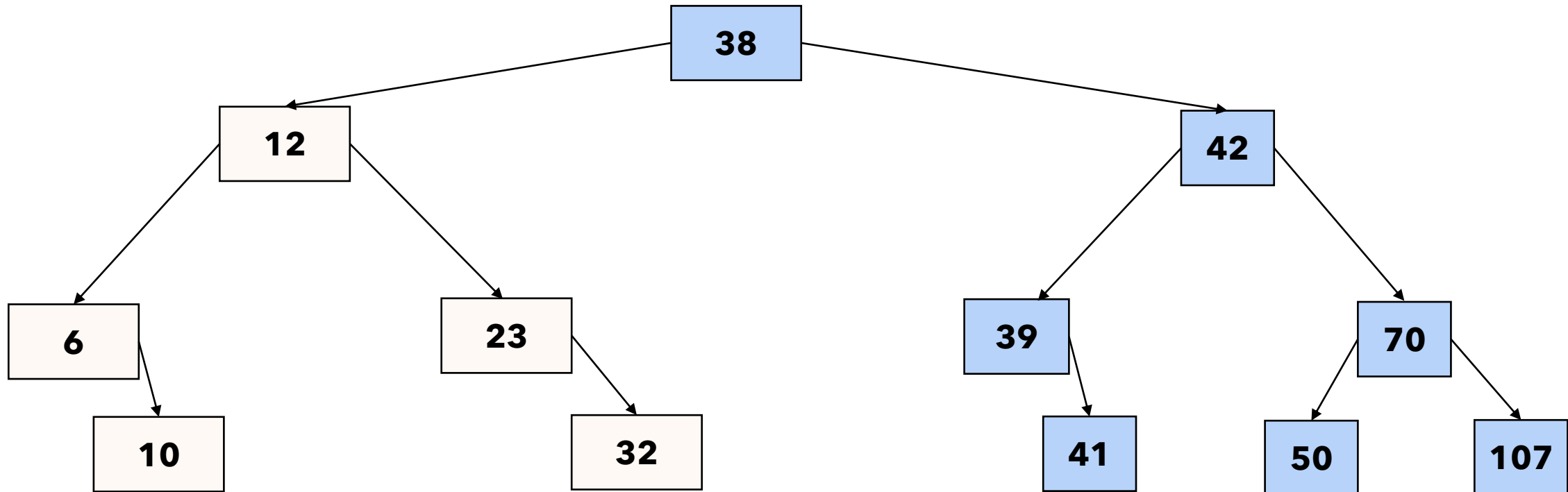
Come sfruttare questo albero

- Partiamo dalla radice dell'albero.
- 39 è uguale al valore della radice? **NO**
- Abbiamo speranze di trovare 39 nel sottoalbero sinistro della radice? **NO**
- Abbiamo speranze di trovare 39 nel sottoalbero destro della radice? **SÌ**



Come sfruttare questo albero

- Abbiamo speranze di trovare 39 nel sottoalbero destro della radice? **SI'**
- Siamo riusciti ad escludere la radice e **potare** il figlio sinistro!
- **È un modo per non perdere tempo nel cercare le cose dove sicuramente non esistono**

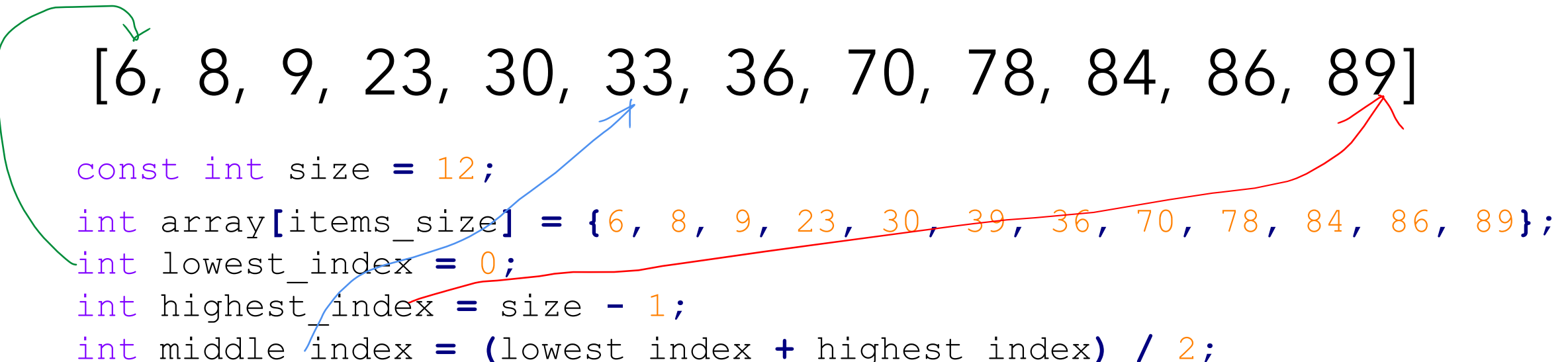


Ricerca binaria o dicotomica (*binary search*)

- Abbiamo speranze di trovare 39 nel sottoalbero destro della radice? **SI'**
- Siamo riusciti ad escludere la radice e **potare** il figlio sinistro!
- **È un modo per non perdere tempo nel cercare le cose dove sicuramente non esistono**
- L'albero che abbiamo visto è solo una nostra costruzione mentale per capire il funzionamento dell'algoritmo che scriveremo
- Quello che dobbiamo fare è giocare con gli indici dell'array in modo da riprodurre la ricerca che abbiamo fatto sull'albero
- Dobbiamo cercare l'elemento **k** (dove k è un numero intero)

[6, 8, 9, 23, 30, 33, 36, 70, 78, 84, 86, 89]

```
const int size = 12;  
int array[items_size] = {6, 8, 9, 23, 30, 39, 36, 70, 78, 84, 86, 89};  
int lowest_index = 0;  
int highest_index = size - 1;  
int middle_index = (lowest_index + highest_index) / 2;
```



Ricerca binaria o dicotomica (*binary search*)

[6, 8, 9, 23, 30, 33, 36, 70, 78, 84, 86, 89]

CASO 1:

$k == \text{array}[\text{middle_index}]$

k trovato all'indice middle_index, la procedura termina

CASO 2:

$k > \text{array}[\text{middle_index}]$

dobbiamo cercare k nel sottoarray destro

CASO 3:

$k < \text{array}[\text{middle_index}]$

dobbiamo cercare k nel sottoarray sinistro

Ricerca binaria o dicotomica (*binary search*)

[6, 8, 9, 23, 30, 33, 36, 70, 78, 84, 86, 89]

CASO 2:

$k > \text{array}[\text{middle_index}]$

dobbiamo cercare k nel sottoarray destro

Come facciamo a specificare via codice che alla prossima iterazione della procedura dobbiamo cercare nel sottoarray destro?

CASO 3:

$k < \text{array}[\text{middle_index}]$

Come facciamo a specificare via codice che alla prossima iterazione della procedura dobbiamo cercare nel sottoarray sinistro?

Ricerca binaria o dicotomica (*binary search*)

k	84												
iteration 1	lowest_index					middle_index				highest_index			k > 33
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 2	lowest_index					middle_index				highest_index			k > 78
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 3	lowest_index					middle_index				highest_index			k < 86
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 4	lowest_index, highest_index, middle_index												found
	6	8	9	23	30	33	36	39	78	84	86	89	

Ricerca binaria o dicotomica (*binary search*)

- Se l'elemento cercato è maggiore di `array[middle_index]`, si aggiorna solo l'indice `lowest_index`:
 - **`lowest_index = middle_index + 1`**
- Se l'elemento cercato è minore di `array[middle_index]`, si aggiorna solo l'indice `highest_index`:
 - **`highest_index = middle_index - 1`**

Ricerca binaria o dicotomica (*binary search*)

k	29												
iteration 1	lowest_index				middle_index				highest_index				k < 33
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 2	lowest_index				middle_index				highest_index				k > 9
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 3	lowest_index, middle_index				highest_index								k > 23
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 4	lowest_index, highest_index, middle_index												k < 30
	6	8	9	23	30	33	36	39	78	84	86	89	
End of loop (highest_index cannot be less than lower index)	highest_index				lowest_index, middle_index								not found
	6	8	9	23	30	33	36	39	78	84	86	89	

Ricerca binaria o dicotomica (*binary search*)

k	29												
iteration 1	lowest_index				middle_index				highest_index				k < 33
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 2	lowest_index				middle_index				highest_index				k > 9
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 3	lowest_index, middle_index				highest_index								k > 23
	6	8	9	23	30	33	36	39	78	84	86	89	
iteration 4	lowest_index, highest_index, middle_index												k < 30
	6	8	9	23	30	33	36	39	78	84	86	89	
End of loop (highest_index cannot be less than lower index)	highest_index				lowest_index, middle_index								not found
	6	8	9	23	30	33	36	39	78	84	86	89	

Ricerca binaria o dicotomica (*binary search*)

k	8												
iteration 1	lowest_index			middle_index					highest_index				k < 33
	6	8	8	23	30	33	36	39	78	84	86	89	
iteration 2	lowest_index		middle_index		highest_index								found
	6	8	8	23	30	33	36	39	78	84	86	89	

Ricerca binaria o dicotomica (*binary search*)

k	8												
iteration 1	lowest_index				middle_index				highest_index				k < 33
	6	7	8	8	30	33	36	39	78	84	86	89	
iteration 2	lowest_index		middle_index		highest_index								found
	6	7	8	8	30	33	36	39	78	84	86	89	

Ricerca binaria o dicotomica (*binary search*)

```
int l = 0;
int h = size - 1;
int m;
bool found = false;
int index_of_key = -1;

while (!found && h >= l) {
    m = (l + h) / 2;
    if (key == v[m]) {
        index_of_key = m;
        found = true;
    }
    else if (key > v[m]) {
        l = m + 1;
    }
    else {
        h = m - 1;
    }
}
```

Benchmarking degli algoritmi di ricerca

- il *benchmarking* è la determinazione delle performance di un algoritmo
- per gli algoritmi di ricerca la performance coincide con la velocità di esecuzione, ossia il tempo che impiegano a rispondere «ho trovato quello che cercavi all'indice i »
- ovviamente il tempo di esecuzione effettivo dipende dalla macchina su cui si effettuano i test
- ovviamente i test vanno effettuati su array piuttosto grandi per vedere differenze importanti (almeno nell'ordine delle migliaia di elementi)

Benchmark degli algoritmi di ricerca

```
const int size = 10000;
int array[size];
for (int i = 1; i <= size; i++) {
    array[i] = i;
}
int searched_item = -1; //does not exist: worst case for linear search
auto t0 = chrono::high_resolution_clock::now();
//binary search
auto t1 = chrono::high_resolution_clock::now();
cout << "Binary search: " << chrono::duration_cast<chrono::nanoseconds>(t1 - t0).count() <<
" nanoseconds passed\n";
auto t2 = chrono::high_resolution_clock::now();
//linear search
auto t3 = chrono::high_resolution_clock::now();
cout << "Linear search: " << chrono::duration_cast<chrono::nanoseconds>(t1 - t0).count() <<
" nanoseconds passed\n";
```

Binary search: 300 nanoseconds passed
Linear search: 17000 nanoseconds passed