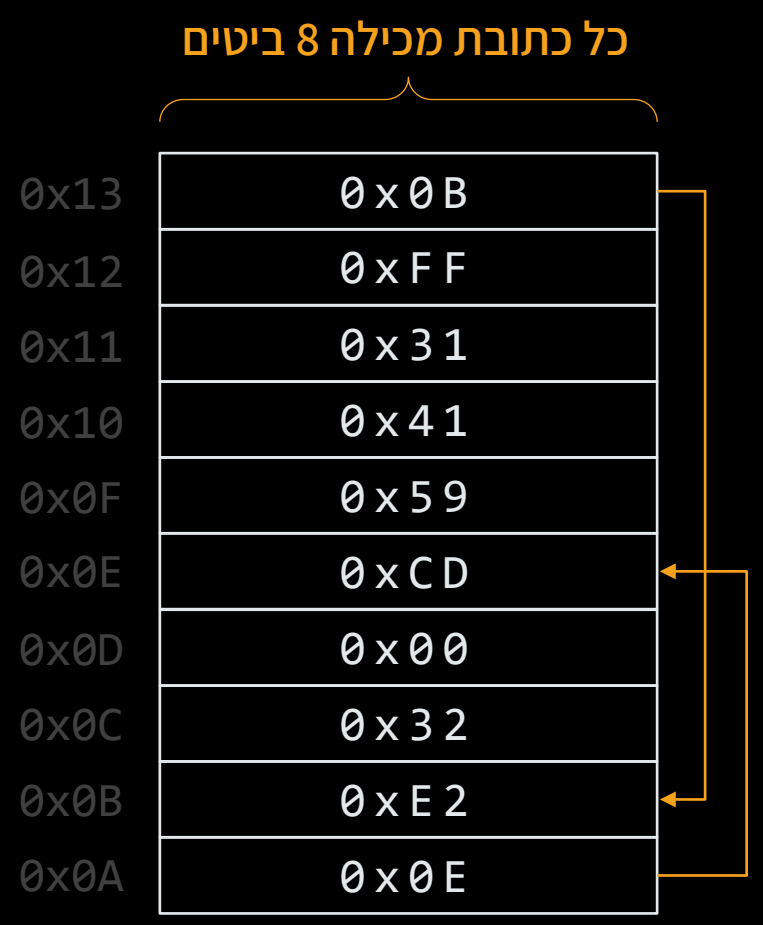
מבני נתונים בסיסיים ו־STL

pair, tuple, vector, list, deque, iterators, set, multiset, map, queue, stack

מבנה הזיכרון במחשב



- הזיכרון כולו הינו לא יותר מאשר סרט אחד ארוך שניתן לקרוא ולכתוב בו אפסים ואחדים
- אי אפשר לגשת לביט יחיד בזיכרון: הגודל הקטן ביותר שניתן לגשת אליו הוא בית (8 ביטים רציפים)
- כל בית או רצף של בתים ניתן לפרש איך שנרצה: בתור רצף בינארי, מספר שלם, מספר ממשי, מבנה נתונים מסובך יותר, או אפילו כתובת למקום אחר בזיכרון
- ניתן לבקש ממערכת ההפעלה מספר בתים מסוים, והיא תקדיש אותו לנו ותביא לנו את הכתובת לבית הראשון
- עבודה בבלוק זיכרון רציף אחד היא נוחה, אך לעיתים לא אפשרית מכיוון שאנו לא יודעים בכמה זיכרון נרצה להשתמש ומתי

קיבוץ נתונים

קיבוץ זוג איברים

```
#include <bits/stdc++.h>
using namespace std;

pair<int, int> minmax(int* arr, int length) {
    int min = arr[0], max = arr[0];
    for (int i = 1; i < length; i++) {
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }
    return {min, max};
}
```

- משתנה מהטיפוס `int*` הוא בעצם מצביע למשתנה מטיפוס `int`
- ניתן לתאר מערך מספרים רציף בזיכרון בתור מצביע לאיבר הראשון במערך ואורכו
- כדי להחזיר יותר מערך אחד מפונקציה נשתמש בטיפוס `pair<A, B>` שמחזיר שני איברים, הראשון מהטיפוס A והשני מהטיפוס B

קיבוץ זוג איברים

```
typedef pair<int, int> ii;

int main() {
    int arr[4] = {4, 2, 3, 1};
    pair<int, int> ans = minmax(arr, 4);
    cout << "Min is " << ans.first << endl;
    cout << "Max is " << ans.second << endl;
}
```

- כדי לגשת לאיבר הראשון בזוג, נשתמש ב־`first`.
- כדי לגשת לאיבר השני בזוג, נשתמש ב־`second`.
- קיבוץ זוג מספרים שלמים נפוץ מאוד בתכנות תחרותי, ולכן בדרך כלל נשתמש בקיצור:

```
typedef pair<int, int> ii;
```

מ־יה סדורה (טאפל)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    tuple<int, double, char> t = {1, 2.0, 'x'};

    cout << get<1>(t) << endl;  // prints 2.0
    get<2>(t) = 'y';  // t = {1, 2.0, 'y'}

    int a;
    double b;
    char c;
    tie(a, b, c) = t;   // a=1, b=2.0, c='y'
    cout << c << endl;  // prints 'y'
}
```

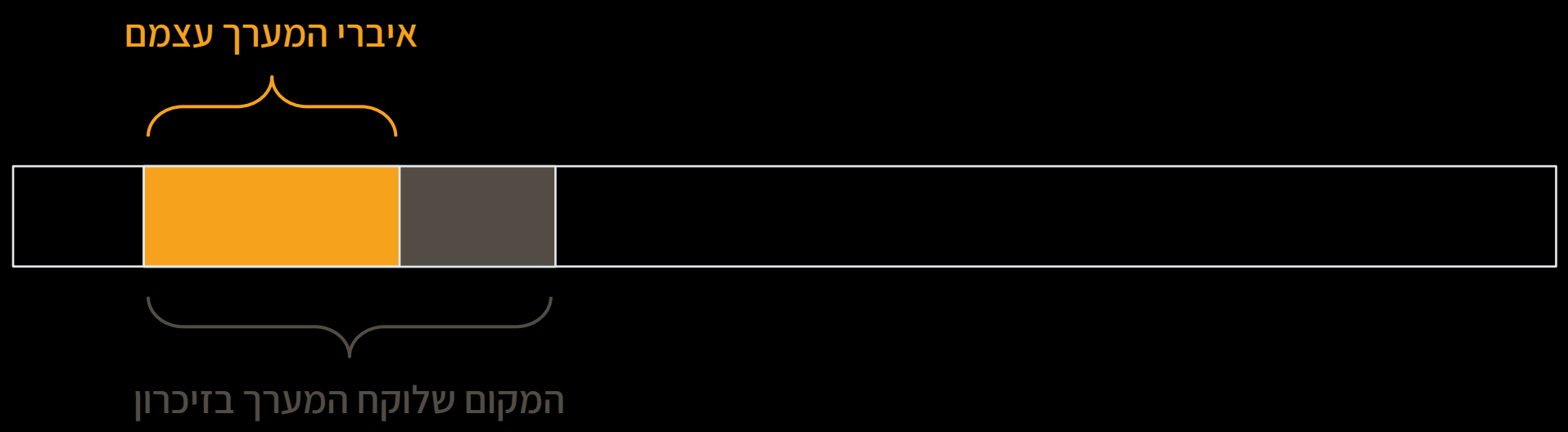
- אם נרצה לקבץ שלושה או יותר איברים, נוכל להשתמש ב־`tuple<A, B, …>`
- כדי לגשת לאיבר ה־`i` בטאפל, נשתמש בפונקציה `get<i>` ולה נעביר את הטאפל
- אם נרצה להעתיק את ערכי הטאפל למשתנים חדשים, נוכל לעשות זאת בעזרת הפונקציה `tie`
- שינוי המשתנים החדשים לאחר `tie` אינו ישפיע על איברי הטאפל

מבני נתונים מסודרים

Ordered containers

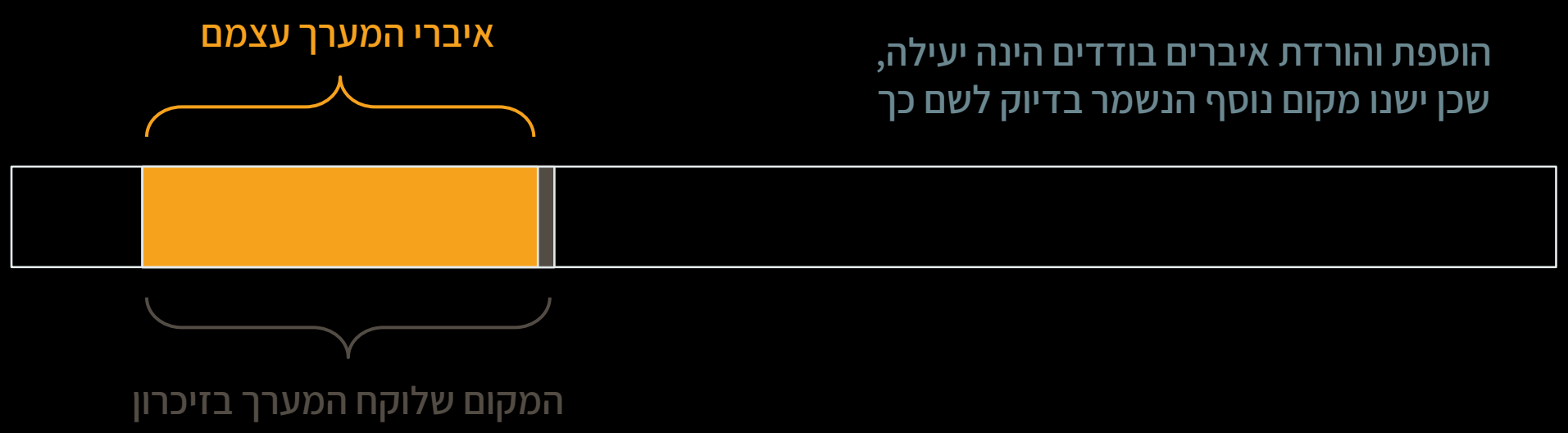
ווקטור

- מערכי C (כמו שראינו עד עכשיו) הם מהירים מאוד, אך השימוש בהם לא נוח
- גודל מערכי C מוגבל וקבוע מראש, והעבודה איתם מסורבלת
- הטיפוס `vector<T>` הוא מעין מעטפת למערך C רגיל
- גודל המערך אינו קבוע: ברגע שנוסיף מספיק איברים, המערך יועתק למקום גדול יותר בזיכרון



ווקטור

- מערכי C (כמו שראינו עד עכשיו) הם מהירים מאוד, אך השימוש בהם לא נוח
- גודל מערכי C מוגבל וקבוע מראש, והעבודה איתם מסורבלת
- הטיפוס `vector<T>` הוא מעין מעטפת למערך C רגיל
- גודל המערך אינו קבוע: ברגע שנוסיף מספיק איברים, המערך יועתק למקום גדול יותר בזיכרון



הוספת והורדת איברים בודדים הינה יעילה,
שכן ישנו מקום נוסף הנשמר בדיוק לשם כך

ווקטור

- מערכי C (כמו שראינו עד עכשיו) הם מהירים מאוד, אך השימוש בהם לא נוח
- גודל מערכי C מוגבל וקבוע מראש, והעבודה איתם מסורבלת
- הטיפוס `vector<T>` הוא מעין מעטפת למערך C רגיל
- גודל המערך אינו קבוע: ברגע שנוסיף מספיק איברים, המערך יועתק למקום גדול יותר בזיכרון

דוגמה לשימוש בווקטור

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;

int main() {
    vi vec(3, 0);                  // vec = {0, 0, 0}
    vec.push_back(1);              // vec = {0, 0, 0, 1}
    vec[1] = 2;                    // vec = {0, 2, 0, 1}
    cout << vec[3] << endl;        // prints 1
    vec.pop_back();                // vec = {0, 2, 0}
    cout << vec.size() << endl;    // prints 3
    vec.clear();                   // vec = {}
    cout << vec.size() << endl;    // prints 0
}
```

דק (DEQUE)

- מבנה נתונים רציף בזיכרון
- בעל גישה קבועה לאיבר רנדומלי כמו בווקטור
- מאפשר הוספת איברים חדשים גם לתחילתו בסיבוכיות ממוצעת טובה
- למרות סיבוכיות תיאורטית זהה, הדק הרבה יותר איטי ובזבזני מווקטור

איטרטורים

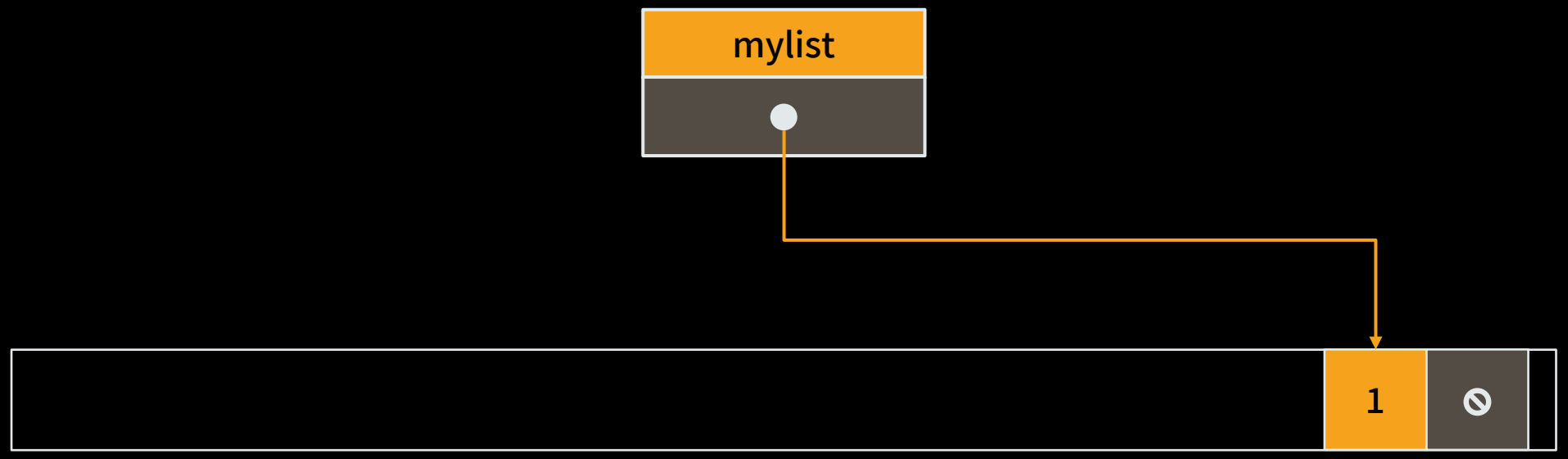
- Iterator־ים הם מעין מצביעים חכמים לתוך אובייקטים במבנה הנתונים
- בהינתן איטרטור `it` (מצביע לאיבר), נוכל לגשת לערכו של האיבר בעזרת `*it`
- לרוב מבני הנתונים ב־STL קיימות המתודות `begin()`, `end()`
- `begin()` מצביע לאיבר הראשון במבנה, בעוד ש־`end()` מצביע לאיבר אחד אחרי האחרון (לא קיים)
- פונקציות כמו `find` מקבלות טווח ערכים בעזרת איטרטורים שעליו תתבצע הפעולה
- בדרך כלל טווח איטרטורים ייוצג על ידי $[a,b)$ ויכלול את האיבר הראשון a, אך לא את האיבר האחרון b
- פונקציות כמו `find` יחזירו איטרטור כפלט. האיטרטור `end()` יוחזר אם התשובה לא נמצאה
- ניתן לקדם את האיטרטור לאיבר הבא באמצעות `it++`, או לאיבר הקודם באמצעות `it--`

רשימה מקושרת

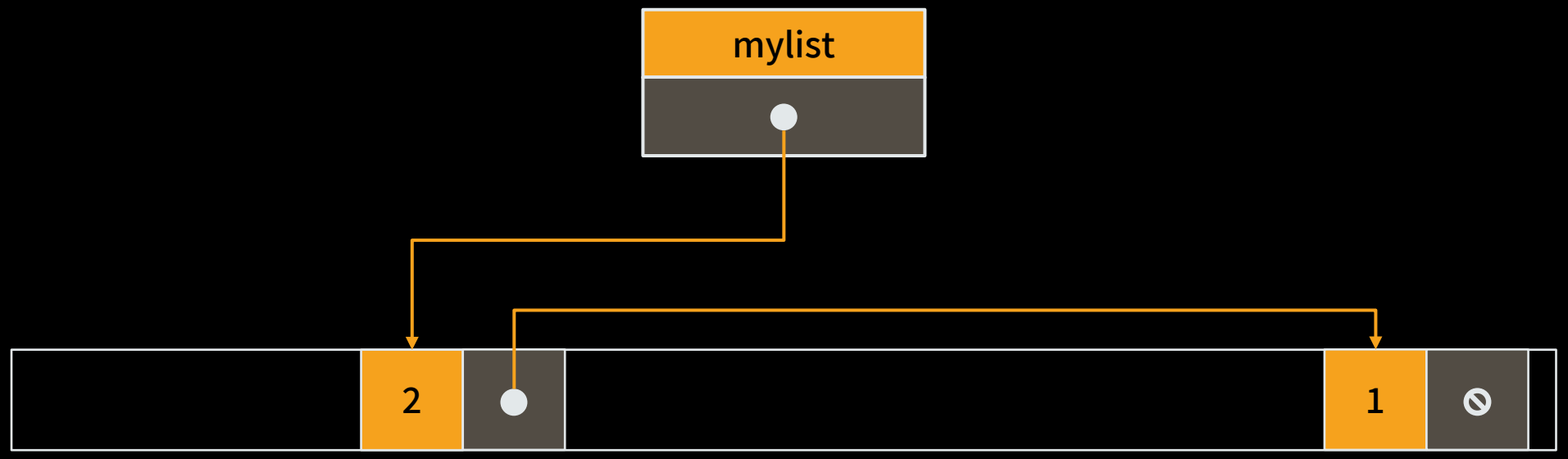
- מבנה נתונים שאינו רציף בזיכרון
- כל איבר ברשימה יחזיק את ערכו ומצביע לאיבר הבא
- בחלק מהמימושים נתחזק גם מצביע לאיבר הקודם
- גישה לאיבר רנדומלי יקרה – נצטרך לעבור על כל הרשימה עד לאיבר שנרצה
- בהינתן מצביע לאיבר נוכל להוסיף איבר חדש לפניו או אחריו בזמן קבוע
- ניתן למזג שתי רשימות לאחת בזמן קבוע
- פחות נפוץ בתכנות תחרותי, אך שימושי לבעיות מסוימות

בניית רשימה מקושרת

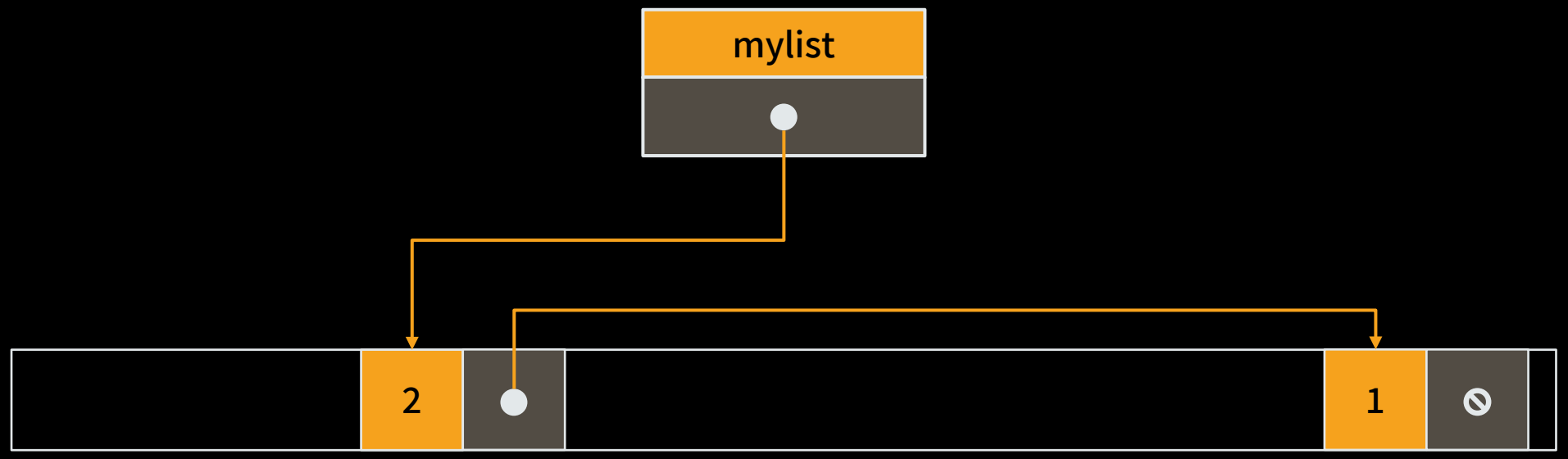
בניית רשימה מקושרת



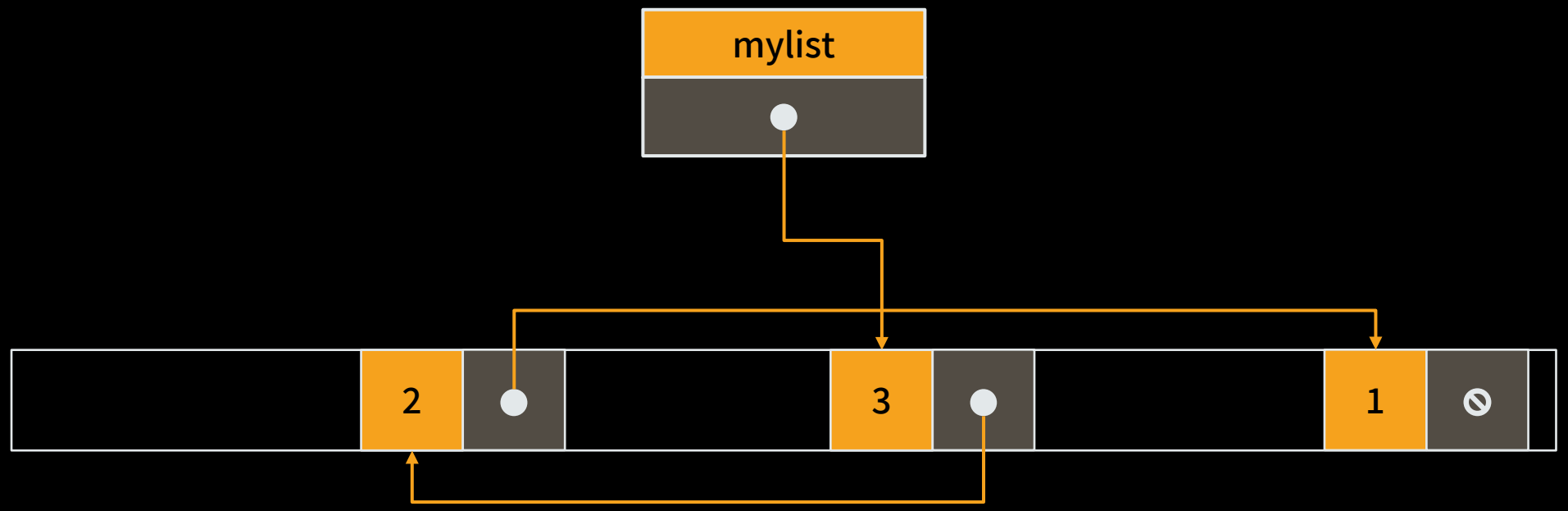
בניית רשימה מקושרת



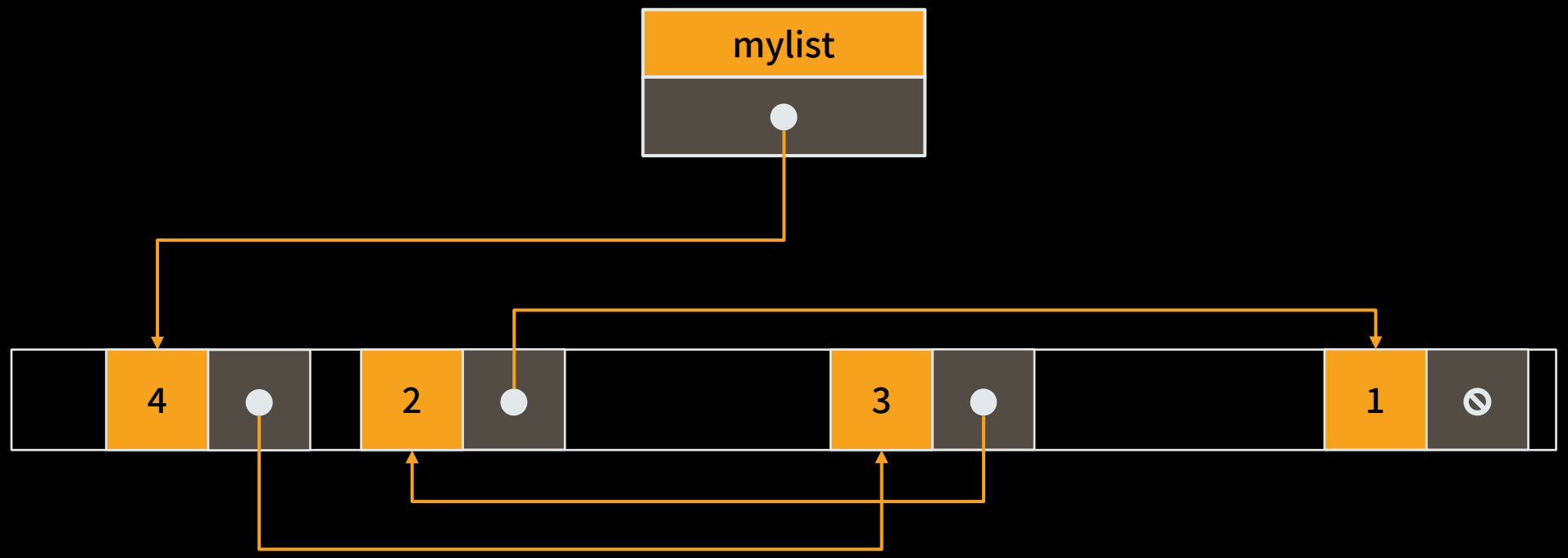
בניית רשימה מקושרת



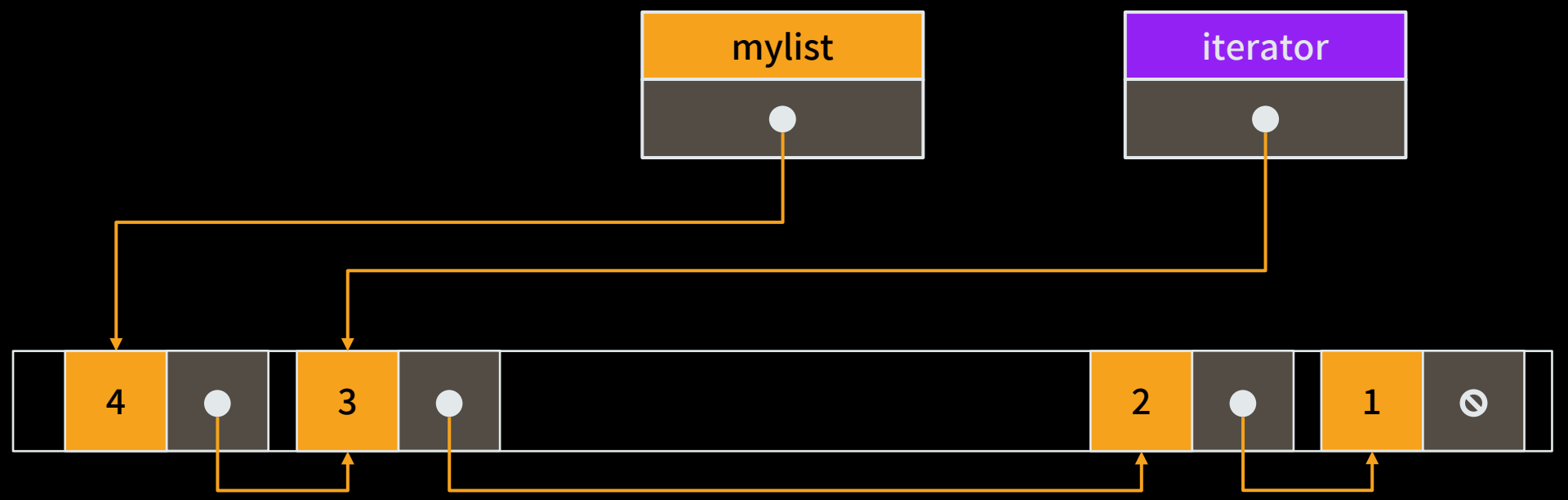
בניית רשימה מקושרת



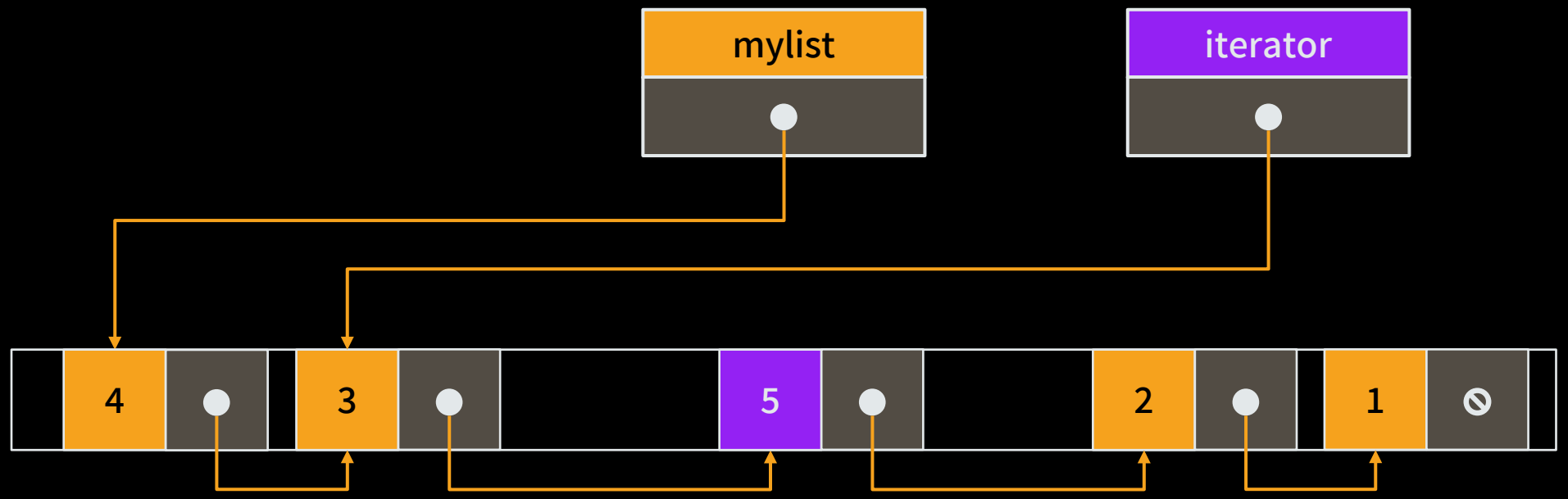
בניית רשימה מקושרת



הוספת איבר לרשימה מקושרת



הוספת איבר לרשימה מקושרת



דוגמה לשימוש ברשימה מקושרת

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    list<int> mylist = {7, 5, 16, 8};

    mylist.push_front(25);
    mylist.push_back(13);
    // mylist = {25, 7, 5, 16, 8, 13}

    // find the iterator (pointer) to 16
    auto it = find(mylist.begin(), mylist.end(), 16);

    // insert 42 after 16
    if (it != mylist.end()) mylist.insert(++it, 42);

    // prints mylist = {25, 7, 5, 16, 42, 8, 13, }
    cout << "mylist = {";
    for (int n : mylist) cout << n << ", ";
    cout << "}" << endl;
}
```

- טיפוס הרשימה המקושרת ב־STL (`list`) ממומש כרשימה מקושרת דו-כיוונית
- ניתן להוסיף לתחילתה ולסופה של הרשימה איברים בסיבוכיות קבועה
- כדי לגשת לאיבר ה־i ברשימה, נצטרך לטייל ברשימה על כל האיברים לפניו בסיבוכיות לינארית
- הכוח של רשימה הוא שבהינתן מצביע בתוך הרשימה, ניתן להוסיף איבר נוסף אחריו או לפניו בזמן קבוע
- בניגוד ל־`vector`, אינו תופס מקום מיותר שלא באמת נמצא בשימוש

מבני נתונים אסוציאטיביים

Associative containers

חיפוש בינארי

79?

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

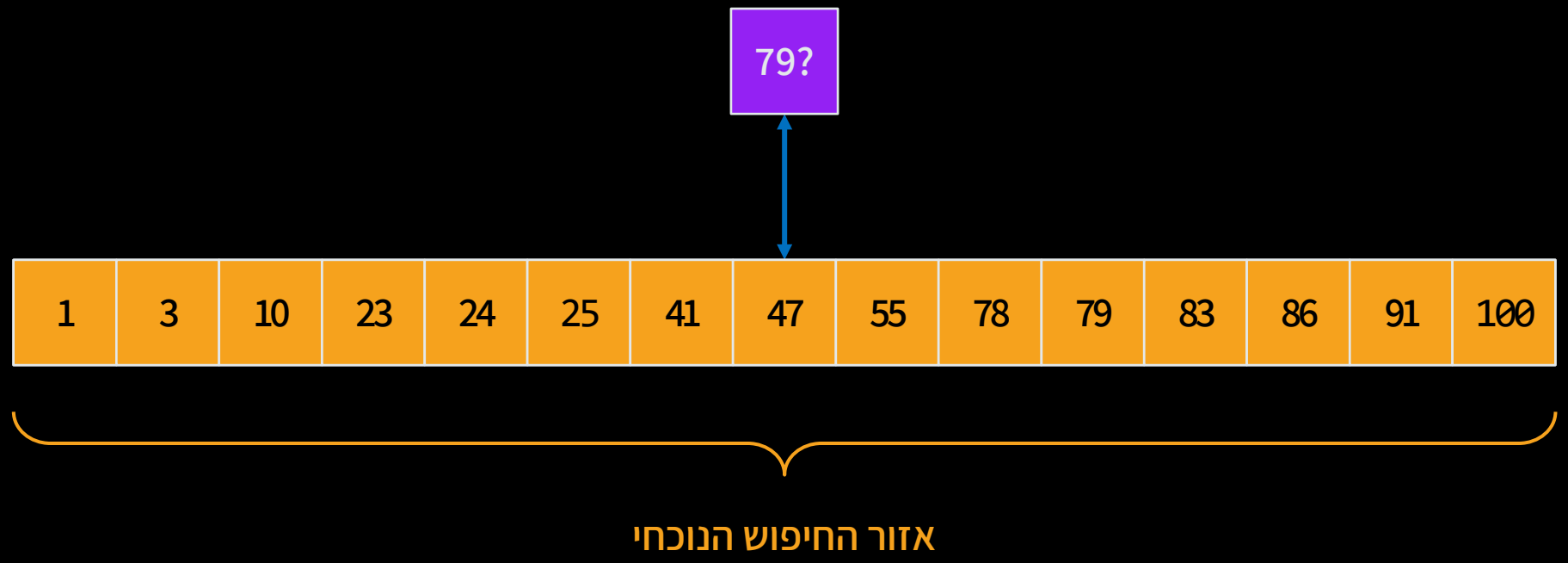
חיפוש בינארי

79?

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

אזור החיפוש הנוכחי

חיפוש בינארי



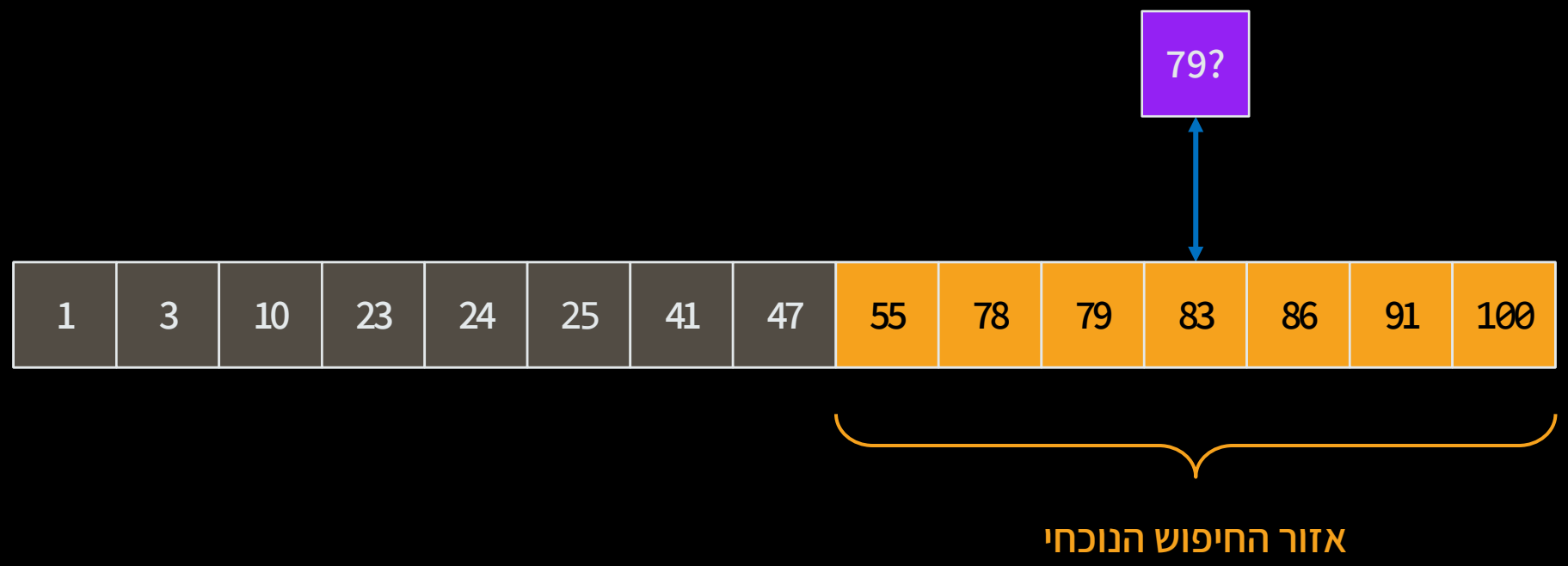
חיפוש בינארי

79?

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

אזור החיפוש הנוכחי

חיפוש בינארי



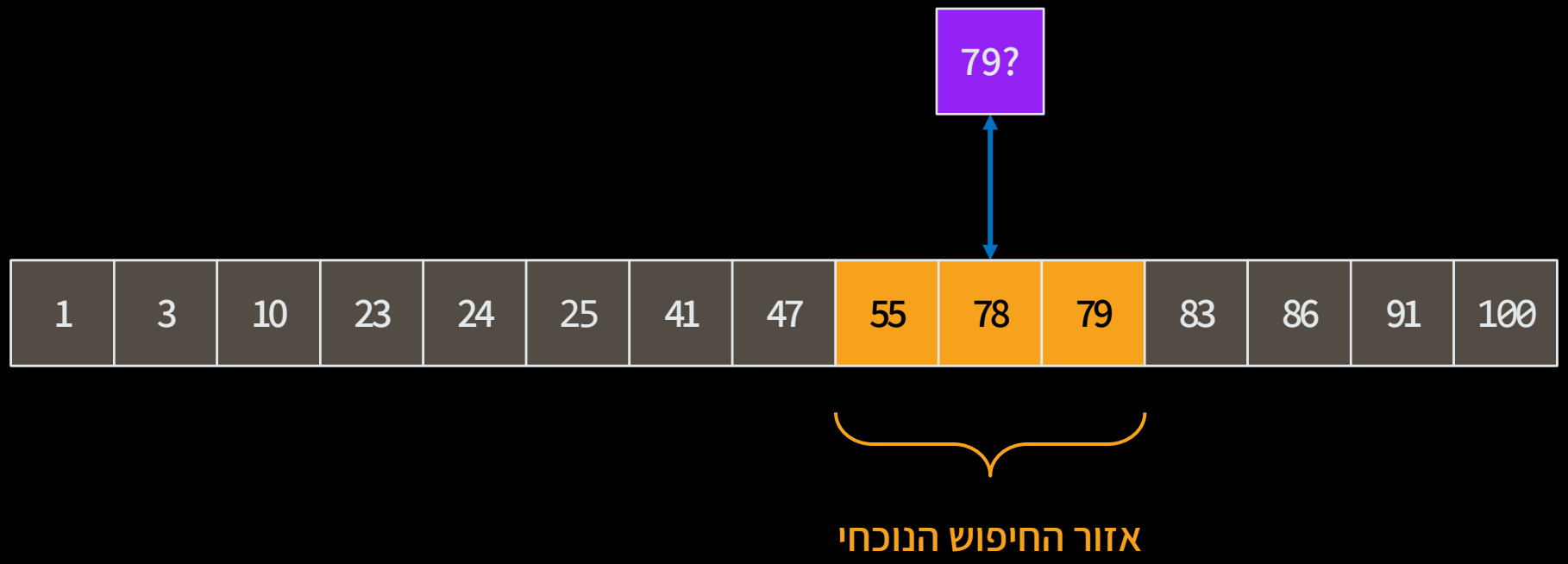
חיפוש בינארי

79?

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

אזור החיפוש הנוכחי

חיפוש בינארי



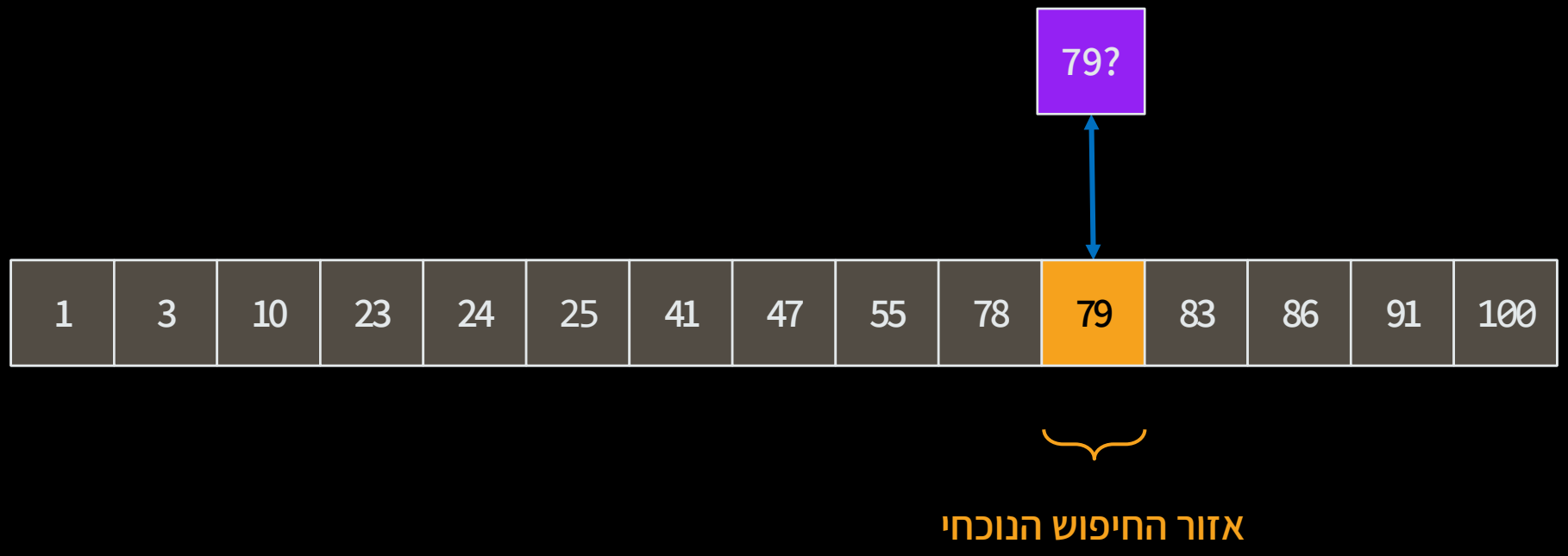
חיפוש בינארי

79?

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

אזור החיפוש הנוכחי

חיפוש בינארי



חיפוש בינארי

79!

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

אזור החיפוש הנוכחי

חיפוש בינארי

- נדרוש רשימת איברים ממוינים המקיימים יחס סדר בניהם
- בהינתן איבר, נוכל לדעת האם הוא נמצא ברשימה או לא, ואם כן נמצא אותו
- בכל בדיקה (השוואה) נחלק את אזור החיפוש שלנו ב~ $^1/_2$
- אם יש לנו 2^n איברים, נוכל לבצע חיפוש בינארי בלכל היותר n פעולות
- נוכל לייצג סיבוכיות זו בעזרת הפעולה המתמטית $\log$: $a^b = c \Leftrightarrow \log_a c = b$
- במערך של $1{,}000{,}000$ איברים, חיפוש בינארי ייקח לכל היותר $\log_2 1{,}000{,}000 \approx 20$ פעולות! 👍
- בעצם, סיבוכיות החיפוש הבינארי על n איברים היא $O(\log_2 n)$

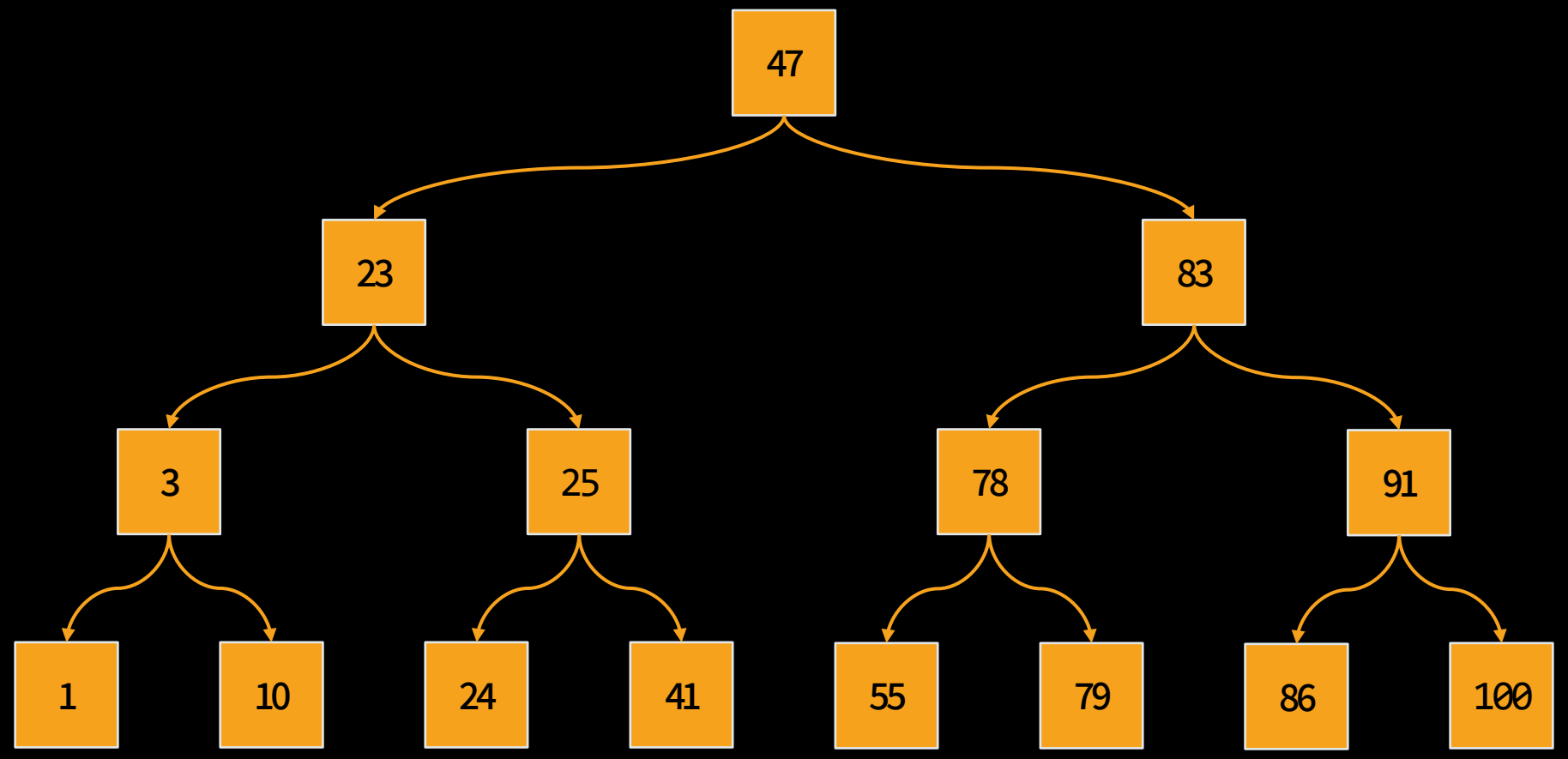
עץ חיפוש בינארי - רציונל

- נרצה מבנה נתונים שנוכל להכניס ולהוציא ממנו איברים, וגם לבדוק האם איבר מסוים נמצא בתוכו
- נוכל לשמור מערך ממוין (vector), ולחפש בו איברים בעזרת חיפוש בינארי בסיבוכיות $O(\log n)$ 👍
- במקרה כזה הבעיה תהיה בשינוי המערך: כיצד נכניס ונסיר איברים ונשמור עליהם ממוינים? 👎
- אם נשתמש ברשימה מקושרת, נוכל להכניס איברים בקלות אך לחפש איבר במערך ייקח $O(n)$ 👎
- ...אנחנו צריכים מבנה נתונים חדש!

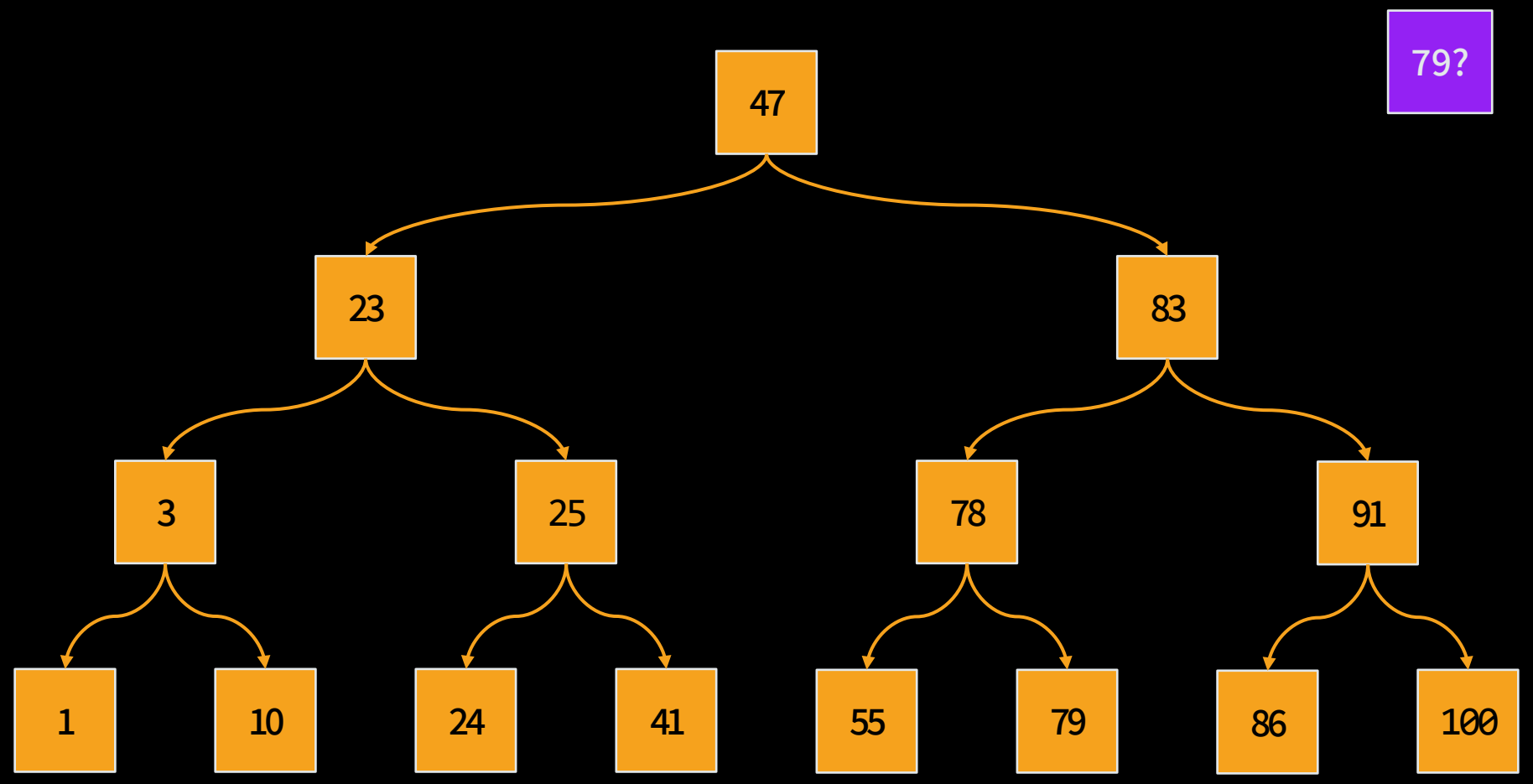
עץ בינארי

1	3	10	23	24	25	41	47	55	78	79	83	86	91	100

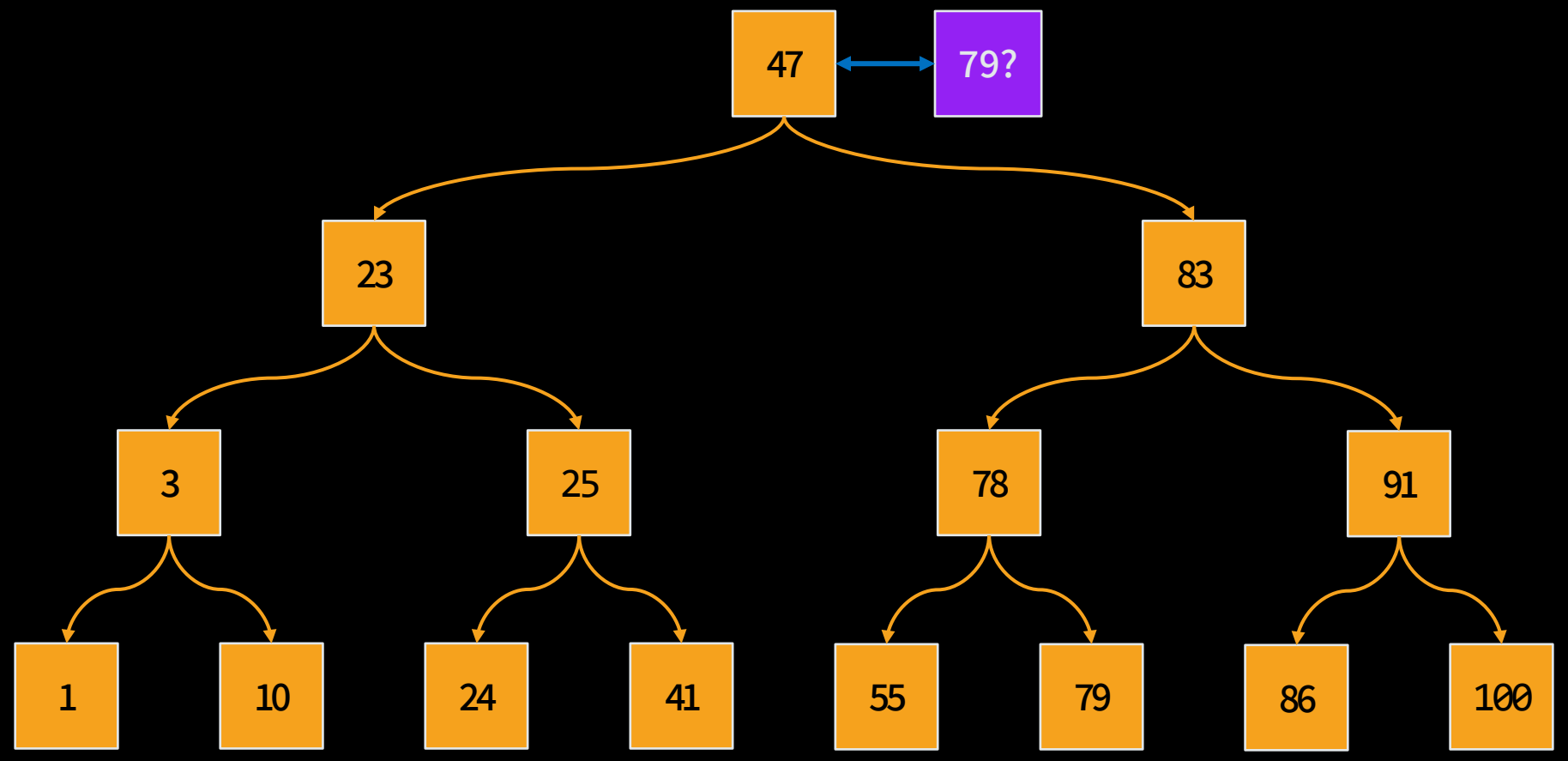
עץ בינארי



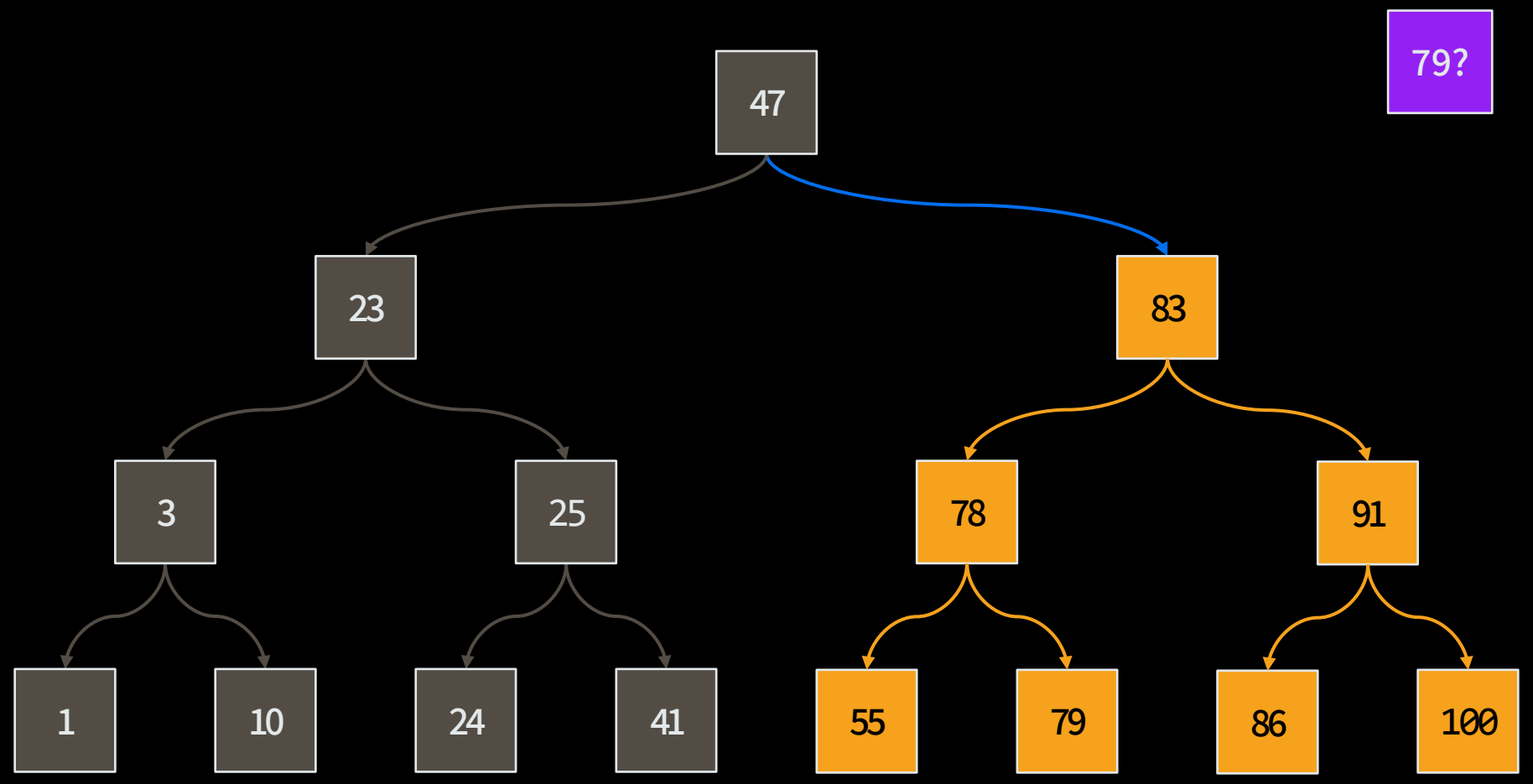
חיפוש בעץ בינארי



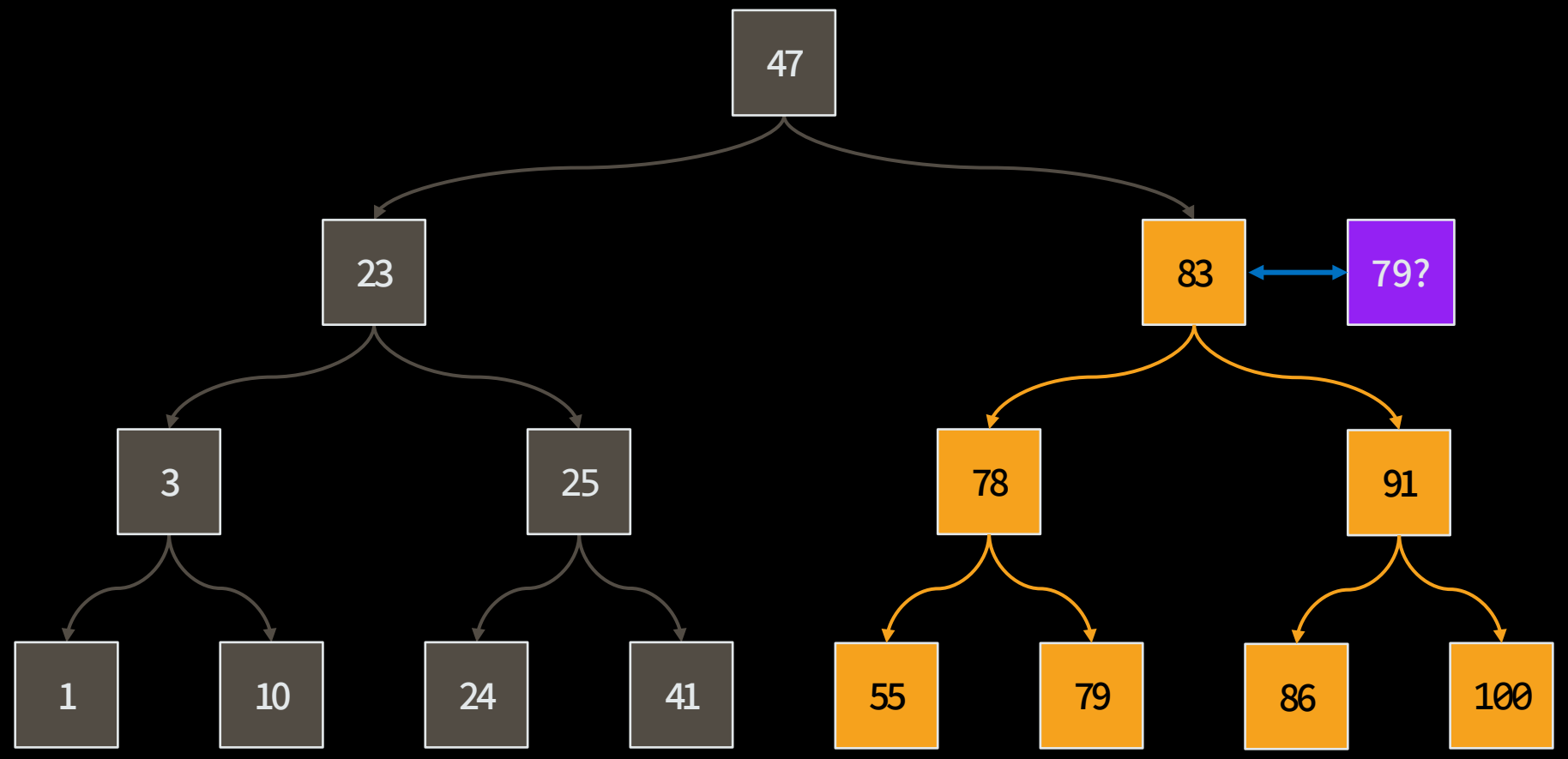
חיפוש בעץ בינארי



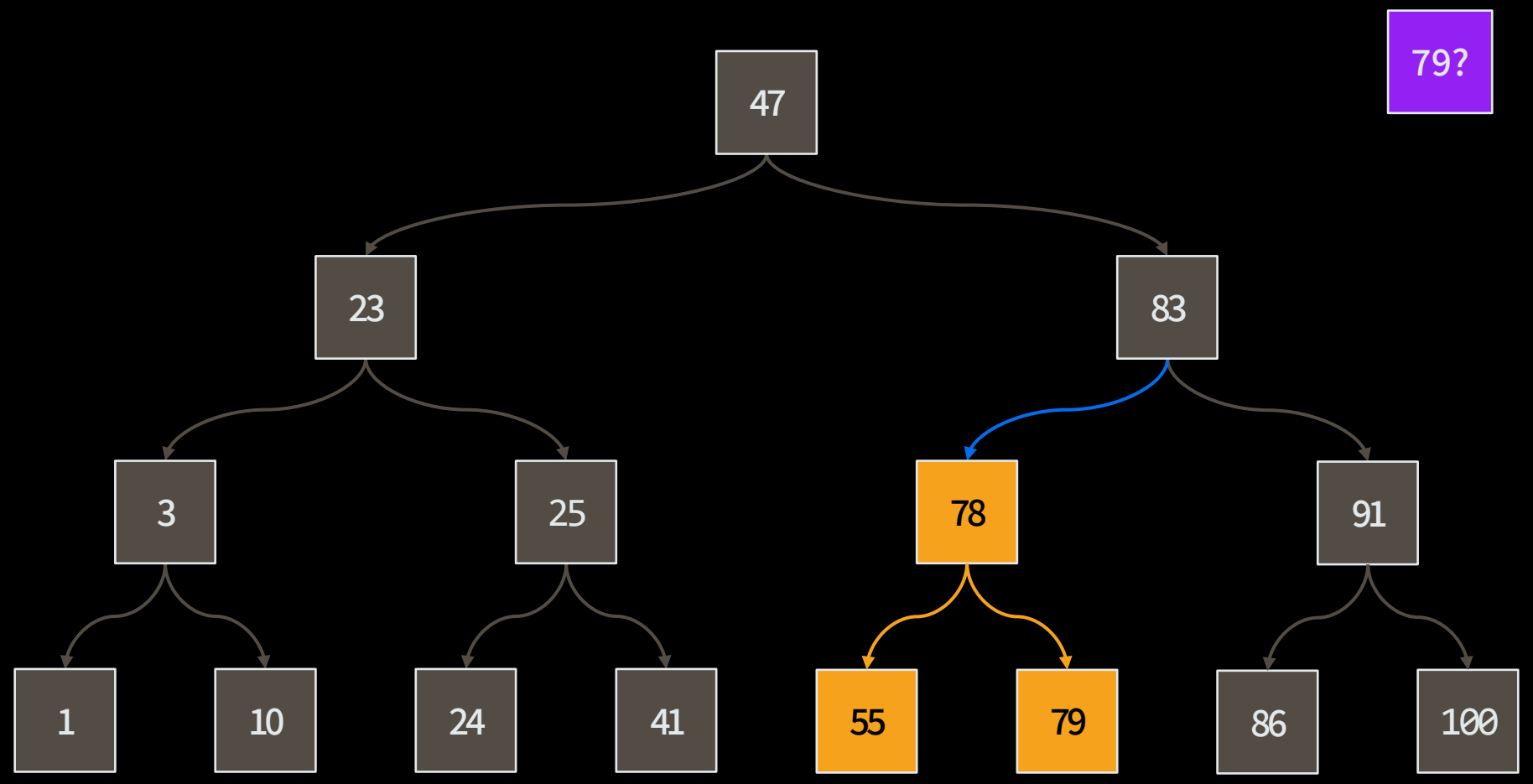
חיפוש בעץ בינארי



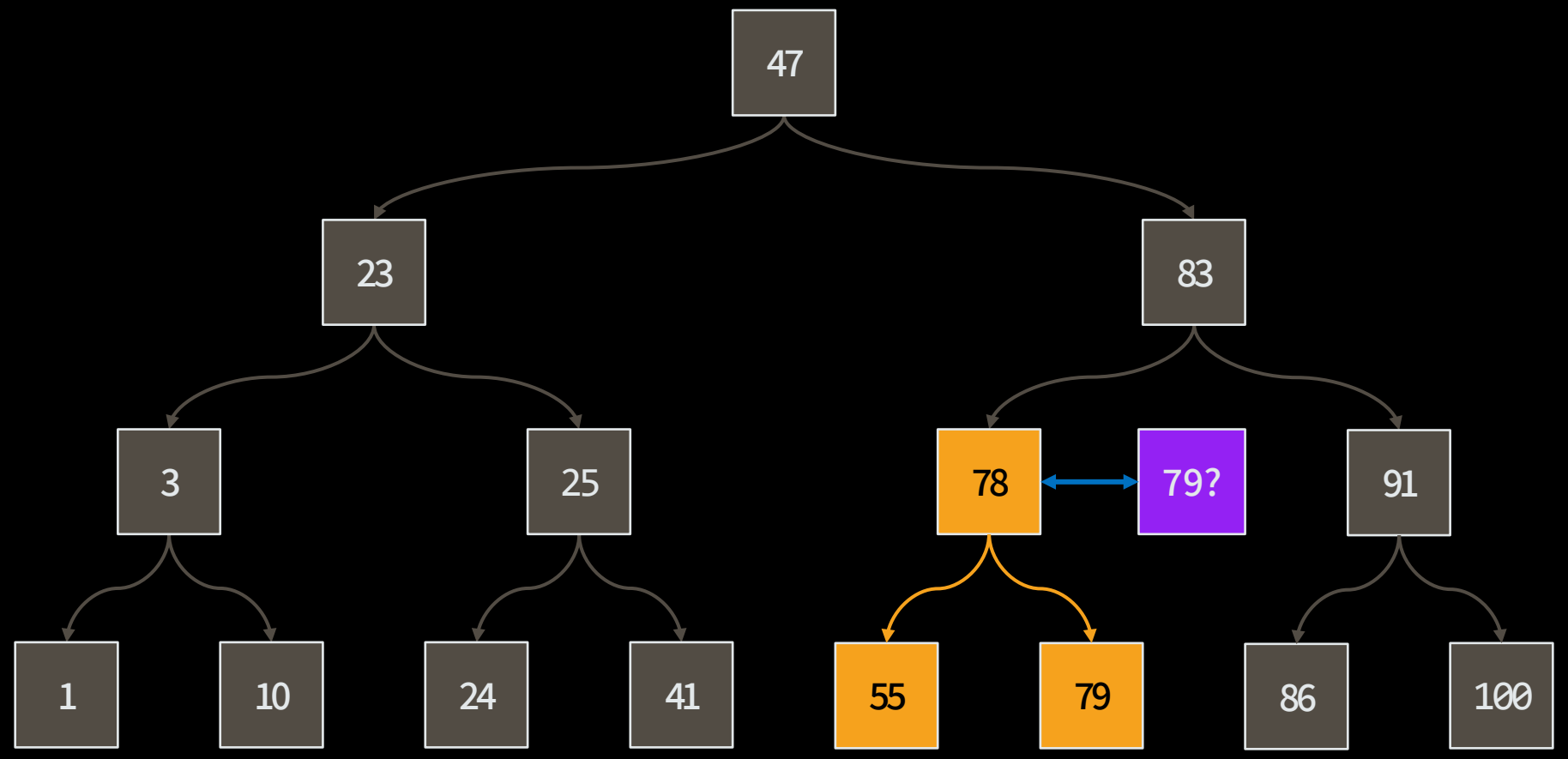
חיפוש בעץ בינארי



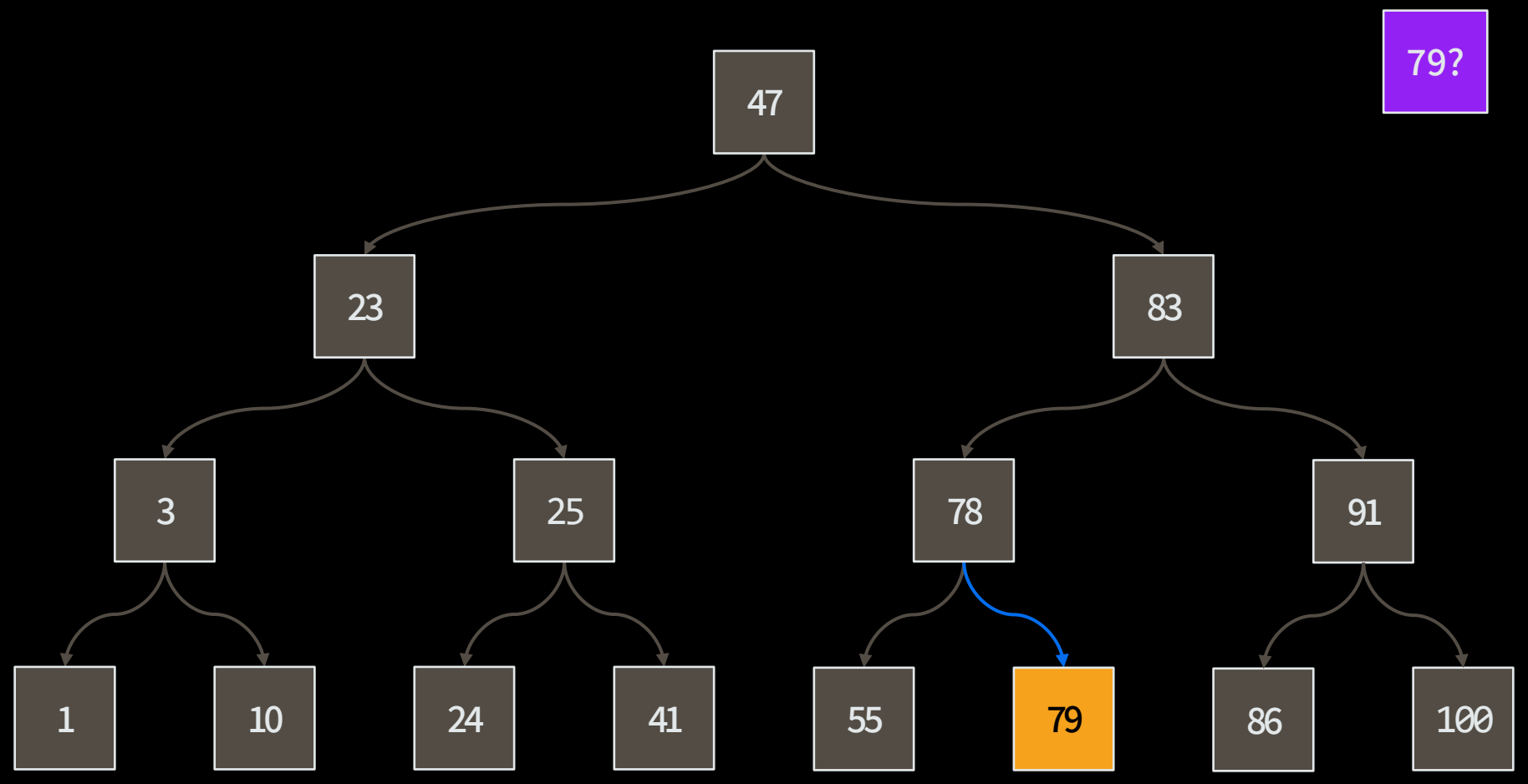
חיפוש בעץ בינארי



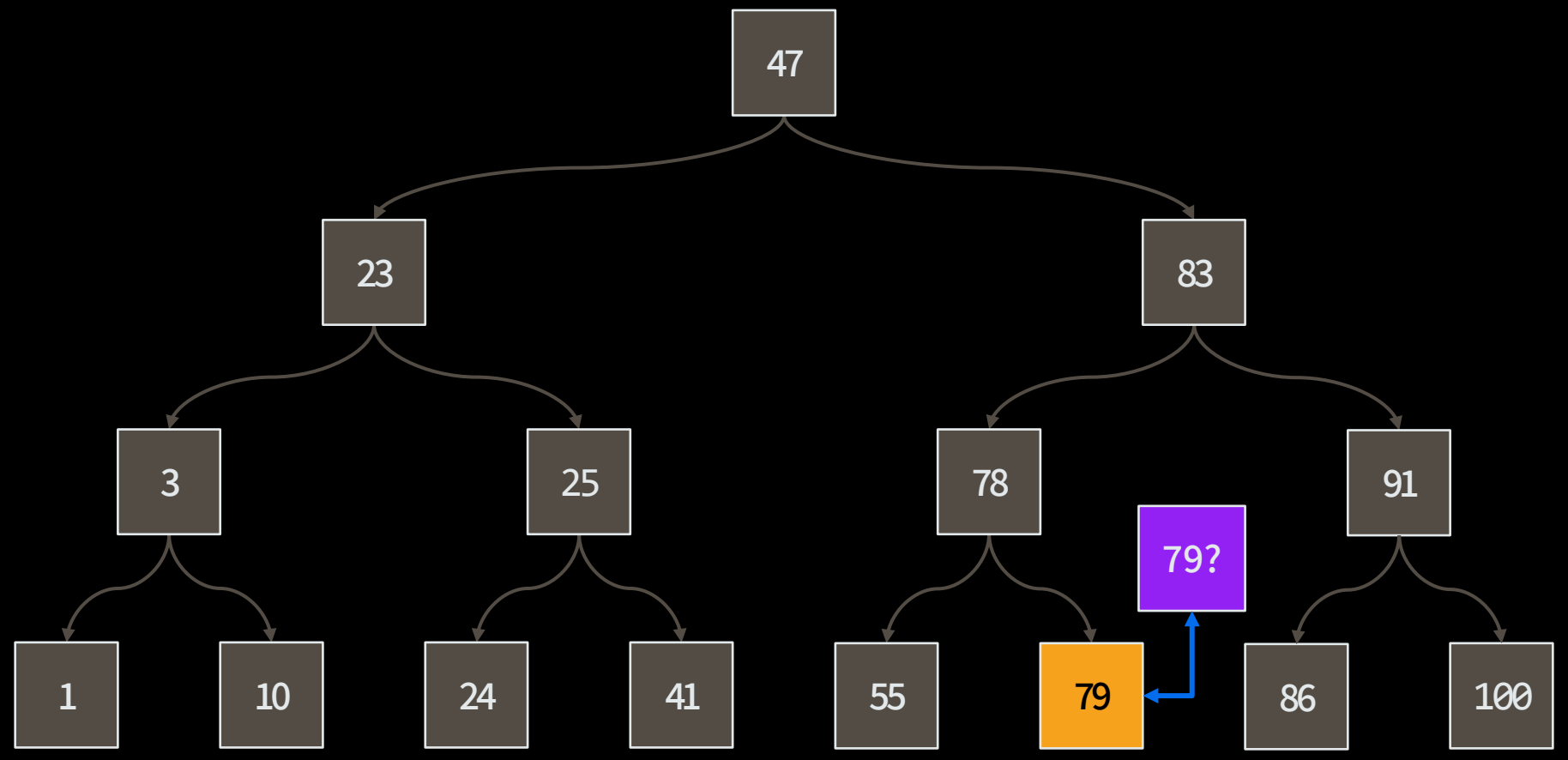
חיפוש בעץ בינארי



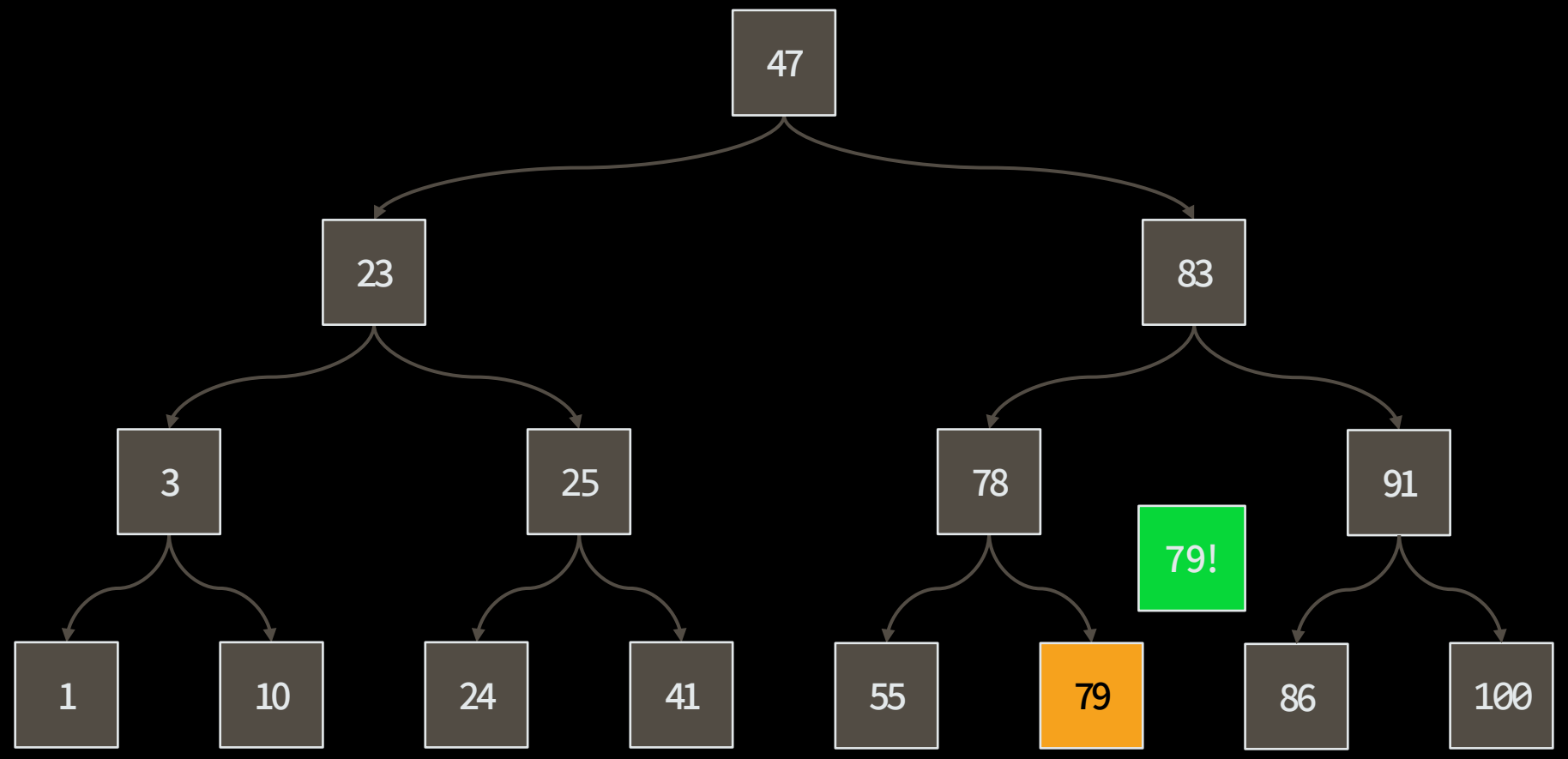
חיפוש בעץ בינארי



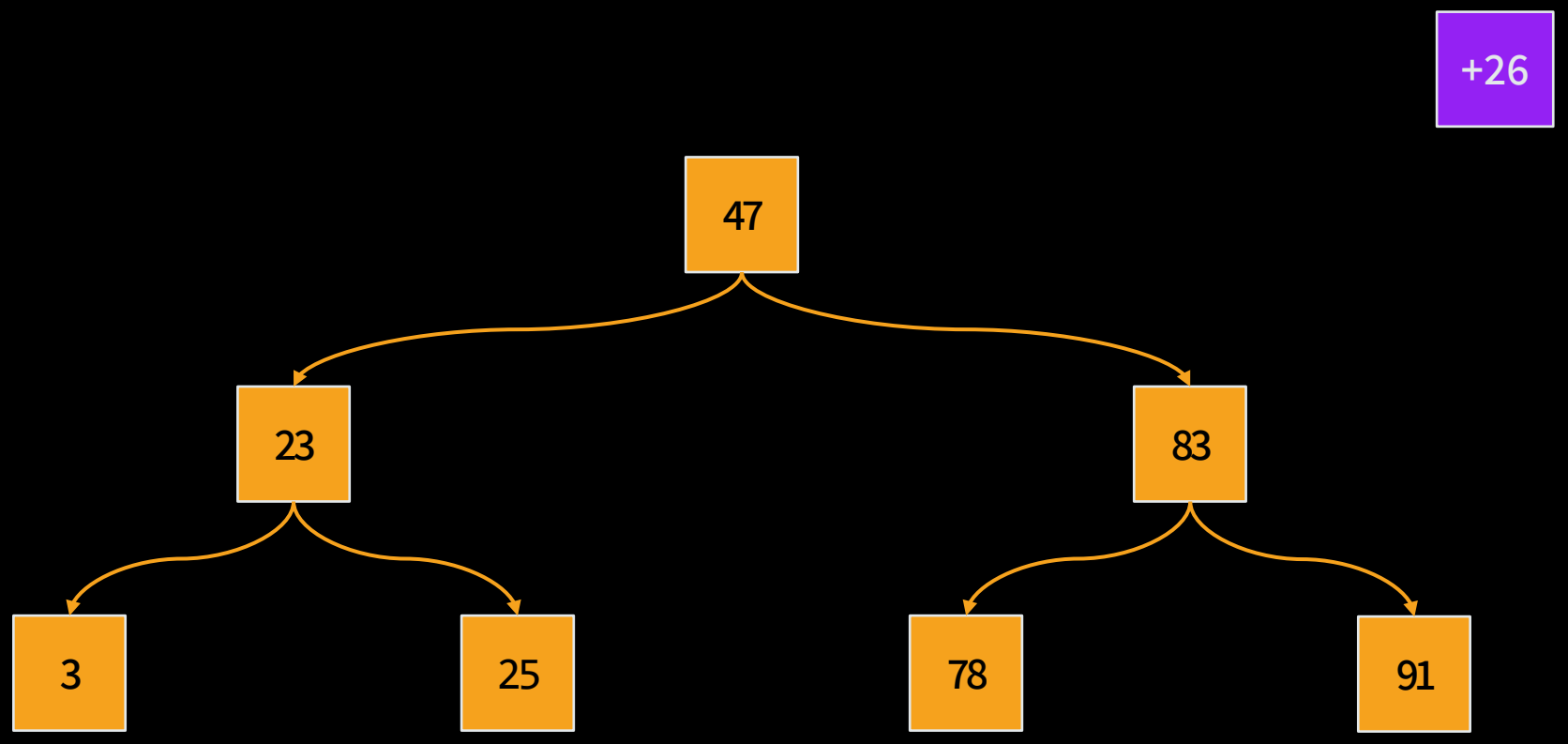
חיפוש בעץ בינארי



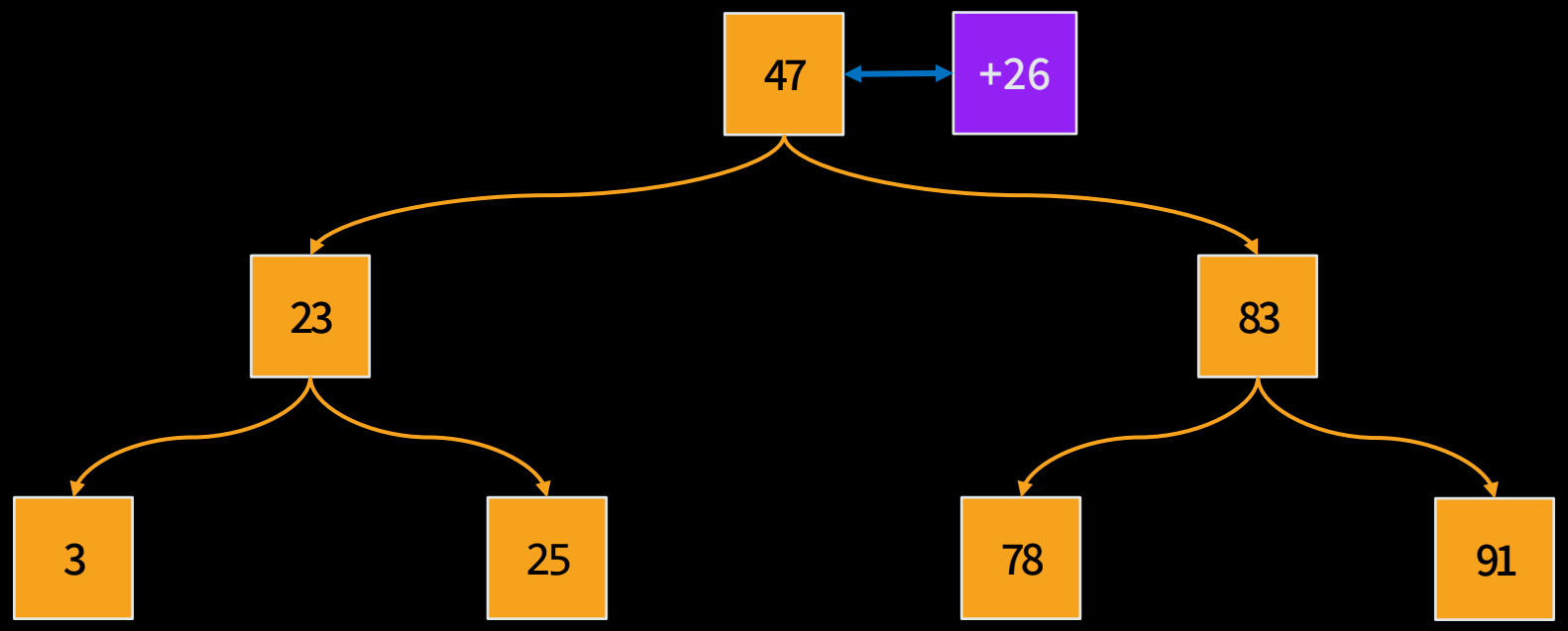
חיפוש בעץ בינארי



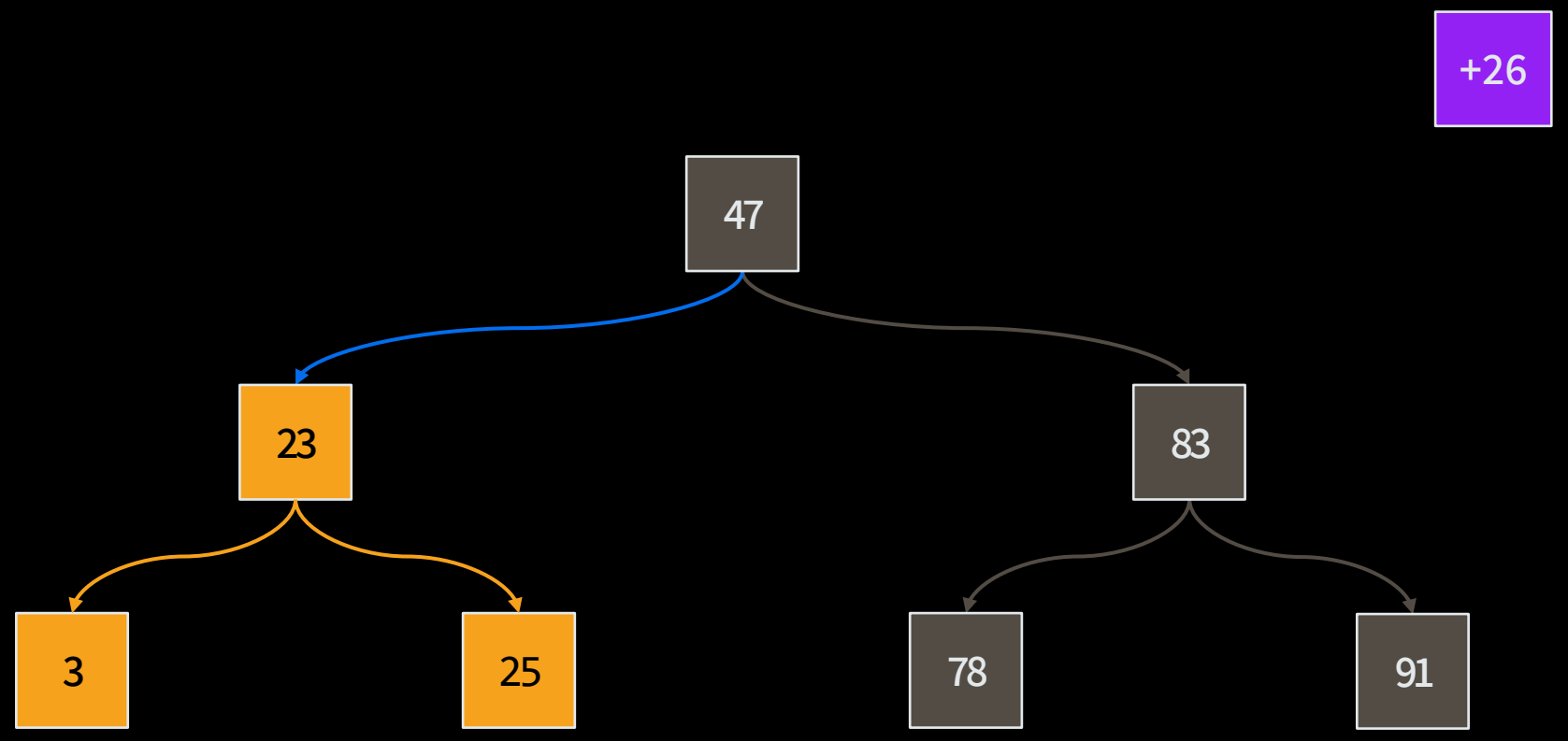
הוספת איבר בעץ בינארי



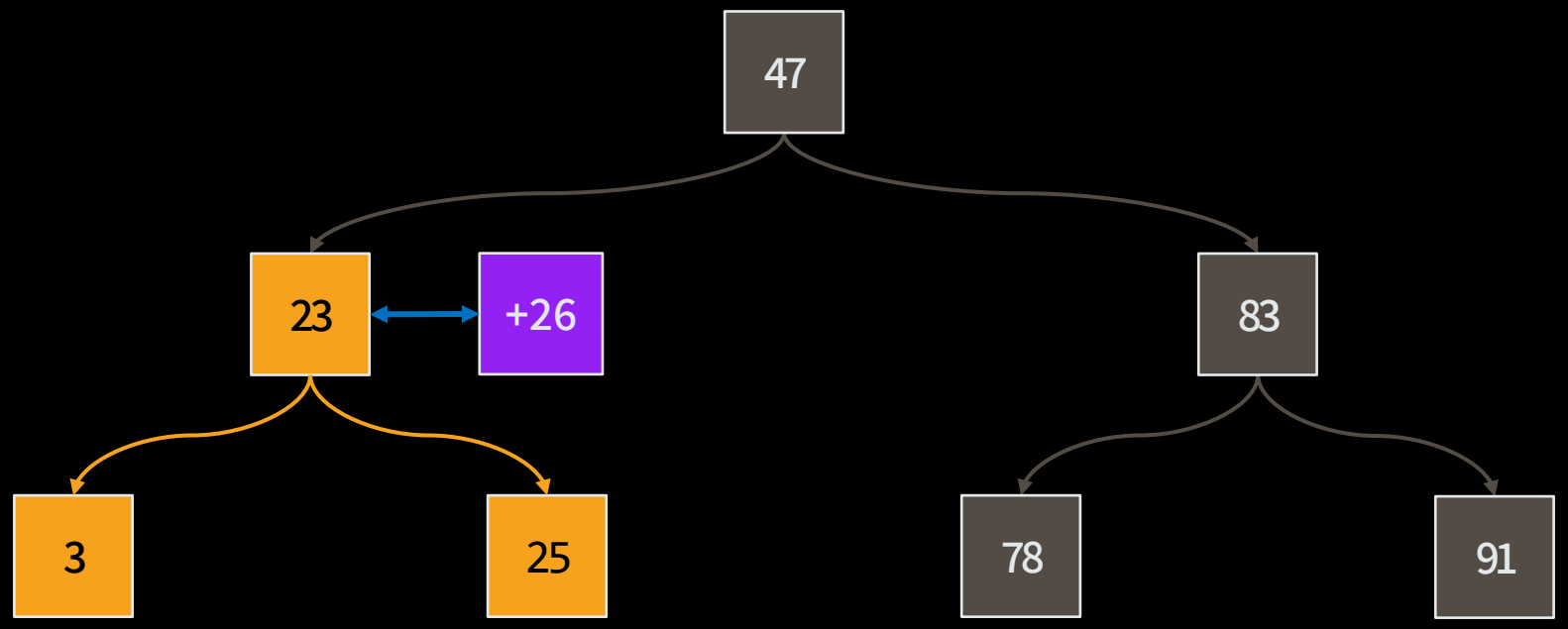
הוספת איבר בעץ בינארי



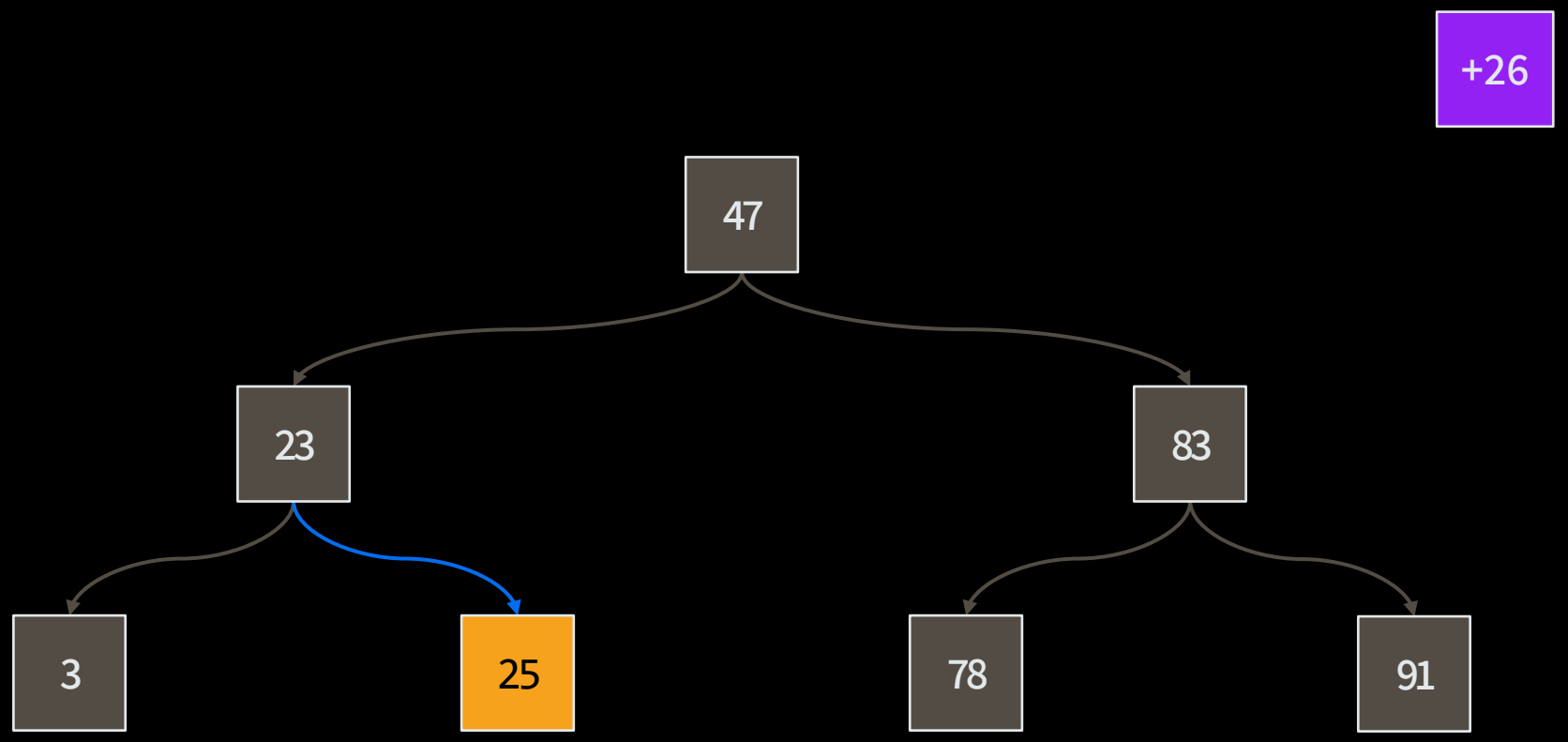
הוספת איבר בעץ בינארי



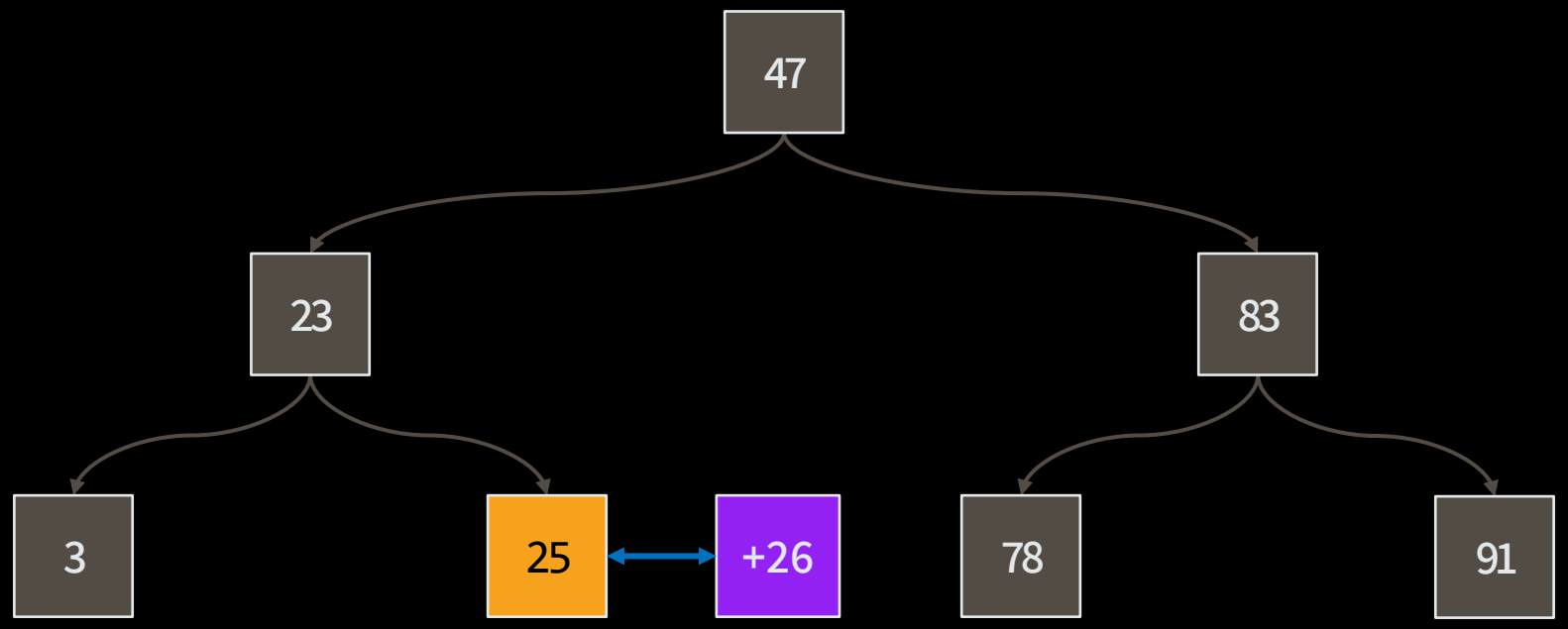
הוספת איבר בעץ בינארי



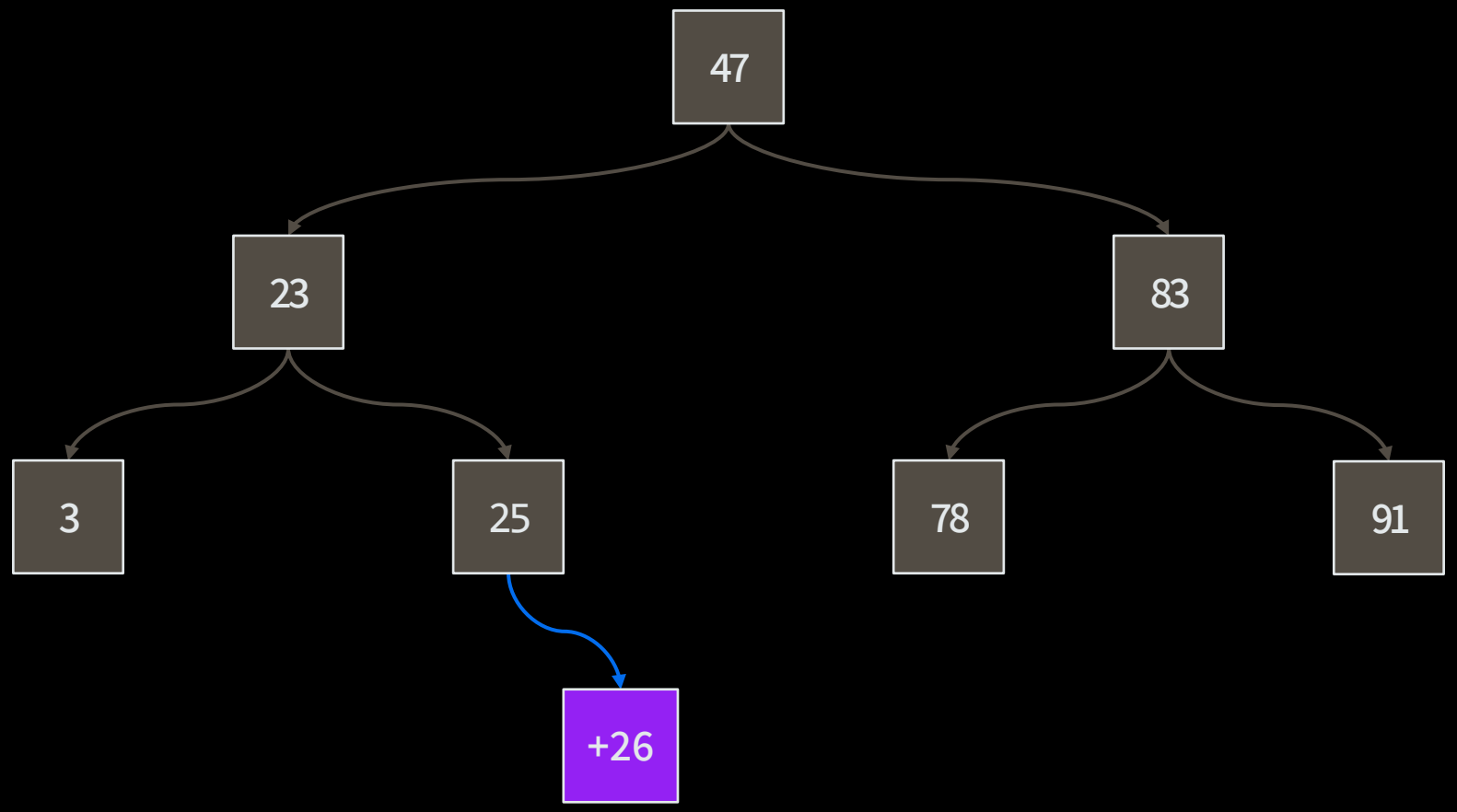
הוספת איבר בעץ בינארי



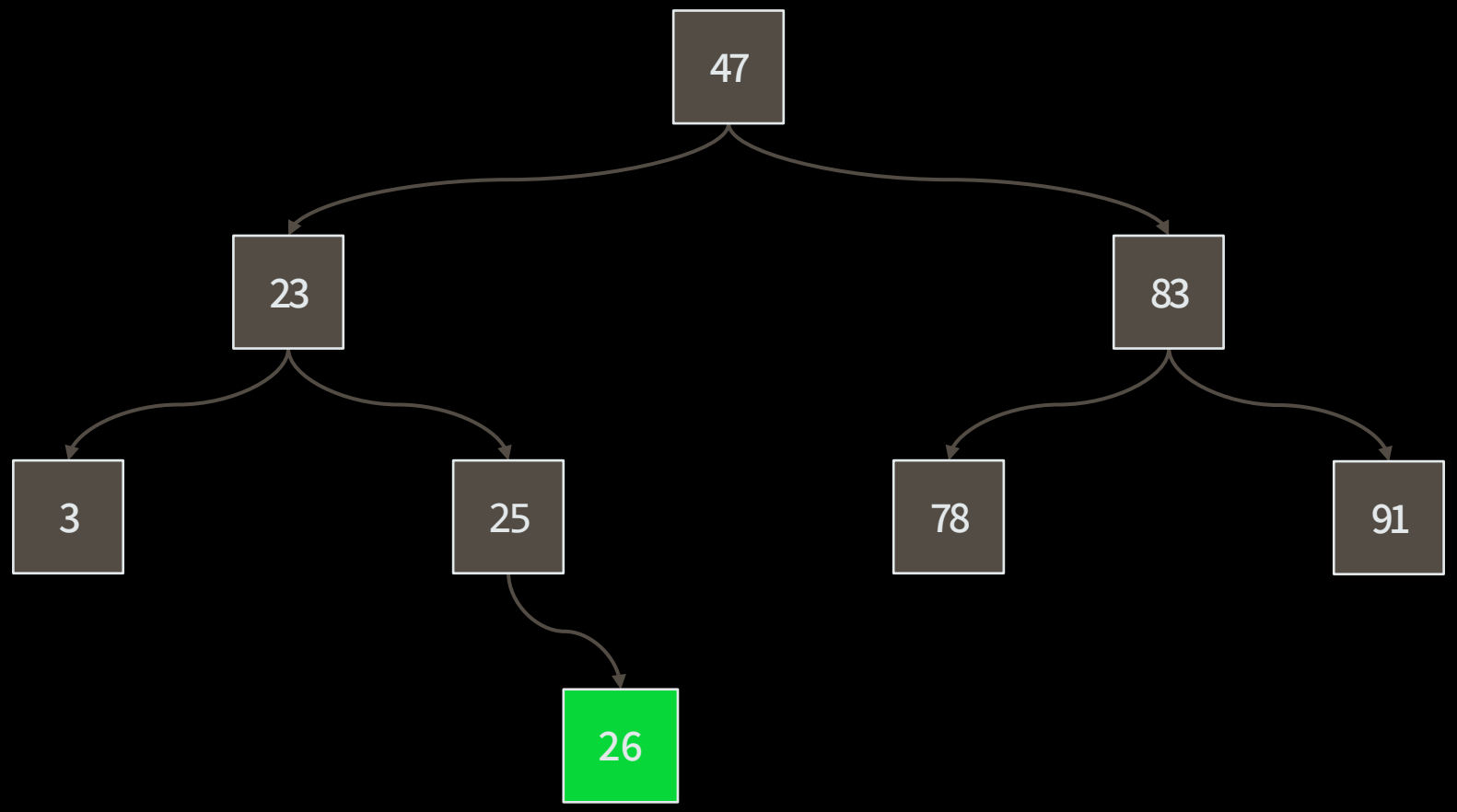
הוספת איבר בעץ בינארי



הוספת איבר בעץ בינארי



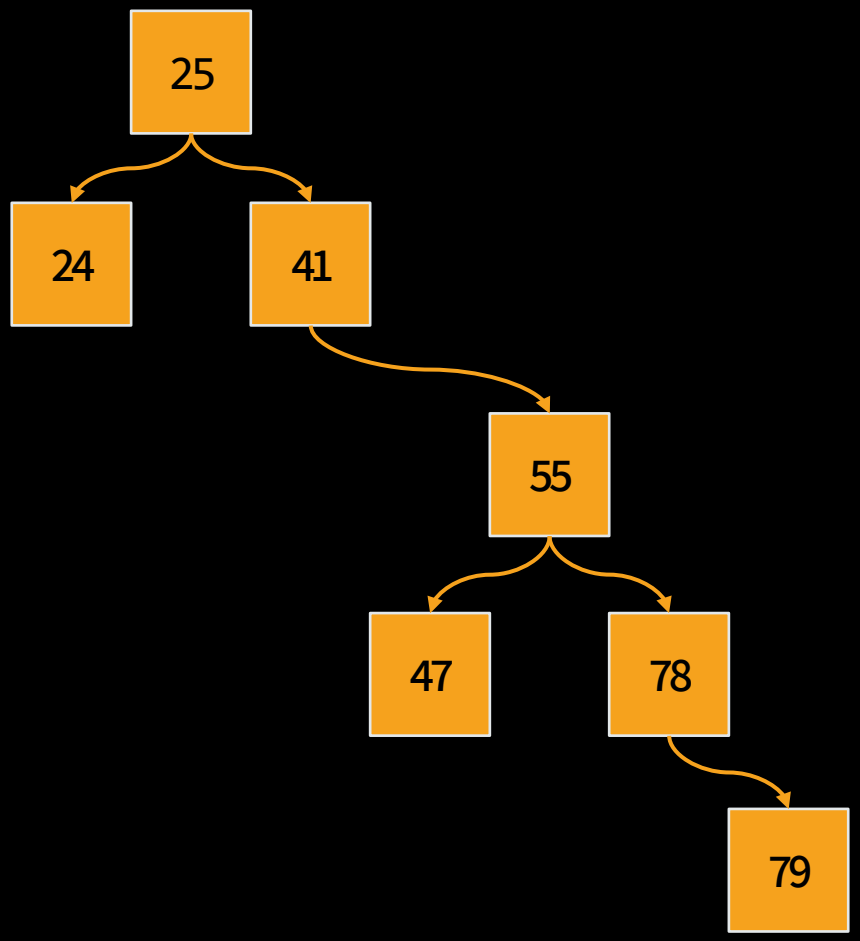
הוספת איבר בעץ בינארי



עץ בינארי שאינו מאוזן

24	25	41	47	55	78	79

עץ בינארי שאינו מאוזן



עץ חיפוש בינארי מאוזן

- עץ בינארי הוא עץ שבו לכל איבר לכל היותר שני בנים – בן ימני ובן שמאלי
- בעץ **חיפוש** בינארי, ערכם של הערכים בתת העץ הימני יהיה גדול ממני, ובתת העץ השמאלי קטן ממני
- גובהו של עץ חיפוש בינארי בעל n איברים הוא לכל הפחות $\log_2 n$ ולכל היותר n
- סיבוכיות חיפוש, הוספת והסרת איברים מהעץ היא כגובה העץ
- ישנם אלגוריתמים שיודעים לשמור על העץ **מאוזן**, כלומר לשמור על גובהו בסביבות $\log_2 n$
- דבר זה הופך את סיבוכיות הפעולות על העץ ללוגריתמית! 👍

קבוצה (SET)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    set<int> set = {1, 2, 3};
    if (set.count(2)) cout << "2 is in the set!" << endl;
    set.insert(4);  // set = {1, 2, 3, 4}
    set.erase(2);   // set = {1, 3, 4}

    cout << "there are " << set.size() << " elements in set!" << endl;

    auto it = set.upper_bound(3);
    cout << "first element greater then 3 in set is " << *it << endl;

    it = set.lower_bound(2);
    cout << "first element greater or equal to 2 is " << *it << endl;

    set.clear();
    cout << "set size is now " << set.size() << endl;
}
```

- עץ חיפוש בינארי מאוזן ממומש ב־STL באובייקט set
- בכדי להכניס איברים לקבוצה נשתמש במתודה insert
- הכנסת איבר שכבר נמצא בקבוצה לא יבצע דבר - כפילויות לא נתמכות
- המתודה erase תמחק איבר מהקבוצה (אם קיים) ותחזיר את מספר האיברים שנמחקו (1 או 0)
- נשתמש במתודות lower_bound, upper_bound למציאת איבר ראשון או אחרון בטווח מסוים

רב-קבוצה (MULTISET)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    multiset<int> ms = {1, 2, 2, 2, 3, 3};
    cout << "ms size is " << ms.size() << endl;  // 6
    cout << "2 appears " << ms.count(2) << " times"
<< endl;  // 3

    ms.insert(4);  // ms = {1, 2, 2, 2, 3, 3, 4}
    ms.erase(2);   // ms = {1, 3, 3, 4}
    // erasing a value erases all its appearances.

    ms.erase(ms.find(3));
    // erasing an iterator erases only one element.
}
```

- רב-קבוצה הוא עץ חיפוש בינארי מאוזן המאפשר כפילויות
- erase המקבל מופע ימחק את כל מופעי המופע מהקבוצה
- find יחזיר iterator למופע הראשון של הערך בקבוצה
- erase המקבל iterator לאיבר בקבוצה ימחק איבר זה בלבד

מפה, מילון (MAP)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // create a map of three (strings, int) pairs
    map<string, int> m = {{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};

    m["CPU"] = 25;  // update an existing value
    m["SSD"] = 30;  // insert a new value

    // using [] with non-existent key inserts default value
    cout << "m[UPS] = " << m["UPS"] << endl;  // m[UPS] = 0

    m.erase("GPU");  // keys are now {CPU, SSD, RAM, UPS}

    // size is the number of keys
    cout << m.size() << endl;
    m.clear();
}
```

- ניתן לחשוב על `map<A, B>` בערך כמו `set<pair<A, B>>`
- מפה ממומשת בעזרת עץ חיפוש בינארי מאוזן, כאשר לכל איבר בעץ מפתח וערך
- העץ ממוין לפי המפתח, והערך של כל איבר יכול להשתנות כרצוננו
- גישה לאיבר על ידי מפתח שלא קיים במבנה הנתונים יצור מופע דיפולטיבי שלו
- בעבור מספרים, גישה למפתח שאינו קיים תיצור איבר חדש שערכו אפס
- ניתן לבדוק האם איבר קיים בעזרת `count`, בדומה ל־`set`

מילה על גיבוב (HASHING)

- בשפות אחרות מבני נתונים דומים ממומשים באמצעות גיבוב (hashing)
- הסיבוכיות התיאורטית של גיבוב טובה יותר מזו של עצי חיפוש – במקרה הממוצע
- עם זאת, בפועל גם סיבוכיות לוגריתמית מספיקה ברוב המקרים
- ב־STL נוכל להשתמש בגיבוב בעזרת `unordered_map` ו־`unordered_set`
- ניתן להנדס טסטים שיפגעו בסיבוכיות הגיבוב ולכן לא נשתמש בגיבוב
- לעוד מידע: codeforces.com/blog/entry/62393

תור ומחסנית

מחסנית

123
65
654
12
-1245
54
-59
312457
0
8

מחסנית

123
65
654
12
-1245
54
-59
312457
0
8

POP

POP

PUSH 98

מחסנית

65
654
12
-1245
54
-59
312457
0
8

POP: 123

POP

PUSH 98

מחסנית

654
12
-1245
54
-59
312457
0
8

POP: 123

POP: 65

PUSH 98

מחסנית

98
654
12
-1245
54
-59
312457
0
8

POP: 123

POP: 65

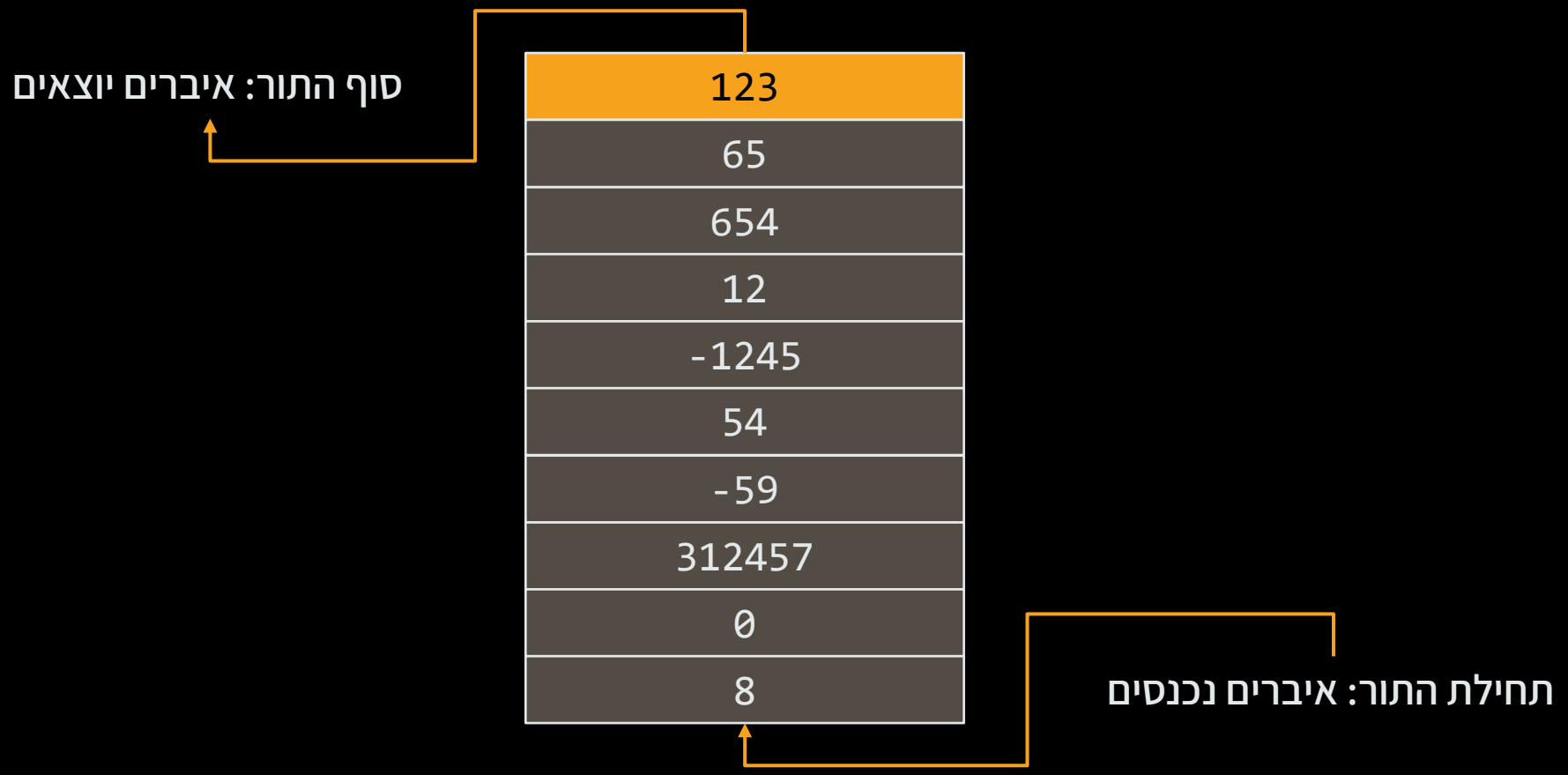
PUSH

מחסנית (STACK)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    stack<int> s({8, 0, 312457, -59, 54, -1245, 12,
654, 65, 123});
    cout << s.size() << endl;  // 10

    cout << s.top() << endl;  // 123
    s.pop();
    cout << s.top() << endl;  // 65
    s.pop();

    s.push(98);
    cout << s.top() << endl;  // 98
    while (!s.empty()) s.pop();
}
```

- תומכת בפעולות הוספה והוצאה של איברים מראש המחסנית
- האיבר האחרון שהוכנס הוא הראשון שיצא (Last In First Out)
- ניתן לממש על ידי `vector` או `list`, אבל כבר ממומש בעזרת האובייקט `stack`
- ניגש לאיבר בראש המחסנית על ידי המתודה `top`
- נכניס איבר על ידי המתודה `push`
- נוציא את האיבר בראש המחסנית על ידי המתודה `pop`

תור


סוף התור: איברים יוצאים
123
65
654
12
-1245
54
-59
312457
0
8
תחילת התור: איברים נכנסים

תור (QUEUE)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    queue<int> s({8, 0, 312457, -59, 54, -1245, 12,
654, 65, 123});
    cout << s.size() << endl;  // 10

    cout << s.front() << endl;  // 123
    s.pop();
    cout << s.front() << endl;  // 65
    s.pop();

    s.push(98);
    cout << s.front() << endl;  // 65
    while (!s.empty()) s.pop();
}
```

- תומכת בפעולות הוספה של איבר לקצה אחת והוצאת איבר מקצהו השני
- האיבר הראשון שהוכנס הוא הראשון שיצא (First In First Out)
- ניתן לממש על ידי deque או list ואפילו בעזרת מחסנית (איך?), אבל כבר ממומש בעזרת האובייקט queue
- ניגש לאיבר בראש התור על ידי המתודה front
- נכניס איבר על ידי המתודה push
- נוציא את האיבר בראש המחסנית על ידי המתודה pop

ויש עוד!

אבל לבינתיים זה מספיק... (: