



Imperativ och objektorienterad programmeringsmetodik

Föreläsning 6 av många

Tobias Wrigstad

Fortsättning länkade strukturer



Insättning och borttagning ur en länkad lista med first och last-pekare

- Prepend — skapa en ny länk först i listan

```
list->first = link_create(element, list->first);
```

Särfall: när listan är tom måste även last-pekaren uppdateras

- Append — skapa en ny länk sist i listan

```
list->last = (list->last->next = link_create(element, NULL)); // fast skriv inte så här
```

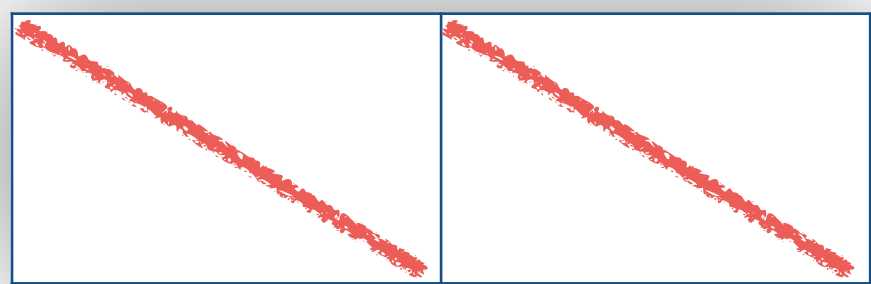
Särfall: när listan är tom måste även first-pekaren uppdateras

- Borttagning — länka ur

```
link_t *prev = list_find_previous_link(list->first, element);  
link_t *to_unlink = prev->next;  
prev->next = to_unlink->next;  
free(to_unlink);
```

Särfall: när listan är tom eller när det första elementet skall tas bort

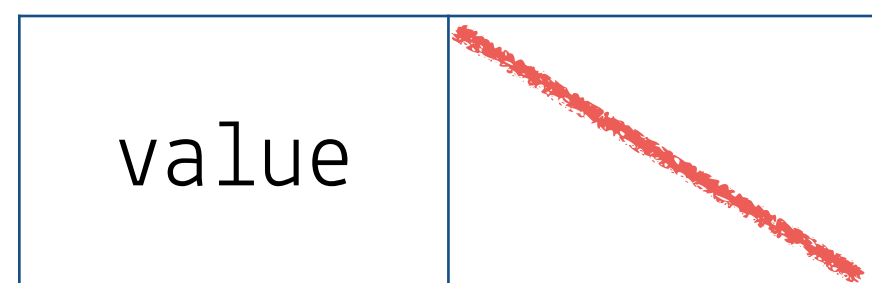
Den tomma listan



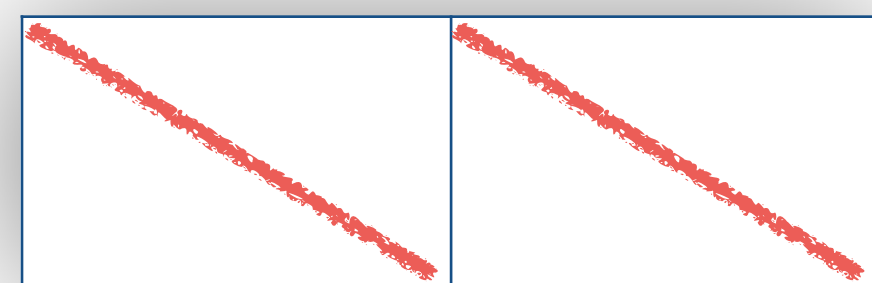
list_t

Prepend i en tom lista

1)

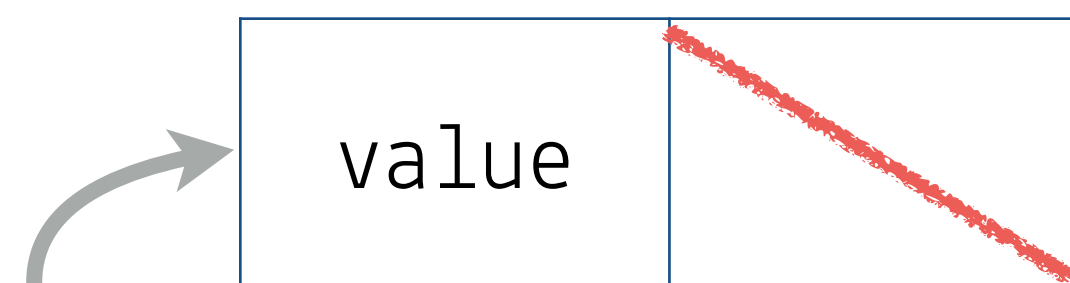


link_t

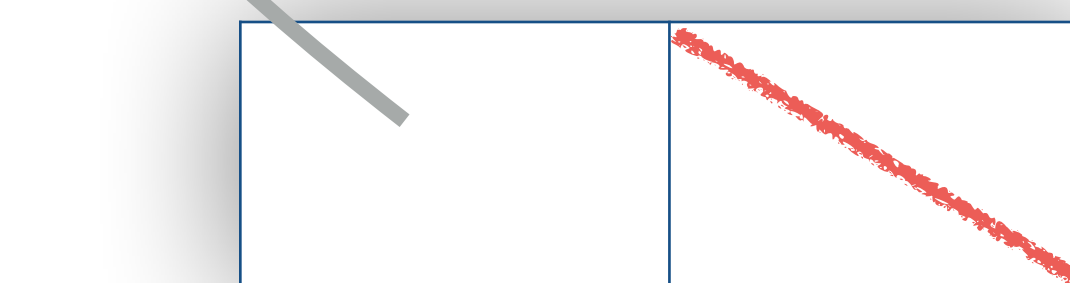


list_t

2)

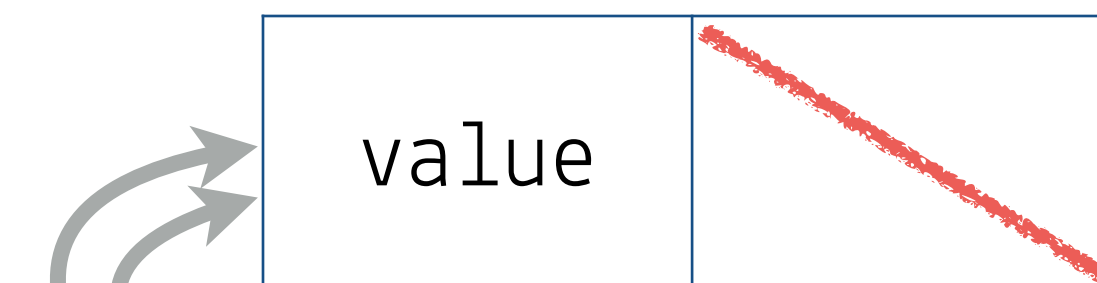


value

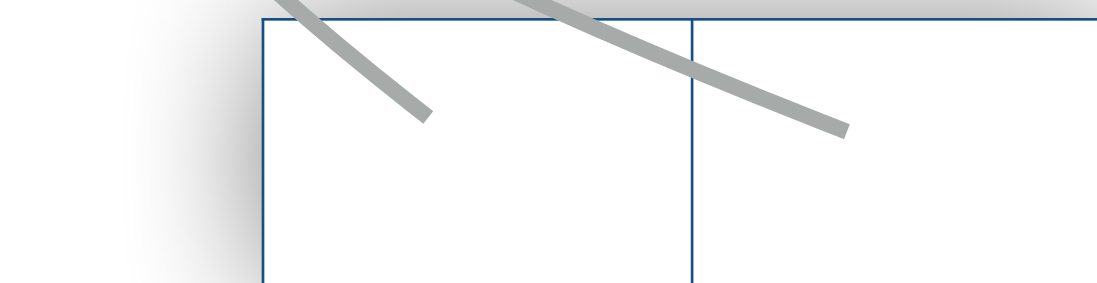


list_t

3)

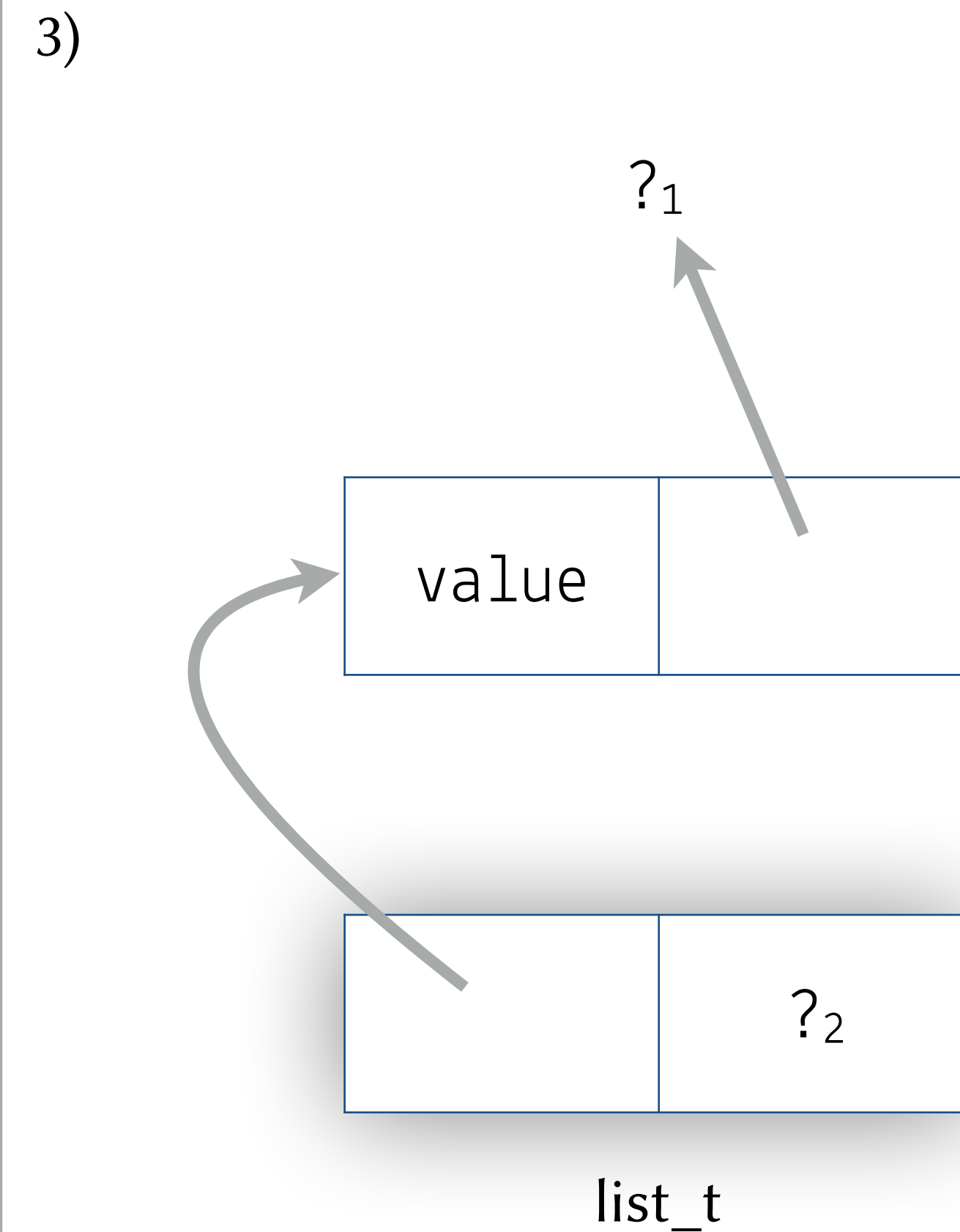
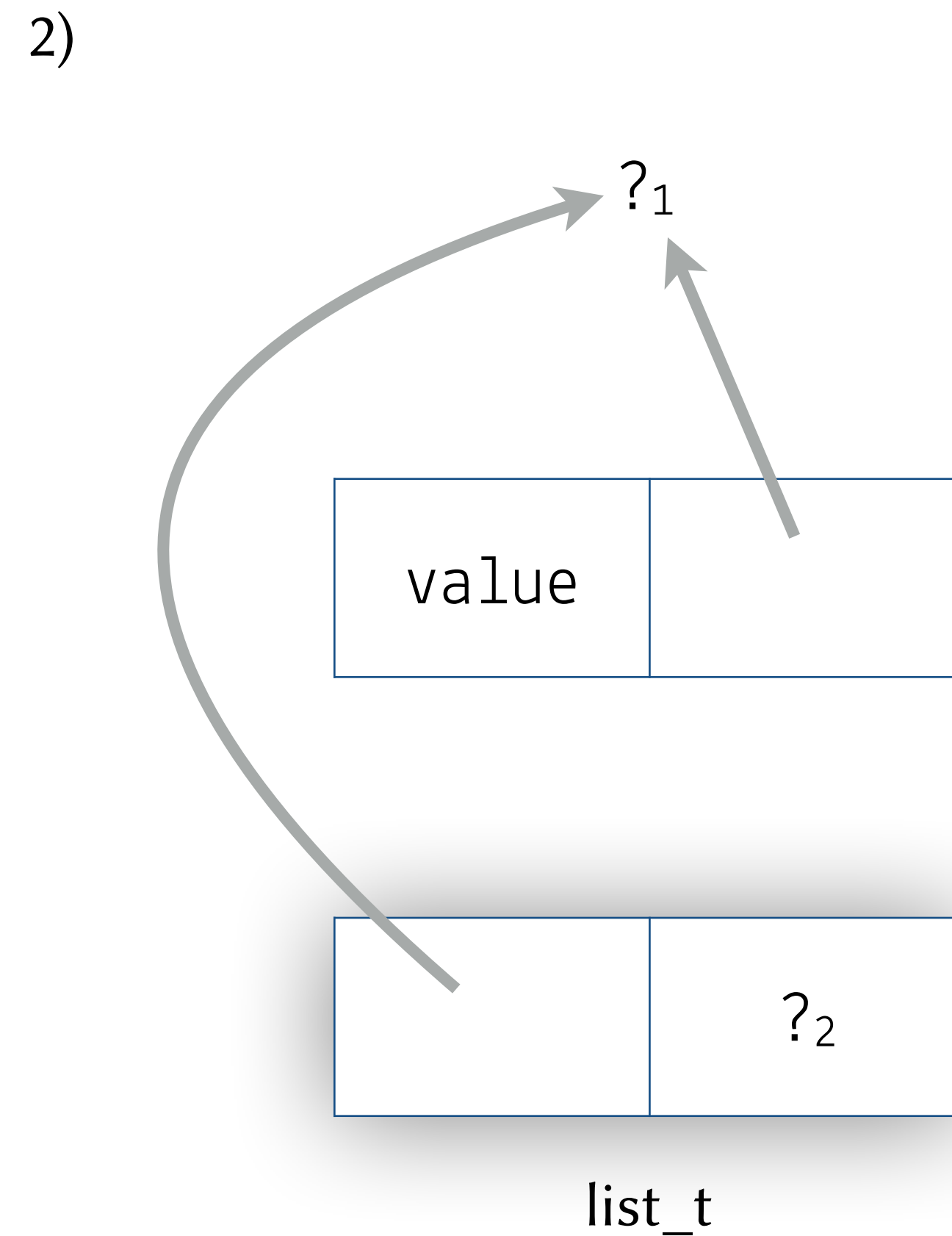
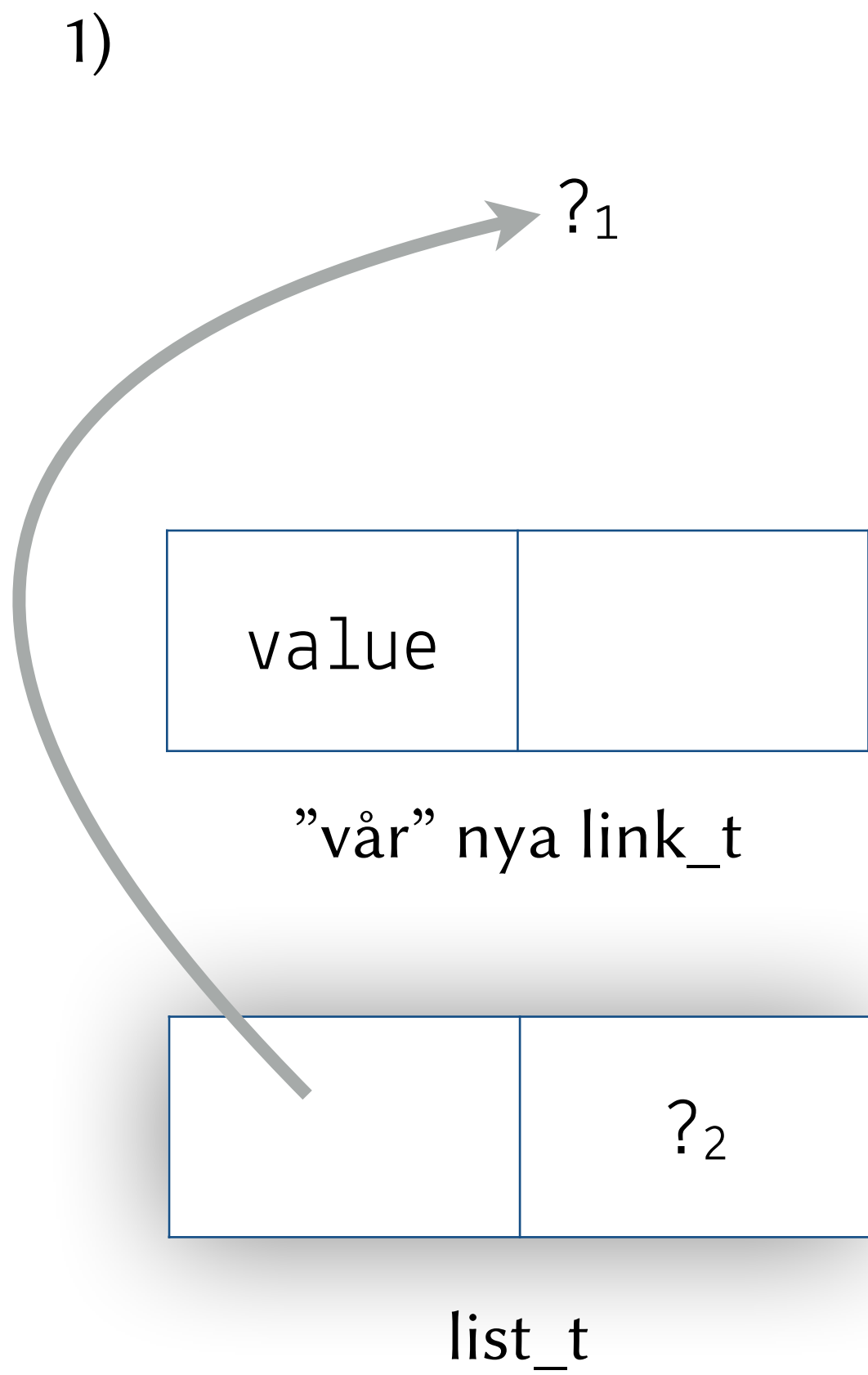


value

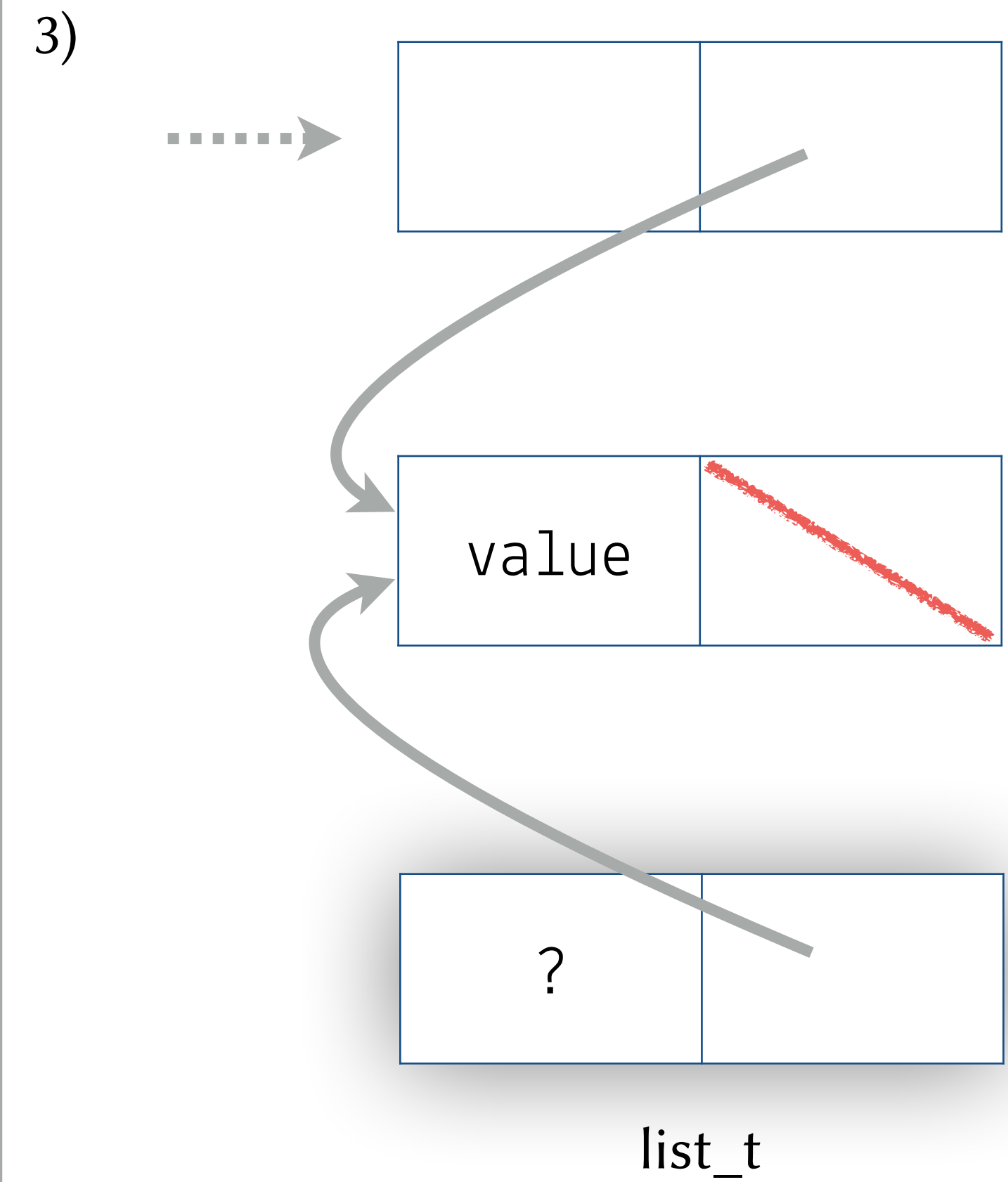
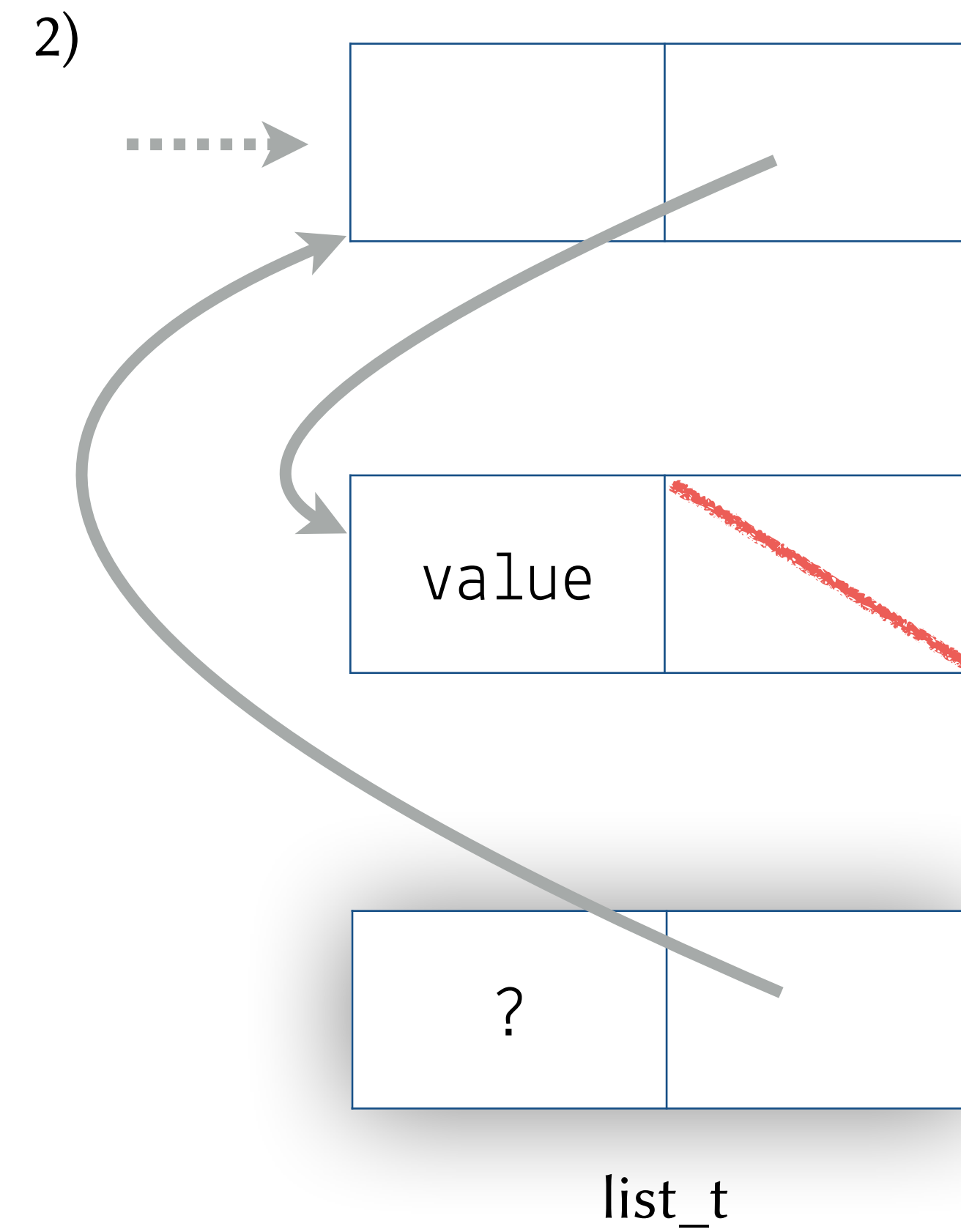
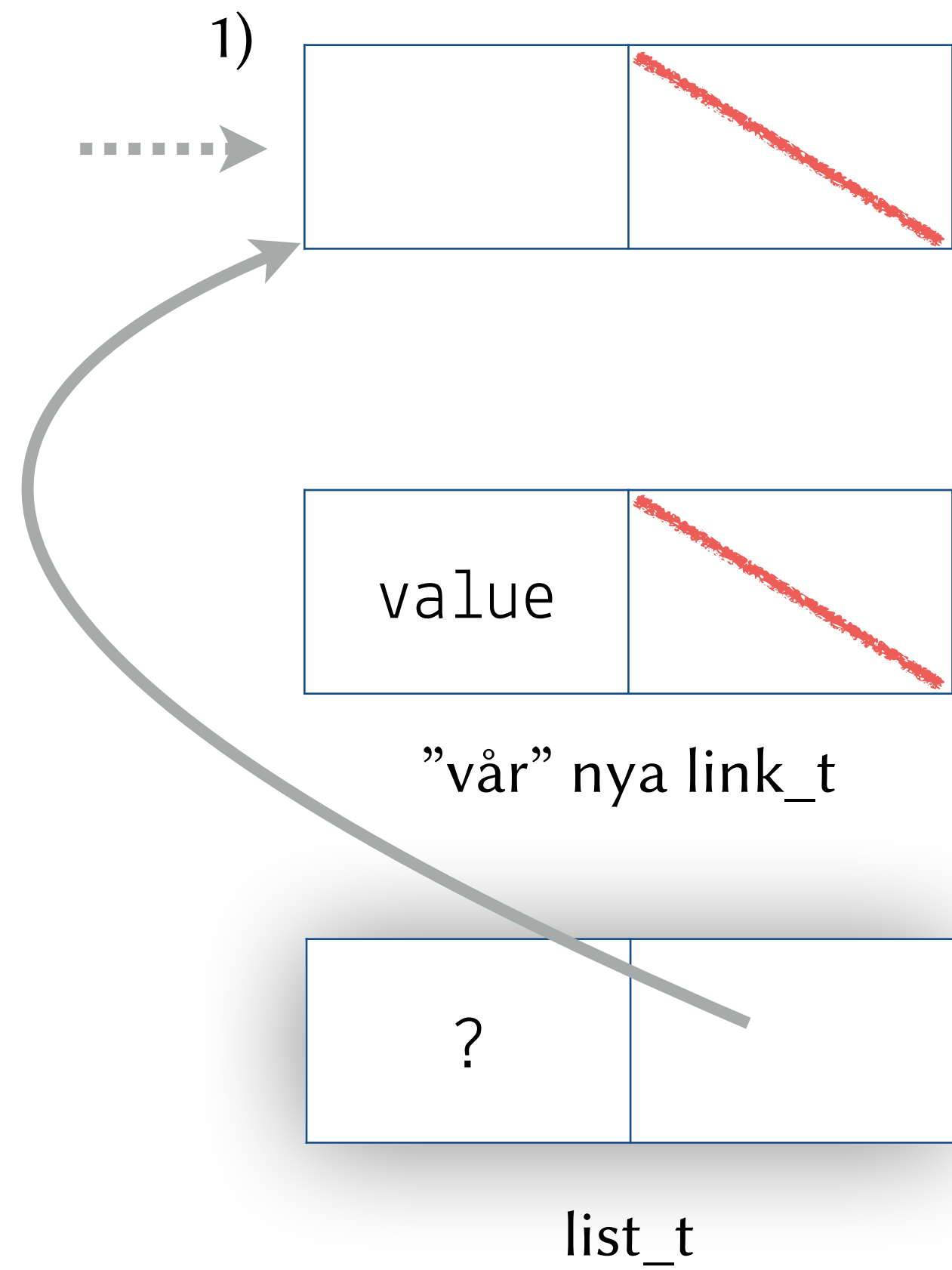


list_t

Prepend — i ev. icke-tom lista

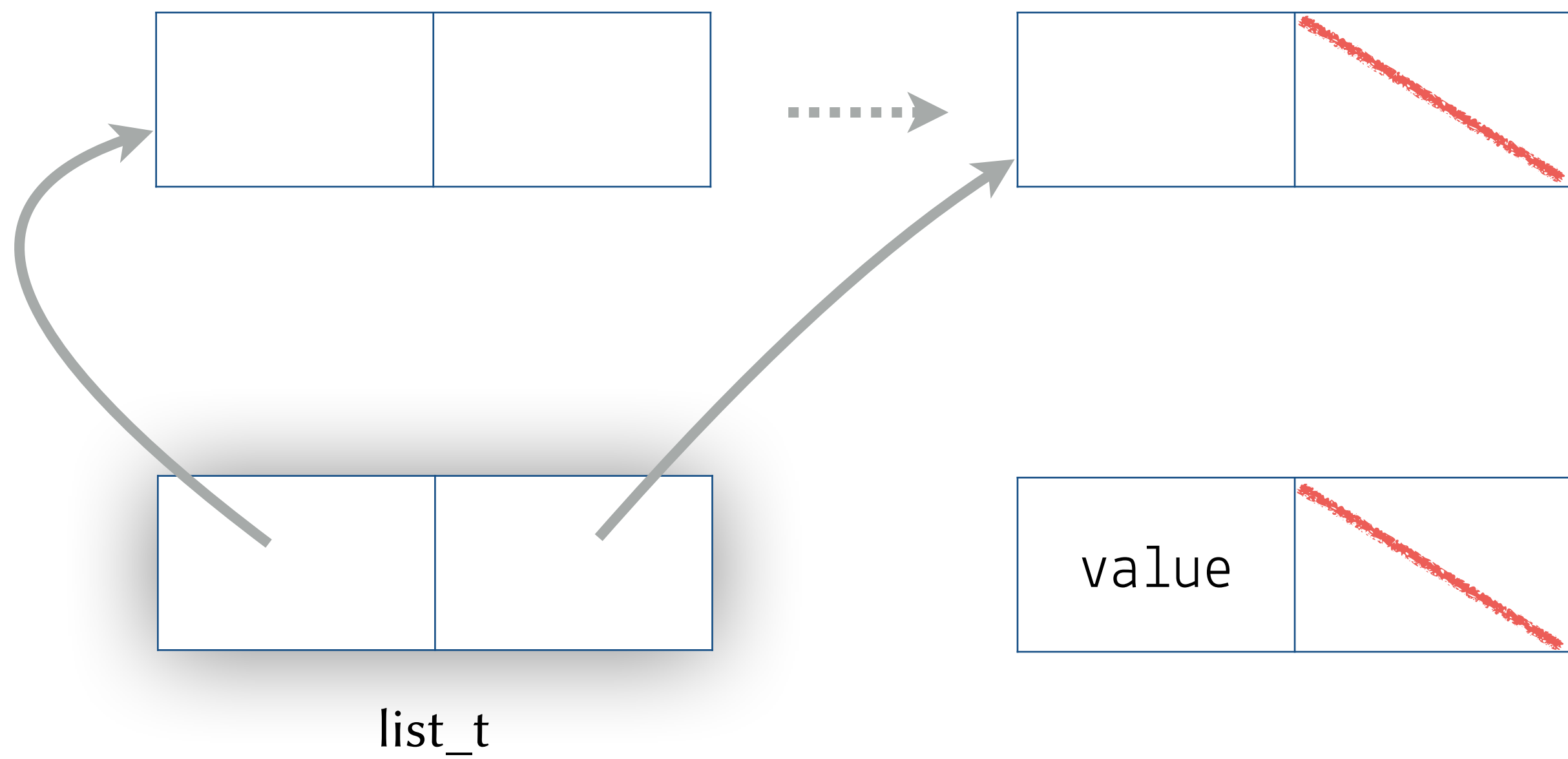


Append

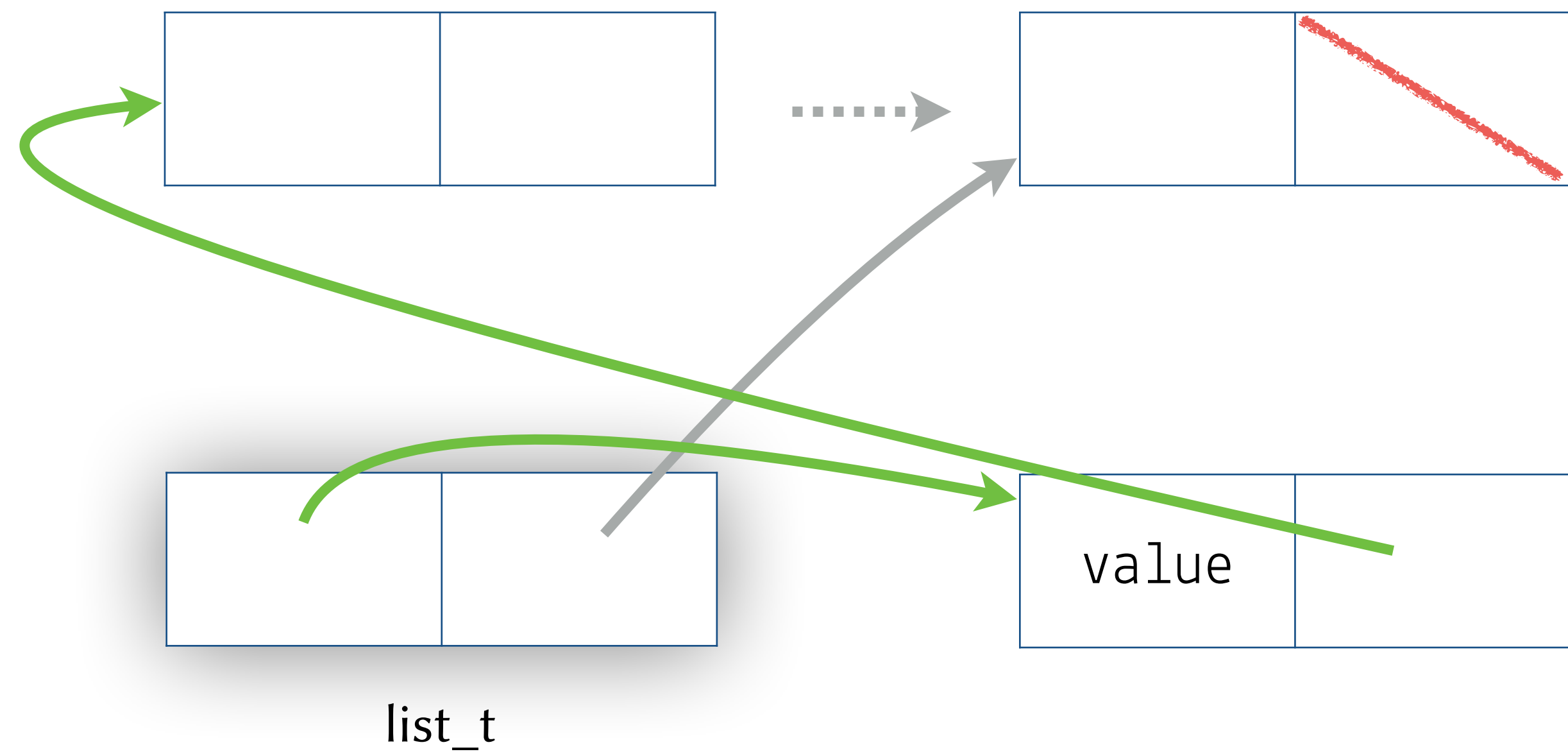


Append

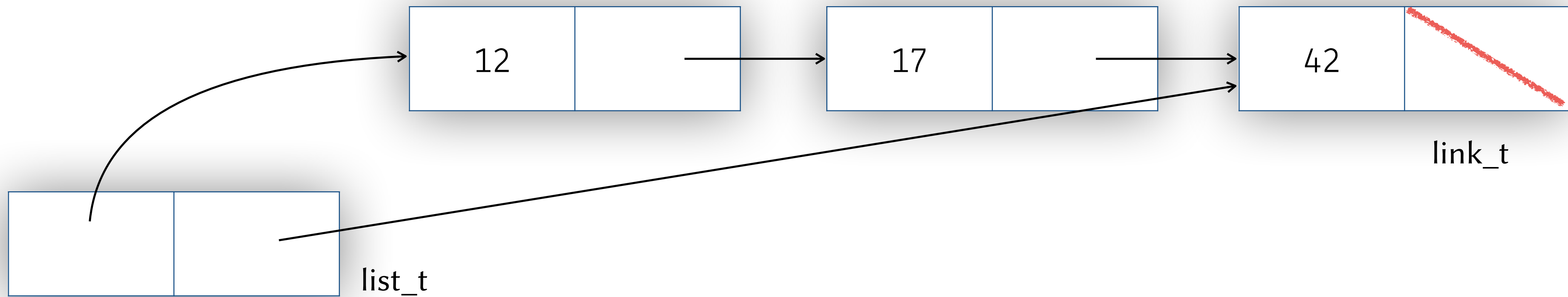
1)



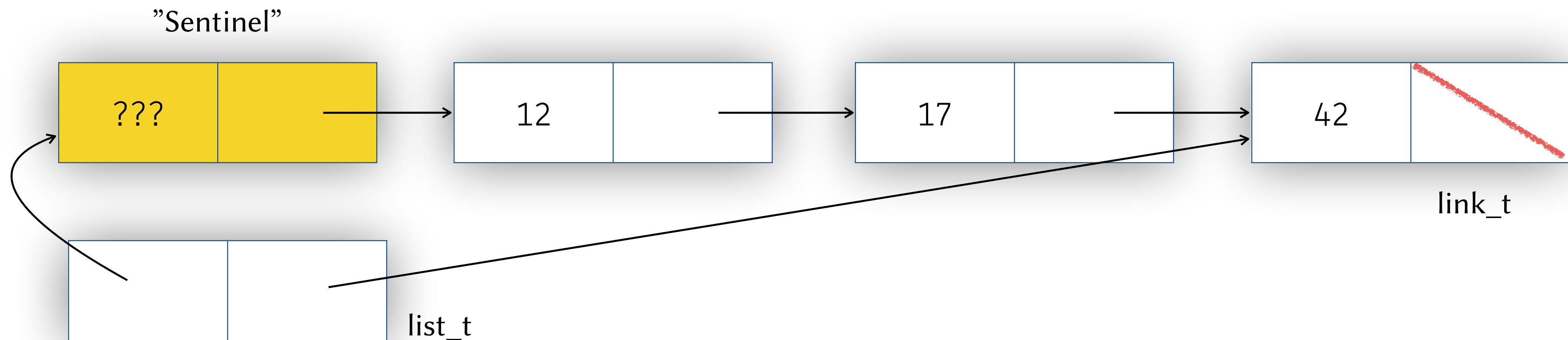
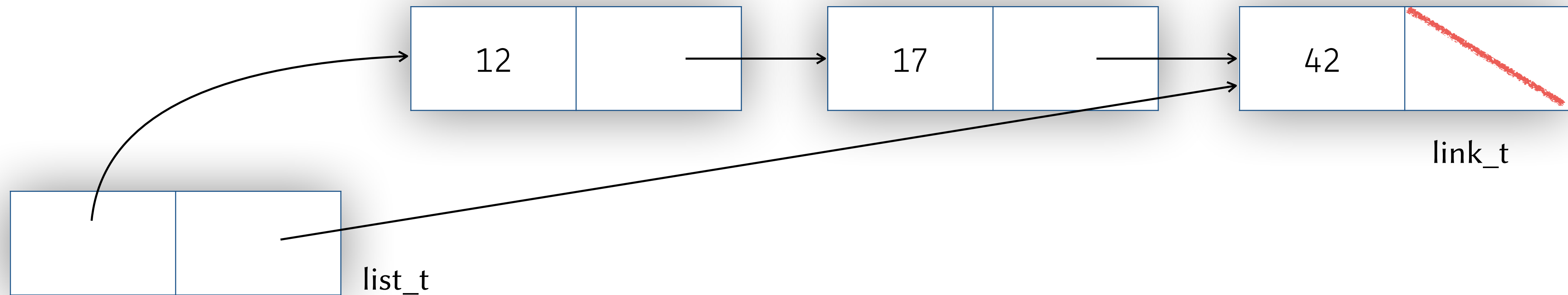
2)



Anpassa datastrukturen



Anpassa datastrukturen



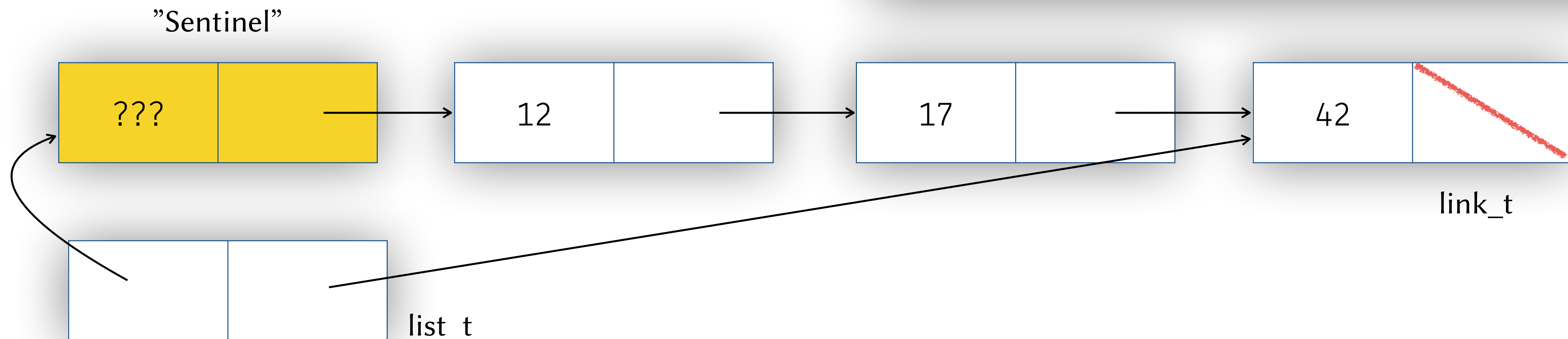
Anpassa datastrukturen eller koden?

- "Trick" — lägg till en extra *tom* länk först i varje lista

På så sätt försvinner sårskapet vid borttagning av first
iom att det alltid finns en föregående länk

`list_find_previous_link()` är numer alltid möjlig
att anropa

```
bool list_remove_first(list_t *l, int value)
{
    link_t *prev = list_find_previous_link(l->first, value);
    if (prev)
    {
        link_t *to_unlink = prev->next;
        prev->next = to_unlink->next;
        int value = to_unlink->value;
        free(to_unlink);
        return true; // ALL OK
    }
    else ...
}
```

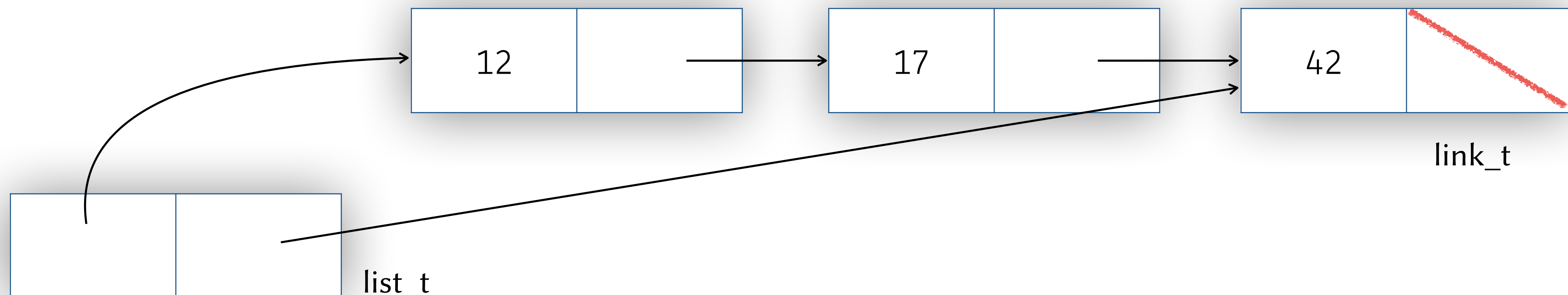


Alternativ till sentinel — pekare in i en struktur

- Problemet med ”föregående länk” är att det inte alltid finns någon sådan

Däremot finns det alltid en ”inkommande pekare”

Insikt: för att länka ur en länk behöver vi veta **var i minnet** pekaren till länken är sparad, så att vi kan skriva en ny pekare till den platsen



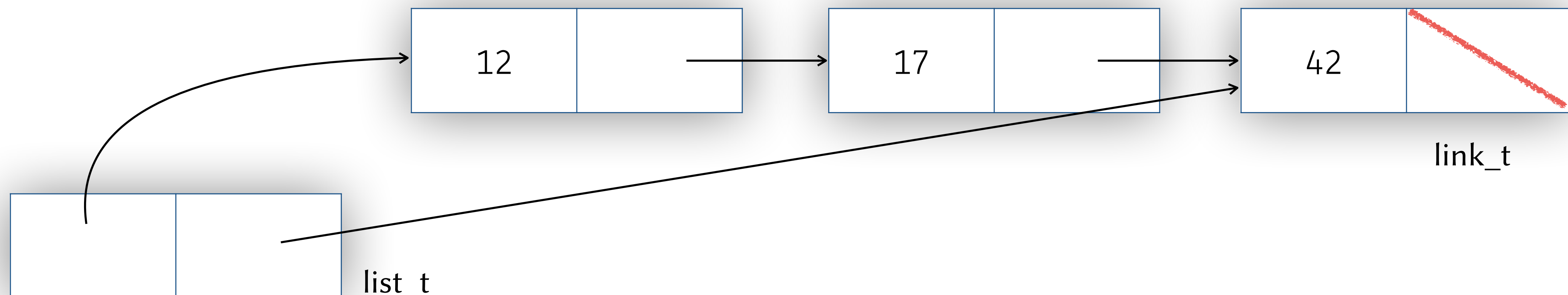
Alternativ till sentinel — pekare in i en struktur

- Problemet med ”föregående länk” är att det inte alltid finns någon sådan

Däremot finns det alltid en ”inkommande pekare”

Insikt: för att länka ur en länk behöver vi veta **var i minnet** pekaren till länken är sparad, så att vi kan skriva en ny pekare till den platsen

```
bool list_remove_first(list_t *l, int value)
{
    link_t **prev = list_find_previous_ptr(&l->first, value);
    if (*prev)
    {
        link_t *to_unlink = (*prev)->next;
        *prev = to_unlink->next;
        int value = to_unlink->value;
        free(to_unlink);
        return true; // ALL OK
    }
    else ...
}
```



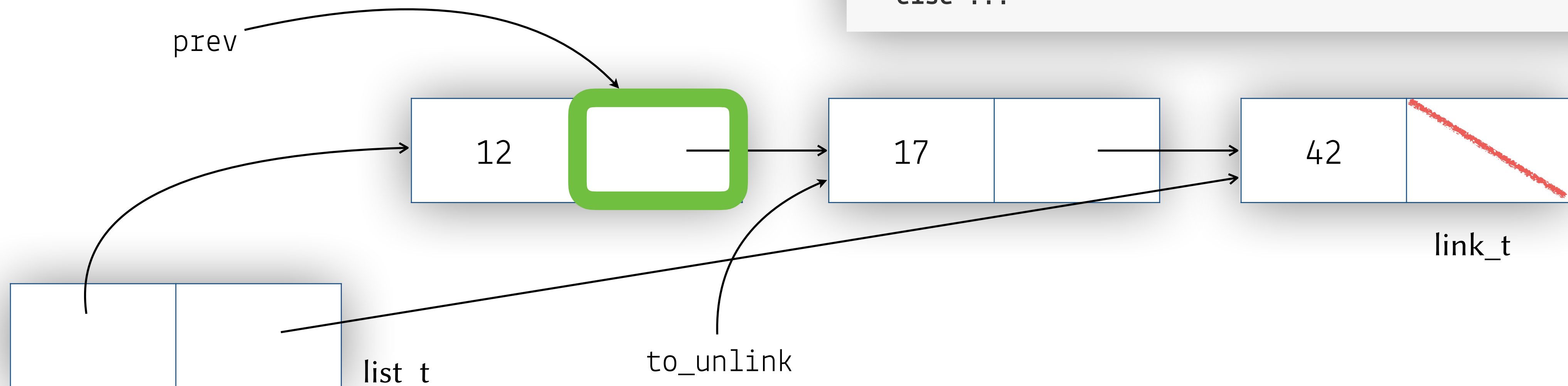
Alternativ till sentinel — pekare in i en struktur

- Problemet med "föregående länk" är att det inte alltid finns någon sådan

Däremot finns det alltid en "inkommande pekare"

Insikt: för att länka ur en länk behöver vi veta **var i minnet** pekaren till länken är sparad, så att vi kan skriva en ny pekare till den platsen

```
bool list_remove_first(list_t *l, int value)
{
    link_t **prev = list_find_previous_ptr(&l->first, value);
    if (*prev)
    {
        link_t *to_unlink = (*prev)->next;
        *prev = to_unlink->next;
        int value = to_unlink->value;
        free(to_unlink);
        return true; // ALL OK
    }
    else ...
}
```

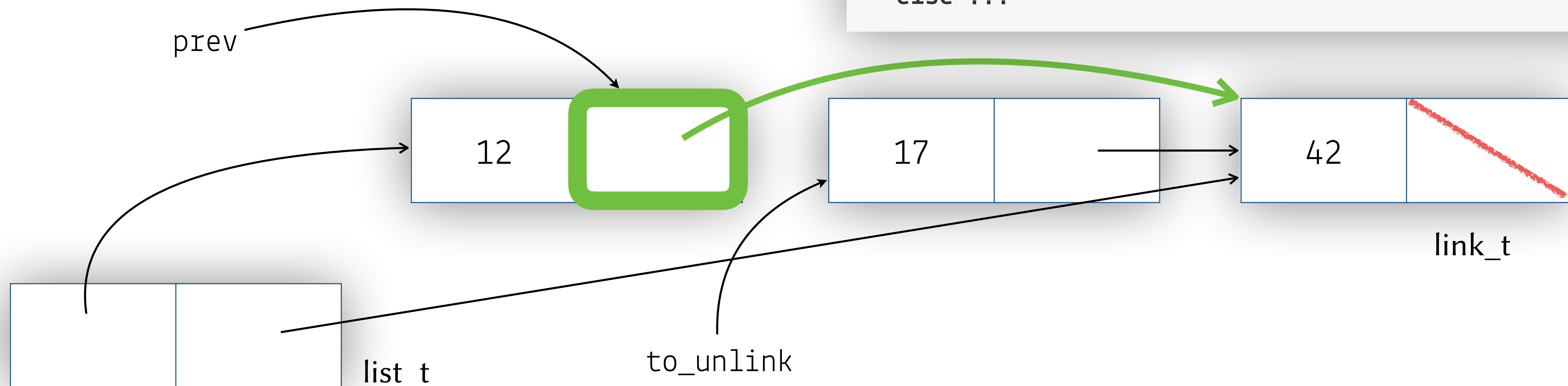


Alternativ till sentinel — pekare in i en struktur

- Problemet med ”föregående länk” är att det inte alltid finns någon sådan

Däremot finns det alltid en ”inkommande pekare”

Insikt: för att länka ur en länk behöver vi veta **var i minnet** pekaren till länken är sparad, så att vi kan skriva en ny pekare till den platsen



```
bool list_remove_first(list_t *l, int value)
{
    link_t **prev = list_find_previous_ptr(&l->first, value);
    if (*prev)
    {
        link_t *to_unlink = (*prev)->next;
        *prev = to_unlink->next;
        int value = to_unlink->value;
        free(to_unlink);
        return true; // ALL GOOD
    }
    else ...
}
```

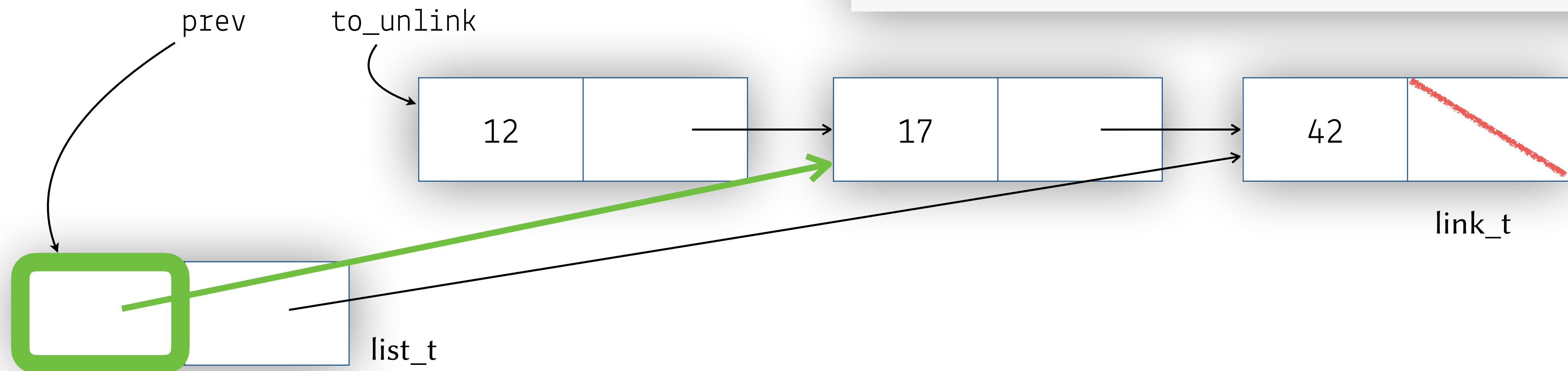
Alternativ till sentinel — pekare in i en struktur

- Problemet med ”föregående länk” är att det inte alltid finns någon sådan

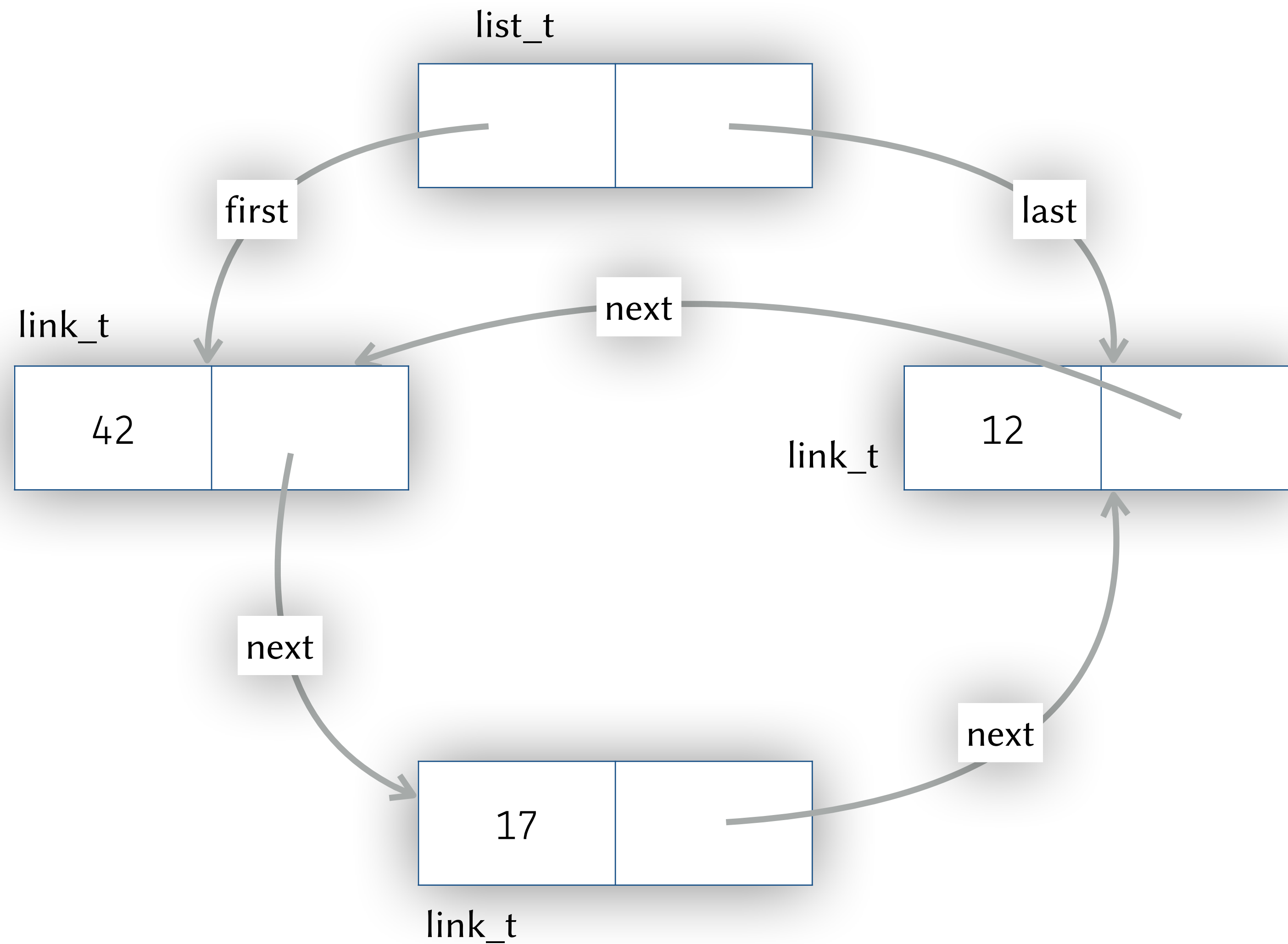
Däremot finns det alltid en ”inkommande pekare”

Insikt: för att länka ur en länk behöver vi veta **var i minnet** pekaren till länken är sparad, så att vi kan skriva en ny pekare till den platsen

```
bool list_remove_first(list_t *l, int value)
{
    link_t **prev = list_find_previous_ptr(&l->first, value);
    if (*prev)
    {
        link_t *to_unlink = (*prev)->next;
        *prev = to_unlink->next;
        int value = to_unlink->value;
        free(to_unlink);
        return true; // ALL GOOD
    }
    else ...
}
```



En icke-tom cirkulärlänkad lista har ett föregående element



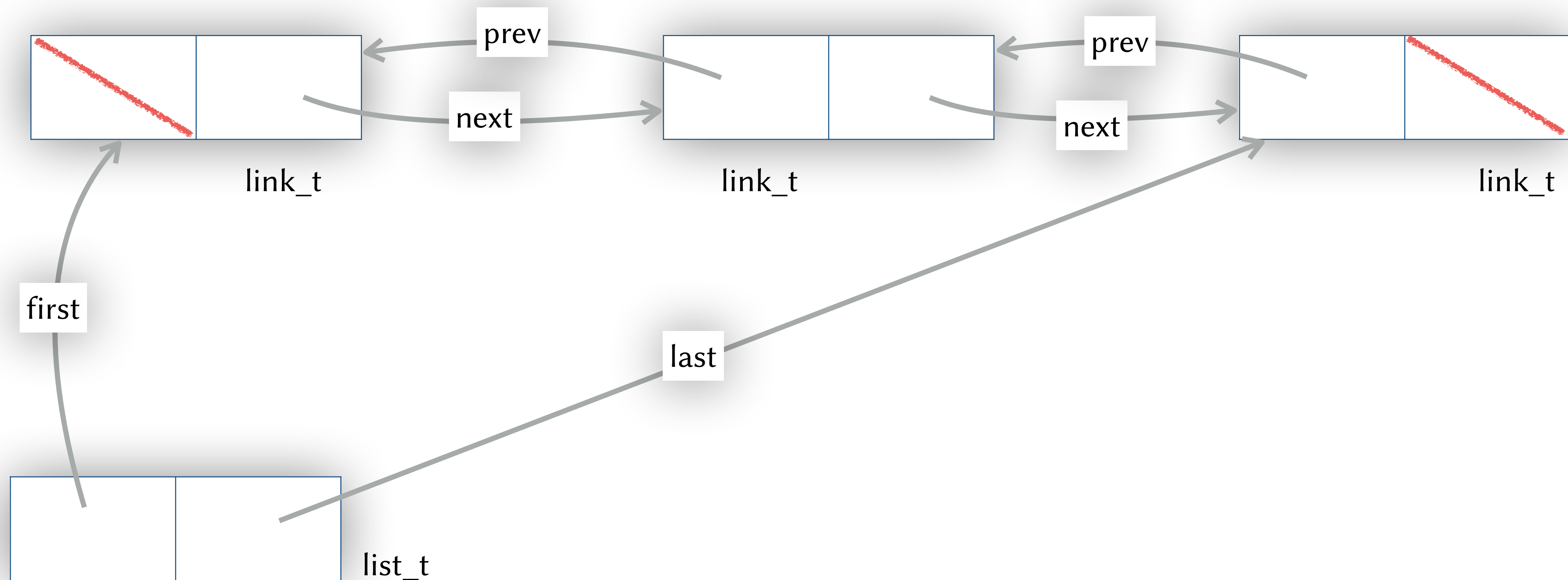
```
link_t *find_previous_link(list_t *list,
                           int value)
{
    link_t *cursor = list->last;

    do
    {
        if (cursor->next->value == value)
        {
            return cursor;
        }

        cursor = cursor->next;
    }
    while (cursor->next != list->first);

    return NULL; // nothing found
}
```


Dubbellänkad lista

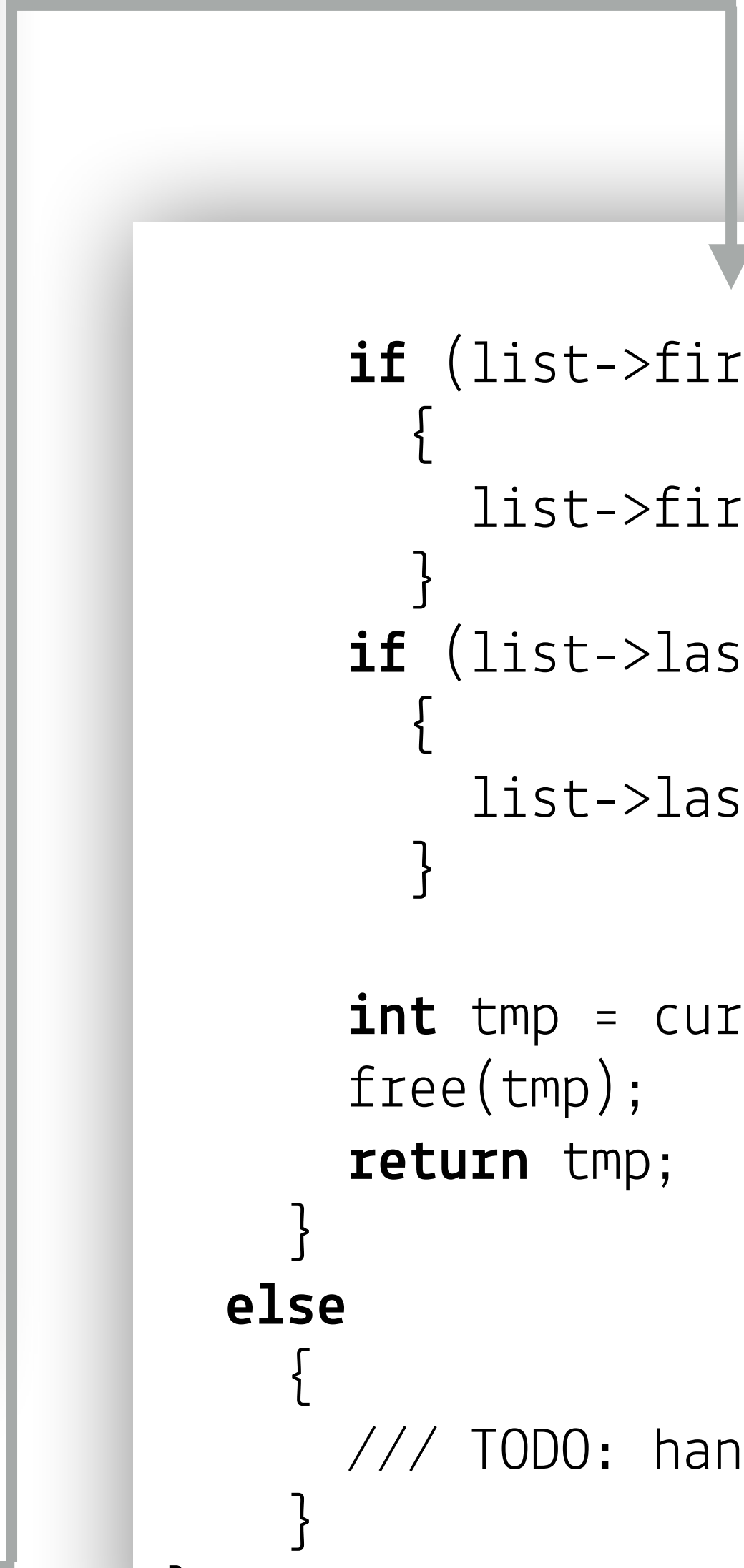


Removal in doubly-linked list

```
int list_remove(list_t *list, int index)
{
    if (list_size(list) > index)
    {
        link_t *cursor = list->first;

        for (int i = 0; i < index; ++i)
        {
            cursor = cursor->next;
        }

        if (cursor->next)
        {
            cursor->next->prev = cursor->prev;
        }
        if (cursor->prev)
        {
            cursor->prev->next = cursor->next;
        }
    }
}
```



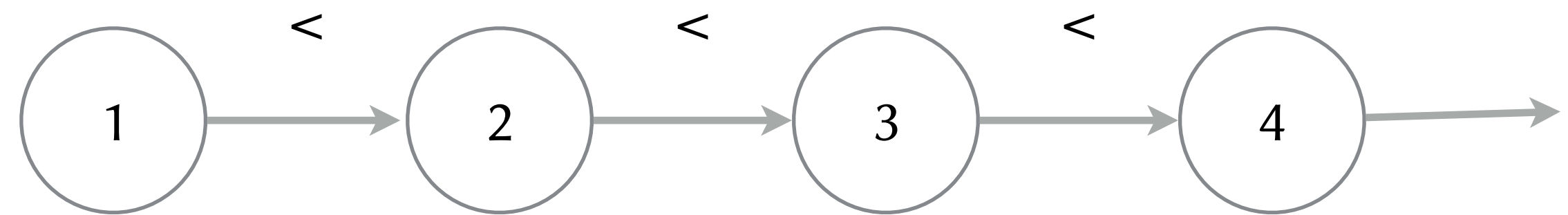
```
    if (list->first == cursor)
    {
        list->first = cursor->next;
    }
    if (list->last == cursor)
    {
        list->last = cursor->prev;
    }

    int tmp = cursor->value;
    free(tmp);
    return tmp;
}
else
{
    /// TODO: handle errors
}
}
```

Trädstrukturer

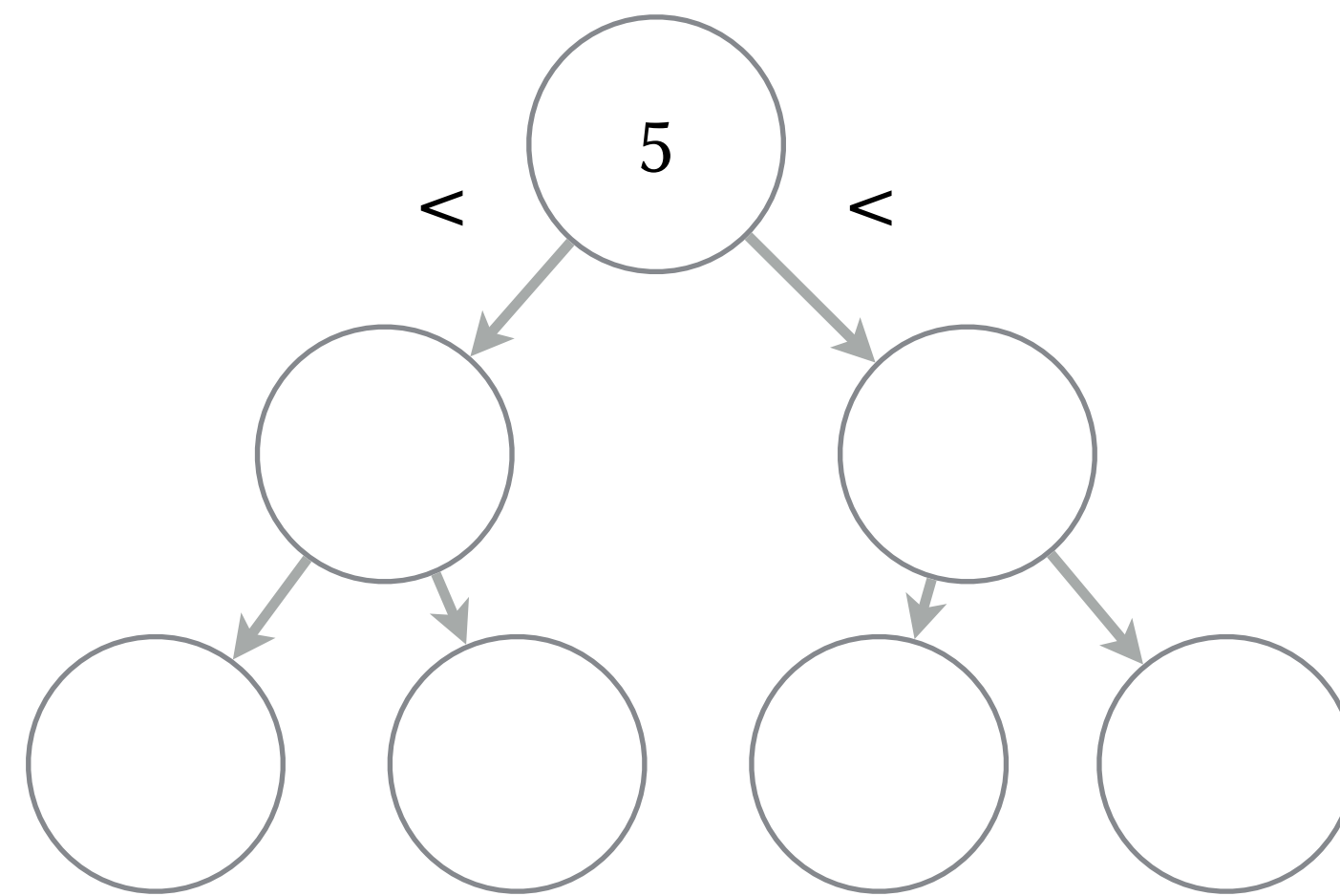
- Sökning i listor har linjär tidskomplexitet — $O(N)$

Varje varv i loopen betar vi av 1 av N element



- Sökning i binära sökträd har tidskomplexitet $O(\log n)$ — meaning the depth of the tree

Varje varv i loopen halverar sökrymden

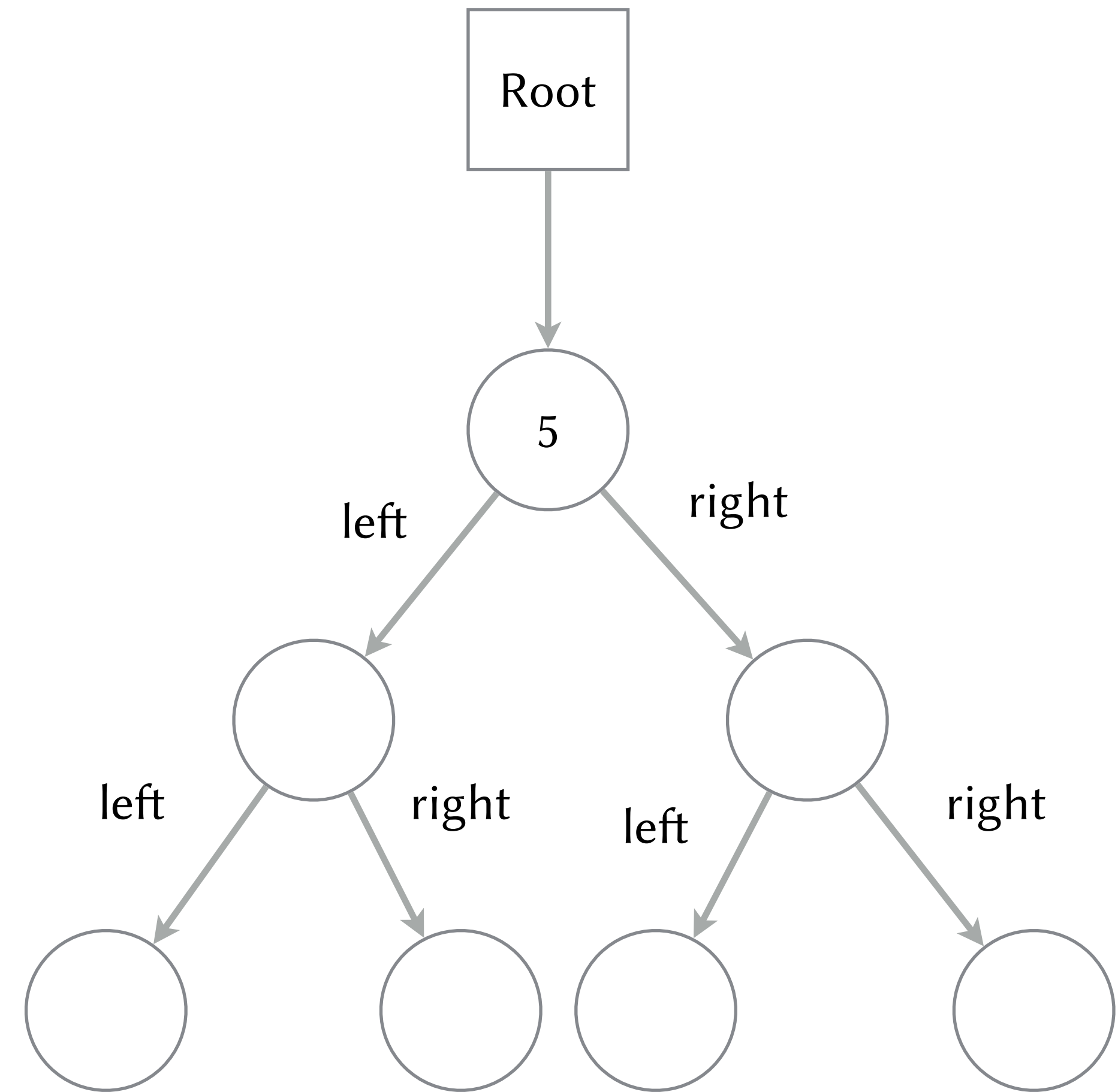


Implementation av binära sökträd i C

```
typedef struct tree tree_t;  
typedef struct node node_t;
```

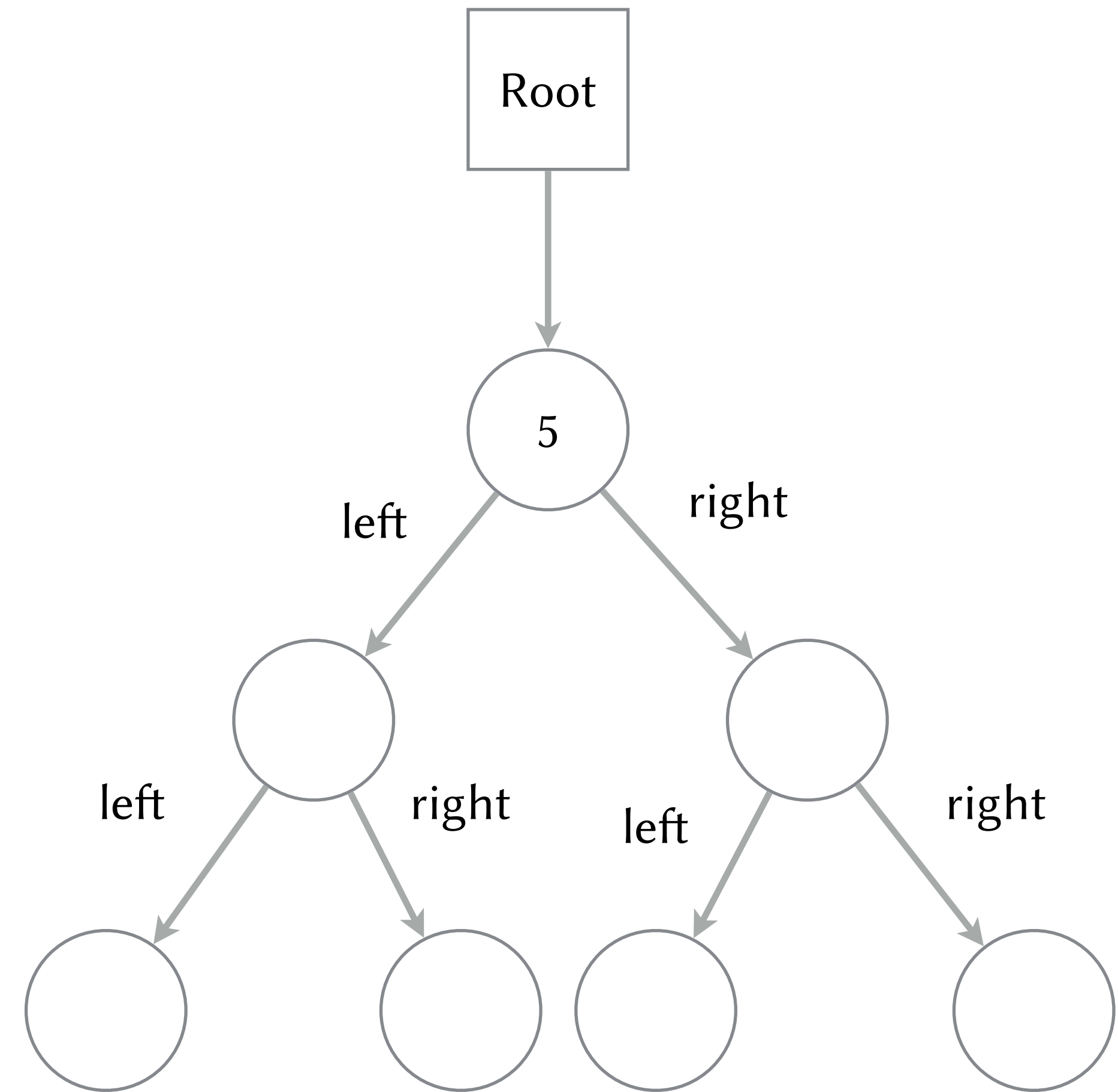
```
struct tree  
{  
    node_t* root;  
};
```

```
struct node  
{  
    int value;  
    node_t* left;  
    node_t* right;  
};
```



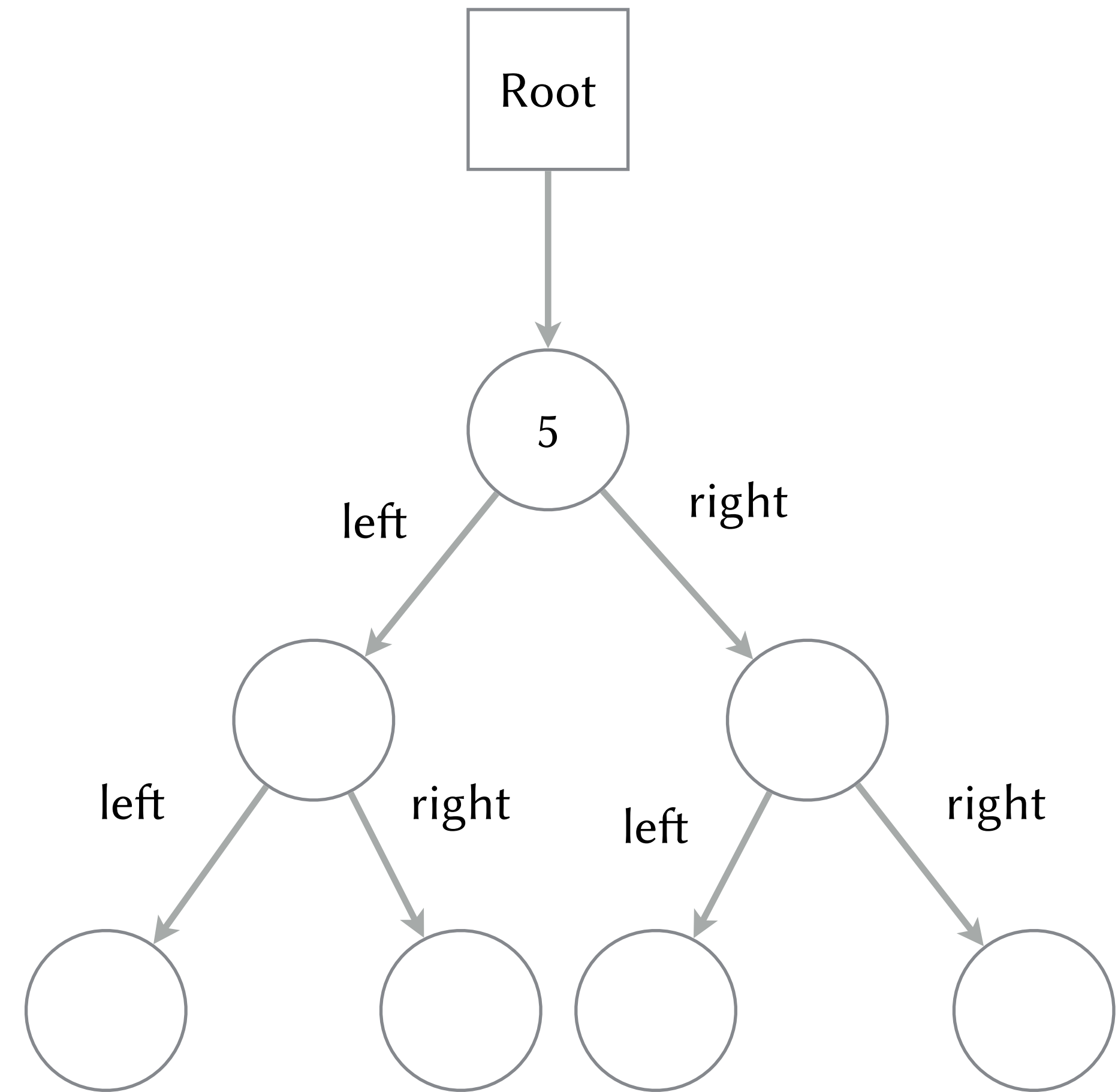
Implementation av binära sökträd i C

```
bool tree_contains(tree_t *t, int value)
{
    if (t->root)
    {
        node_t *n = t->root;
        while (n)
        {
            if (n->value == value) return true;
            n = (n->value > value) ? n->left : n->right;
        }
        return false;
    }
    else
    {
        return false;
    }
}
```



Implementation av binära sökträd i C

```
bool tree_insert(tree_t *t, int value)
{
    if (t->root)
    {
        return node_insert(t->root, value);
    }
    else
    {
        t->root = node_create(value);
        return true;
    }
}
```

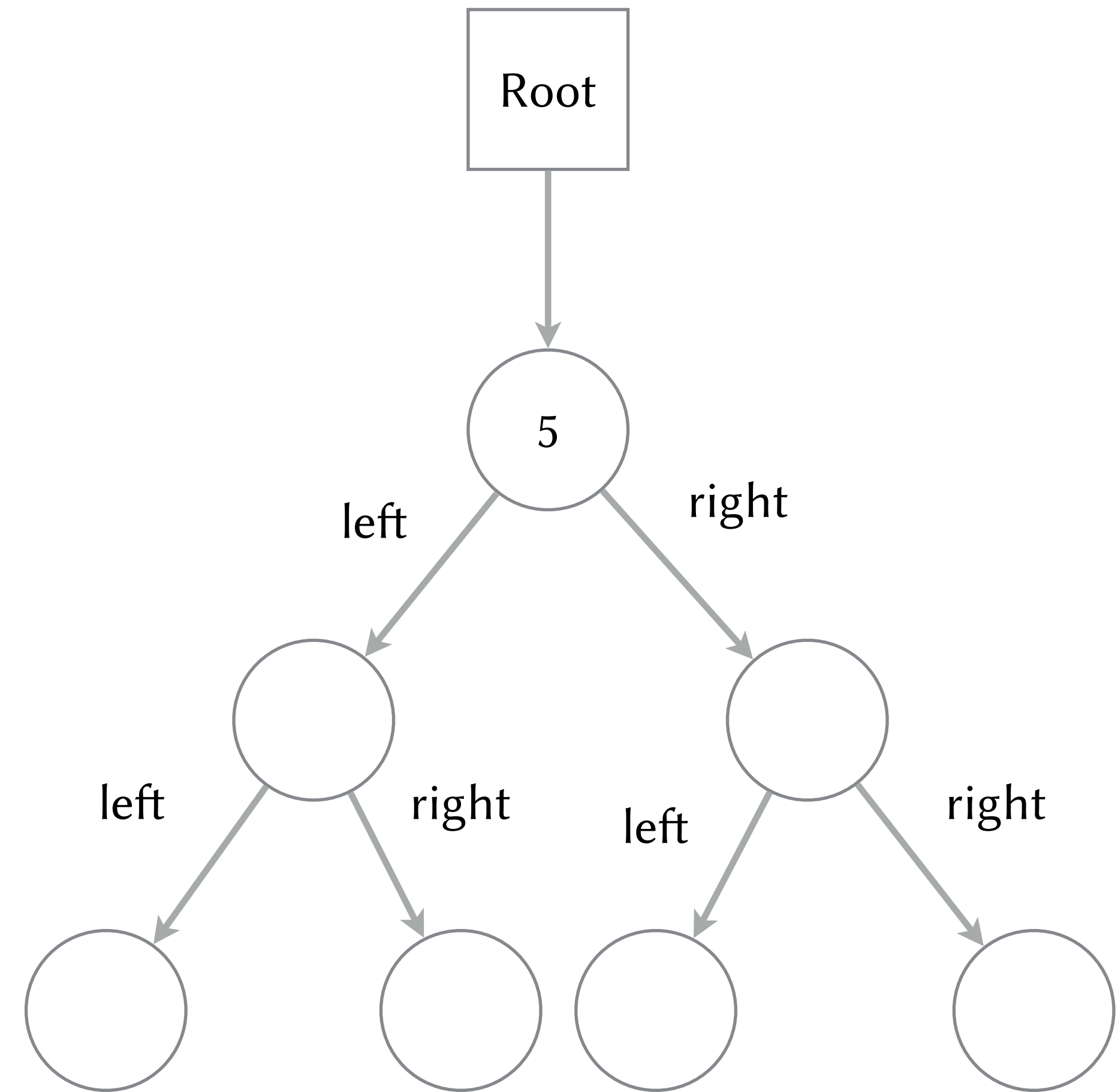


Implementation av binära sökträd i C

```
bool node_insert(node_t *n, int value)
{
    if (n->value == value) return false;

    node_t **w = (n->value > value)
        ? &n->left
        : &n->right;

    if (*w)
    {
        return node_insert(*w, value);
    }
    else
    {
        *w = node_create(value);
        return true;
    }
}
```



Avbildningen map (känd från inlämningsuppgift 1)

- Interface

`put(key, value)`

`get(key)`

- Hash map gör idealiskt uppslagningen i konstant tid $O(1)$

Effektiv implementation iom hashing och backning på en array

- Tree map gör uppslagningen i $O(\log N)$ tid

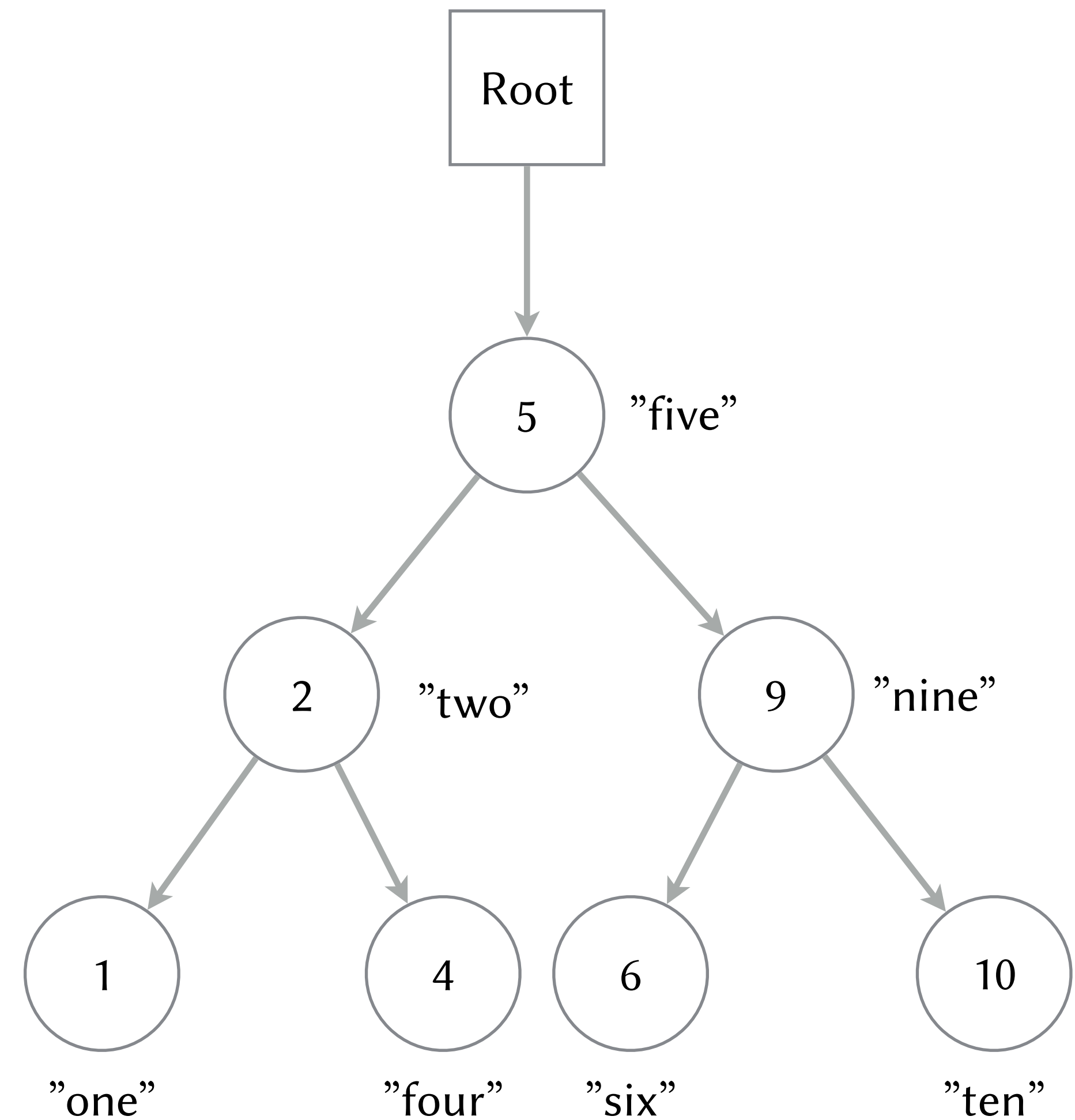
Sökning i trädet

Treemap för ett träd med heltalsnycklar och strängvärden

```
typedef struct tree tree_t;  
typedef struct node node_t;
```

```
struct tree  
{  
    node_t* root;  
};
```

```
struct node  
{  
    int key;  
    char *value;  
    node_t* left;  
    node_t* right;  
};
```

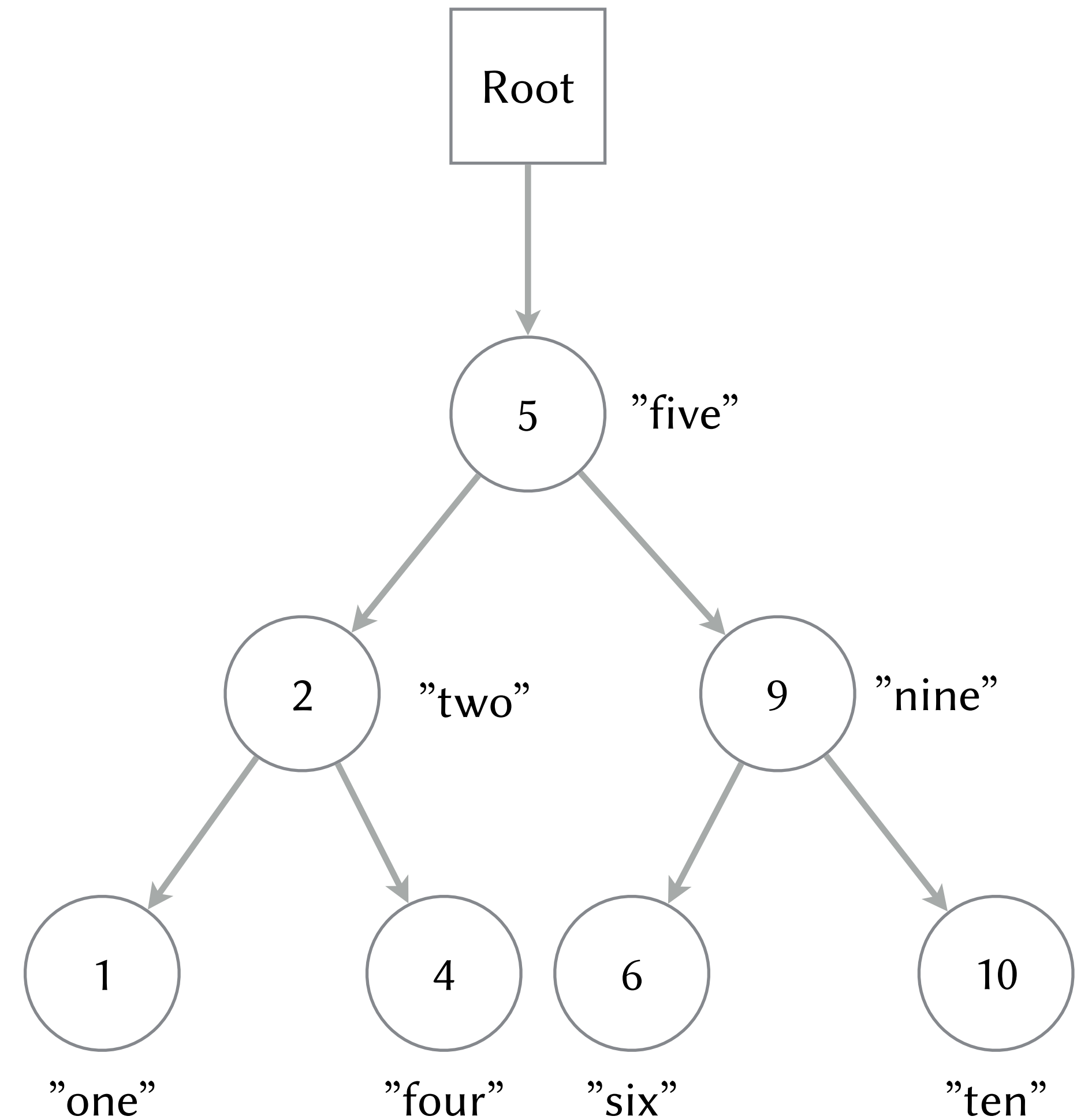


En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}

bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```



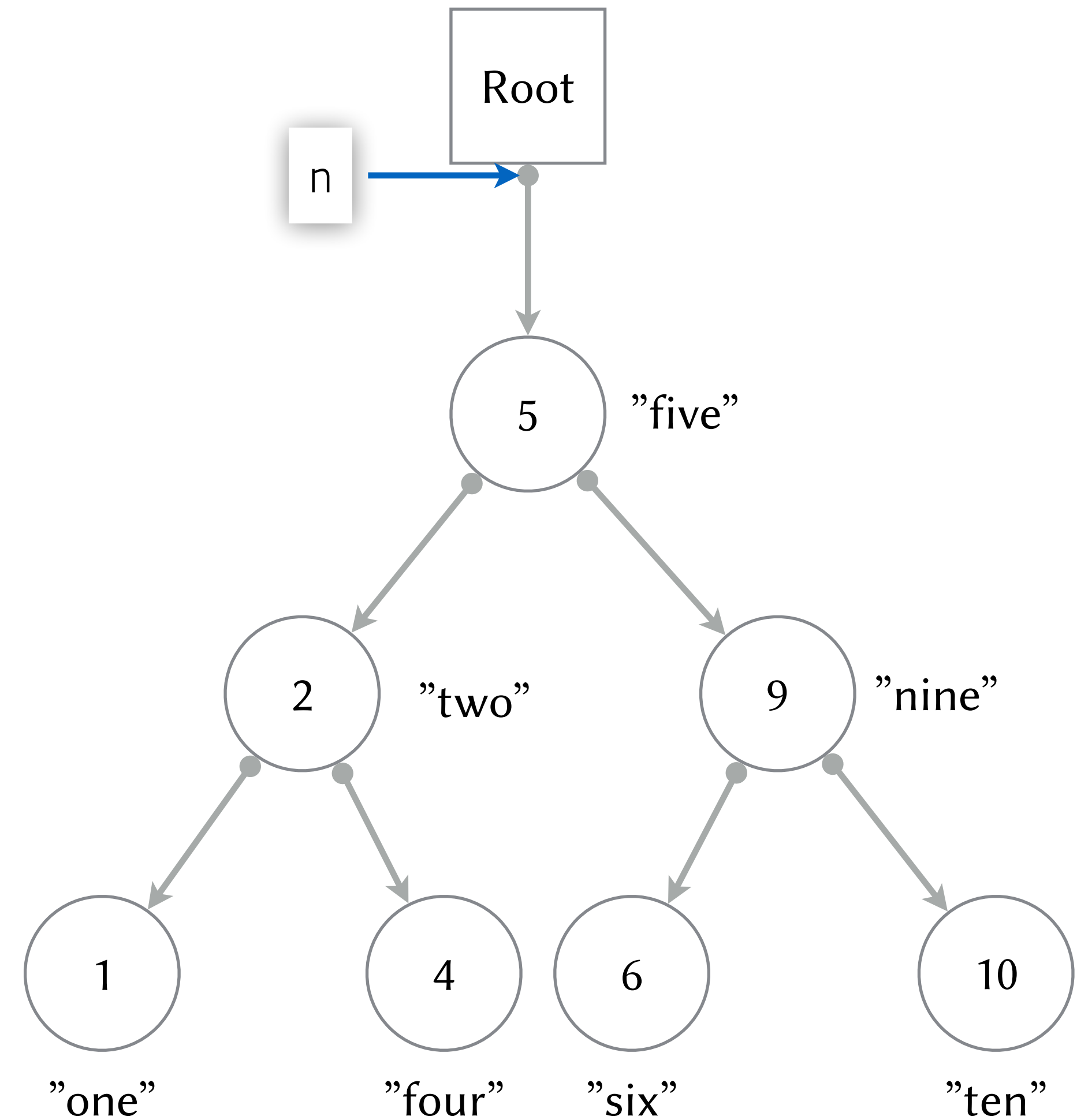
En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}
```

```
bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```

node_find_incoming_ptr(&t->root, 4)

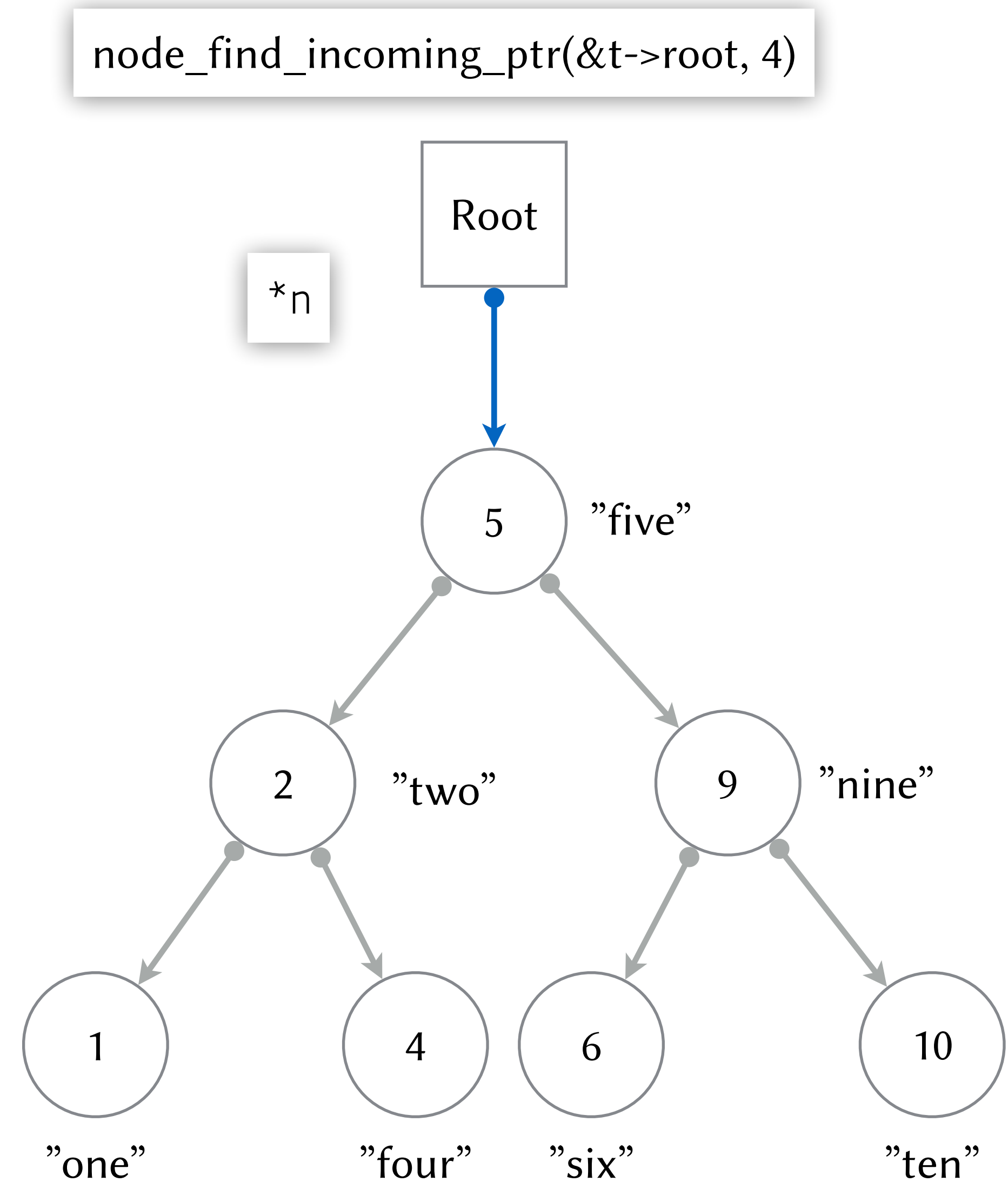


En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}

bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```

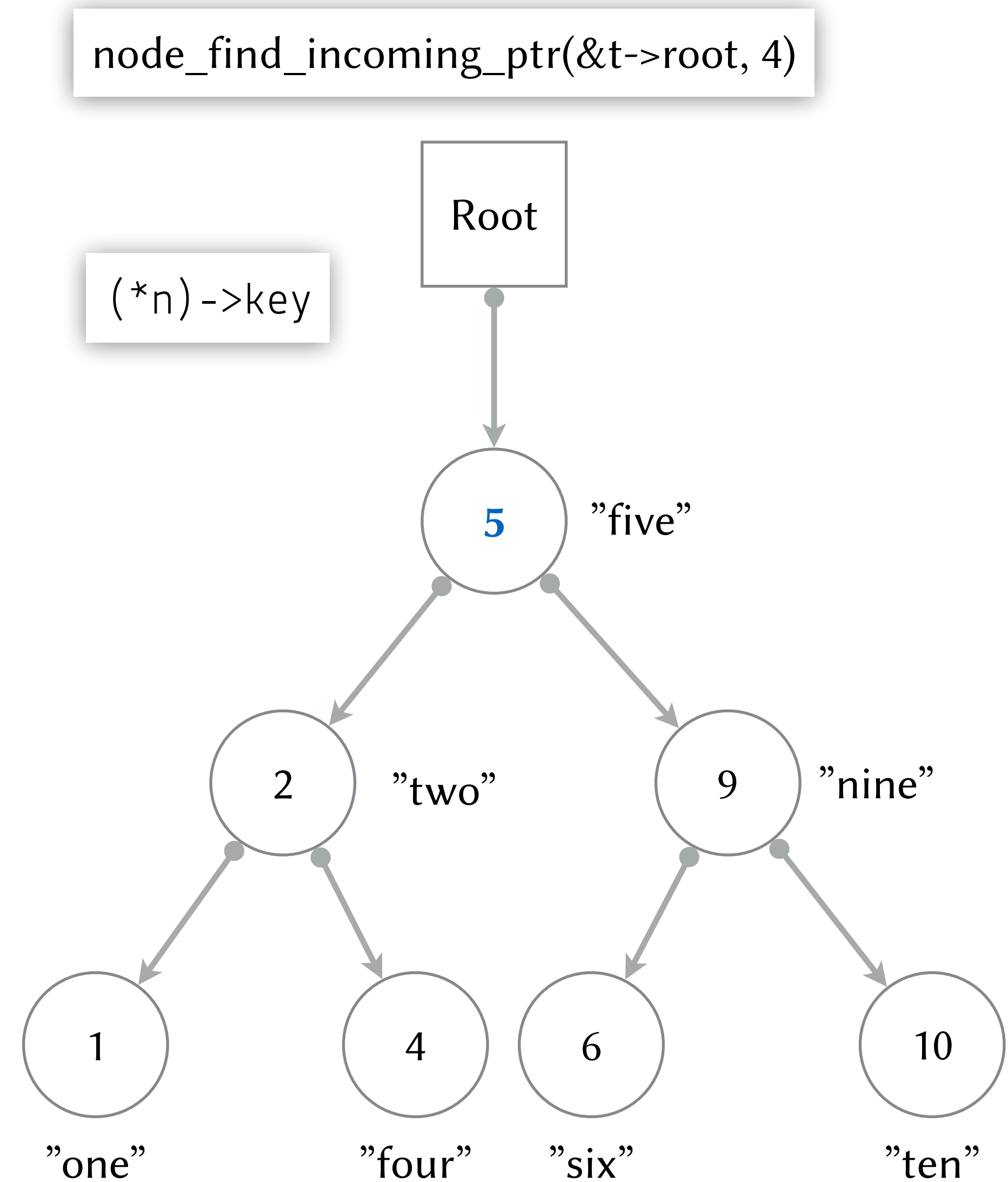


En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}

bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```



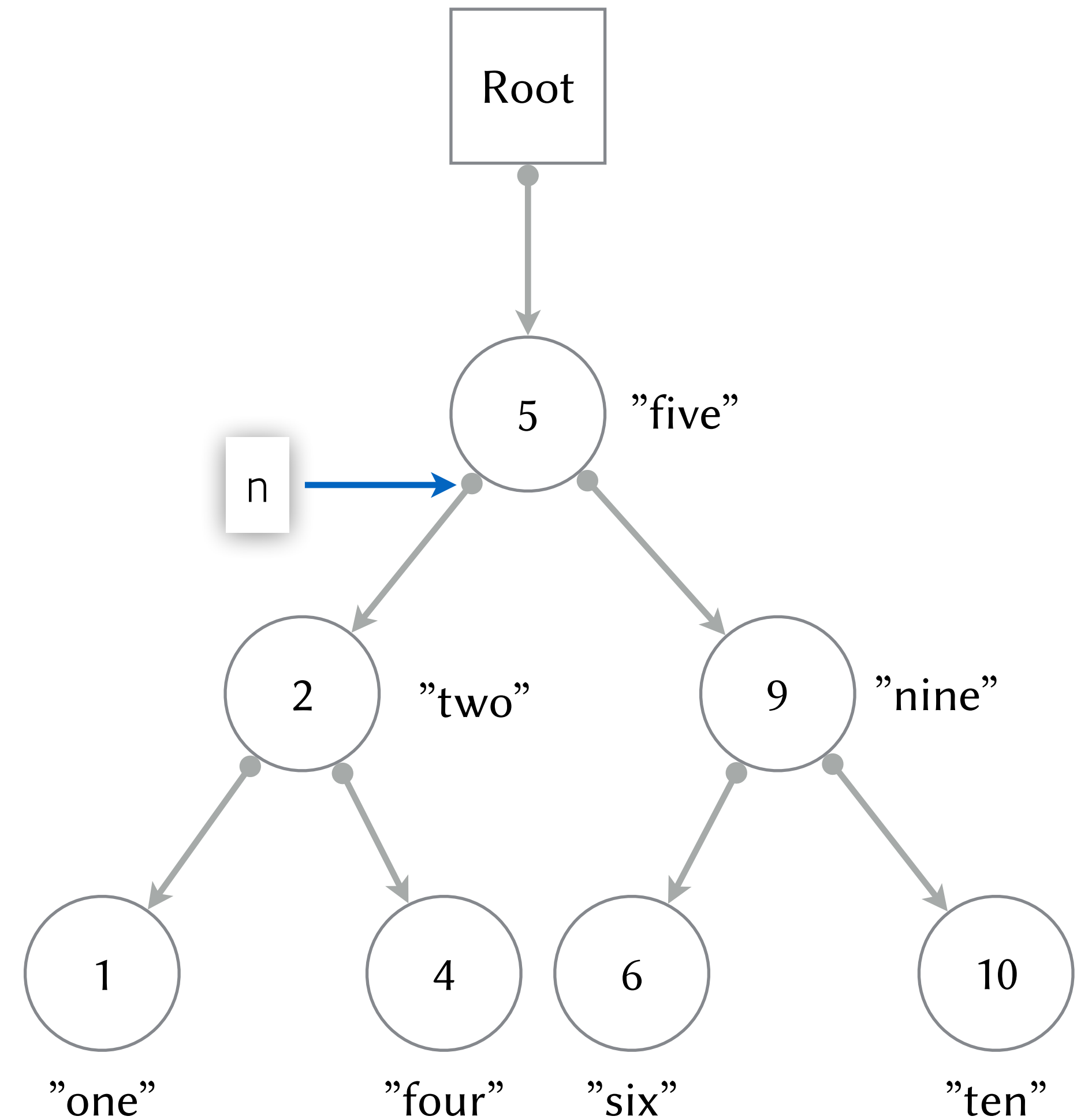
En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}
```

```
bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```

node_find_incoming_ptr(&t->root, 4)



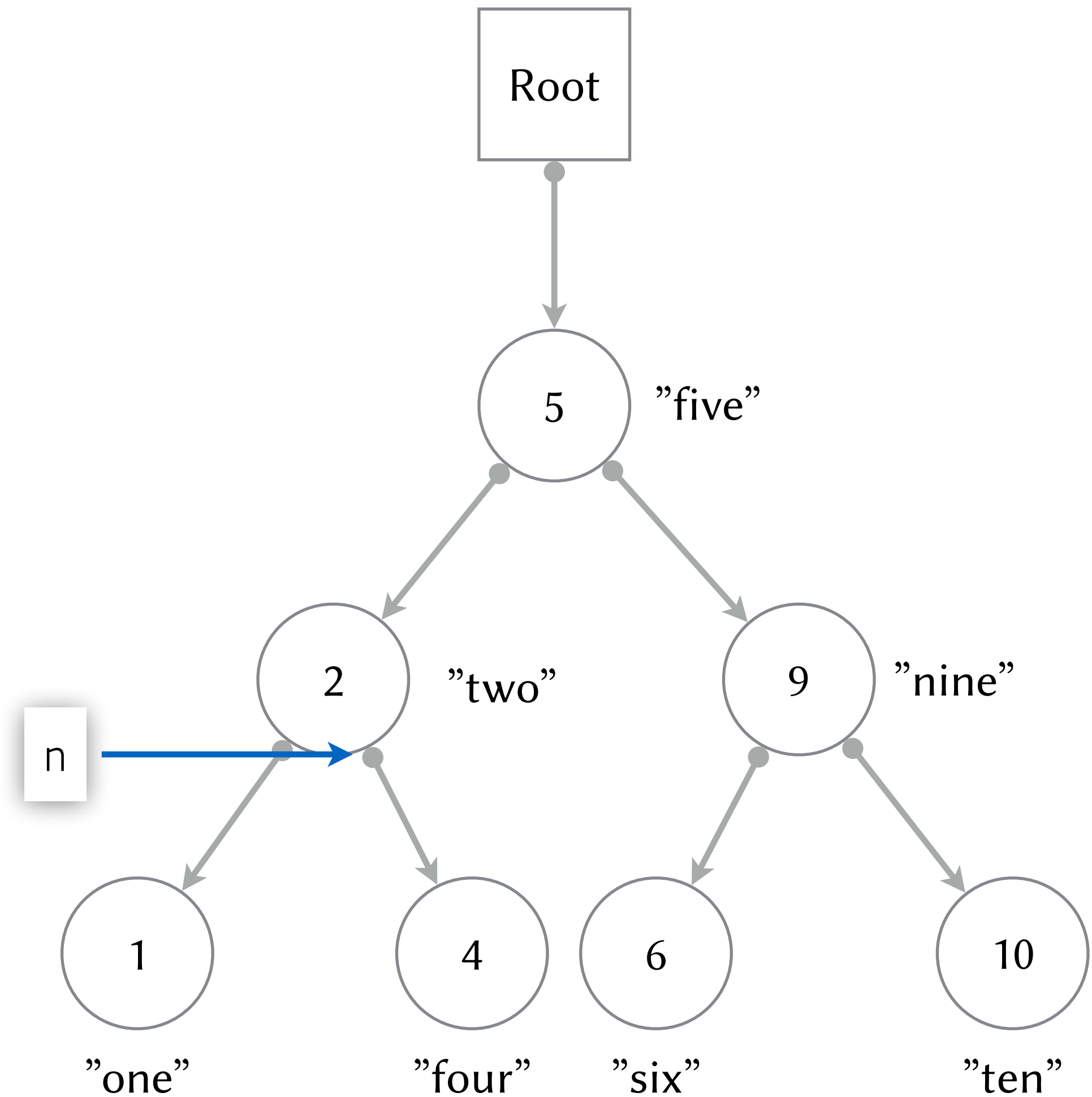
En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}
```

```
bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```

node_find_incoming_ptr(&t->root, 4)



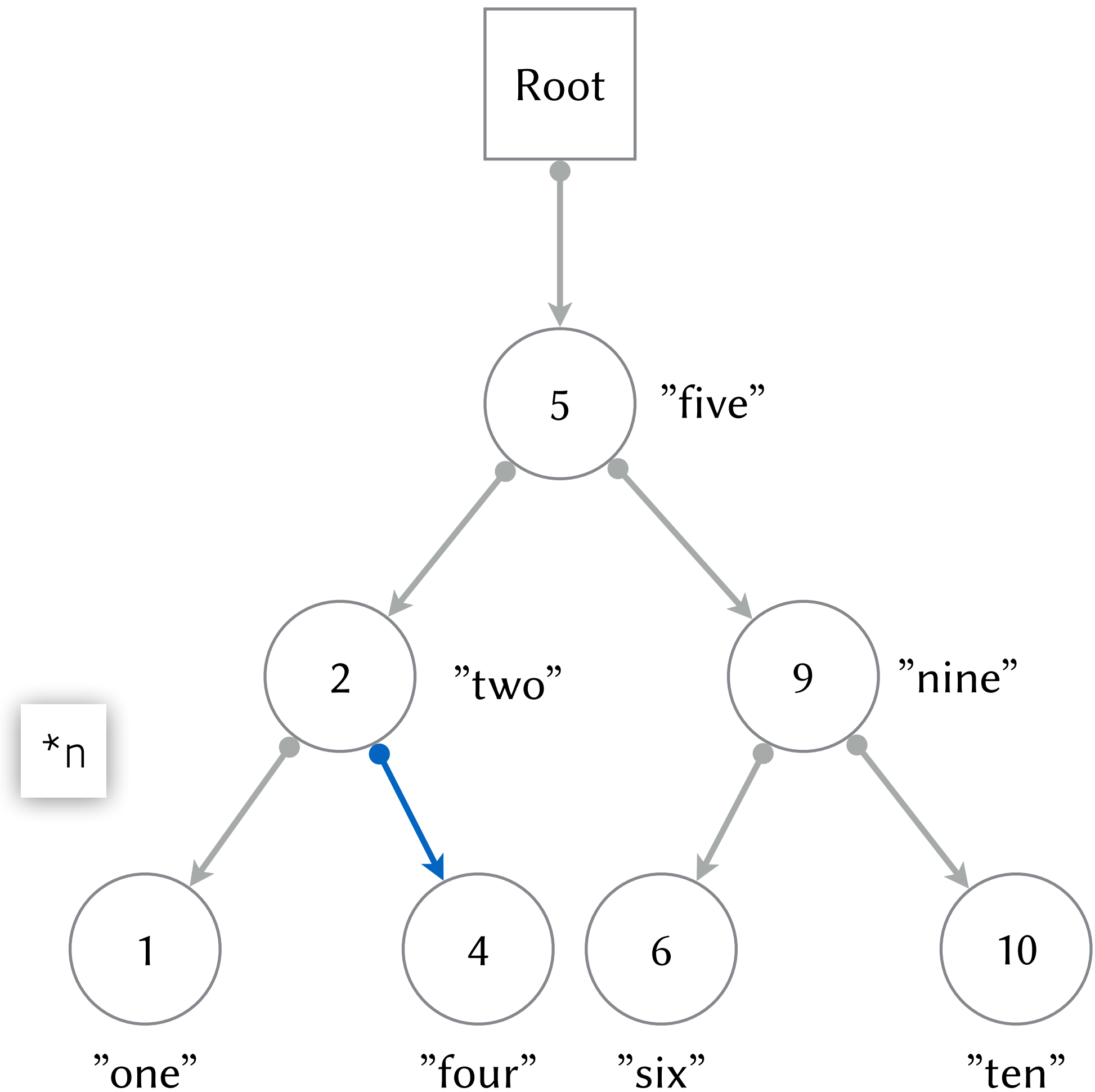
En smart sökrutin för binära sökträd

```
node_t **node_find_incoming_ptr(node_t **n, int key)
{
    while (*n && (*n)->key != key)
    {
        n = ((*n)->key > key)
            ? &n->left
            : &n->right;
    }

    return n;
}

bool tree_contains_key(tree_t *t, int key)
{
    return *node_find_incoming_ptr(&t->root, key) != NULL;
}
```

node_find_incoming_ptr(&t->root, 4)




```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

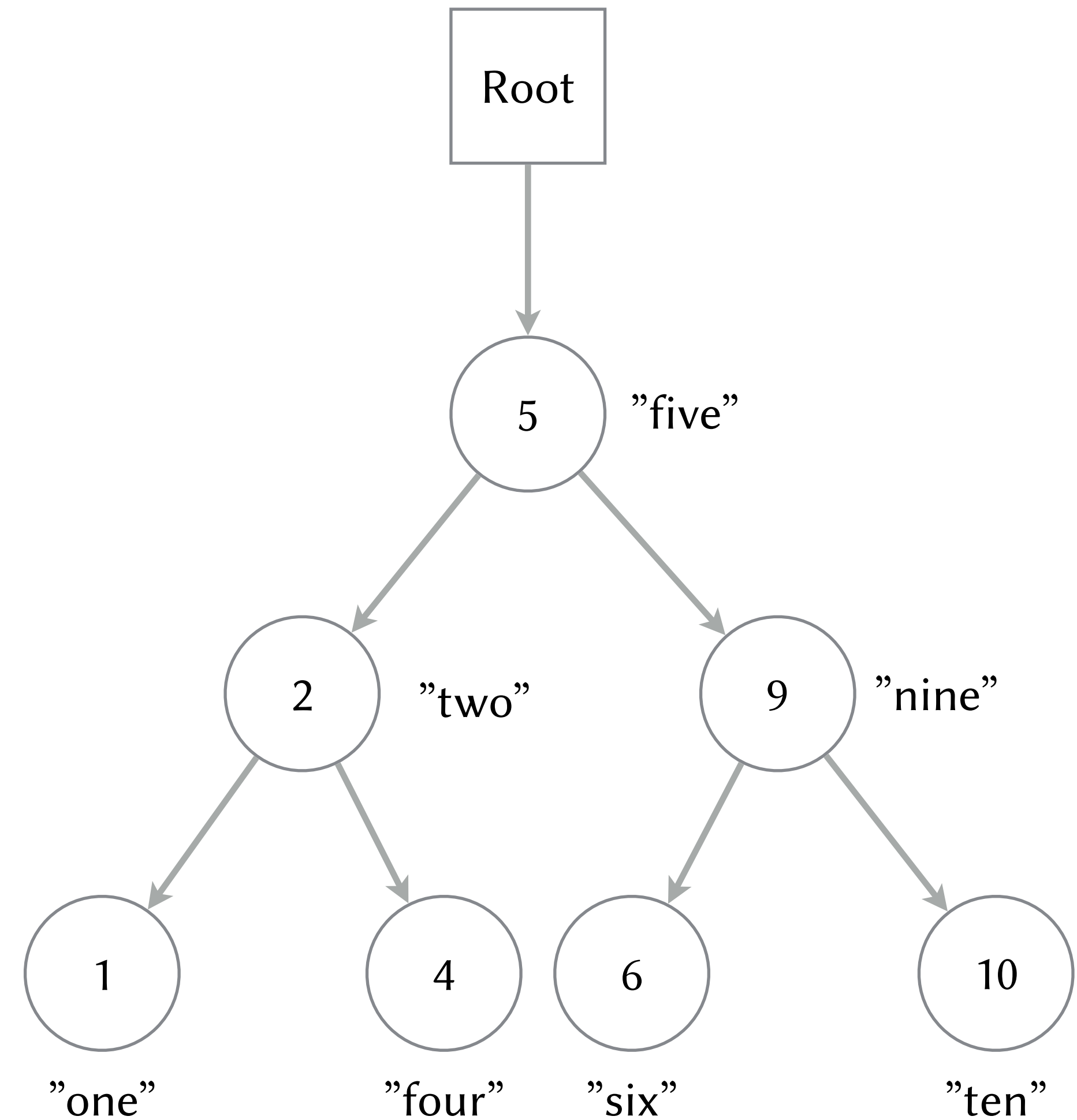
```

```

char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

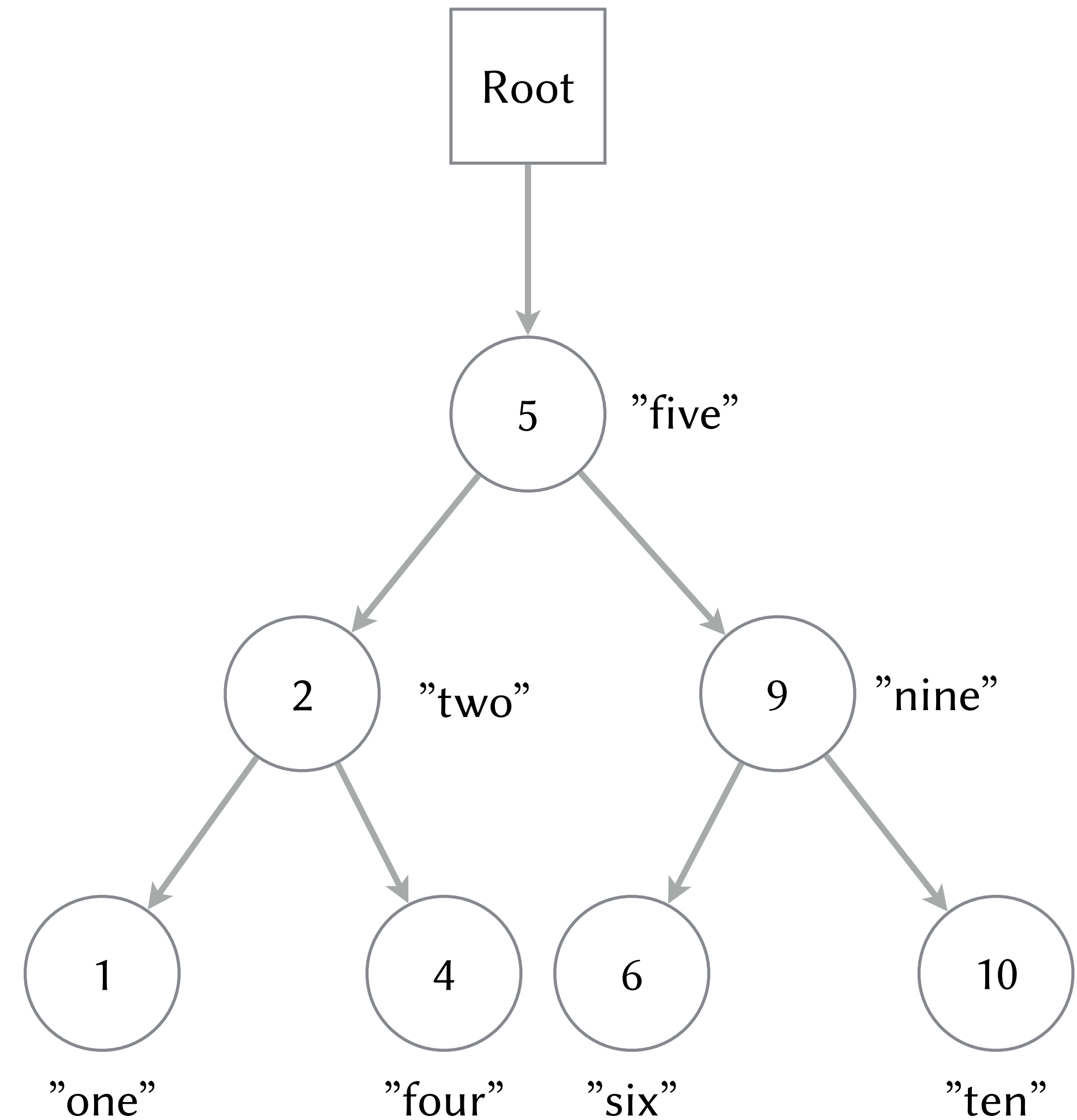
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

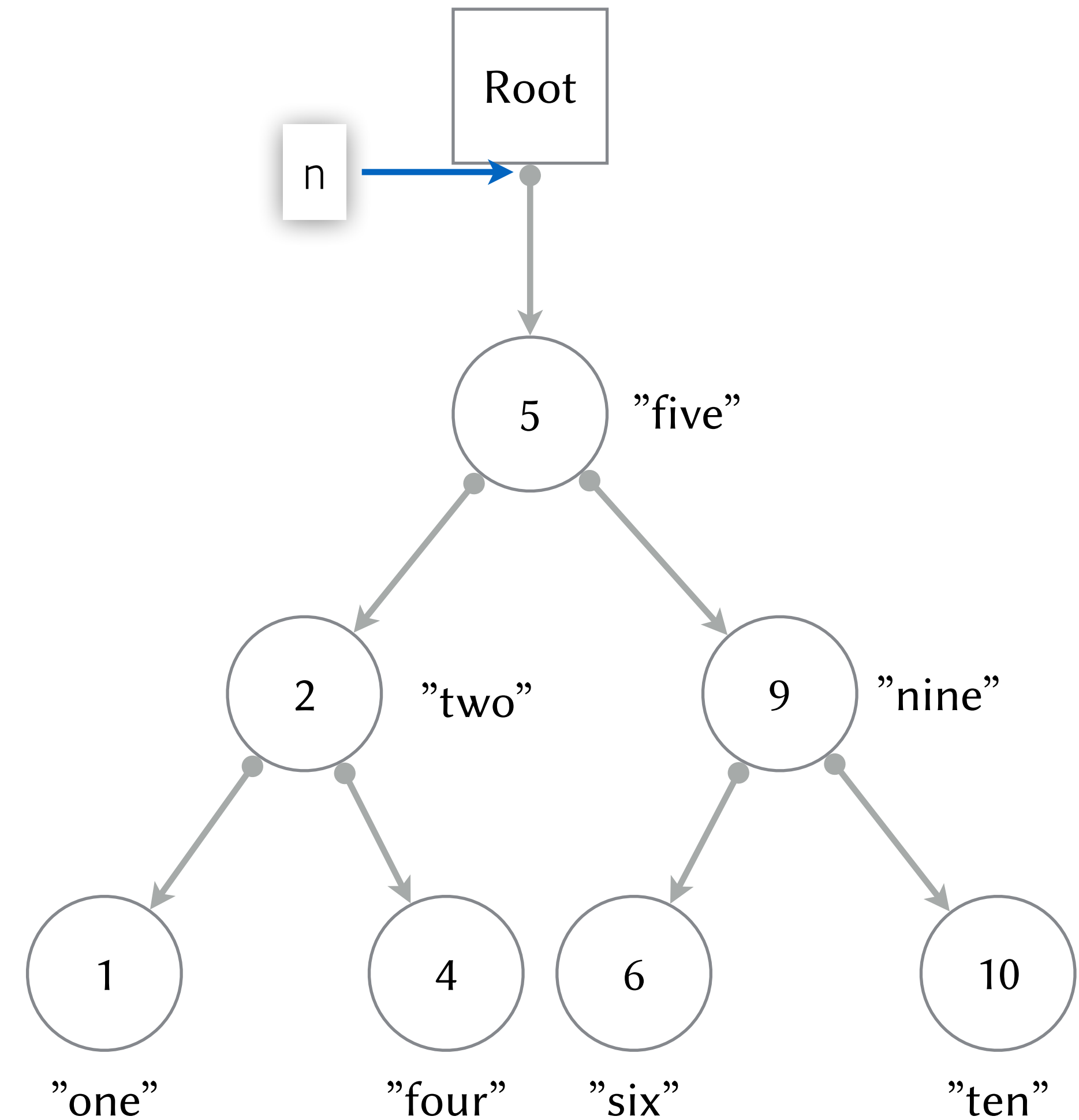
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

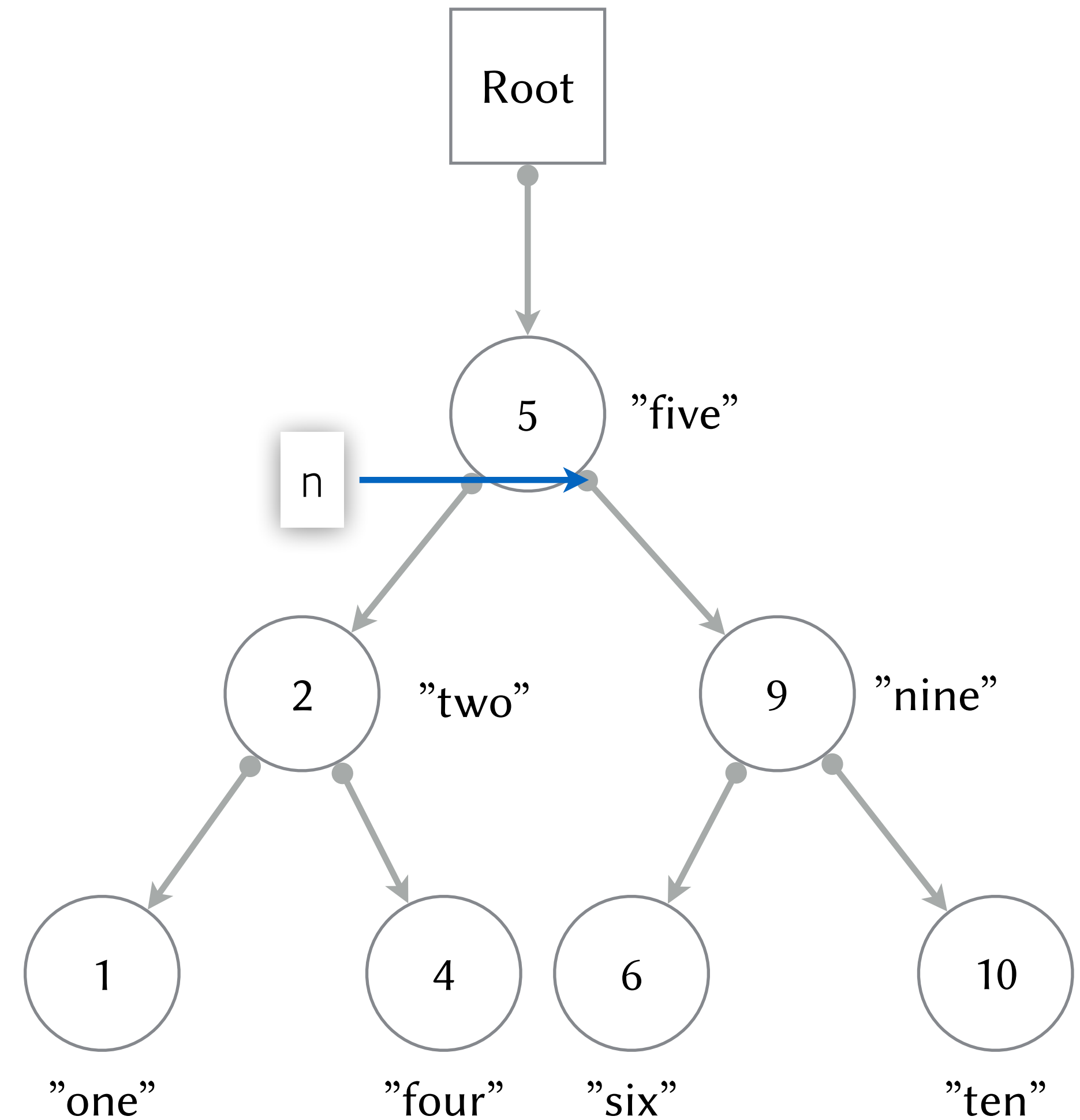
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

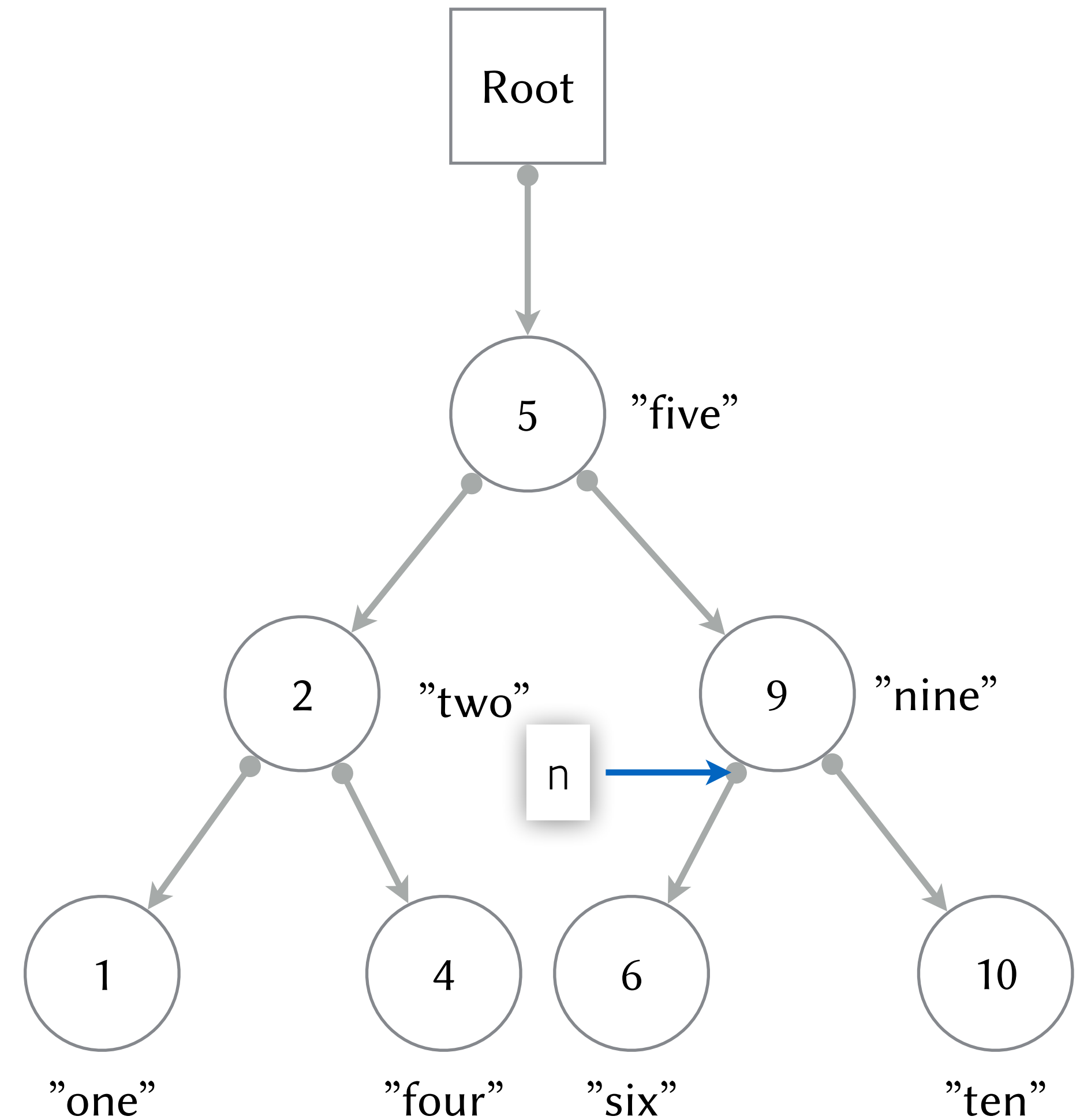
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

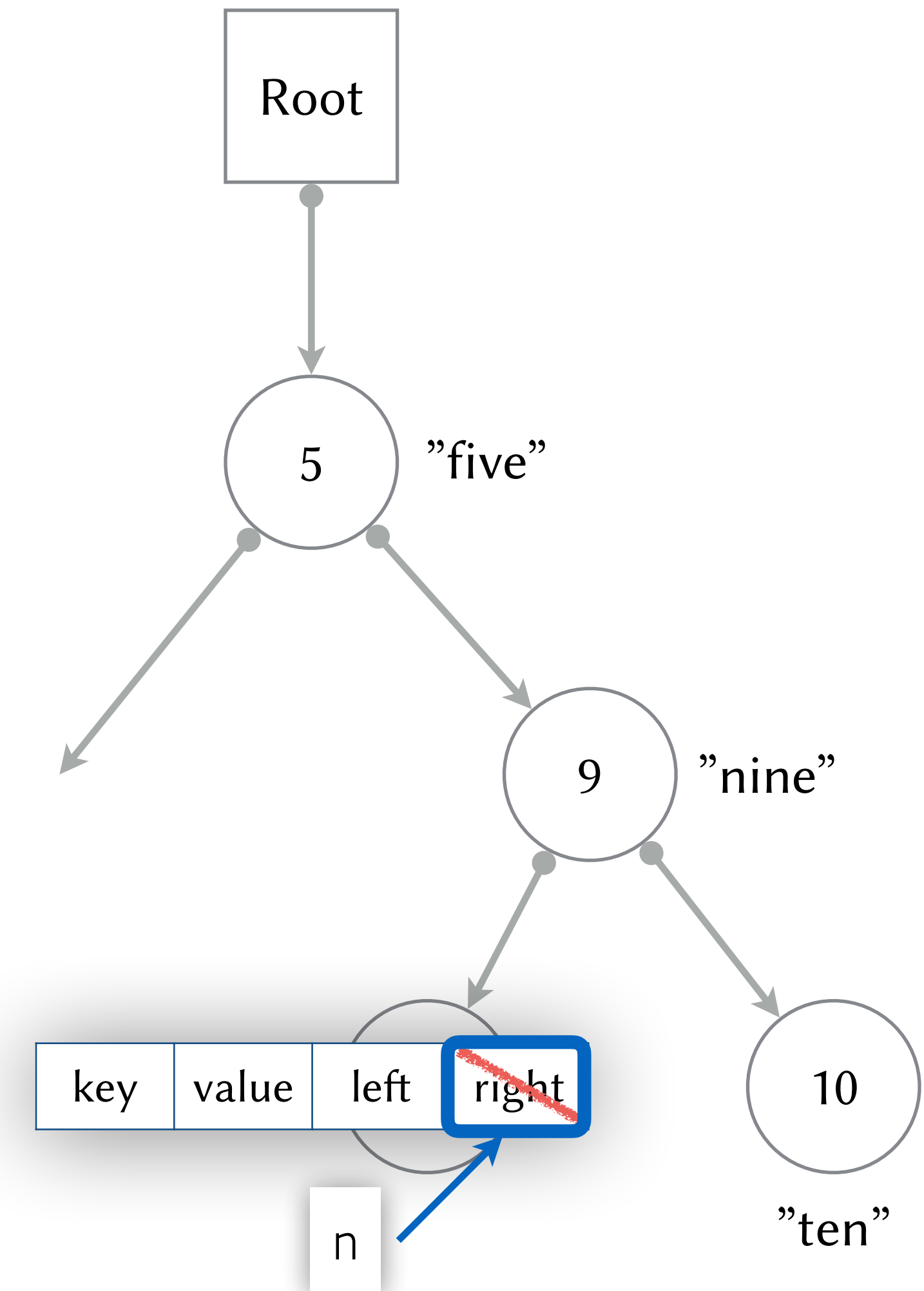
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

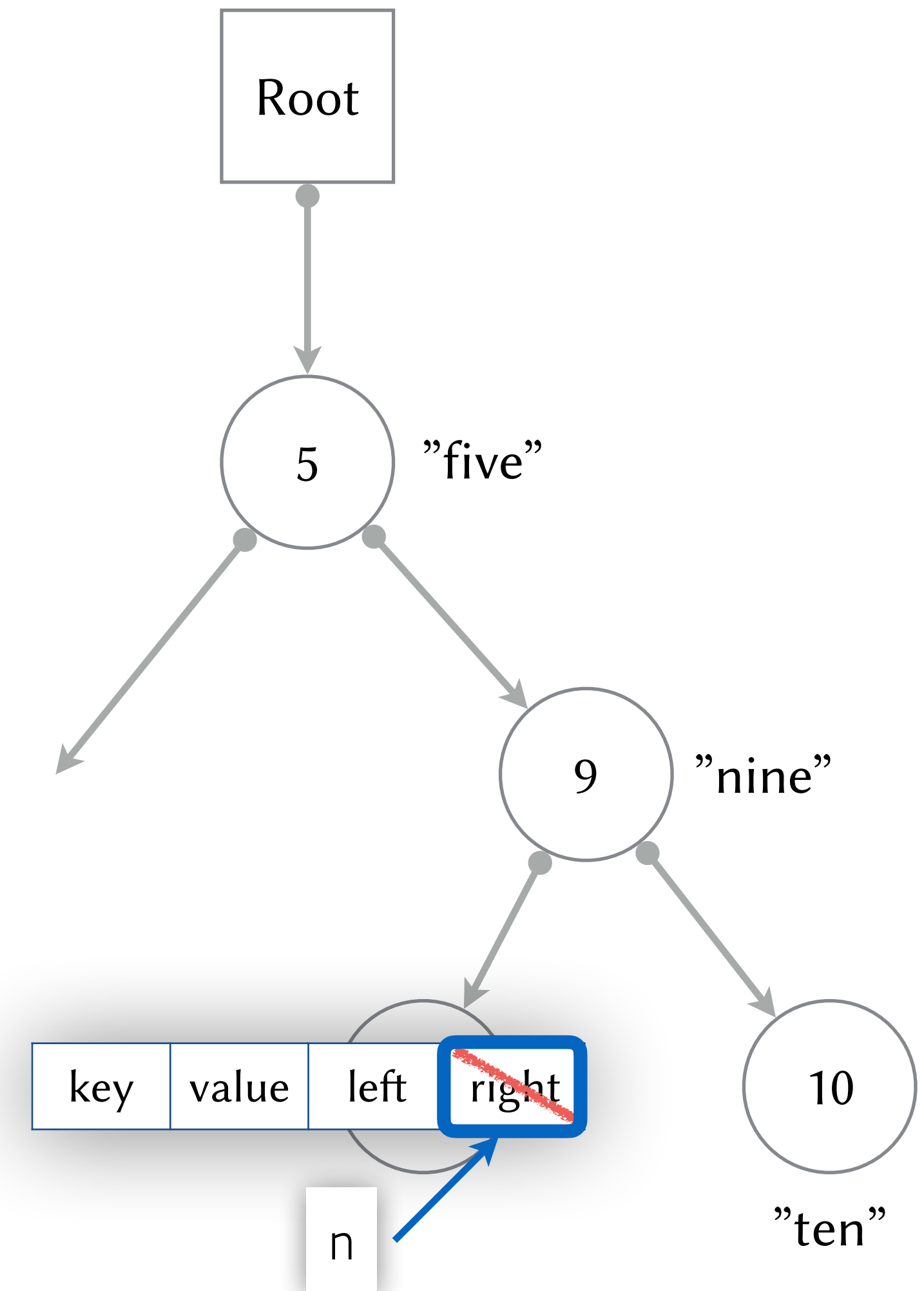
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

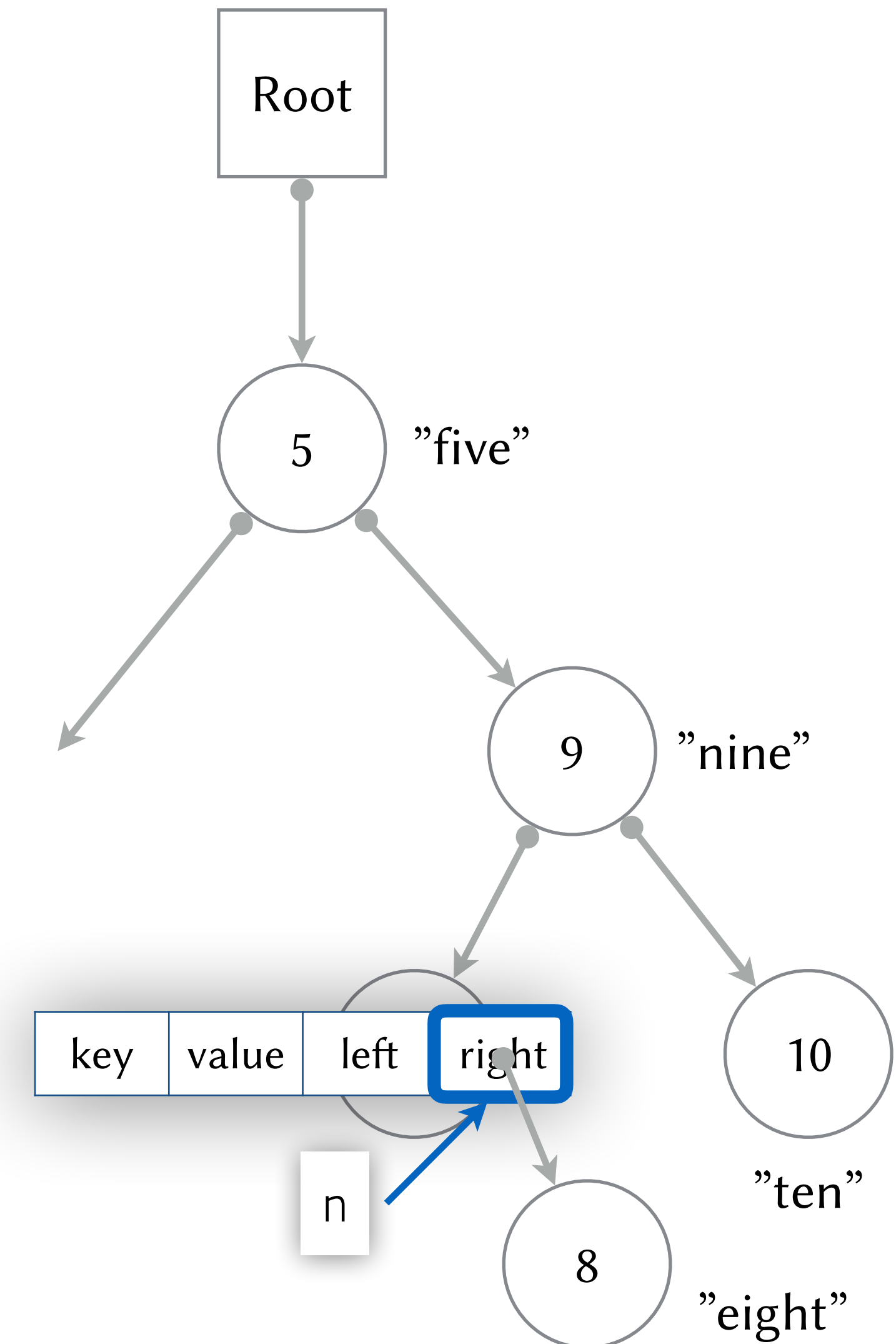
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)




```

char *tree_get(tree_t *t, int key)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    return *n ? (*n)->value : NULL;
}

```

```

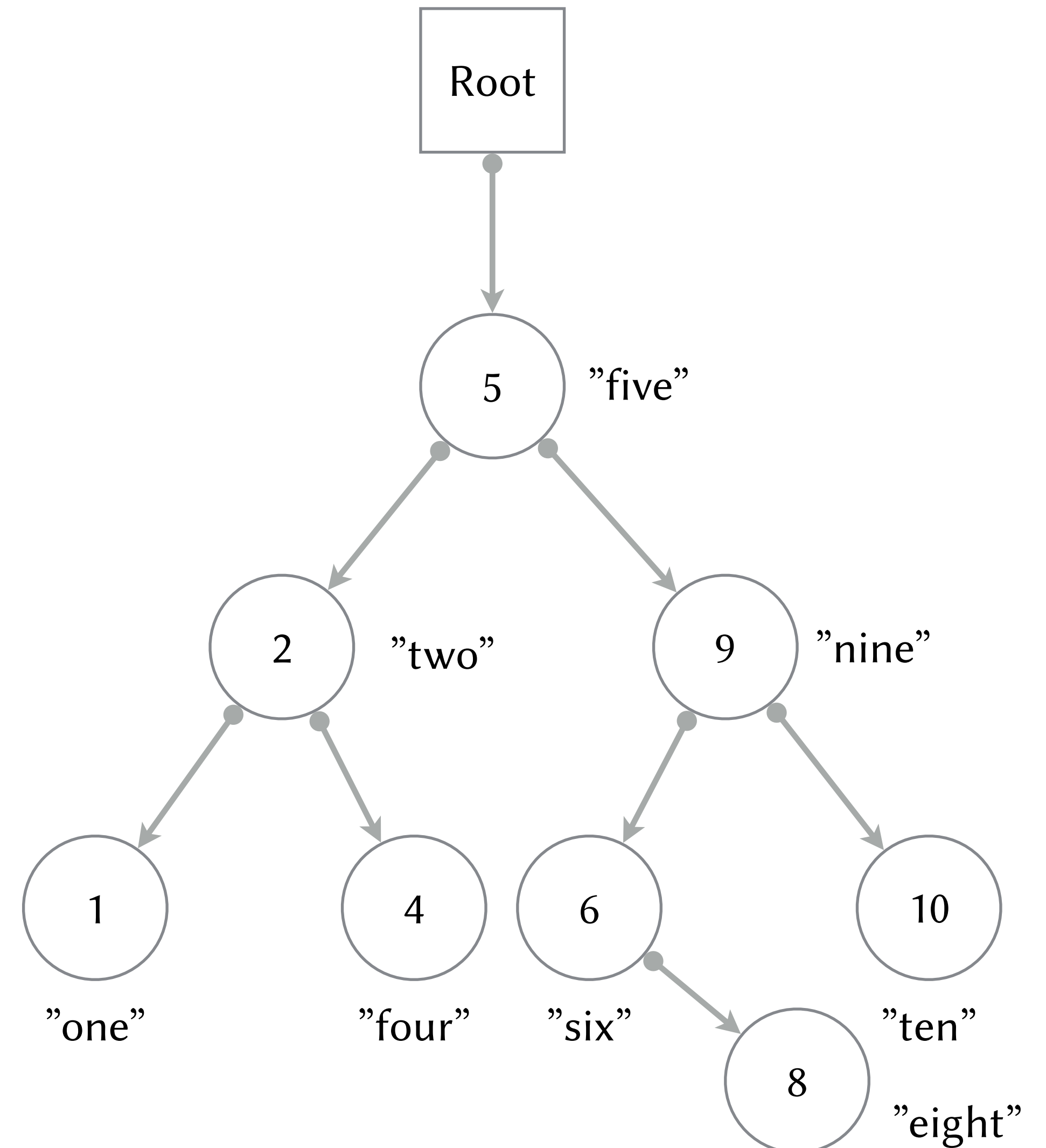
char *tree_put(tree_t *t, int key, char *value)
{
    node_t **n = node_find_incoming_ptr(&t->root, key);

    if (*n == NULL)
    {
        *n = node_create(key, value);
    }
    else
    {
        char *old = (*n)->value;
        (*n)->value = value;
        return old;
    }
}

```

tree_put(t, 8, "eight")

node_find_incoming_ptr(&t->root, 8)



Sammanfattning

Länkade strukturer är extremt vanliga

Länkade listan är enklaste exemplet

Binära träd, n-ställiga träd, grafer, skip lists, etc.

Länkarna i strukturen styr möjligheterna att navigera i datat

Ex: last-pekare, next/previous, parent pointer, etc.

Extra länkar eller pekare kan göra koden enklare (trade-off)

Länkade strukturer är pekartunga

Så ett bra sätt att förstå pekare under kursen