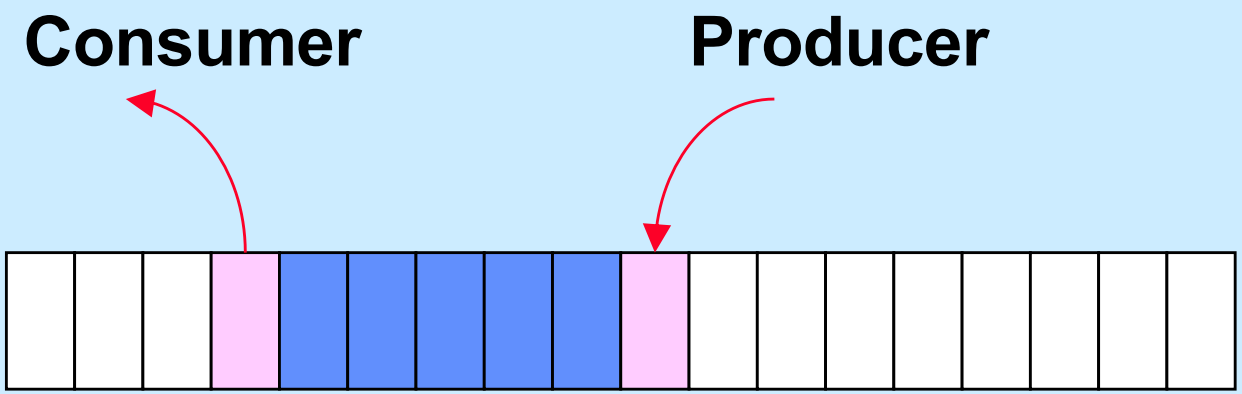
CS 33

Multithreaded Programming III

Producer-Consumer Problem



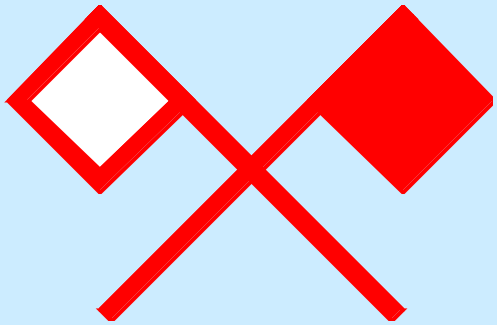
Guarded Commands

```
when (guard) [
  /*
     once the guard is true, execute this
     code atomically
   */

  ...

]
```

Semaphores



- **P(S) operation:**

```
when (S > 0) [
    S = S - 1;
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Producer/Consumer with Semaphores

```
Semaphore empty = BSIZE;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  if (++nextout >= BSIZE)
    nextout = 0;
  V(empty);
  return item;
}
```

POSIX Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared, int init);
int sem_destroy(sem_t *semaphore);
int sem_wait(sem_t *semaphore);
    /* P operation */
int sem_trywait(sem_t *semaphore);
    /* conditional P operation */
int sem_post(sem_t *semaphore);
    /* V operation */
```

Producer-Consumer with POSIX Semaphores

```
sem_init(&empty, 0, BSIZE);
sem_init(&occupied, 0, 0);
int nextin = 0;
int nextout = 0;
```

```
void produce(char item) {
  sem_wait(&empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
    nextin = 0;
  sem_post(&occupied);
}
```

```
char consume( ) {
  char item;
  sem_wait(&occupied);
  item = buf[nextout];
  if (++nextout >= BSIZE)
    nextout = 0;
  sem_post(&empty);
  return item;
}
```

Quiz 1

Does the POSIX version of the producer-consumer solution work with multiple producers and consumers?

a) **Yes**

b) **No, but it can be made to work by using mutexes to make sure that only one thread is executing the producer code at a time and only one thread is executing the consumer code at a time**

c) **It can't easily be made to work**

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s);

void start(state_t *s);

void stop(state_t *s);
```


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  if (s->state == stopped)
    sleep();
}
void start(state_t *s) {
  state = started;
  wakeup_all();
}
void stop(state_t *s) {
  state = stopped;
}
```


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    pthread_mutex_unlock(&s->mutex);
    sleep();
  else pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    sleep();
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Condition Variables

```
when (guard) [
  statement 1;
  …
  statement n;
]
```

```
pthread_mutex_lock(&mutex);
while(!guard)
  pthread_cond_wait(
     &cond_var, &mutex);
statement 1;
…
statement n;
pthread_mutex_unlock(&mutex);
```

```
// code modifying the guard:
…
```

```
pthread_mutex_lock(&mutex);
// code modifying the guard:
…
pthread_cond_broadcast(
    &cond_var);
pthread_mutex_unlock(&mutex);
```

Set Up

```
int pthread_cond_init(pthread_cond_t *cvp,
    pthread_condattr_t *attrp)

int pthread_cond_destroy(pthread_cond_t *cvp)

int pthread_condattr_init(pthread_condattr_t *attrp)

int pthread_condattr_destroy(pthread_condattr_t *attrp)
```

PC with Condition Variables (1)

```
typedef struct buffer {
   pthread_mutex_t m;
   pthread_cond_t  more_space;
   pthread_cond_t  more_items;
   int             next_in;
   int             next_out;
   int             empty;
   char            buf[BSIZE];
} buffer_t;
```

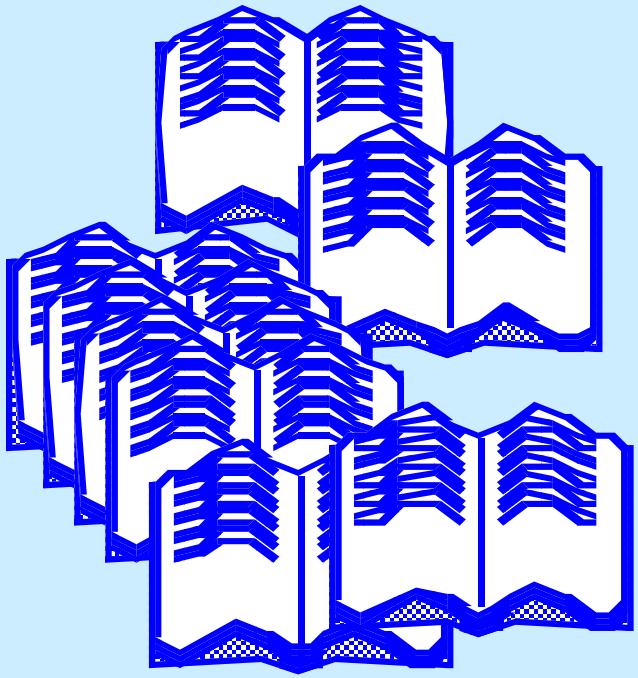
PC with Condition Variables (2)

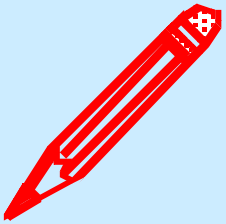
```
void produce(buffer_t *b,
     char item) {

  pthread_mutex_lock(&b->m);
  while (!(b->empty > 0))
    pthread_cond_wait(
        &b->more_space, &b->m);
  b->buf[b->nextin] = item;
  if (++(b->nextin) == BSIZE)
    b->nextin = 0;
  b->empty--;
  pthread_cond_signal(
     &b->more_items);
  pthread_mutex_unlock(&b->m);
}
```

```
char consume(buffer_t *b) {
  char item;
  pthread_mutex_lock(&b->m);
  while (!(b->empty < BSIZE))
    pthread_cond_wait(
        &b->more_items, &b->m);
  item = b->buf[b->nextout];
  if (++(b->nextout) == BSIZE)
    b->nextout = 0;
  b->empty++;
  pthread_cond_signal(
     &b->more_space);
  pthread_mutex_unlock(&b->m);
  return item;
}
```

Readers-Writers Problem





Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]

  /* read */

  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
      (readers == 0)) [
    writers++;
  ]

  /* write */

  [writers--;]
}
```

Pseudocode with Assertions

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]

  assert((writers == 0) &&
     (readers > 0));
  /* read */

  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
     (readers == 0)) [
    writers++;
  ]

  assert((readers == 0) &&
     (writers == 1));
  /* write */

  [writers--;]
}
```

Solution with POSIX Threads

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
       (writers == 0)))
    pthread_cond_wait(
        &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
       &writersQ);
  pthread_cond_broadcast(
     &readersQ);
  pthread_mutex_unlock(&m);
}
```

Quiz 2

If a thread calls *writer*, it will eventually return from writer (assuming well behaved threads).

a) yes, always

b) it will usually return, but it's possible that it will not return

c) it might return, but it's highly likely that it will never return

d) no, never

New Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]

  /* read */

  [readers--;]
}
```

```
writer( ) {
  [writers++;]
  when ((readers == 0) &&
       (active_writers == 0)) [
    active_writers++;
  ]

  /* write */

  [writers--;
  active_writers--;]
}
```

Improved Reader

```
reader( ) {
  pthread_mutex_lock(&m);

  while (!(writers == 0)) {
    pthread_cond_wait(
          &readersQ, &m);
  }
  readers++;

  pthread_mutex_unlock(&m);

  /* read */

  pthread_mutex_lock(&m);

  if (--readers == 0)
    pthread_cond_signal(
          &writersQ);

  pthread_mutex_unlock(&m);
}
```

Improved Writer

```
writer( ) {
  pthread_mutex_lock(&m);

  writers++;
  while (!((readers == 0) &&
        (active_writers == 0))) {
    pthread_cond_wait(
          &writersQ, &m);
  }
  active_writers++;

  pthread_mutex_unlock(&m);

  /* write */

  pthread_mutex_lock(&m);

  writers--;
    active_writers--;
  if (writers)
    pthread_cond_signal(
          &writersQ);
  else
    pthread_cond_broadcast(
          &readersQ);

  pthread_mutex_unlock(&m);
}
```

Quiz 3

If a thread calls *reader*, it will eventually return from writer (assuming well behaved threads).

a) yes, always

b) it will usually return, but it's possible that it will not return

c) it might return, but it's highly likely that it will never return

d) no, never

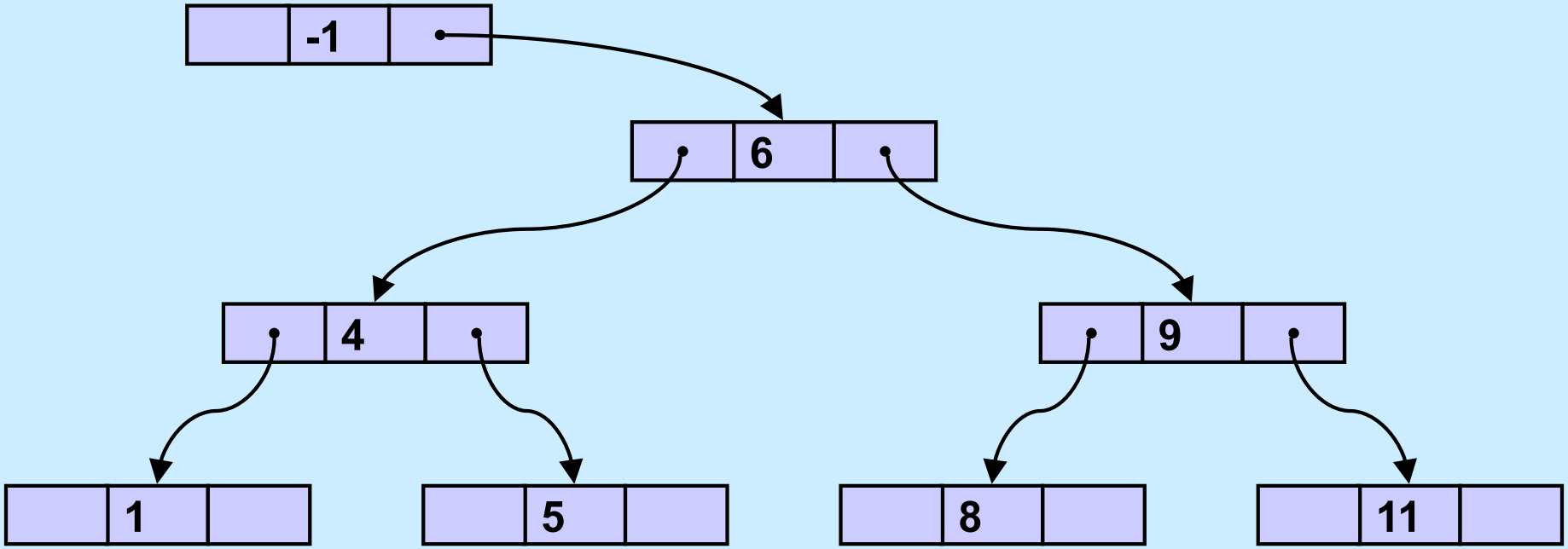
New, From POSIX!

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
      pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(pthread_rwlock_t *lock,
      struct timespec *ts);
int pthread_timedrwlock_wrlock(pthread_rwlock_t *lock,
      struct timespec *ts);
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

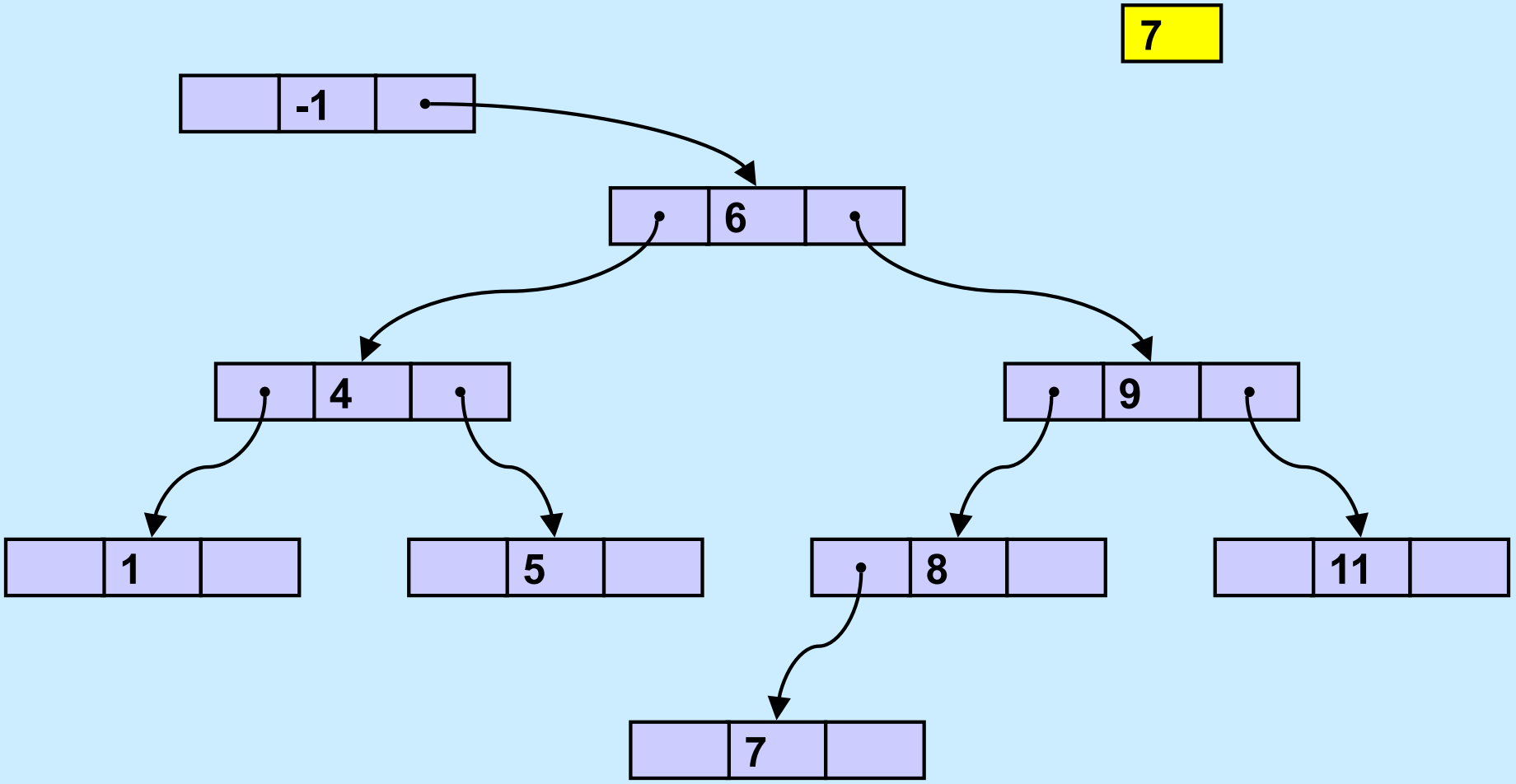
Quiz 4

- Missing in the rwlock API is a function to "upgrade" a readers lock into a writers lock. It's not included because
  a) it's rarely needed, so there's no point to including it
  b) the same effect could be achieved by unlocking the readers lock, then taking a writers lock
  c) using such a function would be likely to cause deadlock

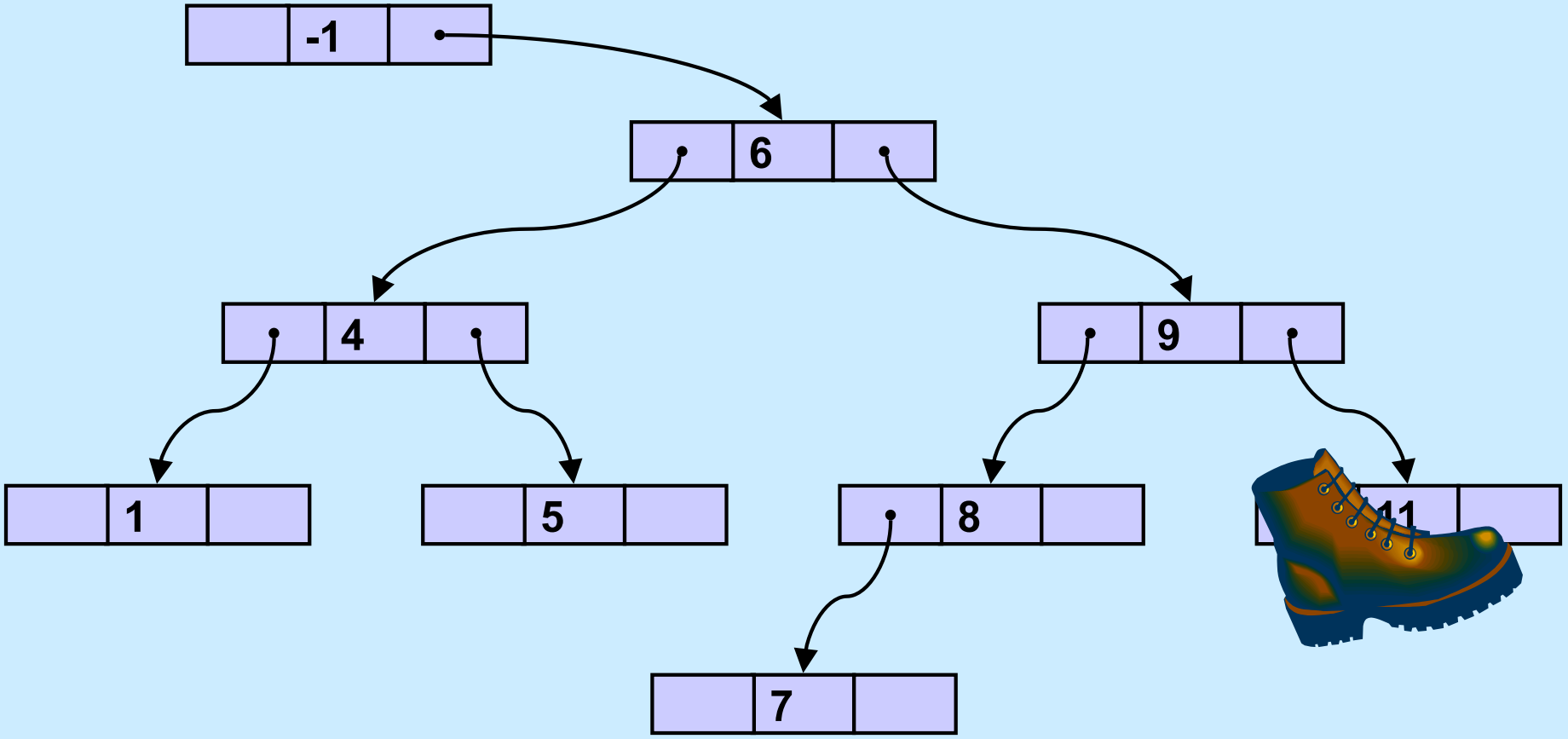
Binary Search Tree



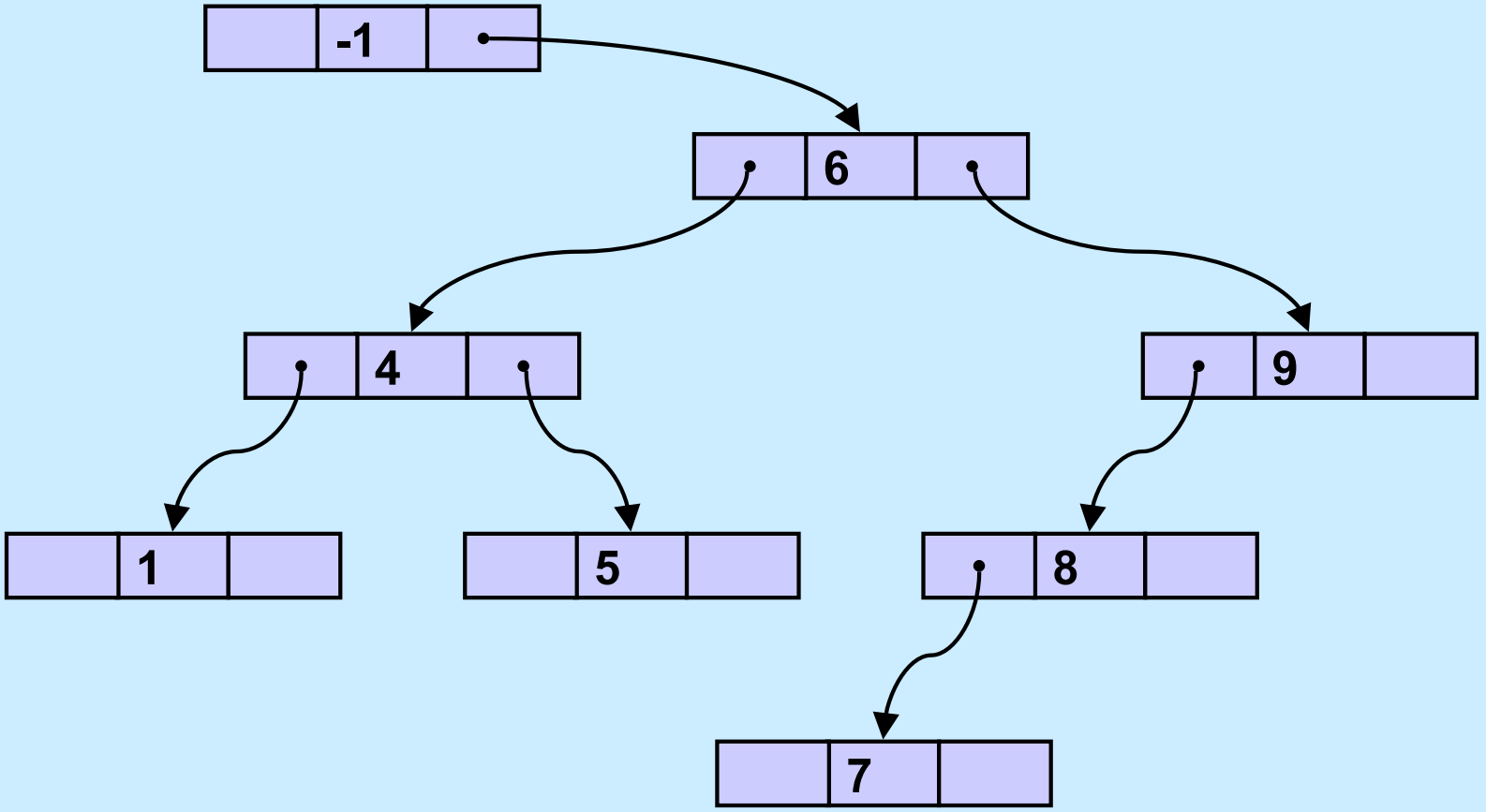
Binary Search Tree: Insertion



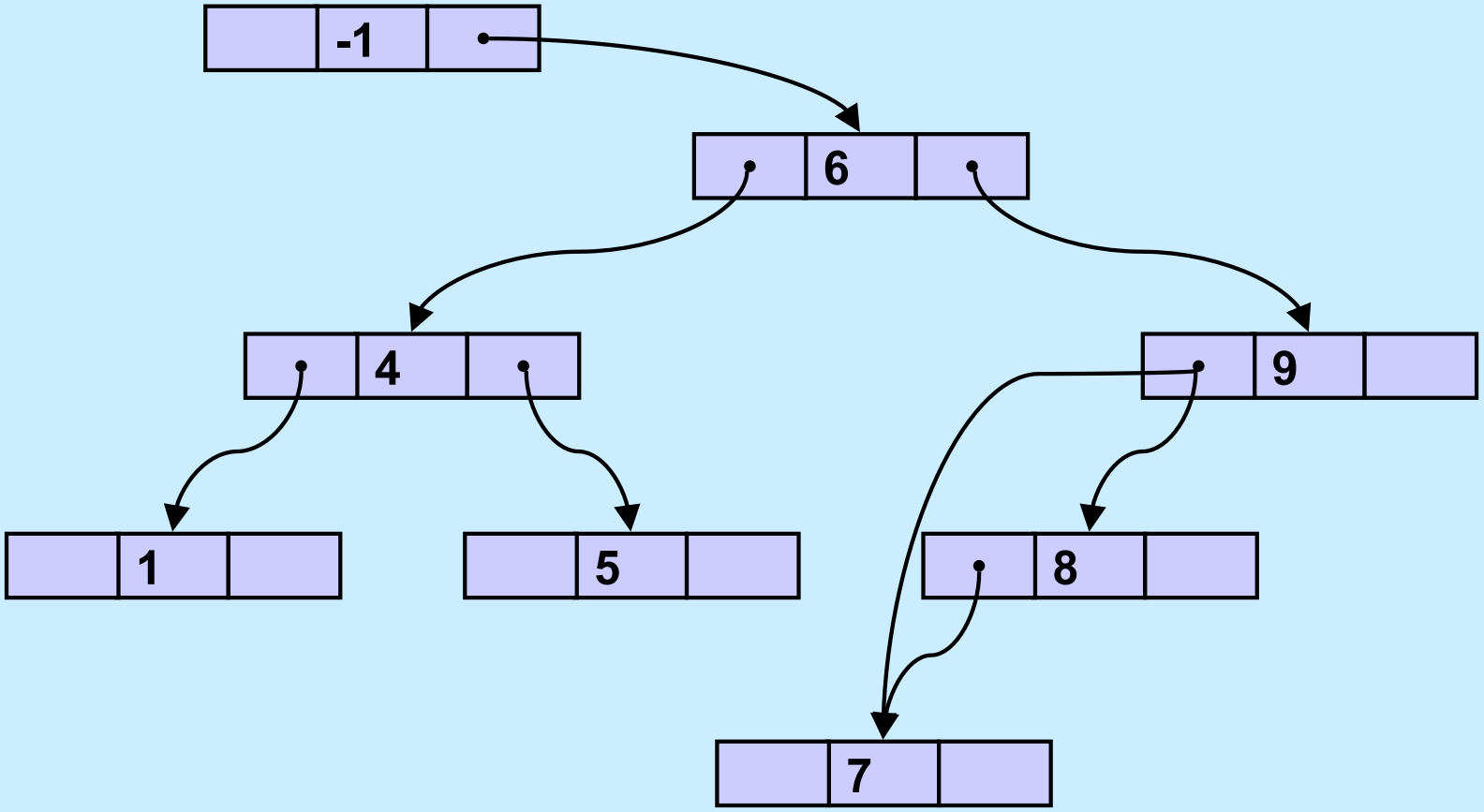
Binary Search Tree: Deletion of Leaf



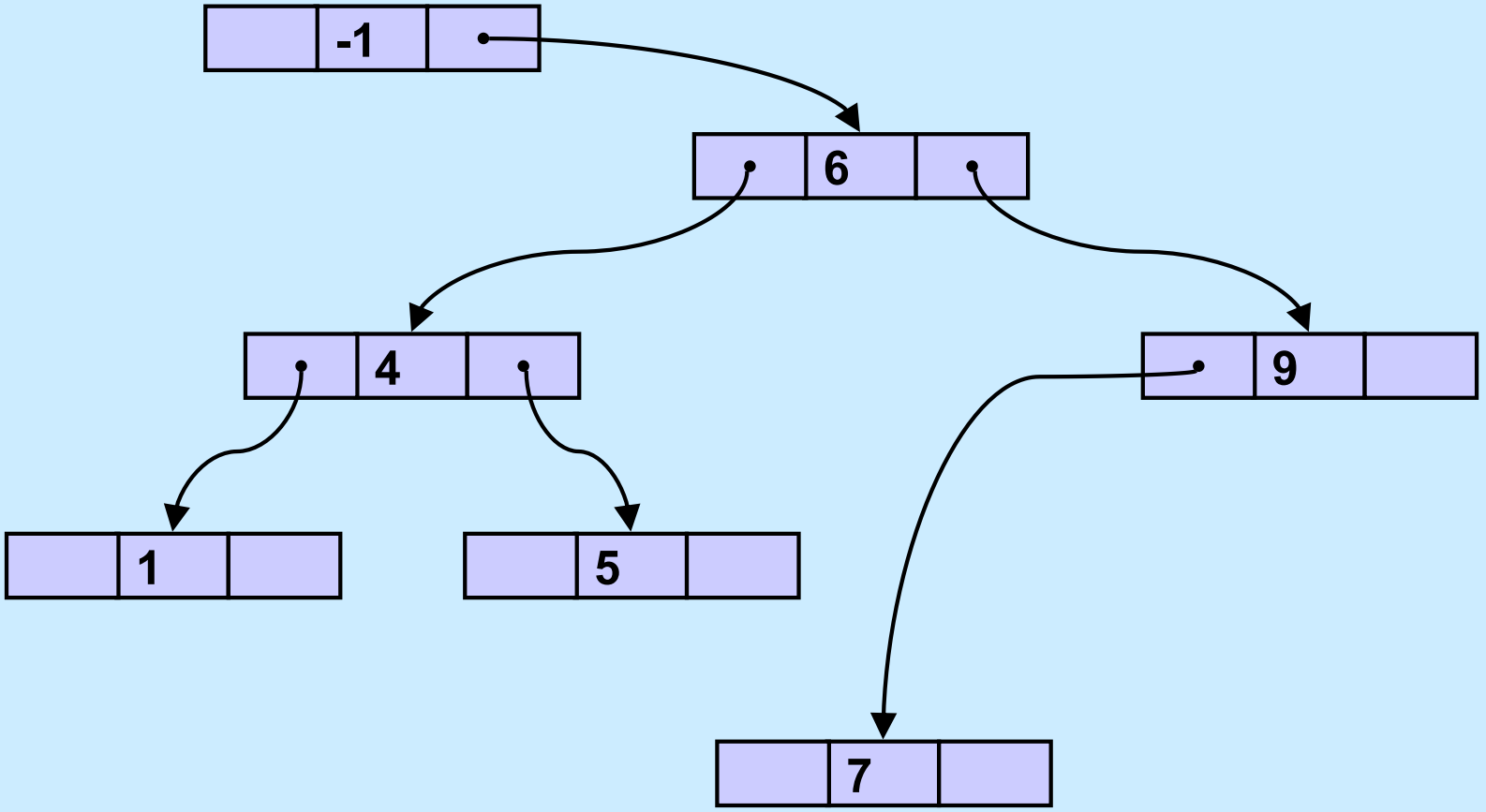
Binary Search Tree: Deletion of Leaf



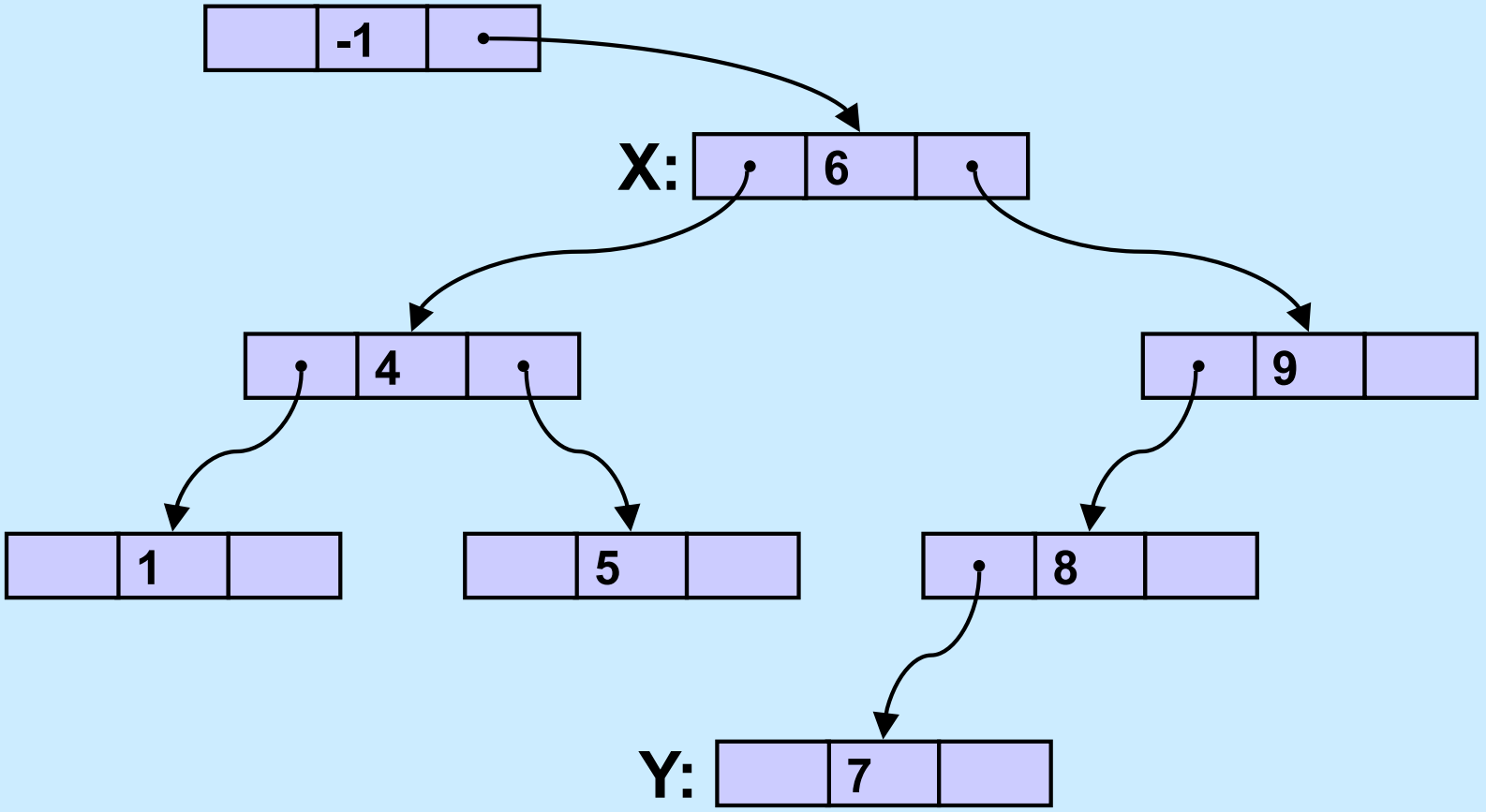
Binary Search Tree: Deletion of Node with One Child



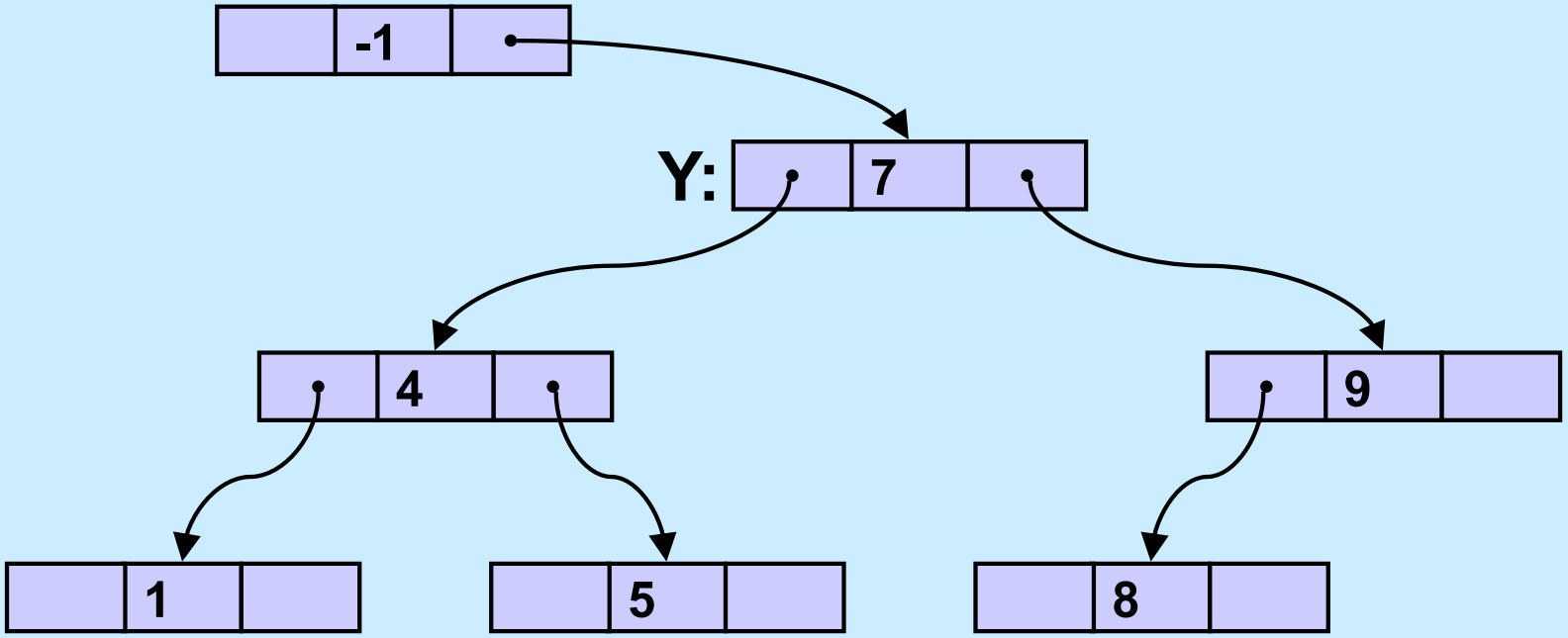
Binary Search Tree: Deletion of Node with One Child



Binary Search Tree: Deletion of Node with Two Children



Binary Search Tree: Deletion of Node with Two Children



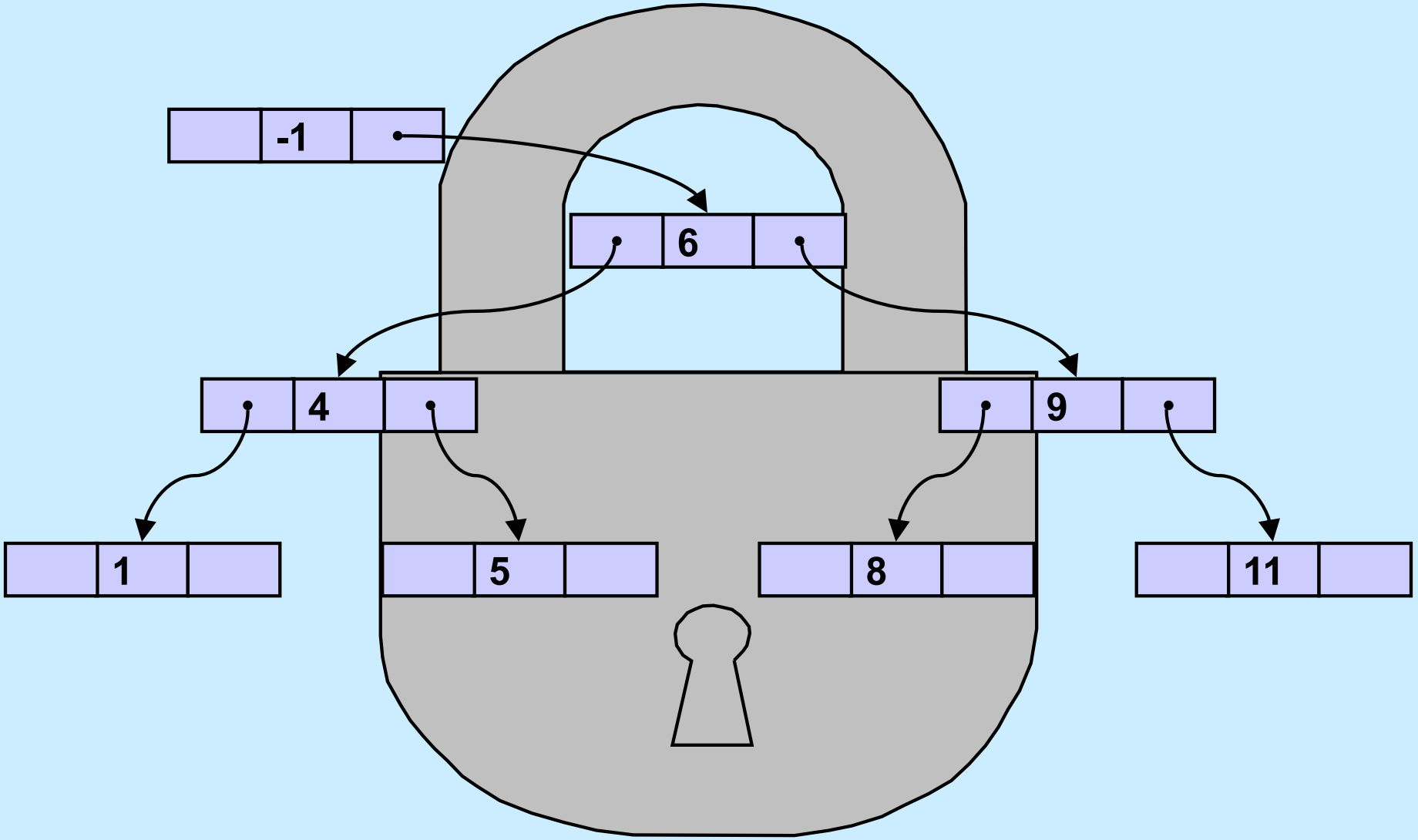
C Code: Search

```
Node *search(int key,
    Node *parent, Node **parentp) {
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
            next, parentpp);
        return result;
      }
    }
  } else {
    if ((next = parent->rchild)
        == 0) {
      result = 0;
    } else {
      if (key == next->key) {
        result = next;
      } else {
        result = search(key,
            next, parentpp);
        return result;
      }
    }
  }
  if (parentpp != 0)
    *parentpp = parent;
  return result;
}
```

C Code: Add

```
int add(int key) {
  Node *parent, *target, *newnode;
  if ((target = search(key, &head, &parent)) != 0) {
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;
  if (name < parent->name)
    parent->lchild = newnode;
  else
    parent->rchild = newnode;
  return 1;
}
```

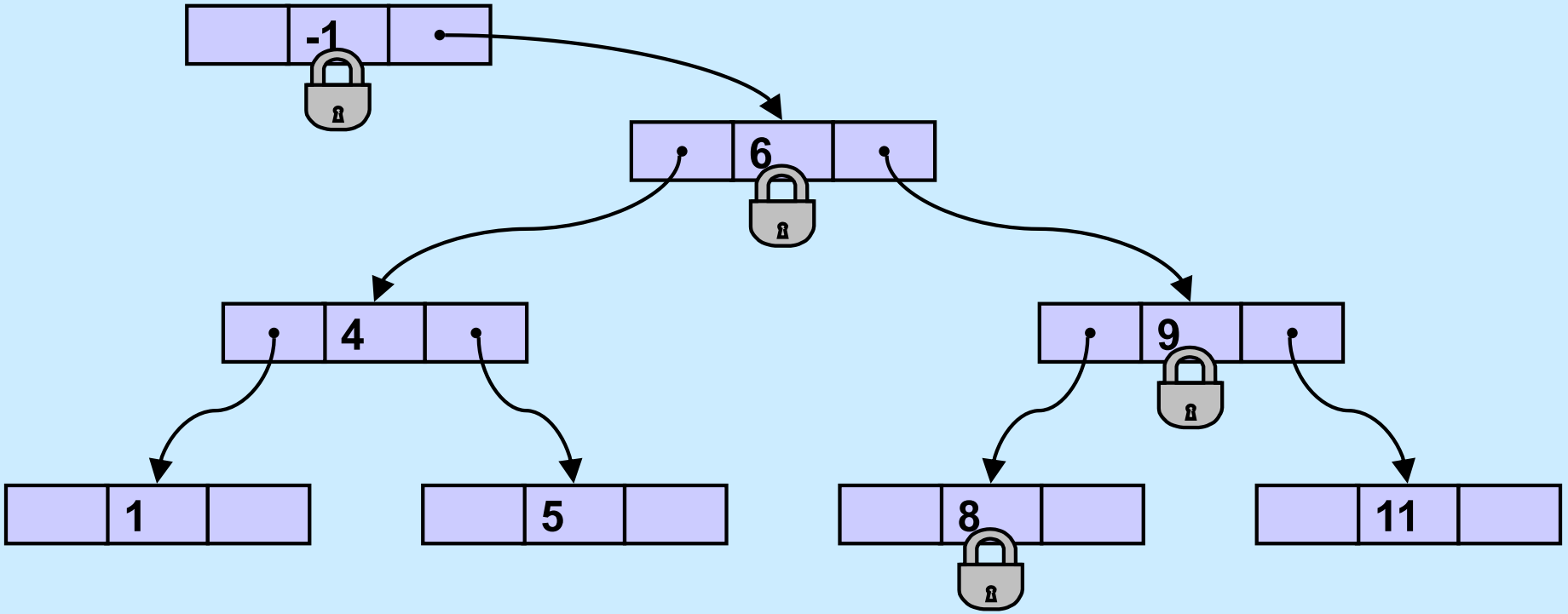

Binary Search Tree
with Coarse-Grained Synchronization



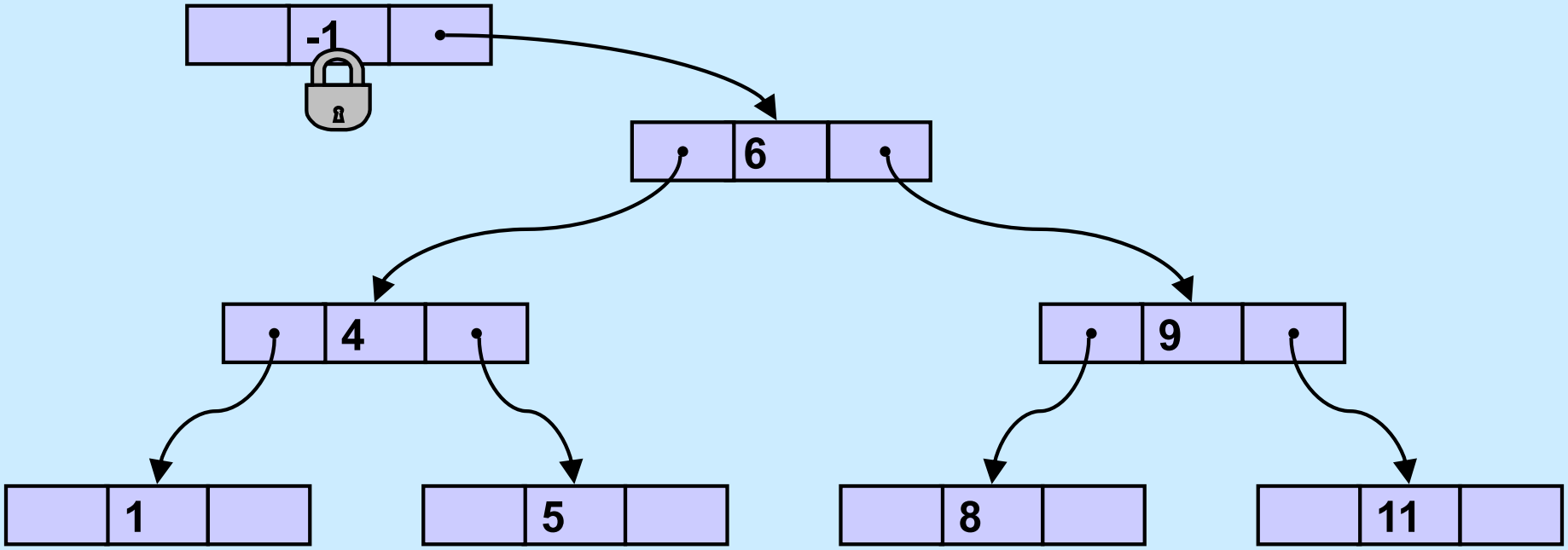
C Code: Add
with Coarse-Grained Synchronization

```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&tree_lock);
  if ((target = search(key, &head, &parent)) != 0) {
    pthread_rwlock_unlock(&tree_lock);
    return 0;
  }
  newnode = malloc(sizeof(Node));
  newnode->key = key;
  newnode->lchild = newnode->rchild = 0;
  if (name < parent->name)
    parent->lchild = newnode;
  else
    parent->rchild = newnode;
  pthread_rwlock_unlock(&tree_lock);
  return 1;
}
```

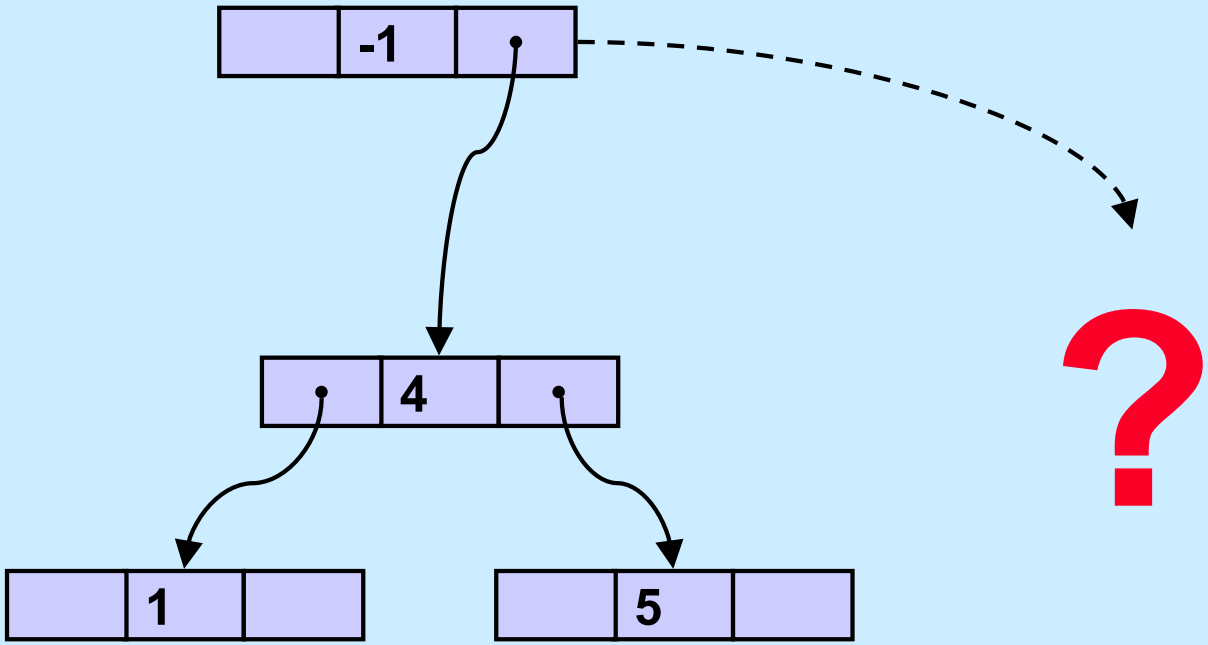
Binary Search Tree
with Fine-Grained Synchronization I



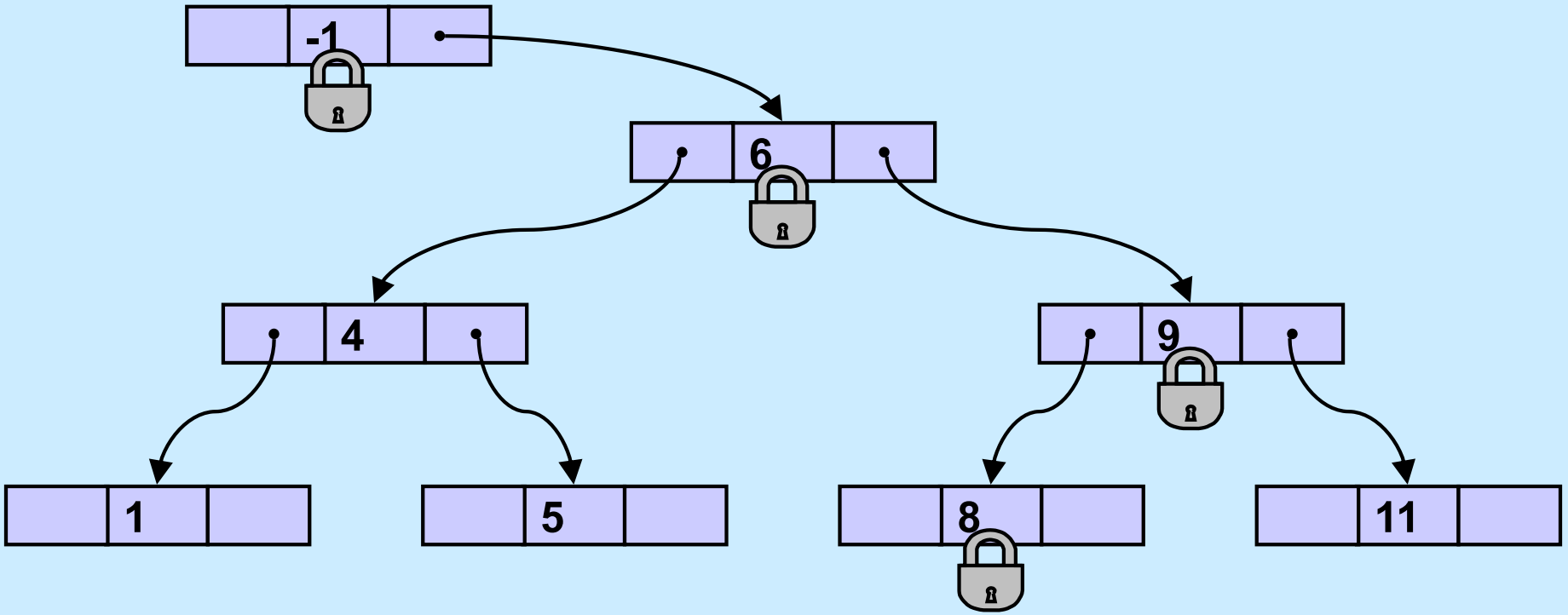
Binary Search Tree
with Fine-Grained Synchronization II



Binary Search Tree
with Fine-Grained Synchronization III



Doing It Right …



C Code: Fine-Grained Search I

```
enum locktype {l_read, l_write};

#define lock(lt, lk) ((lt) == l_read)?
      pthread_rwlock_rdlock(lk):
      pthread_rwlock_wrlock(lk)

Node *search(int key,
    Node *parent, Node **parentp,
    enum locktype lt) {
   // parent is locked on entry
  Node *next;
  Node *result;
  if (key < parent->key) {
    if ((next = parent->lchild)
        == 0) {
      result = 0;
} else {
      lock(lt, &next->lock);
      if (key == next->key) {
        result = next;
      } else {
        pthread_rwlock_unlock(
            &parent->lock);
        result = search(key,
            next, parentpp, lt);
        return result;
      }
    }
```

C Code: Fine-Grained Search II

```
} else {
   if ((next = parent->rchild)
       == 0) {
     result = 0;
   } else {
     lock(lt, &next->lock);
     if (key == next->key) {
       result = next;
     } else {
       pthread_rwlock_unlock(
           &parent->lock);
       result = search(key,
           next, parentpp, lt);
       return result;
     }
   }
 }
 if (parentpp != 0) {
   // parent remains locked
   *parentpp = parent;
 } else
   pthread_rwlock_unlock(
       &parent->lock);
 return result;
}
```


Quiz 5

The search function takes read locks if the purpose of the search is for a query, but takes write locks if the purpose is for an add or a delete. Would it make sense for it always to take read locks until it reaches the target of the search, then take a write lock just for that target?

a) **Yes, since doing so allows more concurrency**

b) **No, it would work, but there would be no increase in concurrency**

c) **No, it would not work**

C Code: Add
with Fine-Grained Synchronization I

```
int add(int key) {
  Node *parent, *target, *newnode;
  pthread_rwlock_wrlock(&head->lock);
  if ((target = search(key, &head, &parent,
       l_write)) != 0) {
    pthread_rwlock_unlock(&target->lock);
    pthread_rwlock_unlock(&parent->lock);
    return 0;
  }
```

C Code: Add
with Fine-Grained Synchronization II

```
    newnode = malloc(sizeof(Node));
    newnode->key = key;
    newnode->lchild = newnode->rchild = 0;
    pthread_rwlock_init(&newnode->lock, 0);
    if (name < parent->name)
      parent->lchild = newnode;
    else
      parent->rchild = newnode;
    pthread_rwlock_unlock(&parent->lock);
    return 1;
}
```

Quiz 6

The add function calls malloc. Could we use the malloc that you'll finish by Wednesday for this, or do we need a different one that's safe for use in multithreaded programs?

a) Since the calling thread has a write lock on the parent of the new node, it's safe to call the standard malloc

b) Even if the calling thread didn't have a write lock on the parent, it would be safe to call the the standard malloc

c) We will need a new malloc, one that's safe for use in multithreaded programs