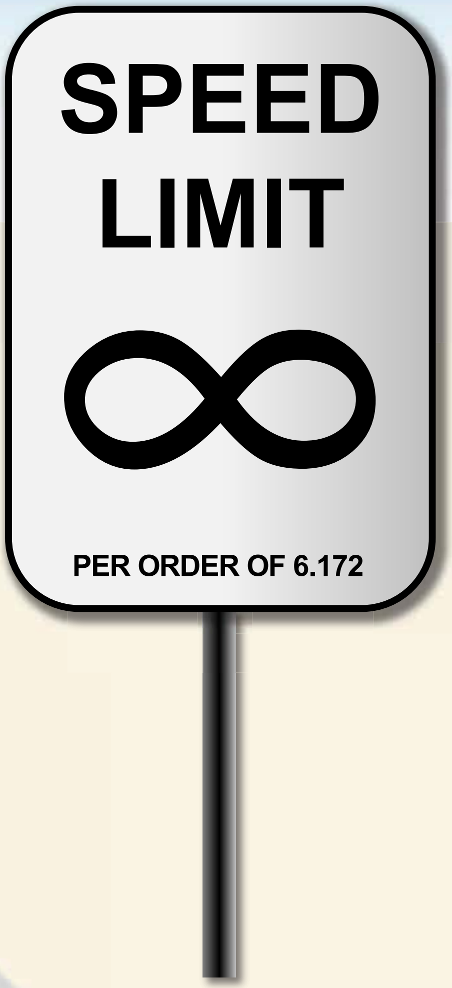
6.172
Performance
Engineering
of Software
Systems

Lecture 20
Speculative Parallelism & Leiserchess

Charles E. Leiserson

SPECULATIVE PARALLELISM



Thresholding a Sum

```
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
  uint sum = 0;
  for (size_t i=0; i<n; ++i) {
    sum += A[i];
  }
  return sum > limit;
}
```

Short-Circuiting

Optimization (Bentley rule)

Quit early if the partial product ever exceeds the threshold.

```
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
  uint sum = 0;
  for (size_t i=0; i<n; ++i) {
    sum += A[i];
    if (sum > limit) return true;
  }
  return false;
}
```

Thresholding a Sum in Parallel

```
#define uint unsigned int

bool sum_exceeds(uint *A, size_t n, uint limit) {
  uint sum;
  CILK_C_REDUCER_OPADD(sum, uint, 0);
  CILK_C_REGISTER_REDUCER(sum);
  cilk_for (size_t i=0; i<n; ++i) {
    REDUCER_VIEW(sum) += A[i];
  }
  CILK_C_UNREGISTER_REDUCER(sum);
  return REDUCER_VIEW(sum) > limit;
}
```

Question

How can we parallelize a short-circuited loop?

Divide-and-Conquer Loop

```
#define uint unsigned int

uint sum_of(uint *A, size_t n) {
  if (n > 1) {
    uint s1 = cilk_spawn sum_of(A, n/2);
    uint s2 = sum_of(A + n/2, n - n/2);
    cilk_sync;
    uint sum = s1 + s2;
    return sum;
  }
  return A[0];
}

bool sum_exceeds(uint *A, size_t n, uint limit) {
  return sum_of(A, n) > limit;
}
```

How might we quit early and save work if the partial sum exceeds the threshold?

Short-Circuiting in Parallel

```
#define uint unsigned int

uint sum_of(uint *A, size_t n, uint limit, bool *abort_flag) {
  if (*abort_flag) return 0;
  if (n > 1) {
    uint s1 = cilk_spawn sum_of(A, n/2, limit, abort_flag);
    uint s2 = sum_of(A + n/2, n - n/2, limit, abort_flag);
    cilk_sync;
    uint sum = s1 + s2;
    if (sum > limit && !*abort_flag) *abort_flag = true;
    return sum;
  }
  return A[0];
}

bool sum_exceeds(uint *A, size_t n, uint limit) {
  bool abort_flag = false;
  return sum_of(A, n, limit, &abort_flag) > limit;
}
```

Short-Circuiting in Parallel

```
#define uint unsigned int

uint sum_of(uint *A, size_t n, uint limit, bool *abort_flag) {
  if (*abort_flag) return 0;
  if (n > 1) {
    uint s1 = cilk_spawn sum_of(A, n/2, limit, abort_flag);
    uint s2 = sum_of(A + n/2, n - n/2, limit, abort_flag);
    cilk_sync;
    uint sum = s1 + s2;
    if (sum > limit &
    return sum;
  }
  return A[0];
}

bool sum_exceeds(uint
  bool abort_flag = f
  return sum_of(A, n,
}
```

Notes:

- Beware: nondeterministic code!
- The benign race on abort_flag can cause true-sharing contention if you are not careful.
- Don't forget to reset abort_flag after use!
- Is a memory fence necessary? *No!*

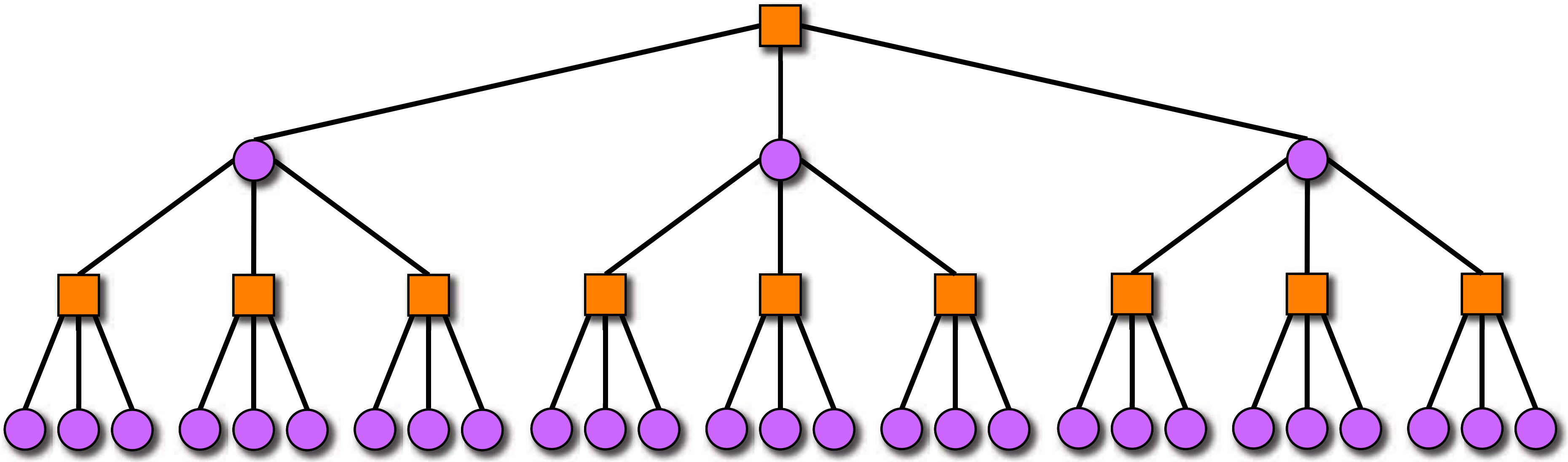
Speculative Parallelism

Definition. Speculative parallelism occurs when a program spawns some parallel work that might not be performed in a serial execution.

Rule of Thumb: Don't spawn speculative work unless there is little other opportunity for parallelism *and* there is a good chance it will be needed.

PARALLEL ALPHA-BETA SEARCH

Review: Alpha-Beta Analysis

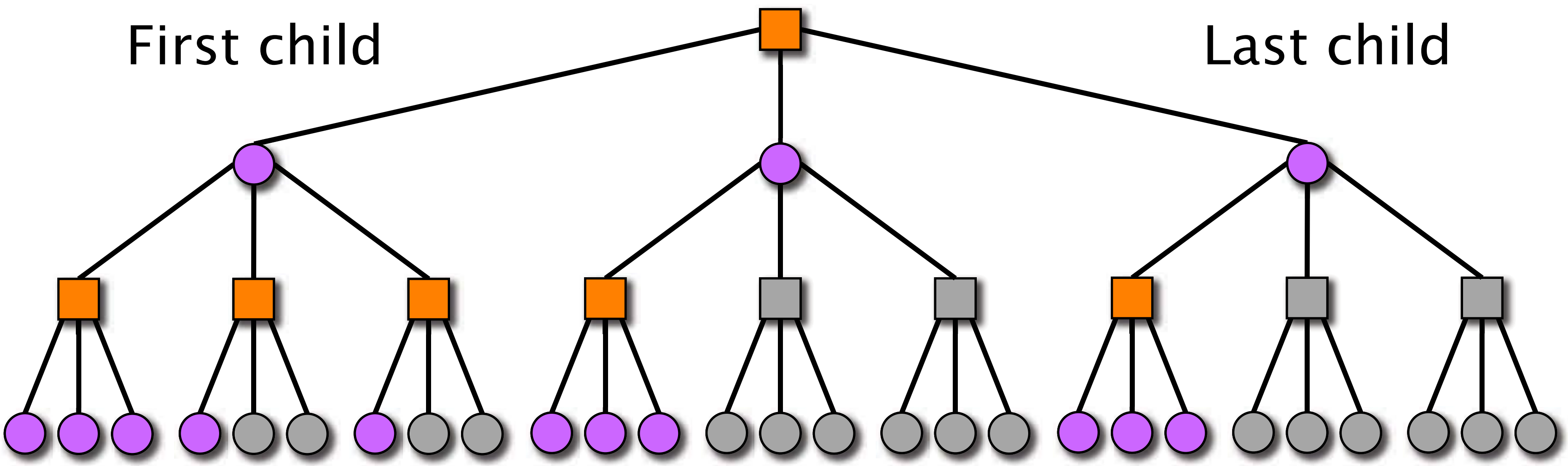


Theorem [KM75]. For a game tree with branching factor b and depth d, an alpha-beta search with moves searched in best–first order examines exactly $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ nodes

Key optimization
Prune the game tree.

The naive algor … s at ply d. For the same work, the search depth is effectively doubled. For the same depth, the work is square-rooted.

Parallel Alpha-Beta



Observation: In a best-ordered tree, the degree of every node is either 1 or maximal.

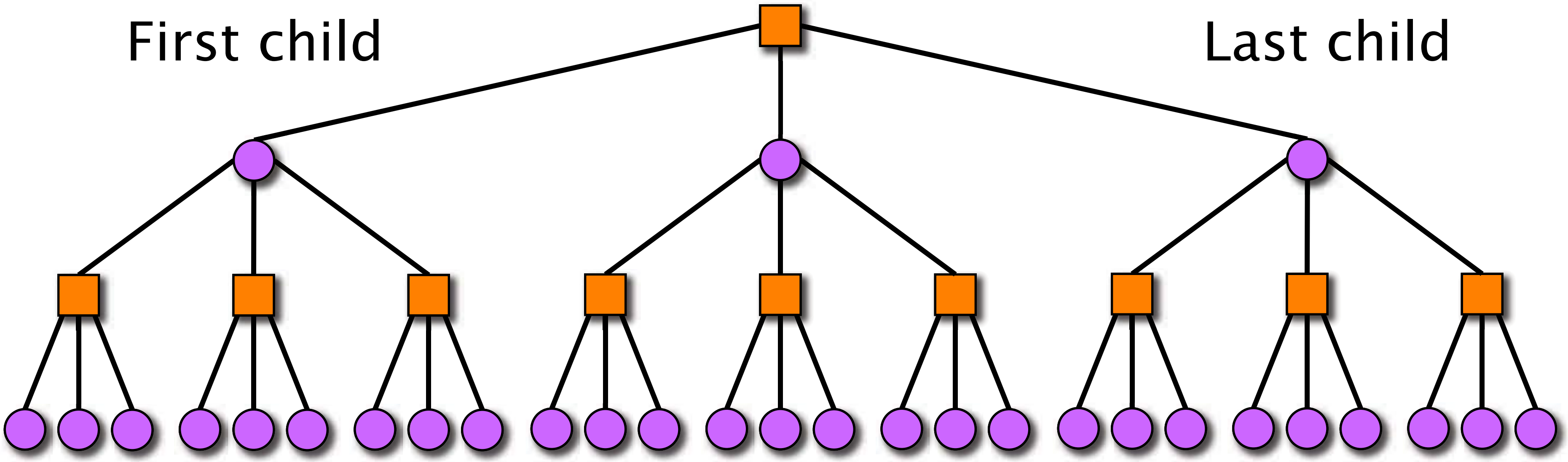
IDEA [FMM91]: If the first child fails to generate a beta cutoff, speculate that the remaining children can be searched in parallel without wasting work: "Young Siblings Wait." Abort subcomputations that prove to be unnecessary.

Abort Mechanism

```
typedef struct searchNode {
  struct searchNode *parent;
  position_t position;
  bool abort_flag;
} searchNode;
```

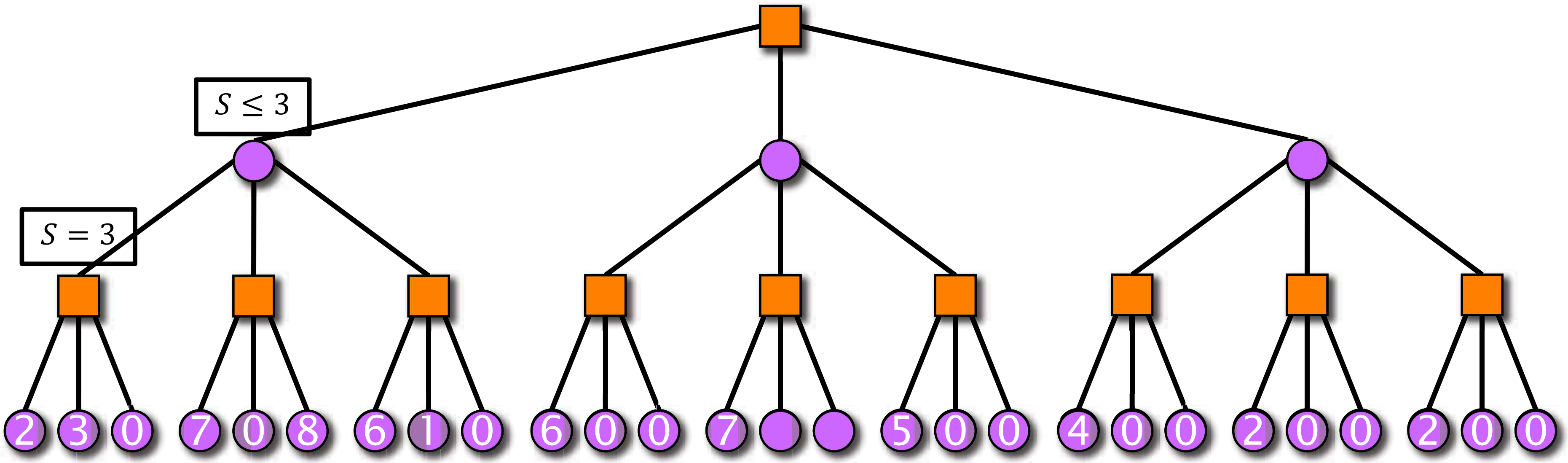
IDEA: Poll up the search tree to see whether any internal node desires an abort.

Problem with Young Siblings Wait

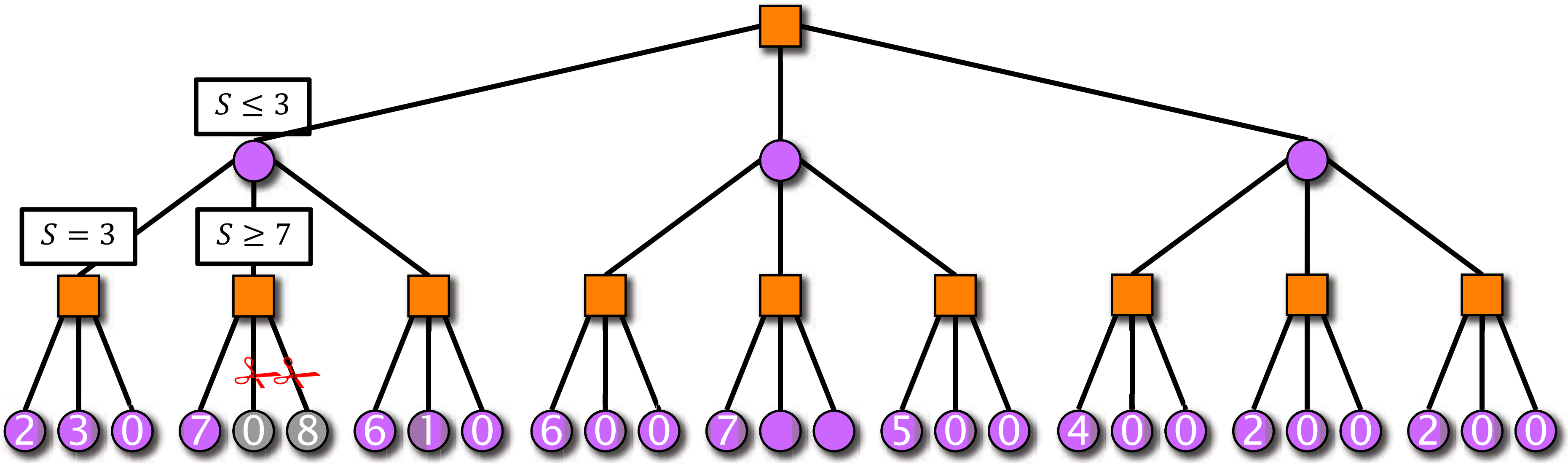


Problem: In general, the game tree is not best-ordered, meaning that parallel alpha-beta search using the "young siblings wait" idea will waste work.

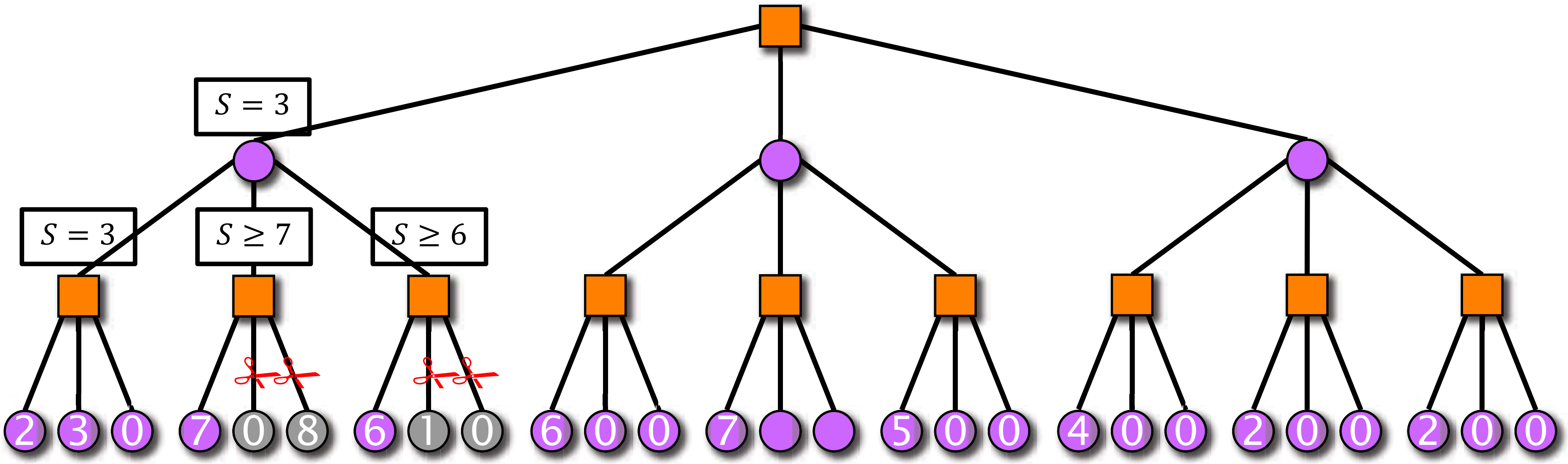
Alpha-Beta Search: Example



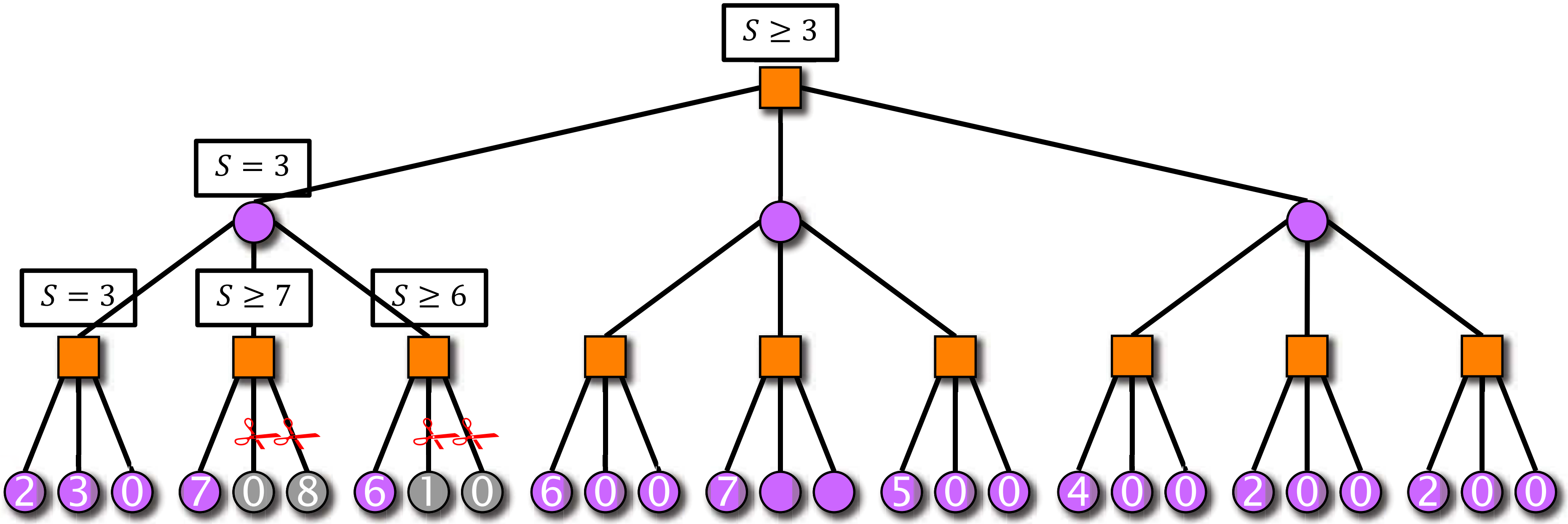
Alpha-Beta Search: Example



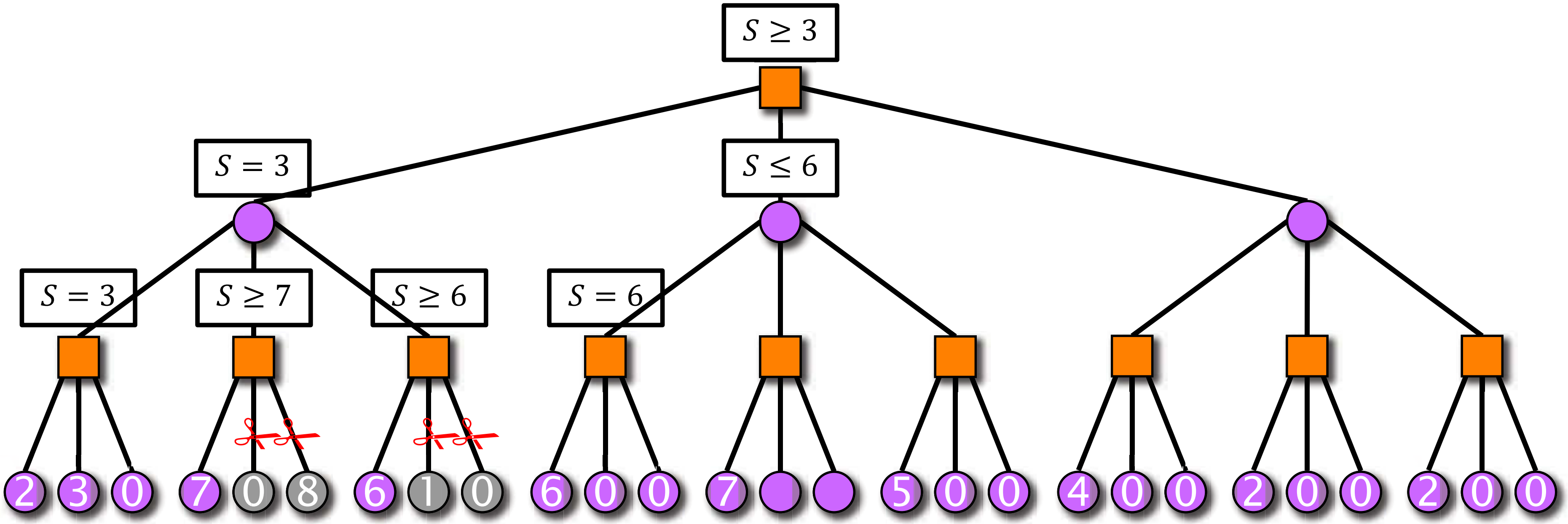
Alpha-Beta Search: Example



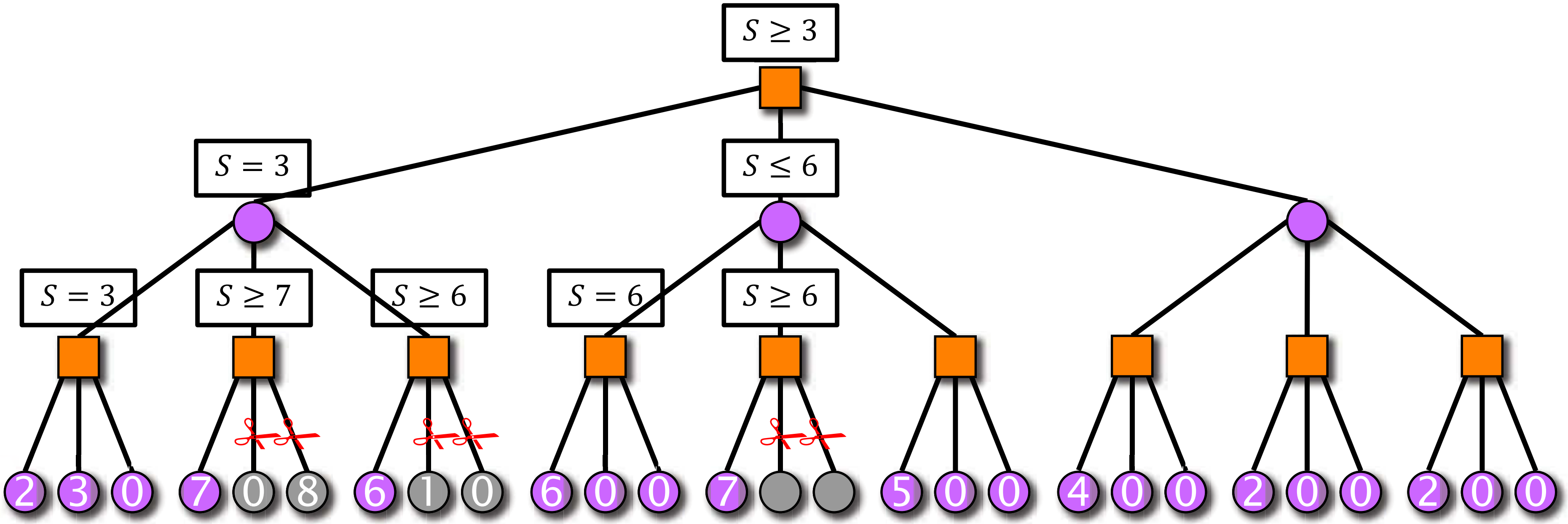
Alpha-Beta Search: Example



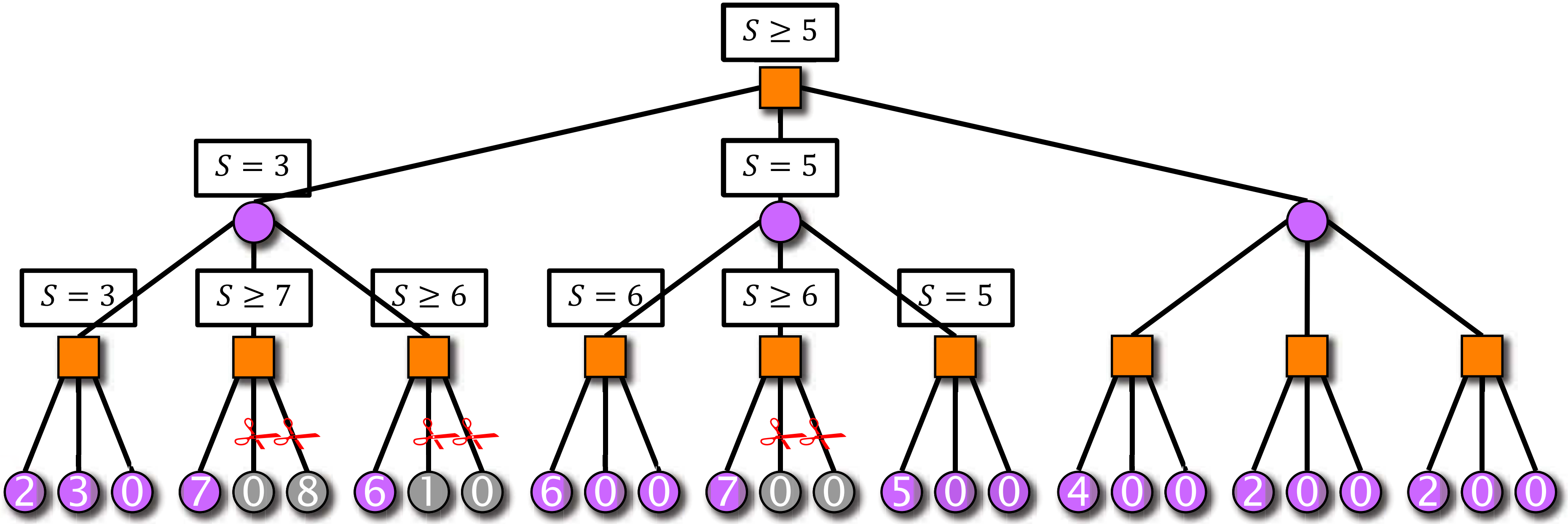
Alpha-Beta Search: Example



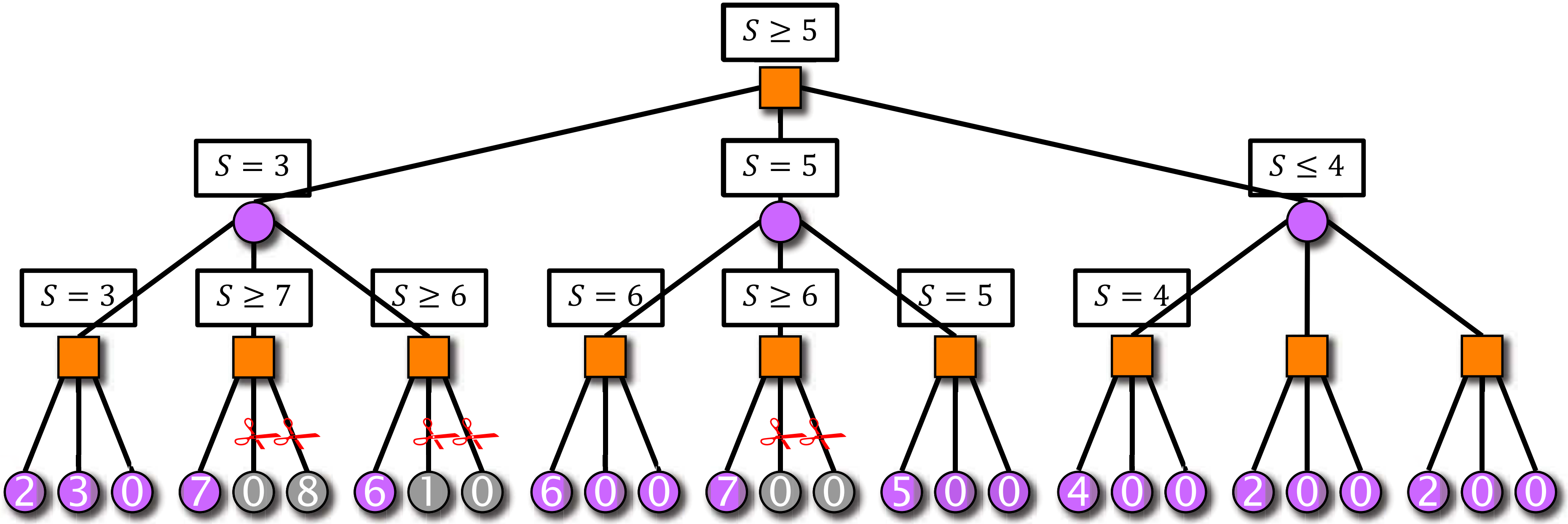
Alpha-Beta Search: Example



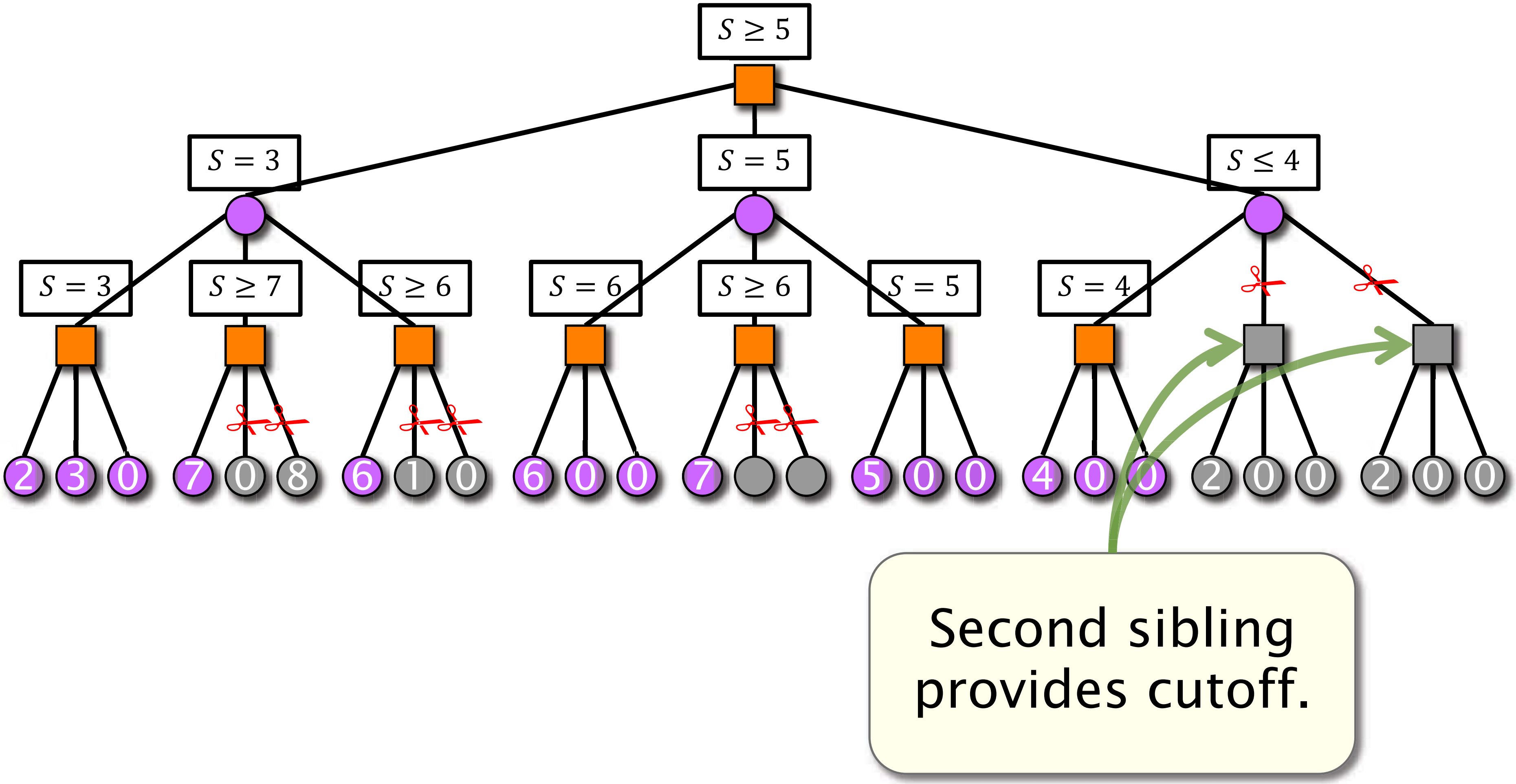
Alpha-Beta Search: Example



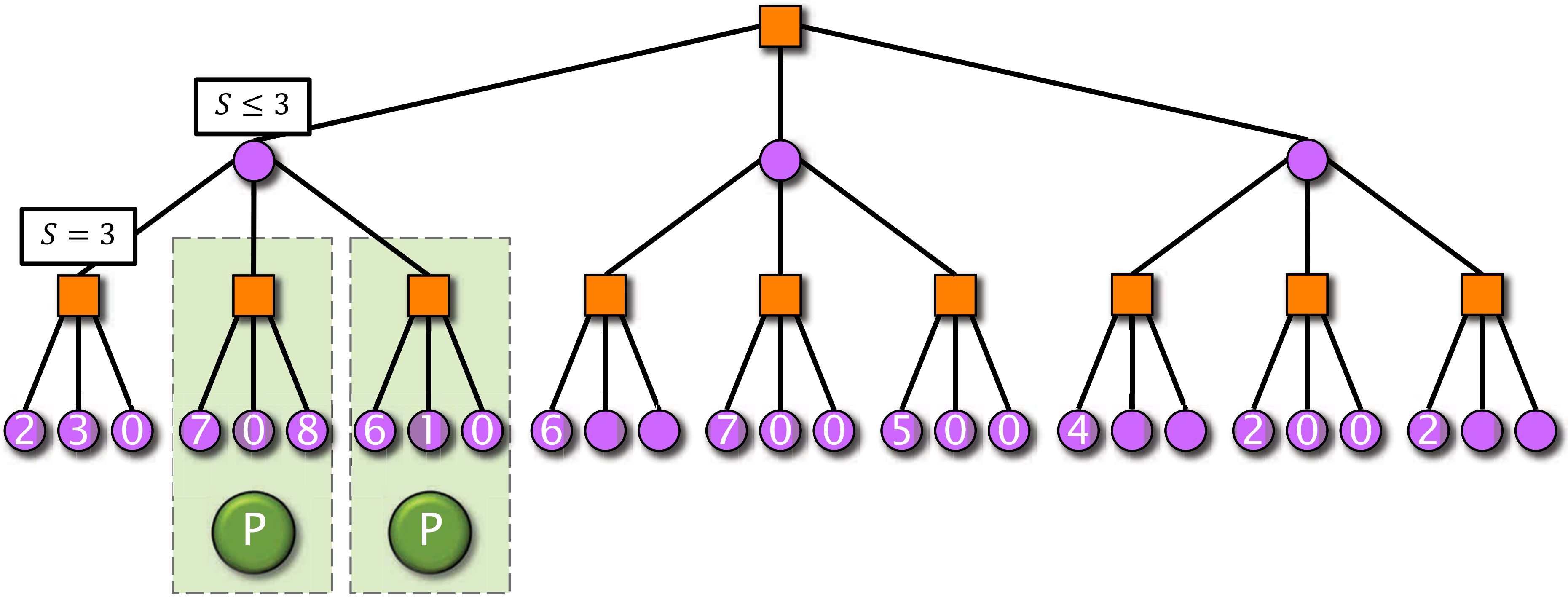
Alpha-Beta Search: Example



Alpha-Beta Search: Example

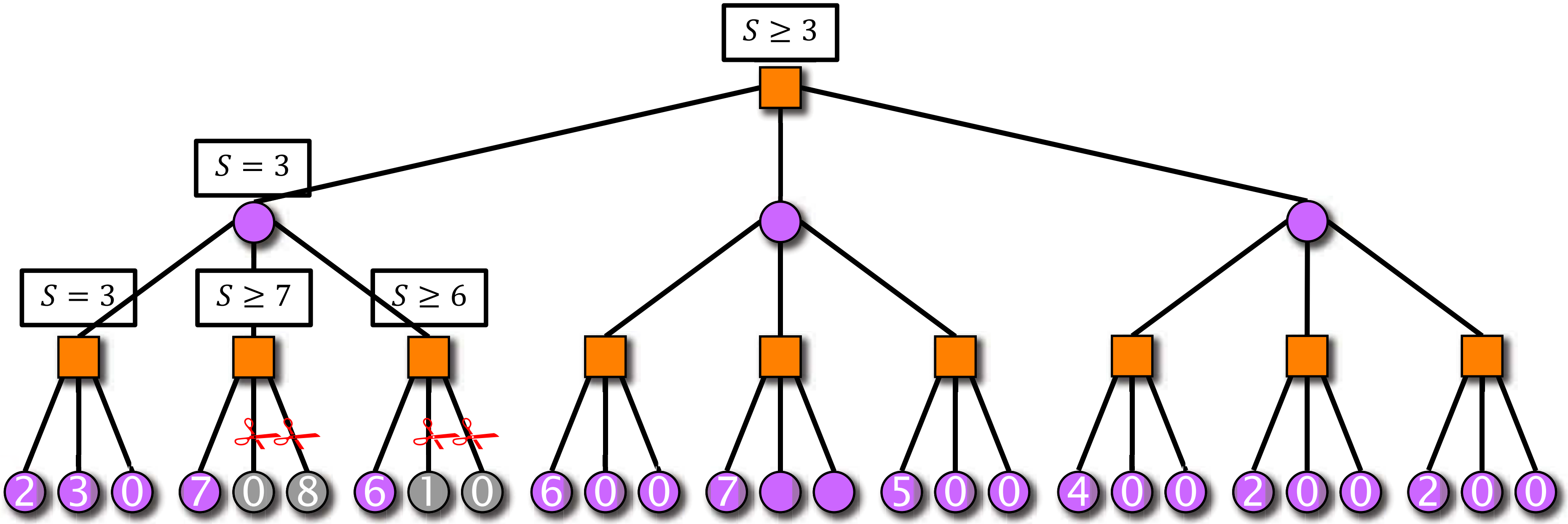


Young Siblings Wait: Example

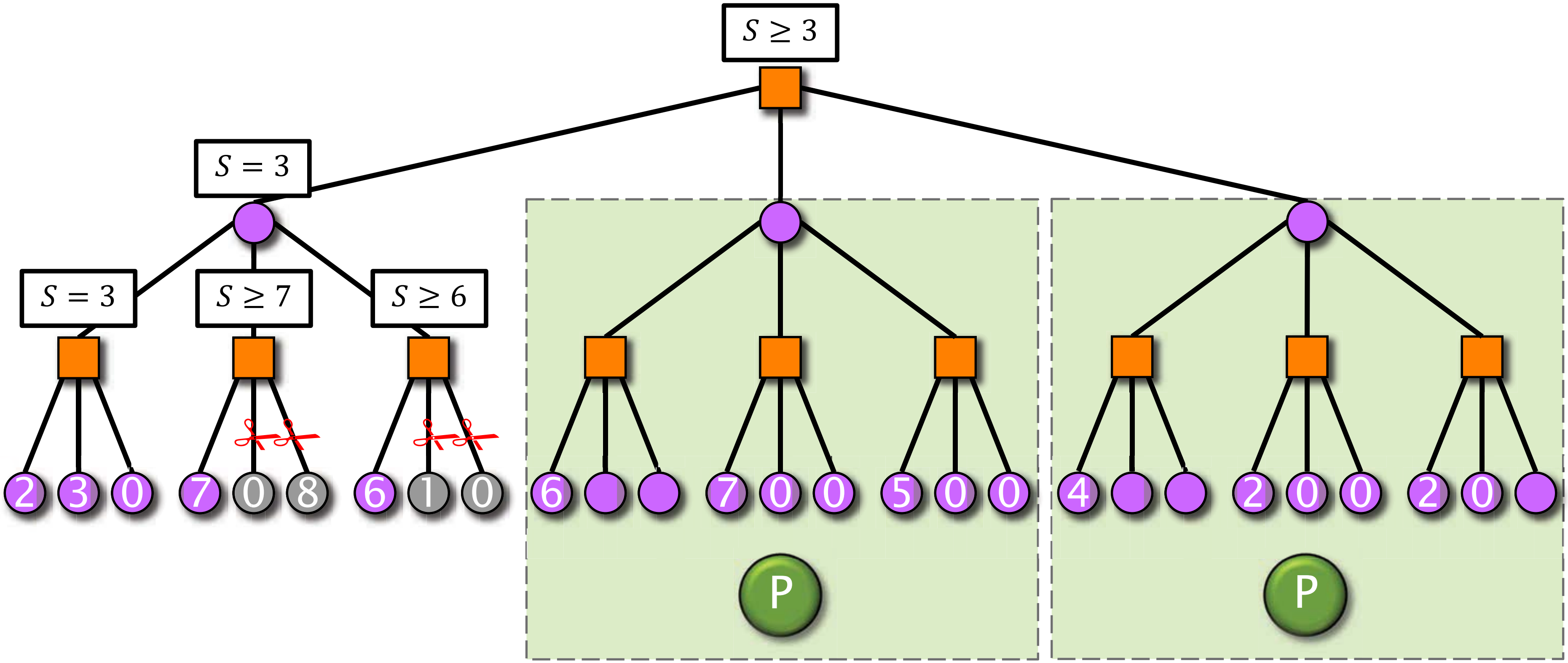


Parallel recursive full-window searches.

Young Siblings Wait: Example

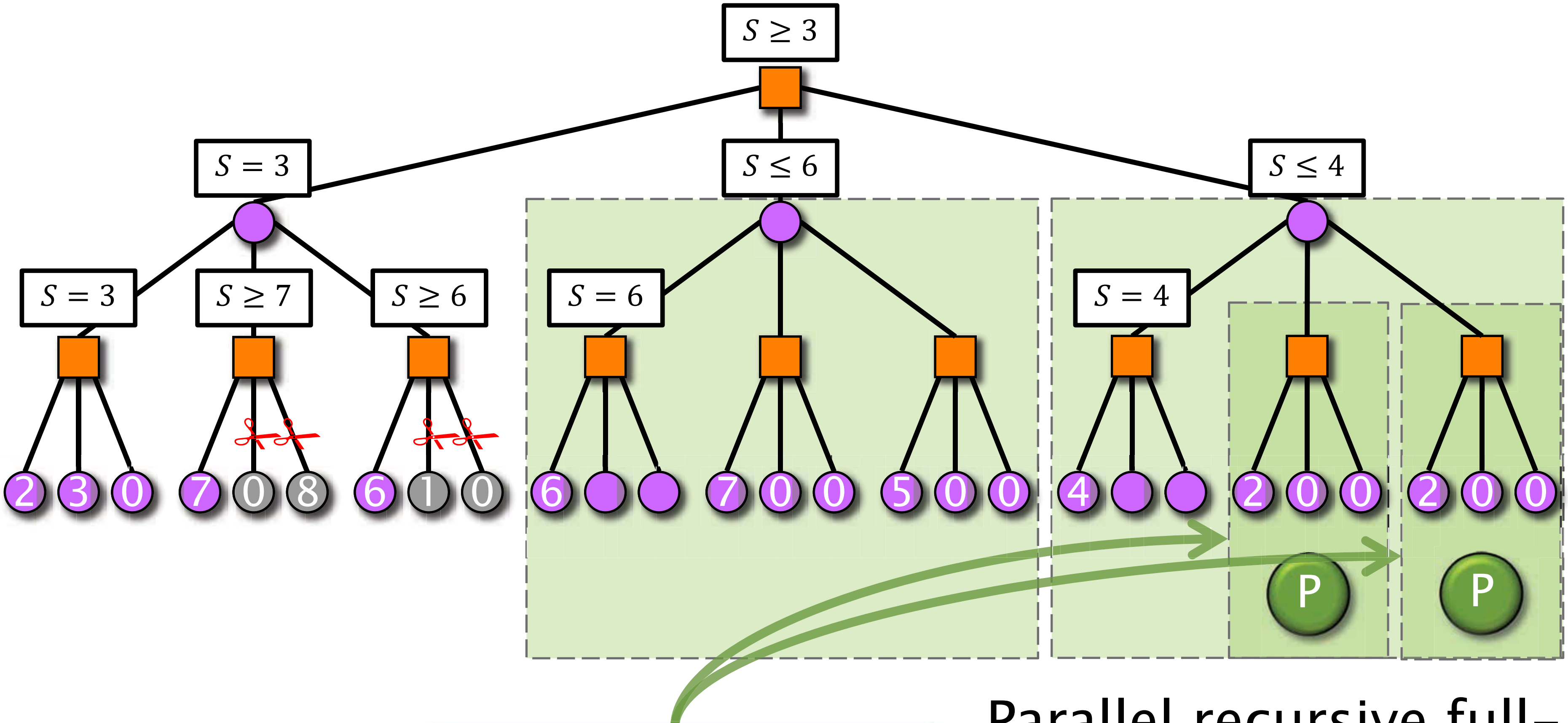


Young Siblings Wait: Example



Parallel recursive full-window searches.

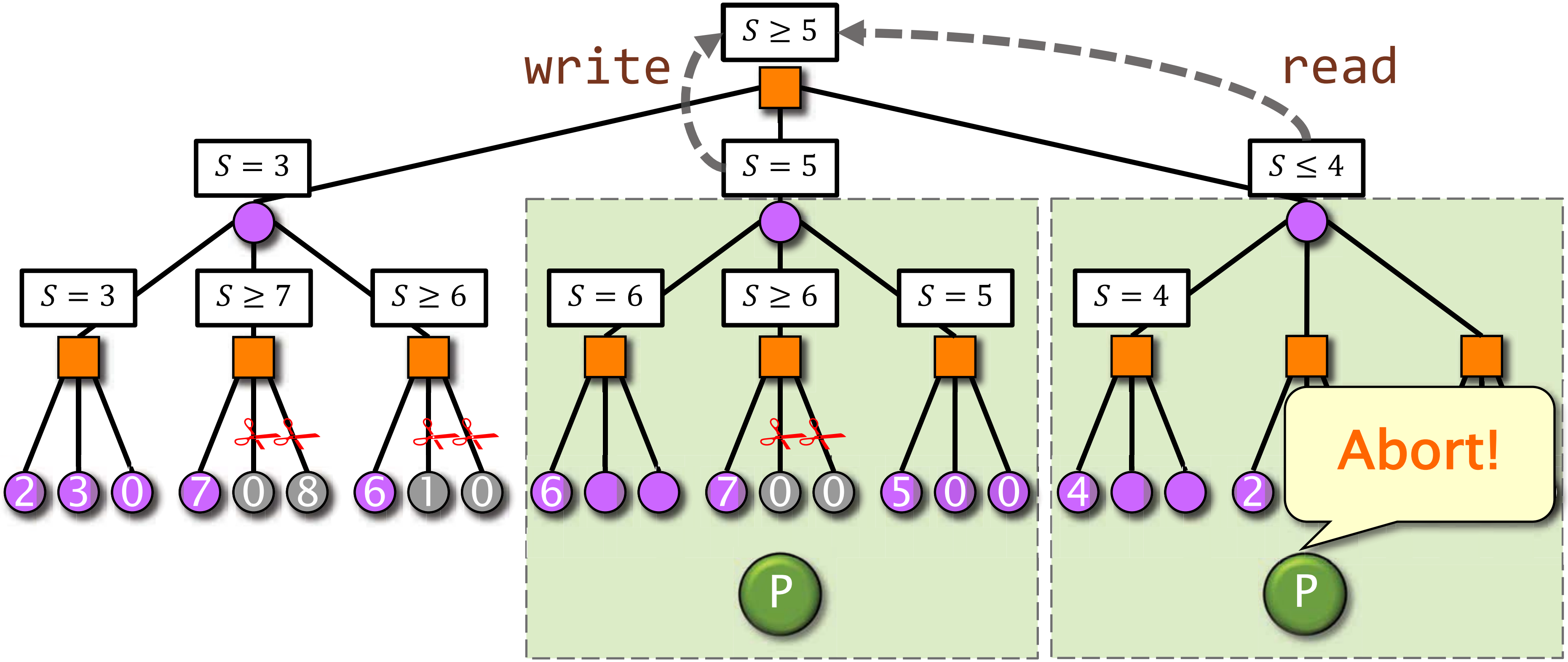
Young Siblings Wait: Example



Cutoff from second child is not available to prune these searches.

Parallel recursive full-window searches.

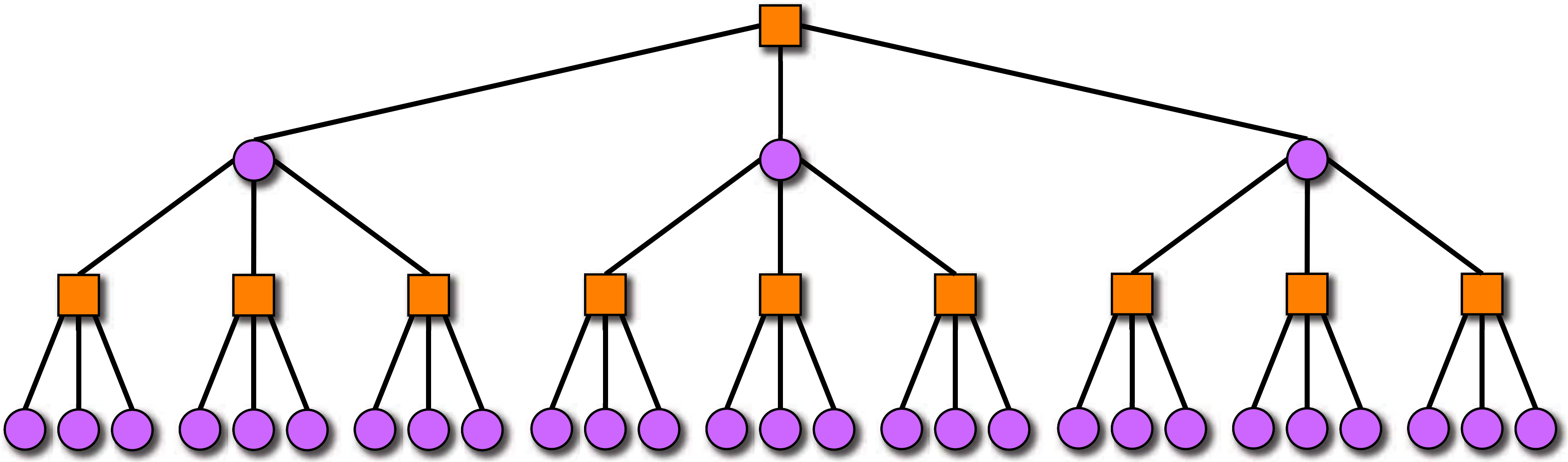
Getting More Aborts



IDEA: Allow children to update parent's alpha/beta value concurrently.

- Children can poll for the alpha/beta value.
- Problem: Difficult to implement efficiently.
- Problem: Efficiency relies on lucky scheduling!

Wasted Work in Parallel Alpha-Beta



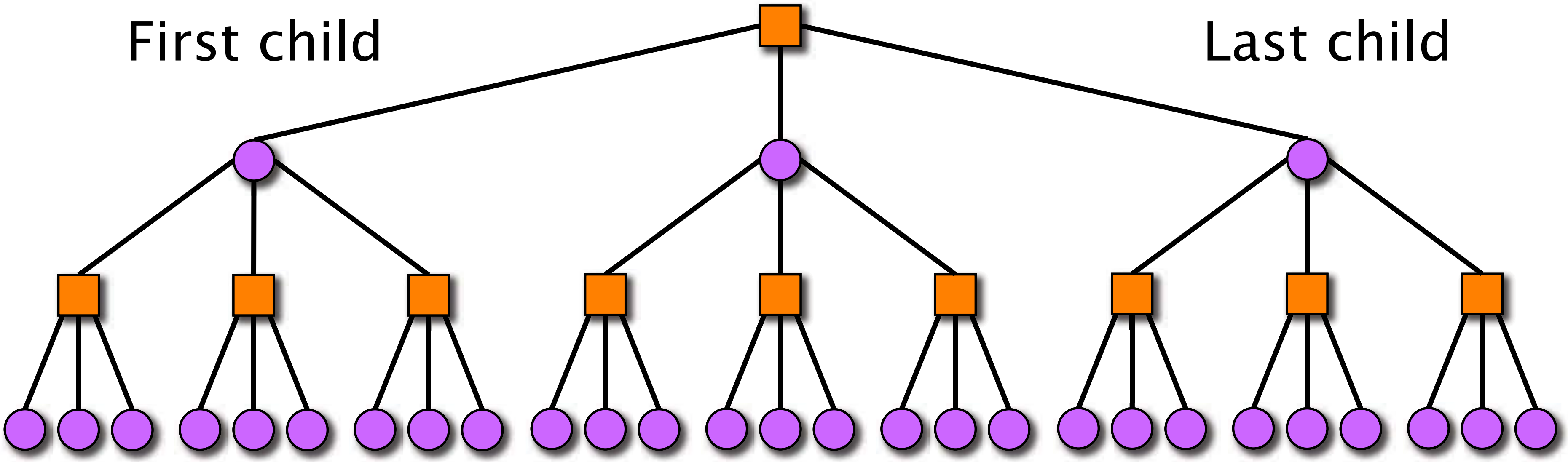
In practice, speculative alpha-beta search of a game tree will always waste some work.

Aim to balance two conflicting goals:

- Generate enough parallel work to get parallel speedup.
- Don't do too much unnecessary work.

JAMBOREE SEARCH

Jamboree Search

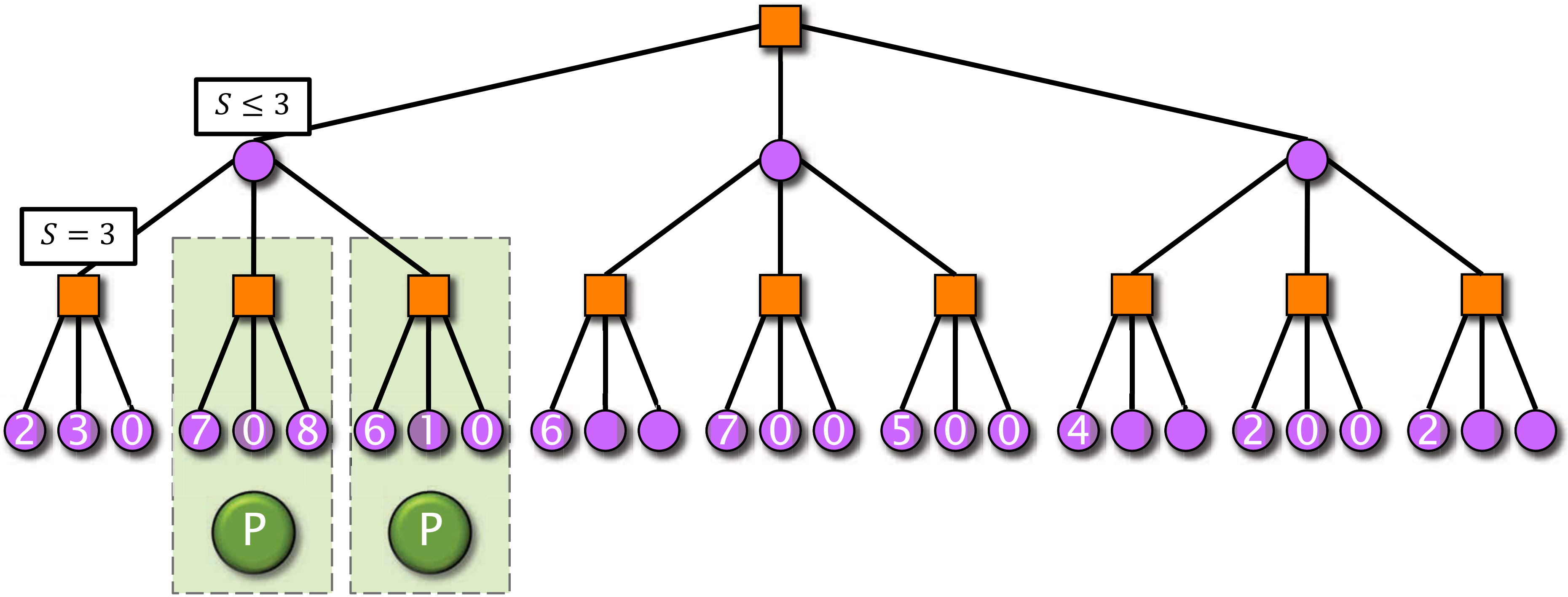


IDEA [K94]: After searching the first child, perform a scout search of the remaining children in parallel, and sequentially value any tests that fail.

- In other words, do `searchPV` serially, and do `scout-search` in parallel.

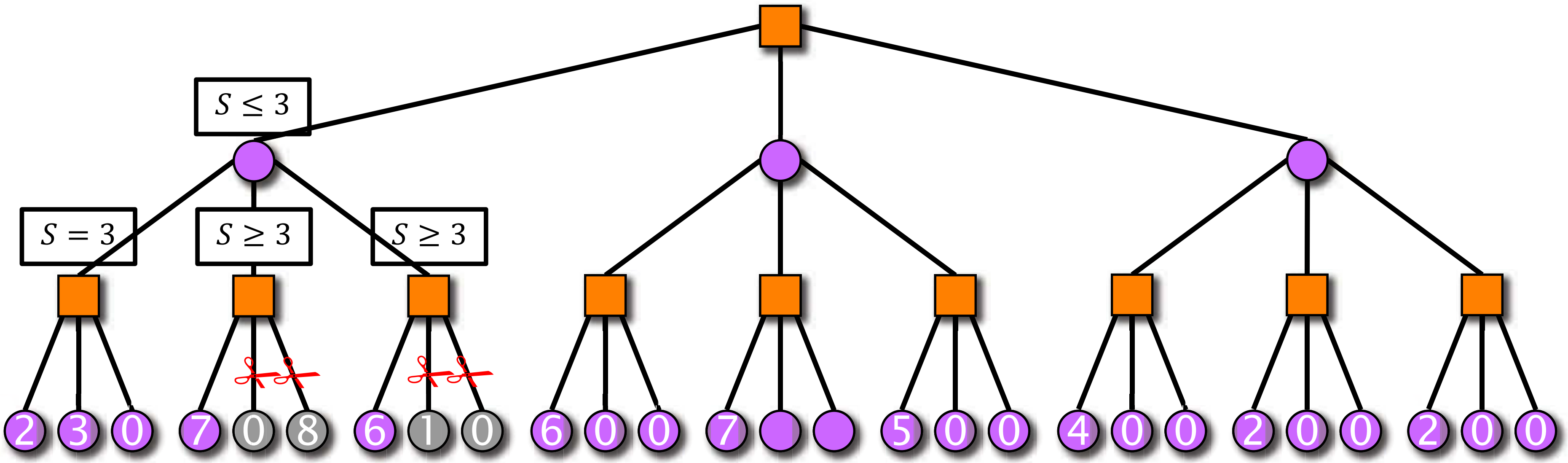
Intuition: It's fine to waste work on a zero-window search, but not on a full-window search.

Jamboree Search: Example

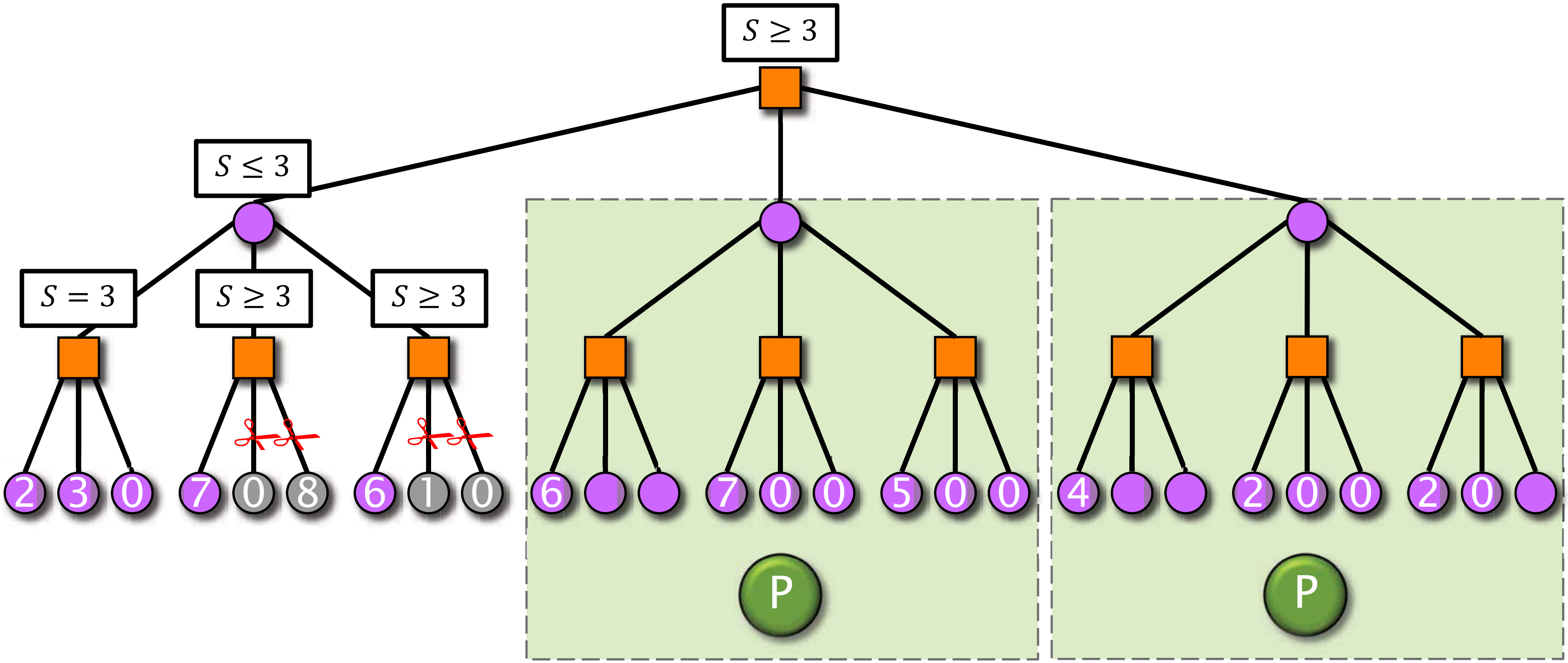


Recursive zero-window search for $S \geq 3$.

Jamboree Search: Example

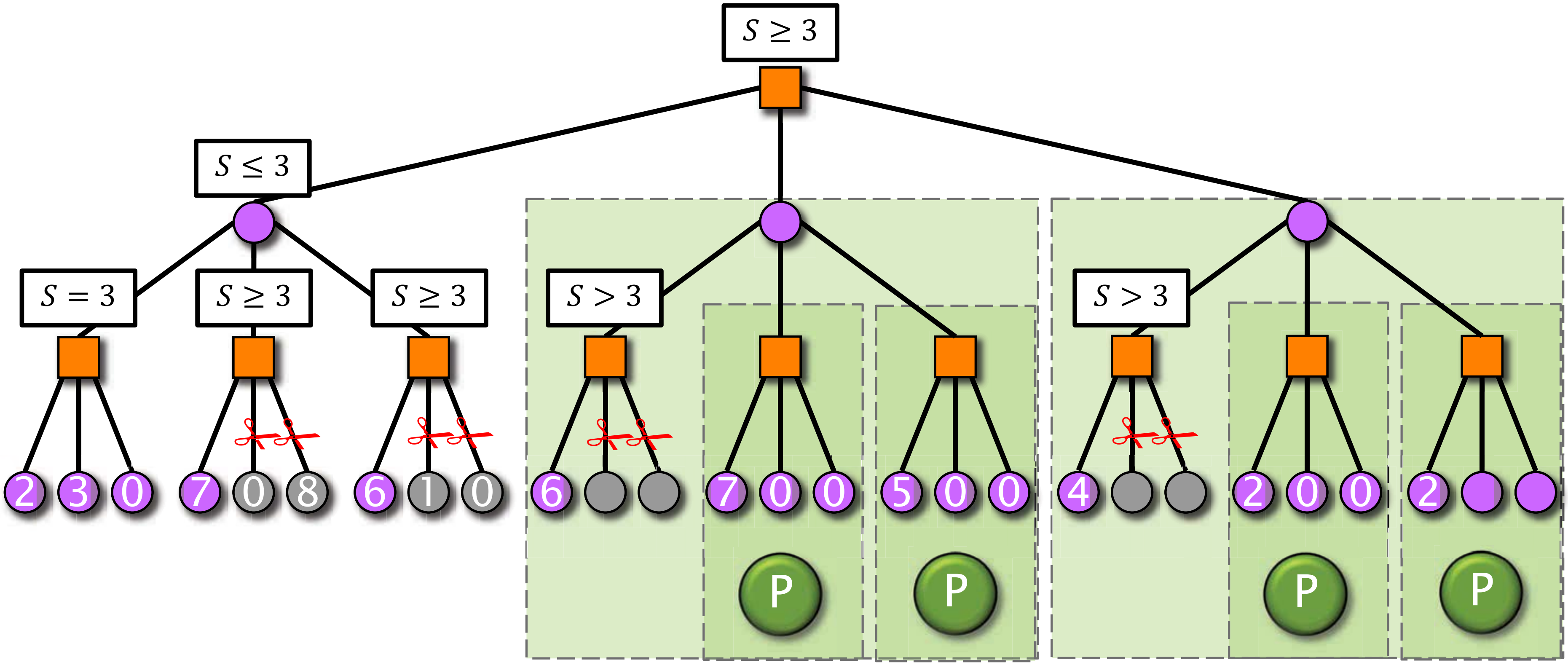


Jamboree Search: Example



Recursive zero–window search for $S \geq 3$.

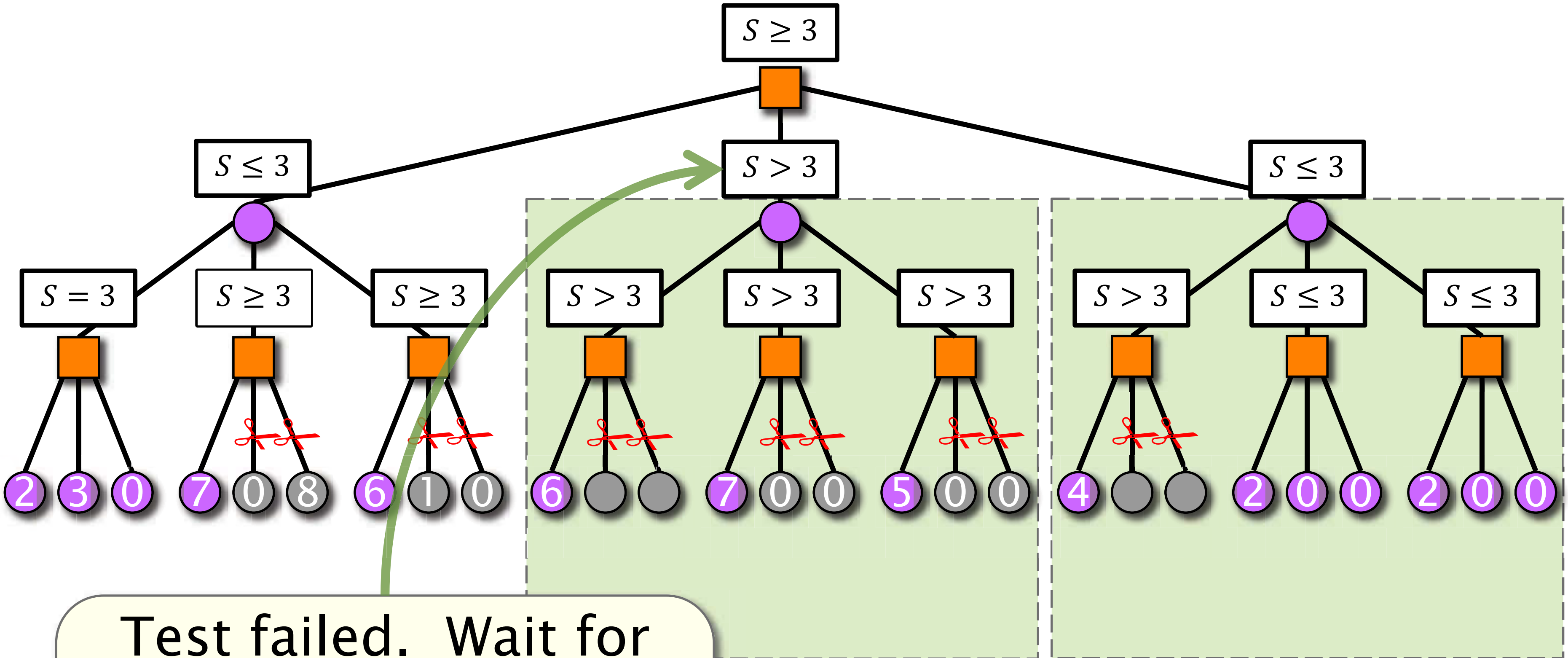
Jamboree Search: Example



Recursive zero-window search for $S \geq 3$.

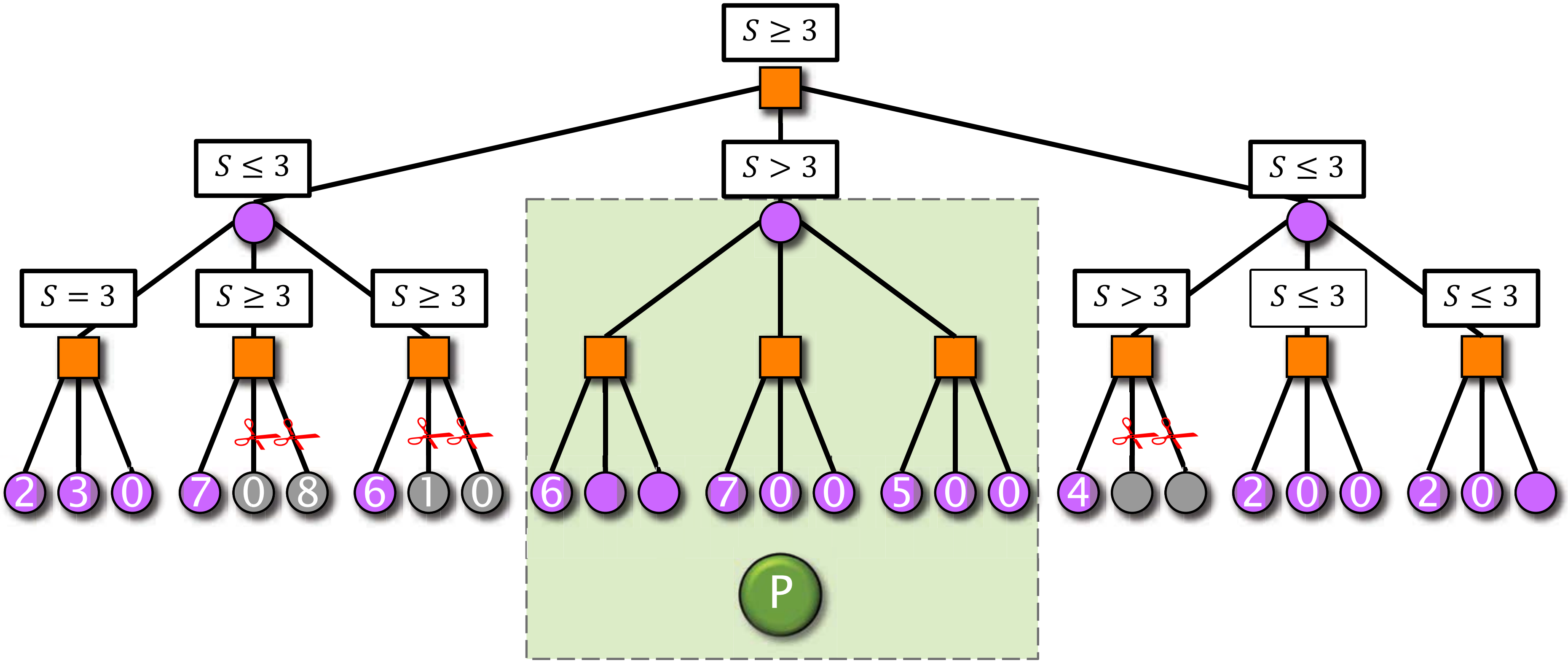
Recursive zero-window search for $S \geq 3$.

Jamboree Search: Example

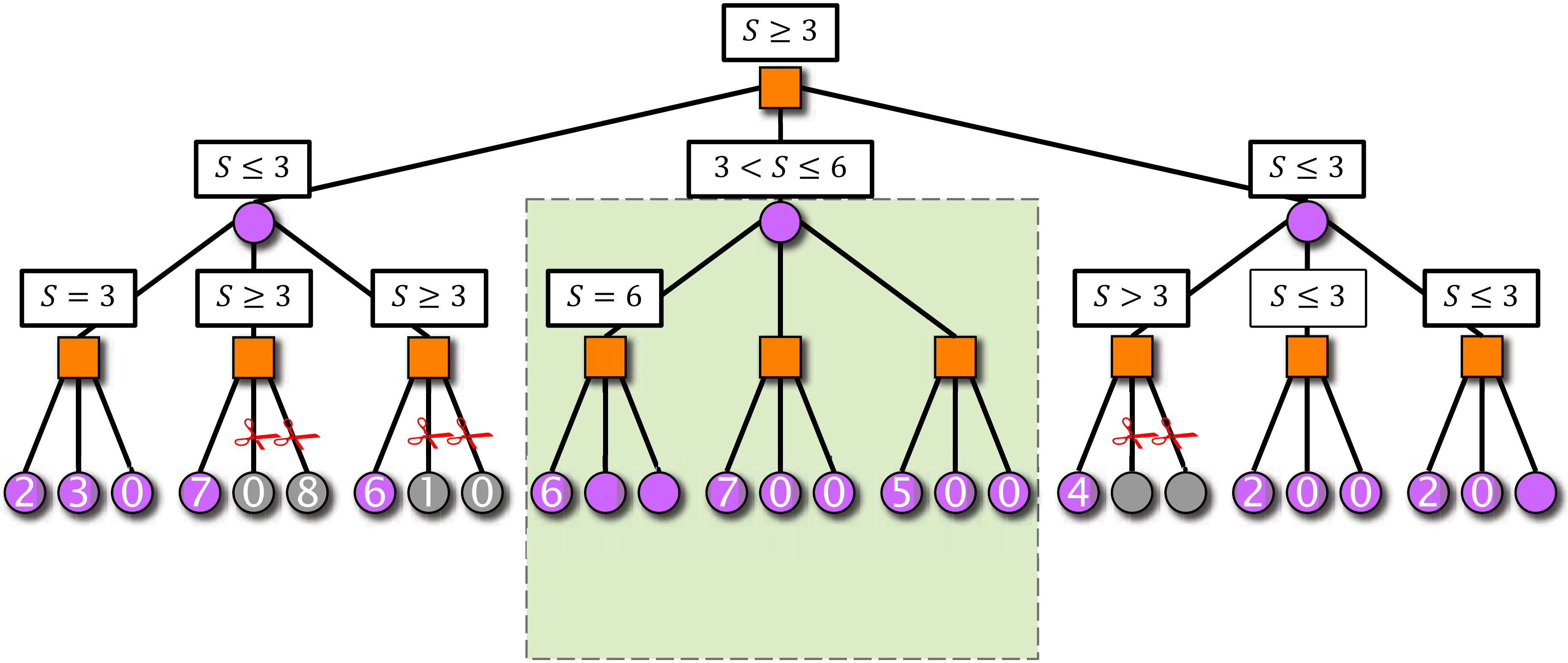


Test failed. Wait for preceding children to finish, then recursively value this tree.

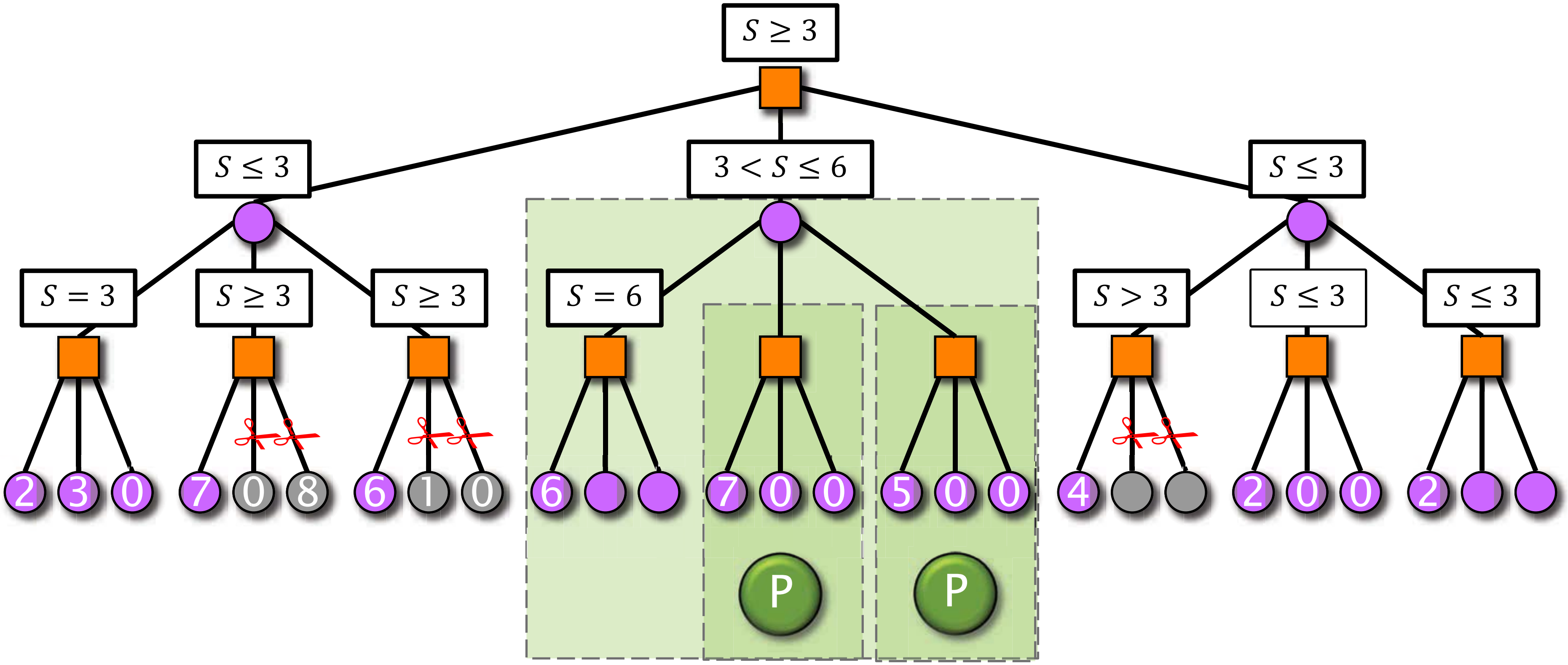
Jamboree Search: Example



Jamboree Search: Example

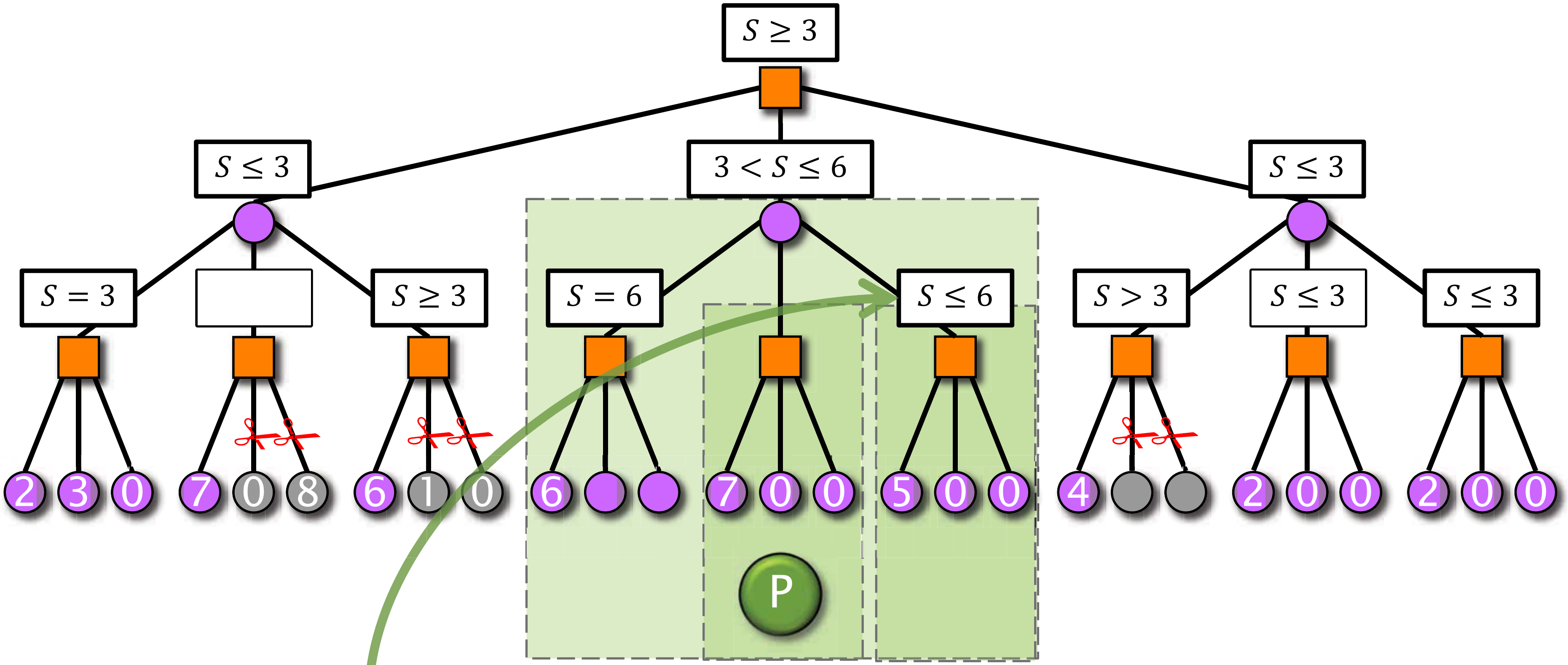


Jamboree Search: Example



Recursive zero–window search for $S \geq 6$.

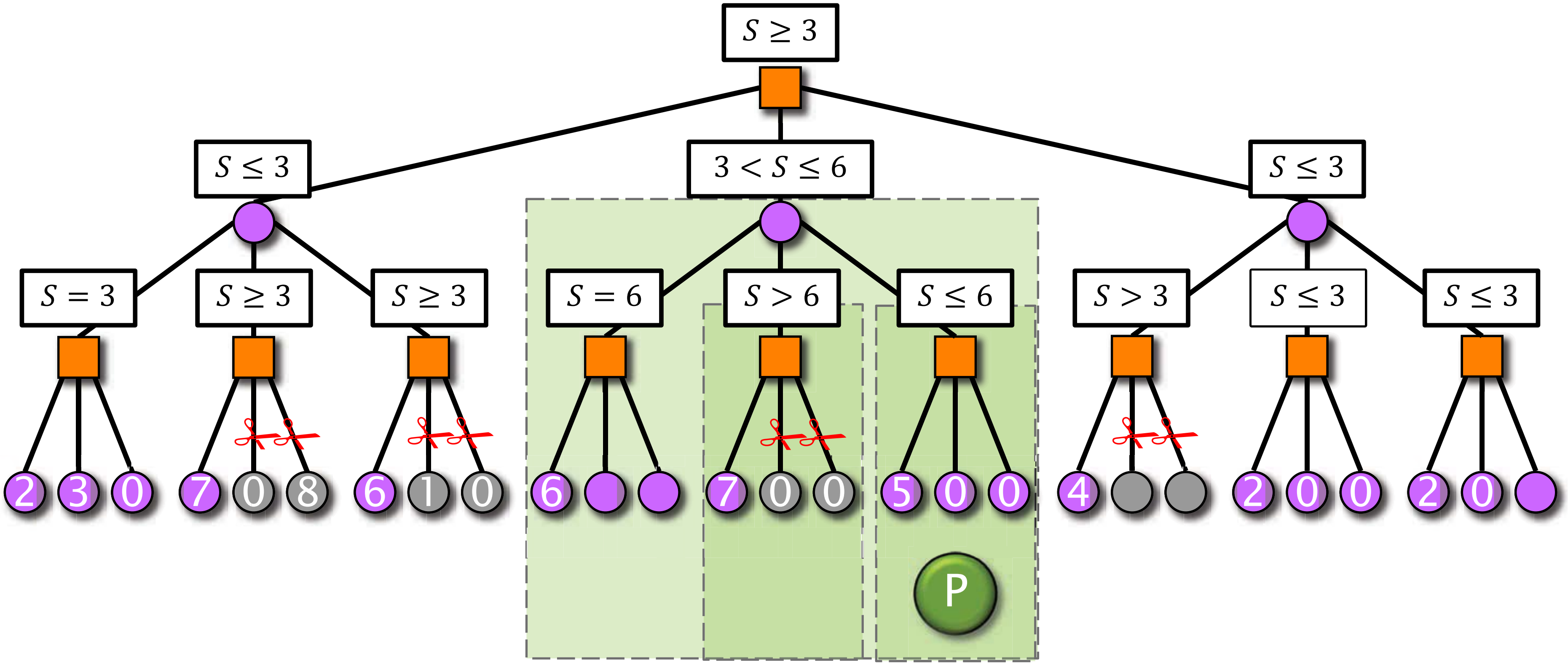
Jamboree Search: Example



Test failed. Wait for preceding children to finish, then recursively value this tree.

Recursive zero-window search for $S \geq 6$.

Jamboree Search: Example



Recursive full-width search.

Jamboree Search Pseudocode [K94]

```
JAMBOREE(n,α,β)
  if n is a leaf then return STATICEVAL(n)
  {c_0,c_1,…,c_k} = Children(n)
  b = -JAMBOREE(c0,-β,-α)
  if b ≥ β then return b
  if b > α then α = b
  parallel_for (c_i in {c_1,c_2,…,c_k})
    s = -JAMBOREE(c_i,-α-1,-α)
    if s > b then b = s
    if s ≥ β then abort-and-return s
    if s > α then
      wait for completion of all c_j where j < i
      s = -JAMBOREE(c_i,-β,-α)
      if s ≥ β then abort-and-return s
      if s > α then α = s
      if s > b then b = s
  return b
```

Full-window search of first child.

Parallel zero-window searches.

Abort all siblings and return.

Why?

Full-window search on failure.

Getting Started with Parallel Leiserchess

The Leiserchess codebase is already structured to support a simple parallelization of `scout_search`.

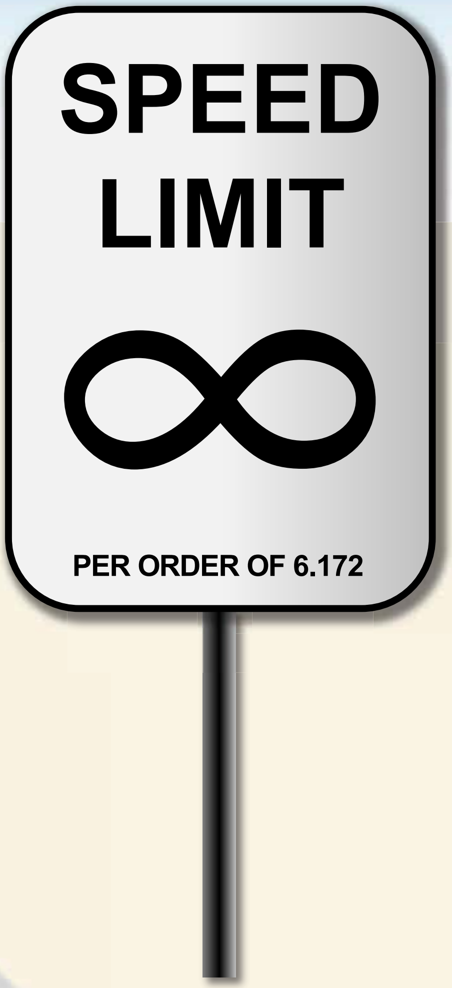
`scout_search.c`

```
static score_t scout_search(searchNode* node, int depth,
                            uint64_t* node_count_serial) {
  …
  cilk_for (int mv_index = 0; mv_index < num_of_moves;
            mv_index++) {
    // Get the next move from the move list.
    int local_index = number_of_moves_evaluated++;
    move_t mv = get_move(move_list[local_index]);
    …
  }
  …
}
```

Resulting search is not the same as Jamboree search, but it's enough to get you started.

Tips for Parallelizing Leiserchess

- Simply parallelizing the loop will produce code with races! Consider how you can address them:
  - Synchronize concurrent accesses, e.g., using locks.
  - Make a thread-local copy when a computation is stolen.
  - Use a thread-local data structure, but don't copy data between threads.
  - Decide the race is benign and leave it be.
- Avoid generating too much wasted work.
  - Duplicate the loop over the moves in scout_search, and make one copy parallel.
  - Switch from the serial loop to the parallel loop when the number of legal moves is high enough.



Computer-Chess Programs

Opening Book

- Precompute best moves at the beginning of the game.
- The [KM75] theorem implies that it is cheaper to keep separate opening books for each side than to keep one opening book for both.

Iterative Deepening

- Rather than searching the game tree to a given depth d, search it successively to depths 1, 2, 3, …, d.
- With each search, the work grows exponentially, and thus the total work is only a constant factor more than searching depth d alone.
- During the search for depth k, keep move-ordering information to improve the effectiveness of alpha-beta during search k+1.
- Good mechanism for time control.

Endgame Database

IDEA: If there are few enough pieces on the board, precompute the outcomes and store them in a database.

- It doesn't suffice to store just win, loss, or draw for a position.
- Keep the distance to mate to avoid cycling.

Quiescence Search

- Evaluating at a fixed depth can leave a board position in the middle of a capture exchange.
- At a "leaf" node, continue the search using only captures — quiet the position.
- Each side has the option of "standing pat."

Null-Move Pruning

- In most positions, there is always something better to do than nothing.
- Forfeit the current player's move (illegal in chess), and search to a shallower depth.
- If a beta cutoff is generated, assume that a full-depth search would have also generated the cutoff.
- Otherwise, perform a full-depth search of the moves.
- Watch out for zugzwang!

Other Search Heuristics

- Killers
  - The same good move at a given depth tends to generate cutoffs elsewhere in the tree.
- Move extensions — grant an extra ply to the search if
  - the King is in check,
  - certain captures,
  - singular (forced) moves.

Transposition Table

- The search tree is actually a dag!
- If you've searched a position to a given depth before, memoize it in a hash table (actually a cache), and don't search it again.
- Store the best move from the position to improve alpha-beta and minimize wasted work in parallel alpha-beta.
- Tradeoff between how much information to keep per entry and the number of entries.

Zobrist Hashing

- For each square on the board and each different state of a square, generate a random string.
- The hash of a board position is the XOR of the random strings corresponding to the states of the squares.
- Because XOR is its own inverse, the hash of the position after a move can be accomplished incrementally by a few XOR's, rather than by computing the entire hash function from scratch.

Transposition-Table Records

- Zobrist key
- Score
- Move
- Quality (depth searched)
- Bound type (upper, lower, or exact)
- Age

Typical Move Ordering

1. Transposition-table move
2. Internal iterative deepening
3. Nonlosing capture in MVV-LVA (most valuable victim, least valuable aggressor) order
4. Killers
5. Losing captures
6. History heuristic

Late-Move Reductions (LMR)

Observation

With a good move ordering, a beta cutoff will either occur right away or not at all.

Strategy

- Search first few moves normally.
- Reduce depth for later moves.

Board Representation

Bitboards

- Use a 64-bit word to represent, for example, where all the pawns are on the 64 squares of the board.
- Use `POPCOUNT` and other bit tricks to do move generation and to implement other chess concepts.

More Good Stuff

https://www.chessprogramming.org/

6.172 Performance Engineering of Software Systems
Fall 2018