```
import math
from typing import TypeVar, Generic, List, Optional

T = TypeVar('T')

class Heap(Generic[T]):
    """
    A generic heap implementation that supports arbitrary number of children per node.
    The heap maintains the max-heap property where parent nodes are greater than their children.
    """

    def __init__(self, child_count: int):
        """
        Initialize the heap with specified number of children per node.

        Args:
            child_count: Number of children each node can have (must be a power of 2)

        Raises:
            ValueError: If child_count is not valid (<=0 or not a power of 2)
        """
        self._validate_child_count(child_count)
        self.child_count = child_count
        self.data: List[T] = []

    def _validate_child_count(self, child_count: int) -> None:
        """
        Validate that the child count is positive and a power of 2.

        Args:
            child_count: Number to validate

        Raises:
            ValueError: If child_count is invalid
        """
        # Ensure child_count is greater than zero
        if child_count <= 0:
            raise ValueError("child_count must be greater than zero")

        # Ensure child_count is a power of 2
        log_child_count = math.log2(child_count)
        if math.ceil(log_child_count) != math.floor(log_child_count):
            raise ValueError("child_count must be a power of 2")
```

```
def insert(self, item: T) -> None:
    """
    Insert an item into the heap and maintain the heap property.

    Args:
        item: Item to insert into the heap
    """
    self.data.append(item)
    item_index = len(self.data) - 1

    # Keep swapping up until we reach the root or heap property is satisfied
    while item_index > 0:
        item_index = self._swap_up(item_index)
        if item_index == -1:
            break

def _swap_up(self, child_index: int) -> int:
    """
    Check a child against its parent and swap if necessary to maintain heap property.

    Args:
        child_index: Index of the child to potentially swap up

    Returns:
        New index of the item after swap, or -1 if no swap occurred
    """
    child_value = self.data[child_index]
    parent_index = math.floor((child_index - 1) / self.child_count)

    if parent_index >= 0:
        parent_value = self.data[parent_index]
        if child_value > parent_value:  # type: ignore
            # Swap the values
            self.data[parent_index] = child_value
            self.data[child_index] = parent_value
            return parent_index

    return -1

def pop_max(self) -> Optional[T]:
    """
    Remove and return the maximum value from the heap.
```

```
        Returns:
            The maximum value in the heap, or None if the heap is empty
        """
        if not self.data:
            return None

        max_item = self.data[0]

        if len(self.data) > 1:
            # Move last item to root and maintain heap property
            self.data[0] = self.data.pop()
            item_index = 0

            # Keep swapping down until leaf node or heap property is satisfied
            while item_index >= 0:
                item_index = self._swap_down(item_index)
        else:
            self.data.pop()

        return max_item

    def _swap_down(self, parent_index: int) -> int:
        """
        Check a parent against all children and swap with the largest if necessary.

        Args:
            parent_index: Index of the parent to potentially swap down

        Returns:
            New index of the item after swap, or -1 if no swap occurred
        """
        parent_value = self.data[parent_index]
        largest_child_index = -1
        largest_child_value = None

        # Find the largest child
        for i in range(self.child_count):
            child_index = self.child_count * parent_index + i + 1
            if child_index < len(self.data):
                child_value = self.data[child_index]
                if largest_child_value is None or child_value > largest_child_value:  # type: ignore
                    largest_child_index = child_index
                    largest_child_value = child_value
        # Perform swap if necessary
        if largest_child_value is not None and parent_value < largest_child_value:  # type: ignore
            self.data[parent_index] = largest_child_value
            self.data[largest_child_index] = parent_value
            return largest_child_index

        return -1

# Example usage:
if __name__ == "__main__":
    # Create a heap with 2 children per node
    heap = Heap[int](2)

    # Insert some numbers
    numbers = [4, 10, 3, 5, 1]
    for num in numbers:
        heap.insert(num)

    # Pop all numbers (will come out in descending order)
    while True:
        max_value = heap.pop_max()
        if max_value is None:
            break
        print(max_value)
```