PA4: Analog Floorplan

Chapter 1

Assignment report

1.1 How to Compile and Execute Your Program

The program is compiled using the provided `Makefile`. The execution requires specifying the input and output file paths via command-line flags.

1.1.1 Compilation

To compile the program, navigate to the source directory in the terminal and simply run the `make` command. This will use the `g++` compiler with `-std=c++11` and `-O3` optimization flags to create an executable file named `floorplanner`.

```
$ make
g++ -std=c++11 -O3 -Wall -o floorplanner 110502529_PA4.cpp
```

Listing 1.1 Compilation Command

1.1.2 Execution

The program is executed from the command line, providing the input block file with the `-i` flag and the desired output file path with the `-o` flag. The provided `Makefile` also includes a convenience target `run` for this purpose.

1.1.2.1 Direct Execution

```
$ ./floorplanner -i <path_to_input.block> -o <path_to_output.out>
# Example:
$ ./floorplanner -i testcases/case1.block -o case1.output
```

Listing 1.2 Direct Execution Command

1.1.2.2 Execution using Makefile

```
$ make run input=<path_to_input.block> output=<path_to_output.out>
# Example:
$ make run input=testcases/case1.block output=case1.output
```

Listing 1.3 Execution using the Makefile 'run' target

1.2 The Completion of the Assignment

All requirements of the assignment have been completed. This includes:

- **B*-Tree Representation:** A B*-Tree is correctly implemented to represent the floorplan, with nodes storing block information, dimensions, and tree pointers (parent, left, right).
- **Packing Algorithm:** A contour-based packing algorithm using Depth-First Search (DFS) is implemented in the `pack()` and `dfs_pack()` methods to determine the coordinates of each block and the total chip dimensions.
- **Simulated Annealing (SA):** A multi-start SA optimization engine is implemented in `run_simulated_annealing()`. It includes a temperature cooling schedule, an acceptance probability function (Metropolis criterion), and adaptive hyperparameters based on problem size.
- **Perturbation Strategy:** A robust perturbation strategy with three distinct move types (Rotate/Resize, Swap, Move) is implemented in `perturb()` to explore the solution space.
- **Cost Function:** A comprehensive cost function is defined in `calculate_cost()`, which is a weighted sum of chip area, aspect ratio penalty, and Integral Non-Linearity (INL). The INL is calculated as specified, involving sorting, cumulative sums, and linear regression.
- **I/O Handling:** The program correctly parses input block files with complex formats using `read_blocks()` and generates the output file in the precise specified format using `write_output()`.
- **Numerical Sorting:** A custom comparator `compareBlockNames()` is implemented to correctly sort module names numerically (e.g., "MM1", "MM2", ..., "MM10"), which is crucial for the INL calculation.

1.3 Perturbation Strategy and Operations

The perturbation strategy is designed to generate a neighboring solution from the current B*-Tree configuration. A random operation is chosen probabilistically to modify the tree. The function `perturb()` selects one of three operations with the following probabilities:

1. **Rotate/Resize Block (Probability $\approx$ 36%):** A random node is selected in the tree. If the block corresponding to this node has multiple predefined dimensions (i.e., different shapes or rotations), one of these dimensions is chosen at random to replace the current one. This changes the width and height of a single block without altering the tree's topology.
2. **Swap two Blocks (Probability $\approx$ 36%):** Two distinct nodes in the tree are chosen randomly. The block information (specifically, `block_id` and `current_dim_idx`) of these two nodes is swapped. This operation changes the placement of two blocks within the existing floorplan topology, but the tree structure itself (the parent-child relationships) remains unchanged.
3. **Move a Node (Probability $\approx$ 28%):** This is the most significant topological change. A random node `u` is selected to be moved, and a different random node `p` is selected as its new parent.
    - **Detach:** Node `u` is first detached from the tree using the `detach()` helper function. This process carefully promotes one of `u`'s children to take its place, ensuring the tree remains valid. If `u` had two children, its right subtree is attached to the rightmost node of its left subtree.
    - **Attach:** Node `u` is then re-inserted into the tree as a new child of node `p`, using the `attach()` helper function. It is randomly attached as either a left or right child. This operation fundamentally alters the relative positions of many blocks.

This mixed strategy ensures a balanced exploration of the solution space by combining small, local changes (rotation, swap) with larger, structural changes (move).

1.4 Cost Function and Acceptance Criteria

The quality of a given floorplan is evaluated by a cost function. The decision to accept a new (potentially worse) solution is governed by the Metropolis criterion within the Simulated Annealing algorithm.

1.4.1 Cost Function

The cost function, implemented in `calculate_cost()`, is a weighted sum of two main components: a combined Area/Aspect Ratio metric and the Integral Non-Linearity (INL).

$$Cost = W_{Area/AR} \cdot Cost_{Area/AR} + W_{INL} \cdot INL$$

Where the weights are set to $W_{Area/AR} = 0.8$ and $W_{INL} = 0.2$.

1. **Area and Aspect Ratio Cost ($Cost_{Area/AR}$):** This component penalizes both large area and undesirable aspect ratios. The aspect ratio (AR) is defined as $AR = \max(\frac{W}{H}, \frac{H}{W})$. A penalty term, $f(AR)$, is applied if the AR is outside the desired range of [0.5, 2.0].

   $$f(AR) = \{ 2 \cdot (0.5 - AR) if AR < 0.5 AR - 2.0 if AR > 2.0 0 otherwise$$

   The combined cost is then: $Cost_{Area/AR} = Area \cdot (1 + f(AR))$.

2. **Integral Non-Linearity (INL):** Calculated in `calculate_inl()`, this term measures the uniformity of the placement of blocks relative to the chip's geometric center. It is the maximum absolute deviation between the cumulative sum of sorted squared distances and its ideal linear regression line.

1.4.2 Acceptance Criteria (Moving to Next State)

The decision to transition from the current state (floorplan) S_{curr} to a new perturbed state S_{next} is made based on the change in cost, $\Delta Cost = Cost(S_{next}) - Cost(S_{curr})$, and the current temperature T. This is the Metropolis criterion:

1. If $\Delta Cost < 0$, the new state is better. The move is always accepted.
2. If $\Delta Cost \geq 0$, the new state is worse. The move is accepted with a probability P:

   $$P(accept) = e^{-\frac{\Delta Cost}{T}}$$

This probabilistic acceptance of worse solutions allows the algorithm to escape local optima, especially at higher temperatures early in the annealing process.

1.5 Reflection on the Assignment Process

Thanks to a clear understanding of the requirements and a solid design strategy from the outset, the implementation of this assignment proceeded smoothly without encountering significant hardness or major obstacles. The overall architecture was planned in advance, which allowed for a methodical and efficient development process.

The implementation was broken down into several manageable components, each addressed systematically:

- **Data Structures:** The B*-Tree and its associated node structures were defined based on the core principles of the algorithm, which made their implementation straightforward.
- **Core Algorithms:** Key algorithms, such as the contour-based packing mechanism and the Simulated Annealing engine, were implemented following well-established patterns. Careful planning of the perturbation operations (Rotate, Swap, Move) and the cost function ensured they integrated seamlessly.
- **Modular Design:** Each function, from file I/O (`read_blocks`) to optimization (`run_simulated_annealing`) and output generation (`write_output`), was designed with a clear and distinct purpose. This modularity prevented complex interdependencies and simplified debugging.

By adopting this structured approach, potential difficulties were preempted, resulting in a robust and correct solution being developed within the expected timeframe. The project served as an excellent exercise in applying algorithmic concepts to a practical VLSI design problem, and the process itself was a valuable and rewarding experience.

1.6 Any Suggestions About This Programming Assignment?

No, I do not have any suggestions. The assignment is well-structured and provides a comprehensive and challenging problem in the domain of VLSI physical design automation. It effectively combines concepts from data structures (B*-Tree), optimization algorithms (Simulated Annealing), and application-specific metrics (INL), making it a valuable learning experience.

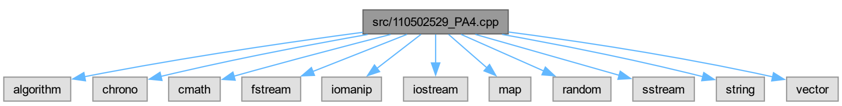
Chapter 2

File Documentation

2.1 src/110502529_PA4.cpp File Reference

```
#include <algorithm>
#include <chrono>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#include <random>
#include <sstream>
#include <string>
#include <vector>
```

Include dependency graph for 110502529_PA4.cpp:



Classes

- struct Dimension

    Stores one possible dimension and layout for a block. A single block can have multiple `Dimension` options, representing different shapes or rotations.
- struct Block

    Represents a fundamental module (or macro) to be placed.
- struct Node

    Represents a node in the B∗-Tree. Each node corresponds to a specific block in the floorplan.
- class Floorplan

    Manages the entire floorplanning process using a B∗-Tree and Simulated Annealing.

Functions

- mt19937 rng (chrono::high_resolution_clock::now().time_since_epoch().count())
  Global random number generator. Seeded with the high-resolution clock for better randomness in each run.
- bool compareBlockNames (const string &a, const string &b)
  Custom string comparison to sort block names numerically (e.g., "MM1", "MM2", ..., "MM10").
- void parse_arguments (int argc, char ∗argv[ ], string &input_file, string &output_file)
  Parses command-line arguments `-i` and `-o`.
- Floorplan run_simulated_annealing (const Floorplan &base_fp, const chrono::seconds &time_limit)
  Executes the multi-start Simulated Annealing (SA) algorithm.
- void print_and_write_results (Floorplan &best_fp, const string &output_file)
  Prints the final results to the console and writes them to the output file.
- int main (int argc, char ∗argv[ ])
  The main entry point of the program.

2.1.1 Function Documentation

2.1.1.1 compareBlockNames()

```
bool compareBlockNames (
            const string & a,
            const string & b)
```

Custom string comparison to sort block names numerically (e.g., "MM1", "MM2", ..., "MM10").

It separates the non-digit prefix from the numeric suffix, compares prefixes first, and then compares the numeric parts if the prefixes are identical. This is crucial for the INL calculation which requires a specific sorted order.

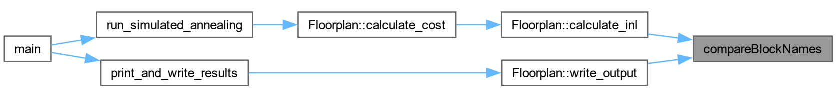
Parameters

a	The first block name string.
b	The second block name string.

Returns

True if `a` should come before `b`, false otherwise.

Here is the caller graph for this function:



2.1.1.2 main()

```
int main (
        int argc,
        char * argv[])
```

The main entry point of the program.

Orchestrates the floorplanning process:

1. Parses command-line arguments.
2. Reads block data from the input file.
3. Runs the simulated annealing optimization within a time limit.
4. Prints and writes the best result found.

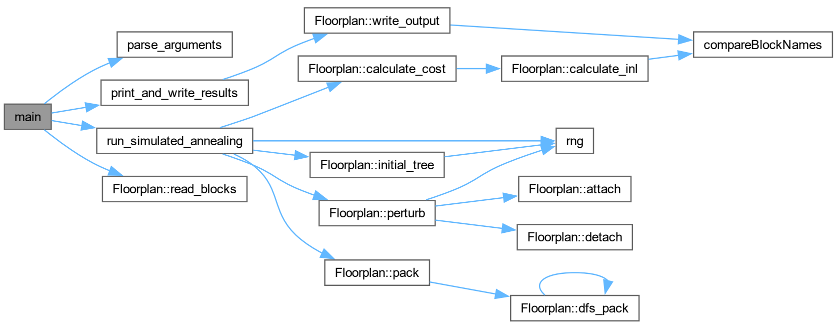
Parameters

argc	Argument count.
argv	Argument vector.

Returns

0 on successful execution.

Here is the call graph for this function:



2.1.1.3 parse_arguments()

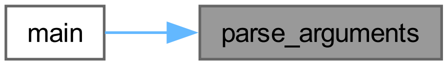
```
void parse_arguments (
        int argc,
        char * argv[],
        string & input_file,
        string & output_file)
```

Parses command-line arguments `-i` and `-o`.

Parameters

argc	Argument count.
argv	Argument vector.
input_file	Reference to a string to store the input file path.
output_file	Reference to a string to store the output file path.

Here is the caller graph for this function:



2.1.1.4 print_and_write_results()

```
void print_and_write_results (
            Floorplan & best_fp,
            const string & output_file)
```

Prints the final results to the console and writes them to the output file.

Parameters

best_fp	The final, best Floorplan object.
output_file	The path to the output file.

Here is the call graph for this function:

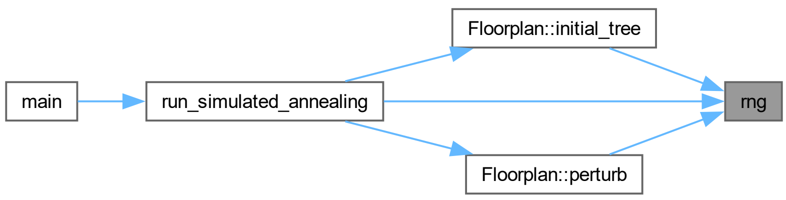
Here is the caller graph for this function:

2.1.1.5 rng()

```
mt19937 rng (
            chrono::high_resolution_clock::now().time_since_epoch().count() )
```

Global random number generator. Seeded with the high-resolution clock for better randomness in each run.

Here is the caller graph for this function:



2.1.1.6 run_simulated_annealing()

```
Floorplan run_simulated_annealing (
            const Floorplan & base_fp,
            const chrono::seconds & time_limit)
```

Executes the multi-start Simulated Annealing (SA) algorithm.

Runs multiple independent SA trials until the time limit is reached. It keeps track of the best solution found across all runs. Hyperparameters (cooling rate, steps per temperature) are adaptively set based on the problem size N.

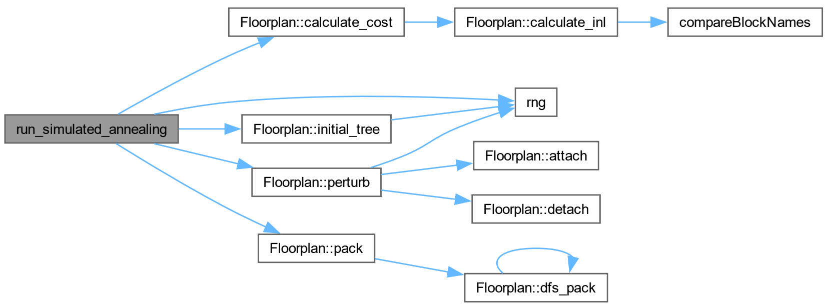
Parameters

base_fp	A Floorplan object containing the initial block data.
time_limit	The total time allowed for the optimization.

Returns

The Floorplan object representing the best solution found.

Here is the call graph for this function:



Here is the caller graph for this function:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

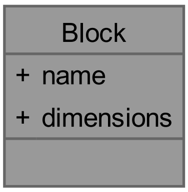
Chapter 4

Class Documentation

4.1 Block Struct Reference

Represents a fundamental module (or macro) to be placed.

Collaboration diagram for Block:



Public Attributes

- string name
    Name of the block (e.g., "MM0").
- vector< Dimension > dimensions
    A vector of possible dimensions for this block.

4.1.1 Detailed Description

Represents a fundamental module (or macro) to be placed.

4.1.2 Member Data Documentation

4.1.2.1 dimensions

```
vector<Dimension> Block::dimensions
```

A vector of possible dimensions for this block.

4.1.2.2 name

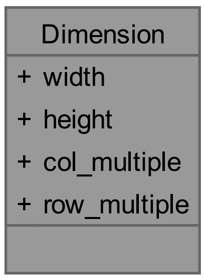
```
string Block::name
```

Name of the block (e.g., "MM0").

4.2 Dimension Struct Reference

Stores one possible dimension and layout for a block. A single block can have multiple `Dimension` options, representing different shapes or rotations.

Collaboration diagram for Dimension:



Public Attributes

- double width
- double height

  Width and height of this dimension option.
- int col_multiple
- int row_multiple

  Column and row multiples (for array-like structures).

4.2.1 Detailed Description

Stores one possible dimension and layout for a block. A single block can have multiple `Dimension` options, representing different shapes or rotations.

4.2.2 Member Data Documentation

4.2.2.1 col_multiple

`int Dimension::col_multiple`

4.2.2.2 height

`double Dimension::height`

Width and height of this dimension option.

4.2.2.3 row_multiple

`int Dimension::row_multiple`

Column and row multiples (for array-like structures).

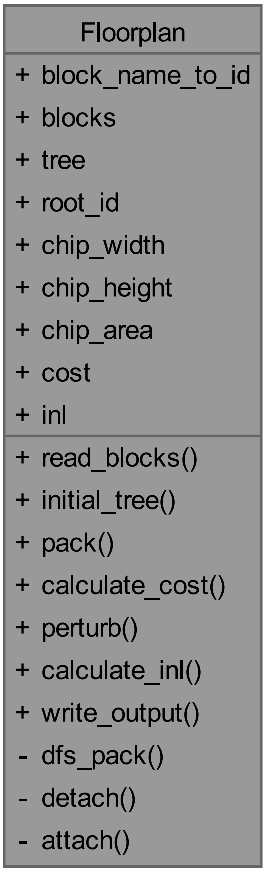
4.2.2.4 width

`double Dimension::width`

4.3 Floorplan Class Reference

Manages the entire floorplanning process using a B∗-Tree and Simulated Annealing.

Collaboration diagram for Floorplan:



Public Member Functions

- void read_blocks (const string &filename)

  Reads block information from a specified .block file.
- void initial_tree ()

  Generates an initial B∗-Tree.
- void pack ()

  Calculates the actual coordinates of all blocks based on the current B∗-Tree.
- void calculate_cost ()

  Calculates the total cost of the current layout.
- void perturb ()

  Applies a random perturbation to the B∗-Tree to generate a new solution.
- void calculate_inl ()

  Calculates the Integral Non-Linearity (INL) as specified by the problem requirements.
- void write_output (const string &filename)

  Writes the final layout to a file in the specified format. The output is sorted by block name before writing.

Public Attributes

- map< string, int > block_name_to_id

  Map for quick lookup of a block's index by its name.
- vector< Block > blocks

  Stores all block definitions read from the input file.
- vector< Node > tree

  The B∗-Tree structure, where each node corresponds to a block.
- int root_id = -1

  Index of the root node of the B∗-Tree.
- double chip_width = 0.0
- double chip_height = 0.0
- double chip_area = 0.0

  Dimensions and area of the resulting layout.
- double cost = 1e18

  The current layout's cost function value, initialized to a large number.
- double inl = 0.0

  The current layout's Integral Non-Linearity value.

Private Member Functions

- void dfs_pack (int node_id, map< double, double > &contour)

  Recursive helper for pack(), placing blocks using DFS and a contour line.
- int detach (int u)

  Detaches a node `u` *from the B∗-Tree.*
- void attach (int u, int p, bool is_left)

  Attaches node `u` *as a child of node* `p`*.*

4.3.1 Detailed Description

Manages the entire floorplanning process using a B∗-Tree and Simulated Annealing.

This class encapsulates all data and algorithms, including file I/O, B∗-Tree representation, packing algorithm, cost calculation, and optimization.

4.3.2 Member Function Documentation

4.3.2.1 attach()

```
void Floorplan::attach (
            int u,
            int p,
            bool is_left) [private]
```

Attaches node `u` as a child of node `p`.

Parameters

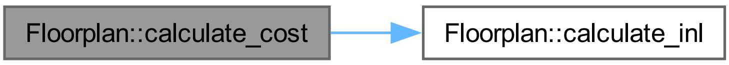
u	The index of the node to attach.
p	The index of the new parent node.
is_left	If true, attach as a left child; otherwise, as a right child.

4.3.2.2 calculate_cost()

```
void Floorplan::calculate_cost ()
```

Calculates the total cost of the current layout.

The cost function is a weighted combination of chip area, aspect ratio (AR), and Integral Non-Linearity (INL). Here is the call graph for this function:



4.3.2.3 calculate_inl()

```
void Floorplan::calculate_inl ()
```

Calculates the Integral Non-Linearity (INL) as specified by the problem requirements.

This method follows these steps:

1. Calculates the geometric center of the layout (Xc, Yc).
2. For each block, calculates the squared Euclidean distance from its center to (Xc, Yc).
3. Sorts these distances based on the block names in ascending order.
4. Computes the cumulative sum of these sorted distances, S_actual(n).
5. Performs a least-squares linear regression on S_actual(n) to find the ideal line S_ideal(n) = an + b.
6. The INL is the maximum absolute deviation between S_actual(n) and S_ideal(n).

4.3.2.4 detach()

```
int Floorplan::detach (
            int u) [private]
```

Detaches a node `u` from the B∗-Tree.

This is a key part of the move operation. It carefully handles the children of `u` to maintain a valid tree structure after `u` is removed.

Parameters

u	The index of the node to detach.

Returns

The index of the child node that was promoted to replace `u`'s position.

4.3.2.5 dfs_pack()

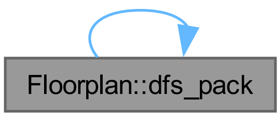
```
void Floorplan::dfs_pack (
            int node_id,
            map< double, double > & contour)  [private]
```

Recursive helper for pack(), placing blocks using DFS and a contour line.

Parameters

node↔_id	The index of the current node to place.
contour	Reference to the contour line map, which is updated during recursion.

Here is the call graph for this function:



4.3.2.6 initial_tree()

```
void Floorplan::initial_tree ()
```

Generates an initial B∗-Tree.

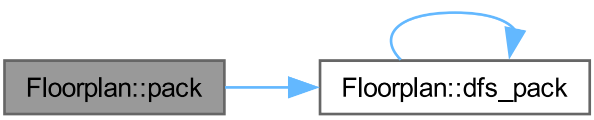
It first shuffles all blocks randomly and then constructs a left-skewed tree as the starting solution. A random dimension is also selected for each block.

4.3.2.7 pack()

```
void Floorplan::pack ()
```

Calculates the actual coordinates of all blocks based on the current B∗-Tree.

This is a core step that translates the abstract tree representation into a concrete physical layout. It uses a contour line algorithm to place blocks as compactly as possible. Here is the call graph for this function:



4.3.2.8 perturb()

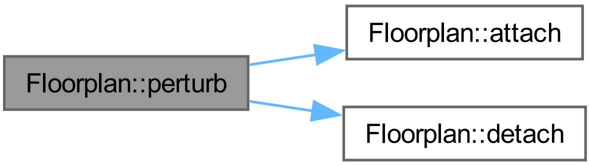
```
void Floorplan::perturb ()
```

Applies a random perturbation to the B∗-Tree to generate a new solution.

It randomly chooses one of three operations with certain probabilities:

1. Change a block's dimension (or rotate it).
2. Swap the contents of two nodes (swapping the blocks they represent).
3. Move a node to a different position in the tree.

Here is the call graph for this function:



4.3.2.9 read_blocks()

```
void Floorplan::read_blocks (
             const string & filename)
```

Reads block information from a specified .block file.

This method parses the input file, which can have complex formats where a single line contains multiple dimension definitions enclosed in parentheses.

Parameters

filename	Path to the input .block file.

4.3.2.10 write_output()

```
void Floorplan::write_output (
             const string & filename)
```

Writes the final layout to a file in the specified format. The output is sorted by block name before writing.

Parameters

filename	Path to the output file.

4.3.3 Member Data Documentation

4.3.3.1 block_name_to_id

```
map<string, int> Floorplan::block_name_to_id
```

Map for quick lookup of a block's index by its name.

4.3.3.2 blocks

```
vector<Block> Floorplan::blocks
```

Stores all block definitions read from the input file.

4.3.3.3 chip_area

```
double Floorplan::chip_area = 0.0
```

Dimensions and area of the resulting layout.

4.3.3.4 chip_height

```
double Floorplan::chip_height = 0.0
```

4.3.3.5 chip_width

```
double Floorplan::chip_width = 0.0
```

4.3.3.6 cost

```
double Floorplan::cost = 1e18
```

The current layout's cost function value, initialized to a large number.

4.3.3.7 inl

```
double Floorplan::inl = 0.0
```

The current layout's Integral Non-Linearity value.

4.3.3.8 root_id

```
int Floorplan::root_id = -1
```

Index of the root node of the B∗-Tree.

4.3.3.9 tree

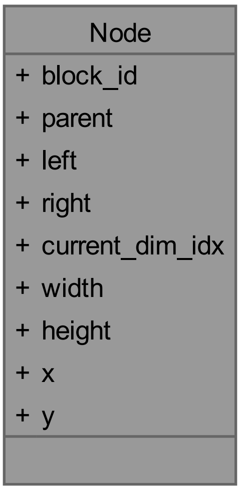
```
vector<Node> Floorplan::tree
```

The B∗-Tree structure, where each node corresponds to a block.

4.4 Node Struct Reference

Represents a node in the B∗-Tree. Each node corresponds to a specific block in the floorplan.

Collaboration diagram for Node:



Public Attributes

- int block_id

    Index into the main `blocks` *vector, identifying the block this node represents.*
- int parent = -1
- int left = -1
- int right = -1

    Indices of parent, left child, and right child in the `tree` *vector.*
- int current_dim_idx = 0

    Index into `blocks[block_id].dimensions`*, specifying the current dimension in use.*
- double width = 0.0
- double height = 0.0

    The width and height corresponding to the `current_dim_idx`*.*
- double x = 0.0
- double y = 0.0

    The final bottom-left coordinates after packing.

4.4.1 Detailed Description

Represents a node in the B∗-Tree. Each node corresponds to a specific block in the floorplan.

4.4.2 Member Data Documentation

4.4.2.1 block_id

```
int Node::block_id
```

Index into the main `blocks` vector, identifying the block this node represents.

4.4.2.2 current_dim_idx

```
int Node::current_dim_idx = 0
```

Index into `blocks[block_id].dimensions`, specifying the current dimension in use.

4.4.2.3 height

```
double Node::height = 0.0
```

The width and height corresponding to the `current_dim_idx`.

4.4.2.4 left

```
int Node::left = -1
```

4.4.2.5 parent

```
int Node::parent = -1
```

4.4.2.6 right

```
int Node::right = -1
```

Indices of parent, left child, and right child in the `tree` vector.

4.4.2.7 width

```
double Node::width = 0.0
```

4.4.2.8 x

```
double Node::x = 0.0
```

4.4.2.9 y

```
double Node::y = 0.0
```

The final bottom-left coordinates after packing.